33 Netzwerk-Programmierung (Sockets)

33.1 Computernetzwerke, Schichtenmodell

Unter einem Computernetzwerk versteht man eine Ansammlung von Computern oder prozessorbestückten Geräten, die miteinander kommunizieren, d.h. Daten versenden und empfangen können. Computernetzwerke spielen in der modernen Industriegesellschaft, die auf den raschen Transport von Daten als Informationsträger angewiesen ist, eine außerordentlich große Rolle. Sie haben sich mit dem **Internet** sowohl weltweit als auch mit dem **Intranet** in einem beschränkten, geschützten Bereich explosionsartig verbreitet. Die meisten Computer sind heutzutage in irgendeiner Form mit anderen Computern oder Peripheriegeräten vernetzt.

Für die Kommunikation in einem Computernetz sind spezielle Programme zuständig, wobei die Programmiersprache Java für Netzwerkanwendungen, die nicht allzu hardwarenahe sind, eine gut ausgebaute, einfach zu benützende Klassenbibliothek innerhalb der JFC zur Verfügung stellt. Dies ist historisch zu verstehen, denn die Spezialisten, die auf dem Gebiet der Web- und Netzwerkprogrammierung tätig waren, bildeten in der Entwicklung von Java eine bevorzugte Zielgruppe. Anfänglich wurde Java fast ausschließlich für **Applets** verwendet.

Man bezeichnet ein einzelnes Gerät, mit dem in einem Netzwerk kommuniziert werden kann, als einen **Knoten** (**node**). Wenn es sich dabei um einen vollwertigen Computer handelt, spricht man sehr allgemein auch von einem **Host**. Von einem höheren Standpunkt aus betrachtet, den wir als Java-Programmierer einnehmen wollen, spielt es keine Rolle, wie die Knoten physikalisch und elektronisch vernetzt und wie die Daten auf den Übertragungsmedien codiert sind. Oft handelt es sich um Drahtleitungen (Netzwerkkabel), immer häufiger sind im lokalen Bereich aber auch drahtlose Hochfrequenzverbindungen (Wireless Local Area Network, **WLAN**) im Einsatz.

Der Austausch von Daten in einem Computernetzwerk ist ein komplexer Prozess, der durch die Einführung eines Schichtenmodells strukturiert werden kann. Analog zum Softwaredesign „versteckt“ man dabei die Komplexität, d.h. die Implementierungsdetails, einer tiefer liegenden Schicht, und legt nur die Schnittstelle zu der darüber liegenden Schicht offen. Obschon in der Netzwerktheorie vom siebenschichtigen **OSI-Modell** (Open System Interconnection-Modell) ausgegangen wird, genügt für die allermeisten Fälle ein **vierschichtiges Netzwerkmodell**, welches nur das Internet-Protokoll (**IP**) berücksichtigt. (Abb. 33.1). Auf

jeder Schicht gibt es eine Sammlung von Regeln, **Protokolle** genannt, an die sich die Hersteller von Hard- und Software halten müssen (Request For Comments, **RFC**).

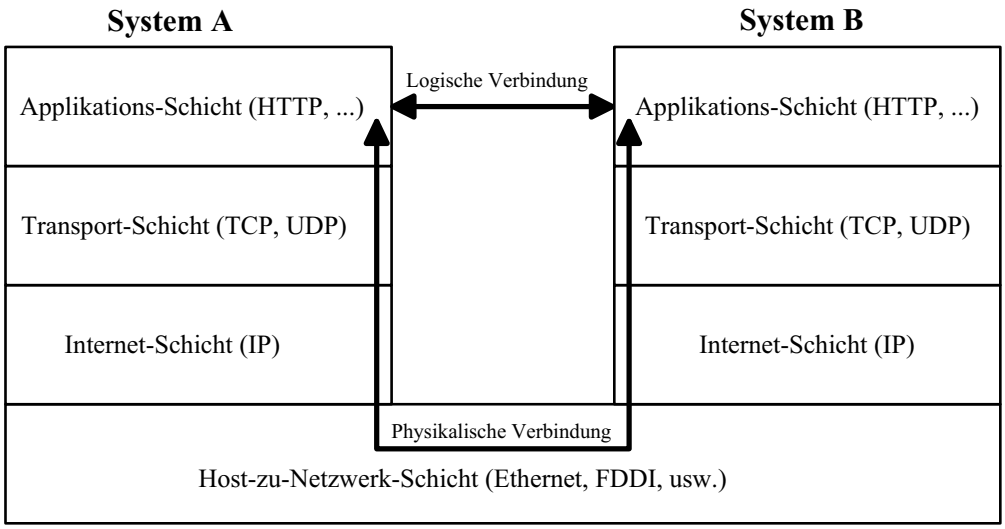

Abb. 33.1 *Vierschichten-Netzwerkmodell*

Vernetzte Computer müssen eine eindeutige Identifikation, eine **Netzwerkadresse** besitzen. Im heute weit verbreiteten **Ethernet-Netzwerk** besitzt jeder Knoten ein Netzwerkinterface mit einer eindeutigen, zwischen den Herstellern vereinbarten **MAC-Adresse** (**Medium Access Control**), welche 48 bit umfasst und meist in 12 Hexziffern angegeben wird. Darüber hinaus wird im Internet-Protokoll festgelegt, dass jeder Knoten zur Identifizierung eine eindeutige **Internet-Adresse** (**IP-Adresse**) besitzen muss, die aus 4 bytes besteht und normalerweise in „gepunkteter" Form (**dotted quad format**) mit 4 Zahlen zwischen 0 und 255 angegeben wird (beispielsweise 130.92.13.177). (Der neue Standard IPv6 sieht für die Internet-Adresse 16 bytes vor.) Der IP-Adresse ist in der Regel ein leichter handhabbarer **Hostname** (**IP-Alias**) zugeordnet, der weltweit von Domain Name Servern (**DNS**) verwaltet wird, beispielsweise `klnt1.unibe.ch`. Ein Host kann auch mehrere Hostnamen besitzen.

In sich abgeschlossene Teile eines Netzwerks können mit dem Internet auch über einen Router verbunden sein, der die IP-Adressen übersetzt (NAT, Network Address Translation) und nach außen nicht die inneren IP- und MAC-Adresse kommunziert. Eine umfassende Eindeutigkeit der IP-Adressen ist also nicht zwingend nötig.

Das IP-Protokoll legt fest, dass die Daten in Form von **Paketen** übertragen werden. Ein einzelnes Paket (**IP datagram**) besteht aus einem **Kopf** (**header**) von 20 bis 60 bytes und nachfolgend bis zu **65565 Datenbytes**. Der genaue Aufbau spielt für die Netzwerk-Programmierung in Java keine Rolle. Wir halten fest, dass der Kopf genügend Informationen enthält, um die Pakete, auch wenn diese über verschiedene Vermittlungswege laufen und daher nicht in der richtigen Reihenfolge beim Emfänger eintreffen, wieder in der ursprüngli-

chen Reihenfolge zusammen zu setzen und fehlerhafte Pakete zu erkennen. **TCP** (Transmission Control Protocol) ist in hohem Maß **fehlertolerant**, da der Sender auf eine Bestätigung (Acknowledgement) wartet, dass die Pakete beim Empfänger korrekt angekommen sind und andernfalls die Pakete neu versendet. **UDP** (User Datagram Protocol) verzichtet auf eine Fehlerkorrektur und ist dadurch rund um den Faktor 3 schneller als TCP. UDP eignet sich vor allem für die schnelle Datenübertragung von Sound und Video, bei der gewisse Fehler zugunsten eines regelmäßigen Datenstroms in Kauf genommen werden. Java unterstützt beide Protokolle gleichermaßen, hingegen nicht das **ICMP** (Internet Control Message Protocol), das vom bekannten Netzwerkwerkzeug **ping** benützt wird.

Auf der obersten Schicht, jener der Applikationsprotokolle, wird beschrieben, wie einzelne **Dienste** miteinander kommunizieren. Zu den wichtigsten gehören: Web, Mail und Datentransfer. Die Kommunikation zwischen einem Web-Server und einem Web-Browser wird durch das **HTTP** (HyperText Transfer Protocol), zwischen einem Mail-Server und einem Mail-Client durch das **SMTP** (Simple Mail Transfer Protocol) und das **POP3** (Post Office Protocol) und für den Datentransfer durch das **FTP** (File Transfer Protocol) beschrieben.

Es darf nicht vergessen werden, dass die Rechnerkommunikation auch bei Echtzeitsystemen eine große Rolle spielt. Heute wird im Zusammenhang mit industriellen Prozessen und wissenschaftlichen Experimenten oft eine Rechnerhierarchie eingesetzt, wobei dedizierte Prozessrechner (oft Microcontroller) die unmittelbare Steuerung des Prozesses oder Experiments übernehmen. Der Prozessrechner kommuniziert in der Regel mit einem konventionellen Computer, der Statusinformationen vom Prozessrechner erhält, diese verarbeitet und Steuerkommandos an den Prozessrechner zurückgibt. Die Verbindung zwischen den beiden Rechnern kann konventionell über eine serielle oder USB-Schnittstelle erfolgen, immer mehr wird aber auch hier das Ethernet eingesetzt. Der Prozessrechner wird vielfach in Assembler oder C, neuerdings auch in Java (Realtime Java, Java Micro Edition) programmiert, der Steuerrechner in einer höheren Programmiersprache oder Applikationssprache (Mathlab, LabVIEW usw.)

33.2 Client-Server-Modell

In vielen Fällen ist es sinnvoll, gewisse Dienste und Informationen zentral zu verwalten und sie über ein Netzwerk anderen Computern zur Verfügung zu stellen. Daher lassen sich Computer in einem Netzwerk einteilen in **Server**, manchmal auch **Hosts** im engeren Sinn genannt, welche Dienste und Informationen anbieten, und **Clients**, welche diese benützen. Ein einzelner Computer kann aber auch gleichzeitig Client- wie Serverfunktionalitäten haben.

*Diese hierarchische Abhängigkeit ist für die Netzwerk-Programmierung von großer Wichtigkeit, allerdings nicht zwingend. Zwei Computer können auch als gleich gestellte Partner Daten austauschen (**peer-to-peer**). Obschon Java dieses Konzept nicht direkt unterstützt, kann es dadurch realisiert werden, dass ein Computer gleichzeitig die Rolle eines Servers und eines Clients übernimmt.*

Typischerweise wartet ein Serverdienst (in der Sprache von Unix ein **Daemon**) darauf, dass ein Client eine Anfrage (**Request**) startet. Der Server öffnet dazu einen individuellen Datenpfad zum Client (**Socket**), verarbeitet die Anfrage und liefert dem Client über den Datenpfad eine entsprechende Antwort (**Response**). Durch Multitasking bzw. Multithreading ist der Server meist in der Lage, gleichzeitig bzw. nebeneinander mehrere Clients zu bedienen (Abb. 33.2).

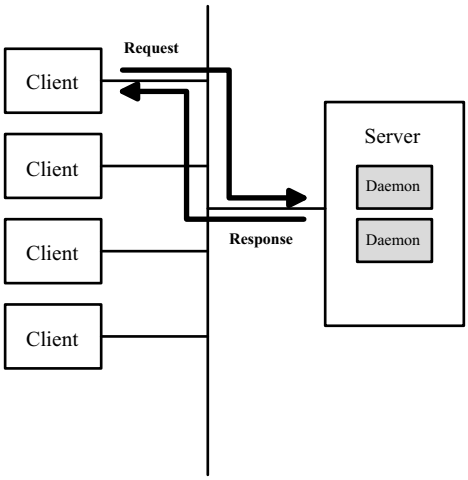

Abb. 33.2 *Client-Server-Modell (Request/response model)*

Die Anfrage und die Antwort müssen gemäß einem zwischen Client und Server vereinbarten Protokoll erfolgen. Obschon der Server seine Dienste ständig zur Verfügung stellen sollte und in diesem Sinn dem Client unterstellt ist, ist es ihm freigestellt, für gewisse Dienstleistungen ein Authentifizierungsverfahren durchzuführen oder sie gänzlich zu verweigern. Daher müssen die Daemons rigorosen Sicherheitsstandards genügen, um zu verhindern, dass ein Client nicht unter Umgehung der Sicherheitsbarrieren über eine **Hintertür** (**backdoor**) Daten einsehen, verändern oder sogar Programme ausführen kann. Das Programmieren von Daemon-Software ist diesbezüglich besonders anspruchsvoll.

Damit auf einem Server mehrere Daemons ihre Dienste anbieten können, gibt es zu einer bestimmten IP-Adresse bis zu 65535 verschiedene **Ports**. Die einzelnen Daemons **binden** beim Starten einen bestimmten Port, an den der Client seine Anfrage richten muss. Bestimmte Portnummern sind standardisiert, die wichtigsten sind in Tabelle 33.1 aufgeführt.

Service	**Port**	**Beschreibung**
daytime	13	Liefert aktuelle Tageszeit des Servers
ftp-data	20	FTP-Datentransfer
FTP	21	FTP-Kommando
Telnet	23	Terminal für Kommandozeilen-Befehle
SMTP	25	E-Mail versenden
HTTP	80	Web-Server

POP3	110	E-Mails abholen
RMI Registry	1099	Remote Method Invocation für Java

Tab. 33.1 *Standardisierte IP-Ports*

33.3 Client- und Server-Sockets

TCP-Sockets sind eine Erfindung von Berkeley Unix und ermöglichen es, eine Netzwerkverbindung wie einen gewöhnlichen Stream mit read- und write-Operationen aufzufassen. Streams sind bekanntlich eine der wichtigsten Erfindungen von Unix mit dem Ziel, alle I/O-Operationen über Tastatur, Bildschirm, Dateien, Netzwerk usw. gleichartig zu behandeln. Sockets schirmen den Programmierer von der enormen Komplexität der tatsächlich auf dem Netzwerk ablaufenden Prozesse ab. Das Verfahren hat sich seit über 40 Jahren bestens bewährt und wurde daher auch in Java übernommen.

Einen **Socket** kann man als Zugangspforte eines IP-Verbindungskanals zwischen zwei Hosts auffassen. Für einen Client sind die wichtigsten Socket-Operationen:

- Verbindung erstellen (öffnen, open)
- Daten senden (schreiben, send)
- Daten empfangen (lesen, read)
- Verbindung beenden (schließen, close).

Für einen Server kommen noch folgende Operationen hinzu:

- Bindung eines IP-Ports (binden, bind)
- Verbindungsanforderung annehmen (accept).

Im Zusammenhang mit der Netzwerkprogrammierung benötigen wir immer einen Client und einen Server. In einem ersten Beispiel wollen wir einen Client programmieren, benötigen aber zum Testen des Programms einen bereits vorhandenen Server. Dazu eignet sich ein **Daytime-Server**, der einzig einige Zeilen Text im ASCII-Format mit aktuellem Datum und Uhrzeit an jeden Client zurück schickt, der eine IP-Verbindung auf Port 13 erstellt.

Es gibt mehrere Daytime-Server im Internet, beispielsweise der Server der Physikalisch-Technischen Bundesanstalt mit dem gegenwärtigen Hostnamen: ntp1.ptb.de. Falls man das Client-Programm ohne Zugang zum Internet entwickeln will, so kann auf dem Entwicklungsrechner der später beschriebene Daytime-Server gestartet werden. Dazu compilieren und starten wir `DaytimeServer.java` *und verwenden für den Client den Hostname 'localhost'.*

Das Client-Programm ist wegen der guten Unterstützung der IP-Sockets im Package `java.net` außerordentlich einfach. Bei der Instanzierung eines Objekts der Klasse `Socket` versucht der Client, eine TCP-Verbindung mit dem Server herzustellen. Falls dies misslingt, wird eine `IOException` geworfen, mit der man den Benutzer über die Schwierigkeiten informieren kann. Kommt die Verbindung zustande, so ist der Socket geöffnet und man kann

mit Hilfe der zurückgegebenen Socket-Referenz mit dem Server kommunizieren. Um vom Server gesendete Daten abzuholen, verschafft man sich am besten mit `getInputStream()` einen InputStream, den wir zweckmäßigerweise in einen BufferdReader umwandeln, da wir ASCII-codierte Daten in Zeilenform (abgeschlossen mit einem `\n`) erwarten.

Mit `readLine()` holen wir Zeile um Zeile vom Socket ab und schreiben diese in ein Console-Fenster. Es ist wichtig zu wissen, dass `readLine()` den weiteren Programmablauf **blockiert** (**program stalling**), solange der Server keine Daten zurückliefert. Nach einer gewissen Wartezeit, die man mit `setSoTimeout()` einstellen kann, wird allerdings eine `SocketTimeoutException` geworfen, die man abfangen kann, um den Benutzer über den Fehlschlag zu orientieren.

Wir sind sorgfältig darauf bedacht, alle Ressourcen im Zusammenhang mit dem geöffneten Socket wieder explizit abzugeben, indem wir in einem finally-Block, der in jedem Fall durchlaufen wird, sowohl den Stream als auch den Socket **schließen**.

Im Normalfall gibt zwar das Java Runtime-System, bzw. der Garbage collector, die Ressourcen bei Programmende wieder frei und ein geschlossener Stream schließt auch den Socket. Wir müssen aber auch mit außerordentlichen Umständen rechnen, bei denen an unerwarteten Stellen eine Exception geworfen wird. Da aufgebrauchte Ressourcen zu den sehr schwierig zu behebenden Programmfehlern gehören, programmieren wir immer vorsichtig und defensiv und geben die Ressourcen explizit frei.

```
// DaytimeClient.java

import java.net.*;
import java.io.*;
import ch.aplu.util.*;

public class DaytimeClient
{
  private static final String HOSTNAME = "localhost";
  private static final int PORT = 13;

  public DaytimeClient()
  {
    BufferedReader reader = null;
    Socket socket = null;
    String line;
    try
    {
      socket = new Socket(HOSTNAME, PORT);
      System.out.println("Connection established");
      socket.setSoTimeout(15000); // 15 sec

      reader = new BufferedReader(
          new InputStreamReader(socket.getInputStream()));

      while ((line = reader.readLine()) != null)
```

```
      {
        System.out.println(line);
      }
    }
    catch (IOException ex)
    {
      System.out.println("Error " + ex);
    }
    finally
    {
      try
      {
        if (reader != null)
          reader.close();
        if (socket != null)
          socket.close();
      }
      catch (IOException ex)
      {}
    }
  }

  public static void main(String args[])
  {
    Console.init();
    new DaytimeClient();
  }
}
```

Die Ausführung des Programms für den lokalen Daytime-Server ist in Abb. 33.3 gezeigt.

Abb. 33.3 *Die Ausgabe von DaytimeClient für den Hostname 'localhost'*

Als Nächstes wenden wir uns der Programmierung eines Servers zu. Dabei erkennen wir, dass es wohl gewisse Analogien zum Client gibt, einige Aspekte aber neu sind. Als Beispiel entwickeln wir einen Daytime-Server, der einem Client, der eine IP-Verbindung auf Port 13 erstellt, einige Zeilen Text mit dem aktuellem Datum und der momentanen Uhrzeit zusendet. Der Konstruktor der Klasse `ServerSocket()` öffnet einen Server-Socket und bindet das Server-Programm (den Daemon) an einen bestimmten IP-Port. Clients, welche jetzt eine IP-Verbindung zu diesem Port des Servers aufbauen, erhalten bereits eine Erfolgsmeldung,

obschon der Server noch gar keine Dienste zur Verfügung stellt. Die Clients werden dabei in eine Verarbeitungs-Warteschlange gefügt.

Um einen Client zu bedienen, muss der Server mit der Methode `accept()` den zuerst angemeldeten Client übernehmen. Hat sich noch kein Client angemeldet, so **blockiert** der Aufruf von `accept()` den weiteren Ablauf des Programms, da `accept()` erst zurückkommt, wenn ein Client angenommen werden kann. Damit der Server mehrere Clients hintereinander bedienen kann, setzen wir `accept()` in eine Endlosschlaufe. Mit der Socket-Instanz, welche `accept()` zurückgibt, können wir einen `OutputStream` holen, den wir zweckmäßigerweise in einen `PrintWriter` verwandeln, um die Daten in Form von einigen Zeilen ASCII-Code an den Client zurück zu schicken. Wir gewöhnen uns daran, mit `flush()` den Stream zu leeren, damit der ganze Inhalt des Streambuffers zum Client übertragen wird, obschon dies beim Schließen des Streams automatisch gemacht würde.

Wir sehen zu Demonstrationszwecken noch vor, den Server künstlich lahm zu legen, indem wir 10 s mit der Antwort zuwarten, falls wir `isLazy` `true` setzen.

```
// DaytimeServer.java

import java.net.*;
import java.io.*;
import java.util.*;
import ch.aplu.util.*;

public class DaytimeServer
{
  private static final int SERVICE_PORT = 13;
  private boolean isLazy = false;

  public DaytimeServer()
  {
    try
    {
      ServerSocket server = new ServerSocket(SERVICE_PORT);
      if (isLazy)
        System.out.print("Lazy ");
      else
        System.out.print("Quick ");
      System.out.println("daytime service started");

      while (true)
      {
        Socket client = server.accept();
        System.out.println("Received request on " +
                           new Date() + " from " +
                           client.getInetAddress() + ":" +
                           client.getPort());
        PrintWriter pr =
            new PrintWriter(client.getOutputStream());
```

```
        if (isLazy)
          Console.delay(10000);  // lazy server
        Date now = new Date();
        pr.println("The current daytime is: " + now);
        pr.println("Goodbye");
        pr.flush();
        pr.close();
        client.close();
      }
    }
    catch (BindException ex)
    {
      System.err.println("Service already running on port " +
                         SERVICE_PORT);
    }
    catch (IOException ex)
    {
      System.err.println("Error " + ex);
    }
  }

  public static void main(String args[])
  {
    Console.init();
    new DaytimeServer();
  }
}
```

Setzen wir `isLazy` auf `true`, so ist der Server so faul, dass ein zweiter Client, den man kurz nach einem ersten startet, die gewünschte Verbindung nicht mehr aufbauen kann, da der Timeout von 15 Sekunden, welcher mit `setSoTimeout()` eingestellt wird, nicht lange genug ist. Es handelt sich dabei um ein grundsätzliches Problem im Zusammenhang mit dem Client-Server-Modell, da man immer damit rechnen muss, dass kurz hintereinander mehrere Clients die Dienste des Servers beanspruchen wollen, währenddessen dieser noch mit dem vorhergehenden Request beschäftigt ist. Wir lösen dieses Problem durch Multithreading.

33.4 Multithreaded Server

Oft ist die Verarbeitungszeit auf dem Server kurz im Vergleich zur Übertragungszeit der Daten vom Server zum Client. Kann der Server nur einen Client bedienen, so ist während der Datenübertragung für längere Zeit der Server oder zumindest der entsprechende Service blockiert. Dies führt zu einem schlechten Antwortverhalten und zur Verschwendung von Rechnerleistung. Viel effizienter ist es, auf dem Server für jeden Client einen eigenen Prozess zu starten.

Obschon jetzt die Antwortzeiten viel besser werden, verschwenden wir viele Rechnerressourcen, da jeder Prozess als eigenständige Applikation geladen und ausgeführt wird.

Eine elegantere Lösung liegt auf der Hand: Man lässt jeden Client in einem eigenen Thread derselben Applikation laufen. Dieser wird entweder bei der Anmeldung eines Clients neu erzeugt oder, um das Programm noch effizienter zu machen, aus einem Pool von bereits vorher erzeugten Threads bezogen. Die Netzwerkprogrammierung liefert darum auch ein perfektes Übungsgelände für den sinnvollen Einsatz von Threads. In Java sind die beiden verwandten Bereiche hervorragend ausgebaut.

Mit unseren guten Kenntnissen über Threads schreiben wir ohne Mühe einen eleganten `DaytimeThreadedServer`, bei dem nach dem `accept()` für jeden Client eine neue Instanz der Klasse `OutputThread` erzeugt wird, welche die Verarbeitung des neu angemeldeten Clients übernimmt. Da wir `OutputThread` nicht als innere Klasse der Applikationsklasse deklarieren wollen, müssen wir bei der Konstruktion zwei Parameter übergeben. Dies ist sowieso besser, da es von der Thread-Klasse mehrere Instanzen geben kann und es gefährlich wäre, diese auf gemeinsame Instanzvariablen der Applikationsklasse zugreifen zu lassen. Starten wir diesen Server mit `isLazy = true`, so werden auch zwei kurz nacheinander gestartete Clients korrekt bedient.

```
// DaytimeThreadedServer.java

import java.net.*;
import java.io.*;
import java.util.*;
import ch.aplu.util.*;

class OutputThread extends Thread
{
  private OutputStream os;
  private boolean isLazy;

  OutputThread(OutputStream os, boolean isLazy)
  {
    this.os = os;
    this.isLazy = isLazy;
  }

  public void run()
  {
    PrintWriter pr =
        new PrintWriter(new OutputStreamWriter(os));

    if (isLazy)
      Console.delay(10000); // lazy server
    Date now = new Date();
    try
    {
      InetAddress address = InetAddress.getLocalHost();
```

```
      pr.println("Current daytime at " + address + " is:");
      pr.println(now);
      pr.println("Goodbye");
      pr.flush();
      pr.close();
    }
    catch (Exception ex)
    {
      System.out.println("Error " + ex);

    }
  }
}

public class DaytimeThreadedServer
{
  private static final int SERVICE_PORT = 13;
  private boolean isLazy = false;

  public DaytimeThreadedServer()
  {
    try
    {
      ServerSocket server = new ServerSocket(SERVICE_PORT);
      if (isLazy)
        System.out.print("Lazy ");
      else
        System.out.print("Quick ");
      System.out.println("daytime service started(threaded)");

      while (true)
      {
        Socket client = server.accept();
        System.out.println("Received request on " +
                           new Date() + " from " +
                           client.getInetAddress() + ":" +
                           client.getPort());
        Thread output = new OutputThread(client.
                                         getOutputStream(),
                                         isLazy);
        output.start();
      }
    }
    catch (BindException ex)
    {
      System.err.println("Service already running on port " +
                         SERVICE_PORT);
    }
```

```
    catch (IOException ex)
    {
      System.err.println("Error " + ex);
    }
  }

  public static void main(String args[])
  {
    Console.init();
    new DaytimeThreadedServer();
  }
}
```

33.5 Telnet, Grundlagen von HTTP

Wir sind bereits in der Lage, ein Clientprogramm zu schreiben, das zum Testen von Netzwerkverbindungen von großem praktischem Nutzen ist. Das Programm baut beim Start eine Socket-Verbindung zu einem vorgegebenen Port des Servers auf und sendet die Tastatureingaben zeichen- oder zeilenweise zum Server. Die vom Server an den Client gesendeten Zeichen werden in einem Console-Fenster des Clients dargestellt.

Früher hat man Geräte hergestellt, welche an Mini- oder Großrechner angeschlossen waren und ausschließlich zur Datenerfassung und Datenanzeige dienten. Da man diese Geräte ***Terminal*** *nannte, bezeichnet man ein zu diesem Zweck entwickeltes Programm als* ***Terminalemulator*** *oder* ***Terminalprogramm****. Für Terminalemulatoren, welche die Daten mittels TCP/IP übertragen, hat sich der Name* ***Telnet*** *eingebürgert. Sie werden meist zusammen mit dem Betriebssystem ausgeliefert. Beispielsweise startet man in einem Console-Fenster von Windows-PCs mit dem Befehl* `telnet` *(unter Angabe des Hostnamens und des Ports) ein einfaches Telnet-Programm.*

Ein Terminalemulator muss in der Lage sein, quasi gleichzeitig auf Tastatureingaben und auf Daten, die vom Server eintreffen, zu reagieren. Da davon ausgegangen werden muss, dass Eingaben und Datenrückgabe unabhängig voneinander zu einem beliebigen Zeitpunkt erfolgen, muss die Programmlogik gut durchdacht sein. Es gehört sich auch, dass ein Telnet-Programm bei einem Zusammenbruch der Verbindung zum Server normal beendet werden kann. Es liegt auf der Hand, die zwei Aufgaben in zwei verschiedenen Threads zu behandeln, wobei einer davon der Applikations-Thread sein kann. Man entscheidet sich ohne wesentliche Vor- und Nachteile, ob die Tastatureingabe oder die Behandlung der Rückgabedaten in einem eigenen Thread verarbeitet wird.

Im nachfolgenden Beispiel verwenden wir für die Tastatureingaben und Datenausgabe ein Console-Fenster und verarbeiten die Datenrückgabe vom Server in der run-Methode der Klasse `InputThread`, die wir als innere Klasse deklarieren, damit der Zugriff auf gemeinsame Variablen erleichtert wird. Das Lesen der Rückgabedaten erfolgt mit dem InputStream der Socket-Instanz, den wir in einen BufferedReader verwandeln, um mit `readLine()` einzelne Zeilen abholen zu können. Wie wir wissen, ist die Methode `readLine()` blockierend, was bedeutet, dass das Programm auf dieser Zeile „hängen“ bleibt, solange keine Daten vom Server eintreffen. Falls allerdings die Verbindung zum Server unterbrochen wird, wirft

`readLine()` eine Exception, die wir dazu verwenden, die run-Methode und damit den Thread unter Freigabe aller Ressourcen zu beenden. Das Hauptprogramm wird darüber informiert, weil es mit `isClosed()` den Zustand der Verbindung ständig überprüft.

Wir leiten `Telnet` aus der Klasse `Console` ab, damit wir die print-Aufrufe ohne vorgestelltes `System.out` verwenden können. Jeder von der Tastatur eingelesenen Zeile fügen wir ein `<cr><lf>` an, wie es für die meisten Internet-Protokolle üblich ist. Mit `flush()` garantieren wir, dass der Zeichenbuffer geleert und damit die Zeile vollständig an den Server verschickt wird.

```
// Telnet.java

import java.io.*;
import java.net.*;
import ch.aplu.util.*;

public class Telnet extends Console
{
  private Socket socket;

  // --------------- Inner class ----------------------------
  class InputThread extends Thread
  {
    public void run()
    {
      BufferedReader in = null;
      try
      {
        in = new BufferedReader(
            new InputStreamReader(socket.getInputStream()));
        String line;
        while ((line = in.readLine()) != null)
          println(line);
      }
      catch (IOException ex)
      {}
      try
      {
        if (in != null)
          in.close();
        if (socket != null)
          socket.close();
      }
      catch (IOException ex)
      {}
    }
  }
  // --------------- End of inner class --------------------
```

```
  public Telnet()
  {
    print("Host? ");
    String host = readLine();
    print("Port: ");
    int port = readInt();
    try
    {
      socket = new Socket(host, port);
      PrintWriter out =
          new PrintWriter(socket.getOutputStream());
      new InputThread().start();
      println("Verbunden mit " + host +
              " am Port " + port);

      while (!socket.isClosed())
      {
        String line = Console.readLine();
        out.print(line);
        out.print("\r\n"); // Line terminator is <cr><lf>
        out.flush();
      }
      println("\nVerbindung unterbrochen.");
      println("Irgendeine Taste zum Beenden...");
      while (!kbhit())
      {}
      System.exit(0);
    }
    catch (UnknownHostException ex)
    {
      println("Verbindungsaufnahme misslungen.");
    }
    catch (SocketException ex)
    {
      println("Verbindungsaufnahme misslungen.");
    }
    catch (IOException ex)
    {
      ex.printStackTrace();
    }
  }

  public static void main(String[] args)
  {
    new Telnet();
  }
}
```

Um den Terminalemulator auszutesten, unternehmen wir einen lehrreichen Versuch und erstellen eine TCP-Verbindung zu irgendeinem Web-Server. Dazu wählen wir beispielsweise die Suchmaschine `www.google.com` auf Port 80. Nachdem die Verbindung zum Web-Server geöffnet ist, müssen wir uns mit dem **HTTP-Protokoll (HyperText Transfer Protocol)** verständigen. Um die Homepage von Google abzuholen, senden wir ihm unter strikter Einhaltung der Groß-Kleinschreibung und der Leerzeichen den Request

```
GET / HTTP/1.0
```

mit einer darauf folgenden Leerzeile (beide müssen mit einem <cr><lf> abgeschlossen sein). Dabei bedeutet GET, dass der Web-Server die Datei, deren URL angegeben wird, an den Client senden soll. Mit dem Slash wird das Wurzelverzeichnis des Web-Servers angesprochen und ohne weitere Dateiangabe sendet der Server üblicherweise die Datei `index.html` (je nach Servereinstellung auch `index.htm`, `default.html` oder `default.htm`). Die Web-Seite, welche in der Regel ASCII-Text mit HTML-Tags enthält, wird allerdings im Console-Fenster nur als Text dargestellt. Das Beispiel zeigt aber, dass das Web auf sehr einfachen Prinzipen aufbaut: Das Client-Programm, üblicherweise Web-Browser genannt, öffnet mit der angegebenen URL beim Web-Server einen IP-Socket und sendet einen HTTP-Request, mit einer relativen oder impliziten Angabe der gewünschten HTML-Datei. Der Server sendet dem Client diese Datei zurück, die der Web-Browser durch Interpretation der HTML-Tags dargestellt.

Wir erkennen auch, dass der Web-Server nach der Datenrückgabe die Socket-Verbindung sofort schließt und damit „vergessen“ hat, welcher Client gerade eben mit ihm in Verbindung war. Man nennt daher das HTTP-Protokoll in der Version 1.0 auch **zustandslos** (**stateless**). Für viele Client-Server-Applikationen ist es wichtig, über längere Zeit eine Kommunikation zwischen einem bestimmten Client und dem Server aufrecht zu erhalten, beispielsweise für Bestell- und Reservationssysteme. Wie wir in Kap. 41 sehen werden, lässt sich eine Session mit dem HTTP-Protokoll nur mit größerem zusätzlichem Aufwand realisieren.

*Die HTTP Version 1.1 erlaubt, eine Verbindung offen (**keep-alive**) zu halten, bis eine der beiden Partner einen close-Request sendet. Besteht eine Web-Seite beispielsweise aus mehreren Bildern, so muss nicht mehr für jedes Bild ein neuer Socket geöffnet werden. Von dieser Technik wird aber noch wenig Gebrauch gemacht.*

Von diesem Erfolg ermutigt, schreiben wir zur Übung ein Programm, mit dem wir die gesellschaftliche Relevanz einiger Begriffe untersuchen können. Wir lassen dazu die Suchmaschine *Google* nach den Links zu einem vorgegebenen Wort suchen, schreiben aber nur die Anzahl Treffer aus. Da in diesem Fall das Senden der Anfrage und das Abholen der Antwort zeitlich hintereinander erfolgen, können wir auf einen eigenen Thread verzichten. Wir schreiben die Antwort des Web-Servers, die ziemlich lang sein kann, zuerst in einen ByteArrayOutputStream, der wesentlich effizienter als ein String ist und die angenehme Eigenschaft besitzt, selbständig bei Bedarf zu wachsen.

Um herauszufinden, welchen Request wir Google bei einer Suche nach einem einzigen Suchwort senden müssen, führen wir eine Abfrage mit einem Browser durch und analysieren die angezeigte URL. Dabei entdecken wir durch einige weitere Versuche, dass beispielsweise für das Suchwort *ferien* die URL

```
http://www.google.com/search?q=ferien
```

genügt. Dies könnte sich zwar ändern, falls Google die Logik der Web-Seite wesentlich überarbeitet. Wir erkennen auch, dass sich die Anzahl Treffer auf einer der obersten Zeilen hinter *of about* befindet, wobei die Zahl in Fettschrift, d.h. zwischen den HTML-Tags und , dargestellt wird. Unser Programm besteht im Wesentlichen aus vier Teilen: Zuerst muss das Suchwort eingelesen werden, nachfolgend wird der HTTP-Request an den Server gesendet und der HTTP-Response abgeholt. Zuletzt müssen die erhaltenen Daten nach der Trefferzahl geparst werden.

```
// ParseHtml.java

import java.io.*;
import java.net.*;
import ch.aplu.util.*;

public class ParseHtml extends Console
{
  private String host = "www.google.com";
  private int port = 80;
  private String startTag = "of about <b>";
  private String endTag = "</b>";

  public ParseHtml()
  {
    ByteArrayOutputStream reply = new ByteArrayOutputStream();
    PrintWriter writer = new PrintWriter(reply, true);
    try
    {
      while (true)
      {
        print("Suchwort: ");
        String searchString = Console.readLine();

        // Send client request
        Socket socket = new Socket(host, port);
        PrintWriter out =
            new PrintWriter(socket.getOutputStream());
        String httpMsg = "GET /search?q=" +
            searchString +
            " HTTP/1.0\r\n\r\n";
        out.print(httpMsg);
        out.flush();

        // Get server response
        reply.reset();
        BufferedReader in = new BufferedReader(
```

```
            new InputStreamReader(socket.getInputStream()));
        String line;
        while ((line = in.readLine()) != null)
          writer.println(line);

        // Parse response
        String dataStr = reply.toString();
        String nbHits = "0";
        int startIndex = dataStr.indexOf(startTag, 0) +
            startTag.length();
        if (startIndex > -1)
        {
          int endIndex = dataStr.indexOf(endTag, startIndex);

          if (endIndex - startIndex < 15 )  // For security
            nbHits = dataStr.substring(startIndex, endIndex);
        }
        println("Trefferzahl: " + nbHits);

        // Release resources
        socket.close();
        in.close();
        out.close();
      }
    }
    catch (IOException ex)
    {}
  }

  public static void main(String[] args)
  {
    new ParseHtml();
  }
}
```

Wir verwenden unser Programm, um einen Anhaltspunkt über die Wichtigkeit einiger gesellschaftsrelevanter Wörter zu erhalten (Abb. 33.4). Das Resultat löst allerdings Unbehagen aus.

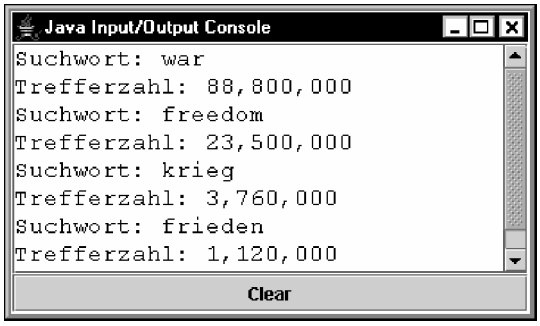

Abb. 33.4 *Trefferquote einiger Suchwörter*

33.6 Echo-Server

Für den Test von Netzwerk-Verbindungen zwischen zwei Hosts ist es sehr zweckmäßig, auf dem einen Host ein Serverprogramm zu starten, dessen einzige Aufgabe darin besteht, empfangene Zeilen ohne Veränderung wieder an einen Client zurückzusenden. Ein Telnet-Programm auf dem anderen Host sollte dann jede gesendete Zeile wie ein Echo wieder empfangen. Es ist mit unseren Kenntnissen nicht schwierig, einen solchen Echo-Server zu schreiben, der auch gleich multithreaded ist und damit mehrere Clients nebeneinander bedienen kann. Wenn wir ihn in einem Kommando-Fenster starten, können wir auch den Port angeben, auf dem der Server „hören“ soll, beispielsweise für Port 333:

```
java EchoServer 333
```

In der Applikationsklasse erzeugen wir eine Instanz von `ServerSocket` und fangen dabei die Exception ab, falls der Port bereits durch einen anderen Dienst verwendet wird. Für jeden neuen Client wollen wir mit der Klasse `SocketHandler` einen neuen Thread erzeugen, mit dem der Client bedient wird. Dazu wäre es am einfachsten in die Applikationsklasse eine Endlosschleife

```
while (true)
{
  Socket s = serverSocket.accept();
  new SocketHandler(s).start();
}
```

zu setzen, da bekanntlich `accept()` blockiert, bis sich ein neuer Client angemeldet hat. Es ergibt sich dadurch allerdings das Problem, wie wir die Applikation beenden wollen, ohne grobe Maßnahmen wie das „Abschießen“ zu ergreifen. Es gibt dazu zwei Möglichkeiten: wir erstellen ein GUI und rufen zum Beenden der Applikation in einer Callbackmethode `System.exit()` auf. Dabei werden zwar möglicherweise nicht alle Socket-Ressourcen freige-

geben. Eine sanfte Variante, um blockierende Methoden in den Griff zu bekommen, besteht darin, sie in einem eigenen Thread aufzurufen. Dadurch läuft das Programm trotz der blockierenden Methode weiter und kann den Abbruch mit geeigneten Maßnahmen veranlassen. In unserem Fall schließen wir mit `close()` den ServerSocket, wodurch `accept()` eine Exception wirft.

Damit alle Threads mit Sicherheit abbrechen, wenn wir die Applikation beenden, lassen wir sie mit `setDeamon()` als Daemon laufen. Um die Wirksamkeit des Verfahrens zu zeigen, beenden wir den Server, falls sich mehr als 3 Clients anmelden. Dazu verwenden wir den int `nbClients`, der die angemeldeten Clients zählt.

Wirft die run-Methode des Threads `ServerHandler` eine Exception, so können wir dies nicht ohne weiteres der Applikationsklasse mitteilen, da sich Exceptions nicht von einem Thread zum anderen propagieren lassen. Wir setzen daher im catch-Block des Threads eine boolesche Instanzvariable `isRunning` auf `false`, die in der Applikationsklasse ständig geprüft wird.

```
// EchoServer.java

import java.net.*;
import java.io.*;

public class EchoServer
{
  // --------------- Inner class SocketHandler -------------
  private class SocketHandler extends Thread
  {
    private Socket socket;

    public SocketHandler(Socket socket)
    {
      this.socket = socket;
      setDaemon(true);
      nbClients++;
      System.out.println("# of clients: " + nbClients);
    }

    public void run()
    {
      BufferedReader in = null;
      PrintWriter out = null;
      try
      {
        in = new BufferedReader(
            new InputStreamReader(socket.getInputStream()));
        out = new PrintWriter(socket.getOutputStream());
        String line;
        while ((line = in.readLine()) != null)
        {
```

```
          System.out.println("Received from " +
                             socket.getInetAddress() + " : " +
                             line);
          out.println(line);
          out.flush();
        }
      }
      catch (IOException ex)
      {}
      try
      {
        if (in != null)
          in.close();
        if (out != null)
          out.close();
        socket.close();
      }
      catch (IOException ex)
      {}
      System.out.println("Client socket closed");
      nbClients--;
      System.out.println("# of clients: " + nbClients);
    }
  }

  // --------------- Inner class ServerHandler -------------
  class ServerHandler extends Thread
  {
    private ServerSocket serverSocket;

    ServerHandler(ServerSocket serverSocket)
    {
      this.serverSocket = serverSocket;
      setDaemon(true);
    }

    public void run()
    {
      try
      {
        while (true)
        {
          Socket socket = serverSocket.accept();
          System.out.println("Connection to " +
                             socket.getInetAddress() +
                             " established");
          new SocketHandler(socket).start();
        }
```

```
      }
      catch (IOException ex)
      {
        isRunning = false;
      }
    }
  }
// --------------- End of inner classes ------------------

  private static int port = 23; // Default telnet port
  private int maxNbClients = 3;
  private int nbClients = 0;
  private boolean isRunning = true;

  public EchoServer()
  {
    try
    {
      ServerSocket serverSocket = new ServerSocket(port);
      System.out.println("Server running on port " + port);
      new ServerHandler(serverSocket).start();
      while (isRunning && nbClients <= maxNbClients)
        Thread.currentThread().yield();
      serverSocket.close();
      System.out.println("Server socket closed");
    }
    catch (Exception ex)
    {
      System.out.println("Port already in use");
    }
    System.out.println("Host exiting...");
  }

  public static void main(String[] args)
  {
    if (args.length > 0)
      port = Integer.parseInt(args[0]);
    new EchoServer();
  }
}
```

Nach dem Start des Echo-Servers freuen wir uns am richtigen Funktionieren, indem wir mit dem vorhin geschriebenen Telnet-Programm eine IP-Verbindung herstellen und eine Zeile eintippen, die vom Server identisch zurückgesendet wird (Abb. 33.5).

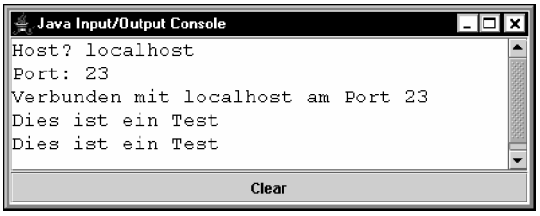

Abb. 33.5 *Verbindung mit dem Echo-Server*

33.7 Chat-Client und -Server

Im Zusammenhang mit der Kommunikation zwischen zwei Computern ergeben sich immer wieder heikle Probleme, die mit der Synchronisierung der zwei unabhängigen Prozesse und der Datenübertragung zusammenhängen, insbesondere wenn genauere Angaben über die Geschwindigkeit der Rechner und die Dauer der Datenübertragung fehlen oder die Werte stark variieren. Dies ist bei Client-Server-Systemen häufig der Fall, da sowohl das Internet wie der Server unterschiedlich ausgelastetet sind. Client- und Server-Applikationen ohne Berücksichtigung des Zeitverhaltens der beteiligten Hosts und der Übertragungsstrecke führen zu katastrophalem Fehlverhalten. Berüchtigt ist das „Hängen“ der Applikation oder sogar des ganzen Computers. Meist ist dieses auf die schlechte Implementierung von Methoden zurückzuführen, die beim Aufruf blockieren, wie die read-Methoden von Streams und `accept()`.

Wie wir beim EchoServer gelernt haben, müssen blockierende Methoden grundsätzlich in einem eigenen Thread aufgerufen werden. Da der Applikationsthread weiterläuft, hat er die Möglichkeit, die blockierende Methode zu überwachen und bei Bedarf nach einem gewissen Timeout einzugreifen.

Als klassische Anwendung der Netzwerkprogrammierung gelten Chat-Systeme, bei denen zwei Partner in Echtzeit miteinander kommunizieren. Das Benutzerinterface besteht üblicherweise aus einem Doppelfenster, wobei im einen Fenster die eigenen Tastatureingaben und im anderen Fenster der übertragene Text des Partners dargestellt werden. Wegen der besseren Effizienz werden bei Internet-Chat-Systemen nicht einzelne Zeichen, sondern jeweils nur ganze Zeilen übertragen.

Die Programmierung von Chat-Systemen ist wegen des gesellschaftlichen Aspekts motivierend, aber auch lehrreich, da die Socketprogrammierung mit Threads, aber auch die Einbettung der Applikation in ein benutzerfreundliches GUI für eine Vielzahl von Anwendungen gleichermaßen wichtig ist. Obschon die beiden Verbindungsknoten vom Benutzer aus gesehen gleiche Funktionalitäten aufweisen, müssen wir wegen des asymmetrischen Client-Server-Modells einen Chat-Server und einen Chat-Client entwickeln, wobei der Server als Erstes gestartet werden muss und der Client nachträglich die Verbindung zum Server aufbaut.

Damit die Applikationsklasse von der Behandlung der Tastatureingaben befreit ist, werden diese in der Klasse `ChatPanel` durch Tastatur-Callbacks behandelt, die bekanntlich in einem weiteren internen Thread ablaufen. Beim Drücken der Enter-Taste wird die eingegebene Zeile zum Partnerknoten gesendet. Dieses Programmkonzept zeigt, dass die Nebenläufigkeit von bestimmten Aufgaben durch die Verwendung von Callbacks und Threads zu einer Entflechtung des Codes führt. Um den Umfang des Codes in vernünftigen Grenzen zu halten, beenden wir Client und Server mit dem Close-Button der Titelleiste, der im Thread des GUI `System.exit()` aufruft und nehmen dabei in Kauf, dass je nach Situation nicht alle Netzwerk-Ressourcen (Sockets, Streams) vollständig freigegeben werden. Immerhin enden damit mit Sicherheit auch alle zur Applikation gehörende Threads.

Wir wenden uns zuerst dem Client-Programm zu. Für den Empfang der übertragenen Zeilen verwenden wir einen eigenen Thread, welcher in der run-Methode solange auf `readLine()` hängt, bis eine Zeile vom Partnerknoten empfangen wird. Da wir nicht wissen, wie lange das Warten dauert, muss der Timeout der blockierenden readLine-Methode mit `setSoTimeout(0)` auf unendlich gestellt werden. Die erhaltene Zeile wird in eine Instanzvariable `line` kopiert, die vorher auf `null` gesetzt ist. Durch ständiges Überprüfen des Stringwerts erfährt der Applikationsthread, dass eine neue Zeile angekommen ist und er diese im unteren Fenster, dem Empfangsfenster, anzeigen muss. Wird die Verbindung unterbrochen, weil beispielsweise der Server ausfällt, so wirft `readLine()` eine IOException, mit der man ein Flag `isConnected` auf `false` setzt. Dadurch wird die Wiederholschleife beendet und man kann anschließend im finally-Block die Ressourcen frei geben. Nach der Rückkehr des Konstruktors wird die Applikation in `main()` mit `System.exit()` beendet.

Die Applikation kann mit einer Kommandozeile gestartet werden, die optional die IP-Adresse des Chat-Servers und den verwendeten IP-Port enthält. Standardmäßig wird `localhost` mit `Port 3000` benützt.

```
// ChatClient.java

import java.net.*;
import java.io.*;

public class ChatClient
{
  class InputThread extends Thread
  {
    public void run()
    {
      try
      {
        while (true)
          line = netIn.readLine(); // Blocking
      }
      catch (IOException ex)
      {
        isConnected = false;
```

```
    }
  }
}

private BufferedReader netIn = null;
private PrintWriter netOut = null;
private Socket socket = null;
private String line = null;
private boolean isConnected = false;
private ChatPanel p = null;

public ChatClient(String hostname, int port)
{
  try
  {
    socket = new Socket(hostname, port);
    socket.setSoTimeout(0);

    netIn = new BufferedReader(
        new InputStreamReader(socket.getInputStream()));
    netOut = new PrintWriter(
        new OutputStreamWriter(socket.getOutputStream()));

    p = new ChatPanel("Simple Chat (Client)");
    p.setOutput(netOut);
    p.appendLine("Connection to '" + hostname +
                 "' on port " + port + " established");

    new InputThread().start();
    isConnected = true;
    p.enableEntry(true);

    while (isConnected)
    {
      try
      {
        Thread.currentThread().sleep(1);
      }
      catch (InterruptedException ex)
      {}
      if (line != null)
      {
        p.appendLine(line);
        line = null;
      }
    }
  }
  catch (IOException ex)
```

```
    {}
    finally
    {
      try
      {
        if (netIn != null)
          netIn.close();
        if (netOut != null)
          netOut.close();
        if (socket != null)
          socket.close();
      }
      catch (IOException ex)
      {}
    }
  }

  void dispose()
  {
    if (p != null)
      p.dispose();
  }

  public static void main(String args[])
  {
    String hostname = "localhost";
    int port = 3000;
    if (args.length > 0)
      hostname = args[0];
    if (args.length == 2)
      port = Integer.parseInt(args[1]);
    new ChatClient(hostname, port);
    System.out.println("Connection to '" + hostname +
                       "' on port " + port + " failed");
    System.exit(0);
  }
}
```

Die graphische Benutzeroberfläche ist in Abb. 33.6 und 33.7 dargestellt und besteht aus einem JFrame mit zwei JTextAreas. Für die Tastatureingabe müsste eigentlich ein Editor-Fenster herangezogen werden. Da die Eingabe aber zeilenorientiert ist, wird der Einfachheit halber eine JTextArea verwendet und ein KeyListener registriert, der die Zeichen einzeln erfasst, wobei mit der Backspace-Taste einzelne Zeichen auch wieder gelöscht werden können. Das Drücken der Enter-Taste wird mit `getKeyCode()` abgefangen und führt zum Abschicken der eingegebenen Zeile.

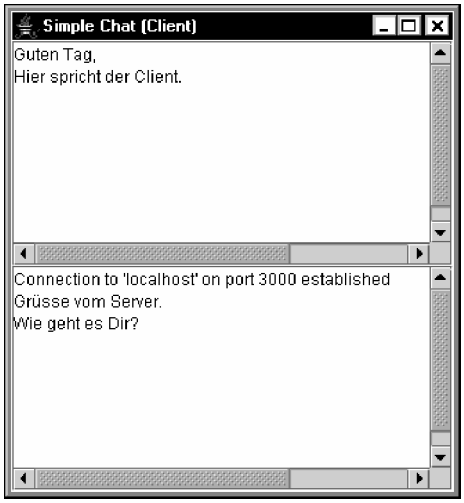

Abb. 33.6 *Chat-Client*

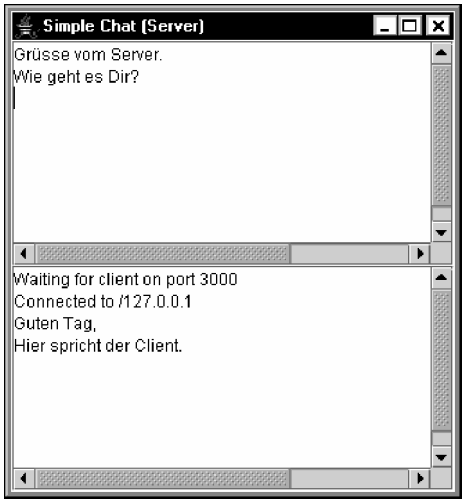

Abb. 33.7 *Chat-Server*

```
// ChatPanel.java

import javax.swing.*;
import javax.swing.text.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;

public class ChatPanel extends JFrame implements KeyListener
{
  private PrintWriter netOut;
  private String line = "";
  private JPanel contentPane;
  private JTextArea inputArea = new JTextArea(10, 40);
  private JScrollPane inputScrollPane =
      new JScrollPane(inputArea,
                      JScrollPane.
                      VERTICAL_SCROLLBAR_ALWAYS,
                      JScrollPane.
                      HORIZONTAL_SCROLLBAR_AS_NEEDED);

  private JTextArea outputArea = new JTextArea(10, 40);
  private JScrollPane outputScrollPane =
      new JScrollPane(outputArea,
                      JScrollPane.
                      VERTICAL_SCROLLBAR_ALWAYS,
                      JScrollPane.
                      HORIZONTAL_SCROLLBAR_AS_NEEDED);
```

```
public ChatPanel(String title)
{
  super(title);
  setDefaultCloseOperation(WindowConstants.
                           EXIT_ON_CLOSE);
  init();
  pack();
  setVisible(true);
}

private void init()
{
  contentPane = (JPanel)getContentPane();
  contentPane.setLayout(new BorderLayout());
  contentPane.add(outputScrollPane, BorderLayout.NORTH);
  contentPane.add(inputScrollPane, BorderLayout.SOUTH);
  inputArea.setEditable(false);
  outputArea.addKeyListener(this);
  enableEntry(false);
}

public void appendLine(String text)
{
  inputArea.append(text);
  inputArea.append("\n");
  scrollToEnd(inputArea);
}

public void setOutput(PrintWriter netOut)
{
  this.netOut = netOut;
}

public void enableEntry(boolean b)
{
  outputArea.setEnabled(b);
  if (b)
    outputArea.requestFocus();
}

private void scrollToEnd(JTextArea ta)
{
  try
  {
    ta.setCaretPosition(
        ta.getLineEndOffset(
        ta.getLineCount() - 1));
```

```
    }
    catch (BadLocationException ex)
    {
      System.out.println(ex);
    }
  }

  public void keyPressed(KeyEvent evt)
  {
    char ch;
    if (evt.getKeyCode() == 8 && line.length() > 0)
      line = line.substring(0, line.length() - 1);
    else
    {
      if (evt.getKeyCode() == 0 || evt.getKeyCode() > 31)
      {
        ch = evt.getKeyChar();
        line += ch;
      }
    }

    if (evt.getKeyCode() == '\n')
    {
      netOut.println(line);
      netOut.flush();
      line = "";
    }
  }

  public void keyReleased(KeyEvent evt)
  {}

  public void keyTyped(KeyEvent evt)
  {}
}
```

Der Chat-Server ist ähnlich wie der Chat-Client aufgebaut. Um gewisse Codeduplikationen zu vermeiden, könnte man die Server- und Client-Programme zusammenlegen, was aber hier aus Gründen der Übersichtlichkeit nicht gemacht wird. Der Server wartet auf die Verbindung mit einem Client, begrenzt aber mit dem Konstruktor `ServerSocket(port, nbConnections)` die Anzahl der Clients auf 1 (für den `localhost` funktioniert diese Begrenzung nicht). Mit dem Flag `isConnected` wird sicher gestellt, dass sich der Server erneut in den Wartezustand begibt, wenn der Client die Verbindung beendet. Gleichzeitig wird im nicht verbundenen Zustand die Benutzereingabe gesperrt. Die Applikation kann optional mit einer Kommandozeile gestartet werden, die den IP-Port enthält. Standardmäßig wird `Port 3000` benützt.

```
// ChatServer.java

import java.net.*;
import java.io.*;

public class ChatServer
{
  class InputThread extends Thread
  {
    public void run()
    {
      try
      {
        while (true)
          line = netIn.readLine(); // Blocking
      }
      catch (IOException ex)
      {
        p.appendLine("Connection lost");
        isConnected = false;
      }
    }
  }

  private static int nbConnections = 1;
  private BufferedReader netIn = null;
  private PrintWriter netOut = null;
  private String line = null;
  private boolean isConnected = false;
  private ChatPanel p = new ChatPanel("Simple Chat (Server)");

  public ChatServer(int port)
  {
    try
    {
      ServerSocket server =
          new ServerSocket(port, nbConnections);
      while (true)
      {
        p.appendLine("Waiting for client on port " + port);
        Socket socket = server.accept(); // Blocking
        isConnected = true;
        p.enableEntry(true);

        netIn = new BufferedReader(
            new InputStreamReader(socket.getInputStream()));
        netOut = new PrintWriter(
            new OutputStreamWriter(socket.getOutputStream()));
```

```
        p.setOutput(netOut);
        p.appendLine("Connected to " +
                     socket.getInetAddress());

        new InputThread().start();

        while (isConnected)
        {
          try
          {
            Thread.currentThread().sleep(1);
          }
          catch (InterruptedException ex)
          {}
          if (line != null)
          {
            p.appendLine(line);
            line = null;
          }
        }
        p.enableEntry(false);
        netIn.close();
        netOut.close();
        socket.close();
      }
    }
    catch (BindException ex)
    {
      p.appendLine("Service already running on port " + port);
    }
    catch (IOException ex)
    {
      p.appendLine("Error " + ex);
    }
  }

  public static void main(String args[])
  {
    int port = 3000;
    if (args.length > 0)
      port = Integer.parseInt(args[0]);
    new ChatServer(port);
  }
}
```

Es ist wenig benutzerfreundlich, dass der eine der beteiligten Partner zuerst den Server und der andere nachher den Client starten muss. Mit sehr wenig Aufwand lässt sich dieser Mangel aber beheben. Dazu schreibt man ein Applikationsprogramm `SimpleChat.java`, das zuerst versucht, als Client mit einem Server in Verbindung zu treten. Misslingt dies, so wird automatisch der Server gestartet. Um den Client zu beenden, rufen wir die Methode `dispose()` auf, welche das JFrame des Clients schließt, was dieselbe Wirkung wie das Klicken des Close-Buttons hat. Verabschiedet sich ein Server als erster von einem Client, so übernimmt dieser automatisch die Server-Funktion.

```
// SimpleChat.java

public class SimpleChat
{
  public static void main(String[] args)
  {
    String hostname = "localhost";
    int port = 3000;
    if (args.length > 0)
      hostname = args[0];
    if (args.length == 2)
      port = Integer.parseInt(args[1]);

    ChatClient cc = new ChatClient(hostname, port);
    cc.dispose();
    new ChatServer(port);
  }
}
```

Auf der Grundlage der erarbeiteten Kenntnisse ist es eine interessante Herausforderung, ein professionelles Chat-System zu entwickeln, bei dem sich beliebig viele Clients mit einem ständig laufenden Chat-Server verbinden können, der lediglich die Aufgabe hat, den empfangenen Text an alle oder ausgewählte Clients weiterzuleiten.