

Inhalt

16	Rechnerkommunikation – verteilte Systeme.....	16-2
16.1	<i>Netzwerktechnologie</i>	16-2
16.1.1	Adressen	16-2
16.1.2	Ports und Sockets	16-3
16.2	<i>Server/Client-Programmierung</i>	16-5
16.2.1	Aufbau einer Server/Client-Verbindung	16-5
16.2.2	Die Klassen <code>java.net.ServerSocket</code> , <code>java.net.Socket</code>	16-7
16.3	<i>Beispiel „Chatroom“</i>	16-9
16.3.1	Modell	16-9
16.3.2	Server	16-10
16.3.3	Client	16-15

16 Rechnerkommunikation – verteilte Systeme

In *Netzwerken* können verschiedene Programme auf unterschiedlichen Rechnern miteinander kommunizieren. Java unterstützt die Programmierung solcher **verteilter Systeme**.

16.1 Netzwerktechnologie

Programme in Netzwerken treten über **Protokolle** miteinander in Verbindung. Diese regeln *Verbindungsaufbau*, *Datentransfer* und *Verbindungsabbau*. Kommunizierende Programme müssen sich zur Kontaktaufnahme im Vorfeld auf ein Protokoll einigen. Es gibt mehrere Standards. Hier soll ein in der Praxis häufig verwendetes Internetprotokoll, der **TCP/IP-Standard** (**TCP – Transmission Control Protocol, IP – Internet Protocol**), besprochen werden.

Rechner können mit Hilfe dieses Protokolls verlustfrei Daten untereinander austauschen. Verlustfrei bedeutet, dass das Anwendungsprogramm, welches die Daten versendet bzw. empfängt, sich nicht um die Reihenfolge oder Vollständigkeit der verschickten Datenpakete kümmern muss. Diese Aufgabe wird vom Betriebssystem übernommen. Daten werden im Internet in Form kleiner Pakete verschickt (1500 Byte) und können dann auf verschiedenen Wegen vom Sender zum Empfänger kommen (Lastverteilung und Stauvermeidung).

16.1.1 Adressen

Für die Abwicklung eines Protokolls müssen **Adressen** aller miteinander kommunizierenden Rechner im Netzwerk bekannt sein. Eine numerische **IP-Adresse** besteht aus 4 Bytes, jedes Byte wird durch einen Punkt getrennt dargestellt. Da eine IP-Adresse eines Rechners international eindeutig sein muss, werden diese von einer zentralen Organisation vergeben und verwaltet (**ICANN - Internet Corporation for Assigned Names and Numbers**).

Beispiel:	userv1	139.18.13.221
	userv2	139.18.13.222
	userv3	139.18.8.201
	userv4	139.18.8.202
	arion	139.18.13.201
	isun	139.18.13.50
	localhost	127.0.0.1

Einprägsamer als IP-Adressen sind **Hostnamen (Domainnamen)**. Dabei kann ein Rechner mehrere solche besitzen. Diese *Aliasnamen* werden zum Gebrauch im Internet durch einen speziellen Dienst (**DNS - Domain Name Service**) ihren tatsächlichen IP-Adressen zugeordnet.

IP-Adresse \longleftrightarrow ^{DNS} **Hostname**

Ein kleines Programm gestattet eine DNS-Anfrage. Als Kommandoparameter ist entweder eine IP-Adresse oder ein Hostname zu übergeben.

DNSAnfrage.java

```
// DNSAnfrage.java
import java.net.*;

/**
 * Ermittelt IP-Adresse bzw. Hostname.
 * Aufruf: java DNSAnfrage <Hostname>
 * oder:   java DNSAnfrage <IP-Adresse>
 */
class DNSAnfrage
{
    public static void main( String[] args)
    {
        try
        {
            InetAddress ip =
                InetAddress.getByName( args[ 0] );
            System.out.println
            ( "Angefragter Name: " + args[ 0] );
            System.out.println
            ( "IP-Adresse:      " + ip.getHostAddress() ; );
            System.out.println
            ( "Host-Name:      " + ip.getHostName() ; );
        }
        catch( ArrayIndexOutOfBoundsException e)
        {
            System.out.println
            ( "Aufruf: java DNSAnfrage <Hostname>" );
            System.out.println
            ( "oder:   java DNSAnfrage <IP-Adresse>" );
        }
        catch( UnknownHostException e)
        {
            System.out.println
            ( "Kein DNS-Eintrag fuer " + args[ 0] );
        }
    }
}
```

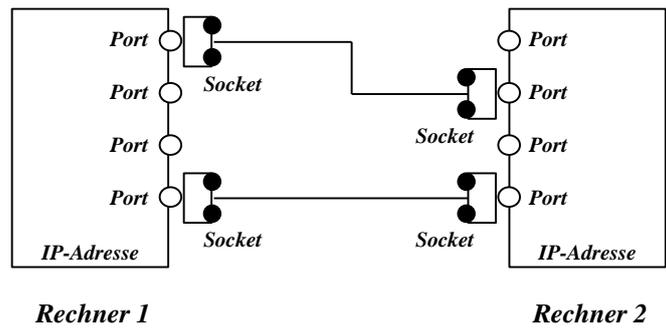
MM 2015

// InetAddress

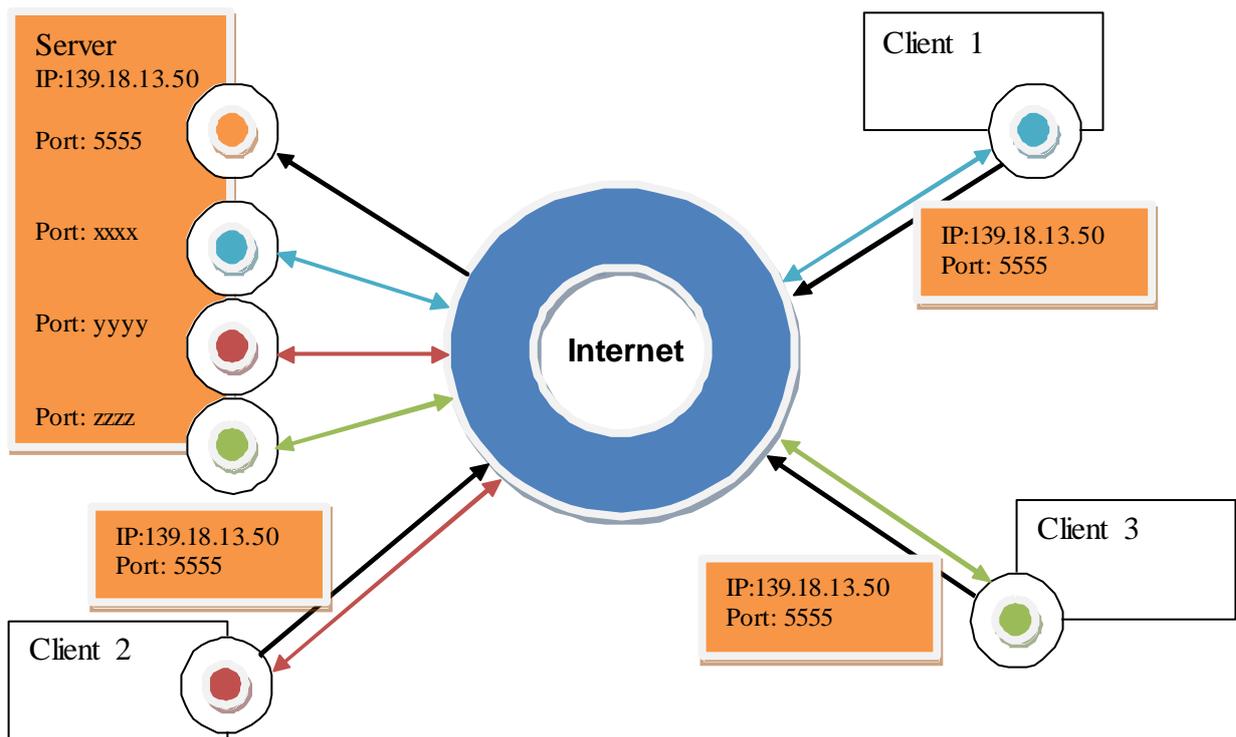
16.1.2 Ports und Sockets

Da auf einem Rechner durchaus *mehrere* Anwendungen *gleichzeitig* Internetkommunikation betreiben können, ein Rechner aber in der Regel nur über *eine* physikalische Verbindung zum Internet und nur über damit *eine IP-Adresse* verfügt, wird mittels einer sogenannten **Portnummer** festgelegt, für welche *Anwendung* die übermittelten Daten bestimmt sind. Portnummern sind ganze Zahlen zwischen 0 und 65535, wobei die im Bereich von 0 bis 1023 für Standardanwendungen reserviert sind. Alle anderen Werte sind frei verfügbar.

IP-Adresse und *Portnummer* der miteinander kommunizierenden Rechner bilden jeweils den **Socket** (Buchse) als Endpunkt einer Datenübertragung. Ein Port eines Rechners stellt somit eine Kommunikationsschnittstelle zur Außenwelt zur Verfügung.



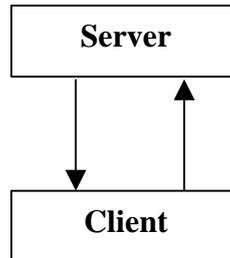
Bei der Kommunikation über Sockets unterscheidet man zwischen **Server** und **Client**. Ein Server *eröffnet die Verbindung* über einen speziellen Socket und bietet Dienste an. Ein Client *dockt über diesen Socket an*. Mit dieser Struktur können Dienste eines Servers von vielen Clients genutzt werden. Dabei muss abgesichert werden, dass sich die Clients *nicht* gegenseitig stören. Deshalb bekommt jeder Client zum Datenaustausch auf dem Server einen eigenen Socket zugeordnet.



16.2 Server/Client-Programmierung

Rechner im Netz bieten anderen Rechnern Dienste an bzw. nutzen angebotene Dienste. So ist es möglich, Aufgaben als Dienste auf wenigen Rechnern zu installieren und mehreren Rechnern gleichzeitig zur Verfügung zu stellen. Die Wartung und Pflege solcher Dienste wird durch die Konzentration auf wenige Rechner vereinfacht.

Client - Server - Architektur



Server (Diener): Ein Programm auf einem Rechner (**Server-Rechner, Server-Host**), welches einen *Service* (Dienst) über ein Netzwerk *anbietet*.

Client (Kunde): Ein Programm auf einem Rechner (**Client-Rechner, Client-Host**), welches einen *Service* (Dienst) eines anderer Rechner über ein Netzwerk *in Anspruch nimmt*.

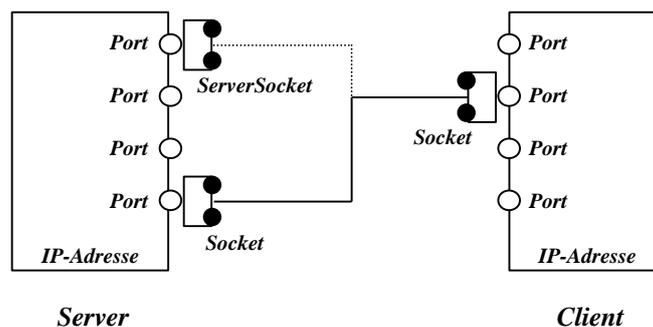
Ein Rechner kann sowohl Server- als auch Client-Rechner sein.

16.2.1 Aufbau einer Server/Client-Verbindung

Server

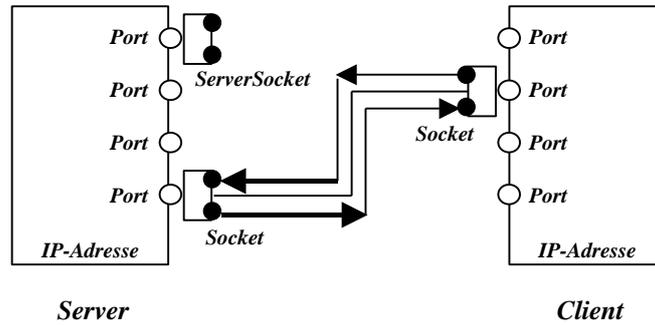
Verbindungsaufbau

1. Ein portgebundener Serversocket wird erzeugt. Dieser wird für Clientanmeldungen verwendet: IP-Adresse des Servers und Portnummer muss als Zugang für die Dienstleistung dem Client bekannt sein.
2. Der Server wartet auf Anmeldungen von Clients.
3. Hat sich ein Client angemeldet, so wird *serverseitig* ein weiterer Socket zur Abwicklung der Kommunikation mit dem Client eingerichtet.



Datentransfer

4. Über diesen Socket werden Ein- und Ausgabestrom zum Client geöffnet.
5. Der Datenaustausch erfolgt über diese Datenströme durch ein festgelegtes Protokoll.



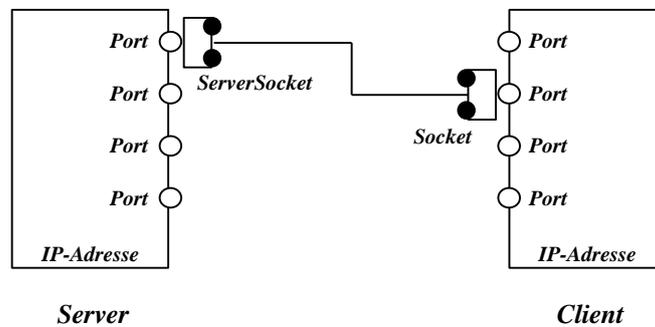
Verbindungsabbau

6. Datenströme und Socket werden geschlossen.
7. Entweder wartet der Server auf weitere Clientanmeldungen oder er wird beendet.

Client

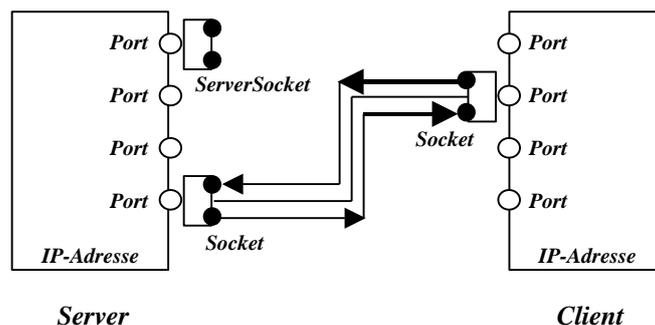
Verbindungsaufbau

1. Zum Abwickeln der Kommunikation mit dem Server wird *clientseitig* ein Socket erzeugt.
2. Ein Client nimmt über IP-Adresse und Portnummer mit einem Server Kontakt auf.



Datentransfer

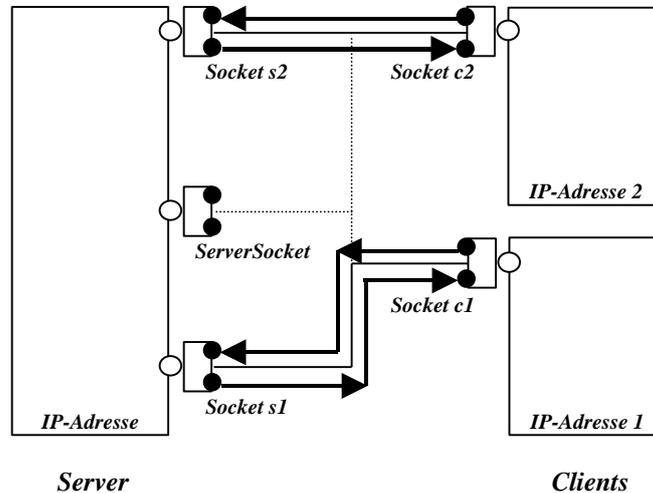
3. Über diesen Socket werden Ein- und Ausgabestrom zum Server geöffnet.
4. Der Datenaustausch erfolgt über die Datenströme durch ein festgelegtes Protokoll.



Verbindungsabbau

- 5. Datenströme und Socket werden geschlossen.

Server mit mehreren Client-Verbindungen



16.2.2 Die Klassen `java.net.ServerSocket`, `java.net.Socket`

Java stellt zwei Klassen für die Erzeugung von Sockets zur Verfügung. Die Klasse **ServerSocket** dient der Konstruktion spezieller Serversockets, die Klasse **Socket** erzeugt einfache Sockets, sowohl für Server als auch für Clients.

Methoden der Klasse `ServerSocket`

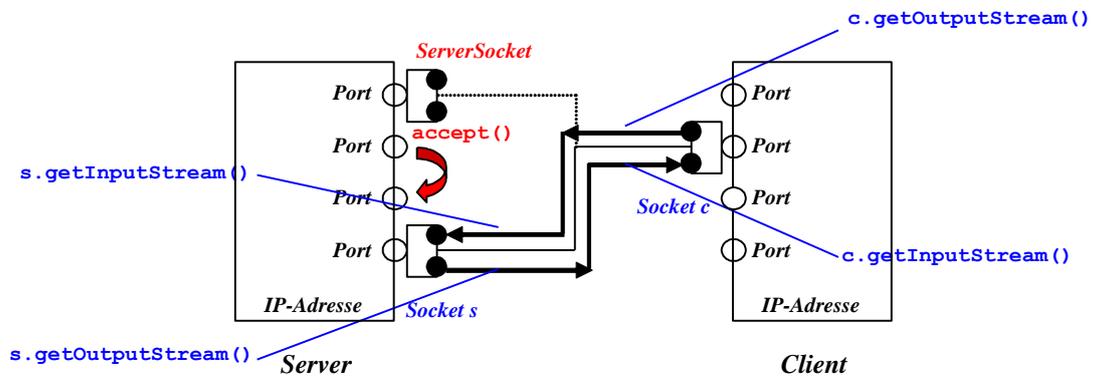
Name	Parameteranzahl	Parameter-typ	Ergebnis-typ	Beschreibung
ServerSocket	1	int		Konstruktor, erzeugt Serversocket am angegebenen Port
accept	0		Socket	wartet auf Anfrage eines Clients und erzeugt dann einen Socket für den Datentransfer, serverseitig

Bemerkung: Bei Vorhandensein einer *Firewall* für den angegebenen Port akzeptiert der Server keine Clients, d.h. die Methode **accept** liefert kein Ergebnis!

Methoden der Klasse `Socket`

Name	Parameteranzahl	Parameter-typ	Ergebnis-typ	Beschreibung
Socket	2	Inet Adress, int		Konstruktor, erzeugt Socket, verbindet Anwendung unter Adresse und Port

Socket	2	String, int		Konstruktor, erzeugt Socket, verbindet Anwendung unter Hostrechners und Port
getInputStream	0		InputStream	liefert einen Byteeingabestrom über den Socket
getOutputStream	0		OutputStream	liefert einen Byteausgabestrom über den Socket
close	0		void	schließt Socket und die dazugehörigen Datenströme



16.3 Beispiel „Chatroom“

In einem **Chatroom** können mehrere Nutzer an unterschiedlichen Rechnern miteinander kommunizieren. Ein Server verwaltet die Gesprächsrunde der Benutzer, die sich über verschiedene Clients an- bzw. abmelden. Er nimmt Beiträge von Gesprächsteilnehmern in Empfang und leitet sie an alle anderen weiter.

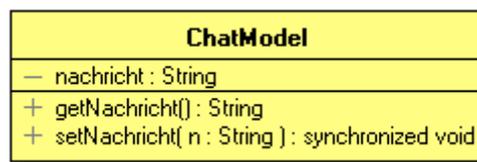
Hier soll eine sehr einfache Konsolenanwendung programmiert werden.

16.3.1 Modell

Eine Klasse **ChatModel** ist der Knotenpunkt der Gesprächsrunde. Sie verwaltet die Nachrichten, die zwischen Clients und Server übermittelt werden sollen. Hier kommt die Nachricht eines Nutzers an und wird anschließend an alle Nutzer weitergeleitet.

Eine ankommende Nachricht eines Gesprächsteilnehmers wird dem Modell durch eine Methode **setNachricht** registriert. Um einen gleichzeitigen Zugriff mehrerer Nachrichten und damit **Deadlocks** (Blockierungen) zu vermeiden, ist eine **Synchronisation** dieser notwendig.

Erhaltene Nachrichten sind vom Modell an alle Nutzer weiterzuleiten. Deshalb wird durch Überwachen des Modells der Empfang einer Nachricht durch eine Änderungsmeldung registriert. Der Überwachungsmechanismus übernimmt die Weiterleitung der Nachricht.



ChatModel.java

```
//ChatModel.java
import java.util.*;
import java.util.concurrent.*;

/**
 * Model fuer Chatroom, speichert letzte Nachricht.
 */
public class ChatModel
    extends Observable
{
    // coreData
    /**
     * Zuletzt gesendete Nachricht bzw. Startnachricht.
     */
    private String nachricht = "Chatroom leer";

    /* ----- */
    // getData-Methoden
    MM 2015
    // Observable
```

```

/**
 * Lesen der Nachricht.
 * @return Nachricht
 */
public String getMessage()
{
    return nachricht;
}

/* ----- */
// service-Methoden
/**
 * Schreiben der Nachricht, synchronisiert.
 * @param n empfangene Nachricht
 */
public synchronized void setMessage( String n)
{
    nachricht = n;

    setChanged(); // Aenderungsmeldung
    notifyObservers( nachricht);
}
}

```

16.3.2 Server

Verbindungsaufbau

Zunächst wird ein Objekt der Klasse **ChatModel** für die *Nachrichtenaktualisierung* instanziiert. Eine Portnummer wird als Kommandozeilenparameter übergeben und mit dieser ein *ServerSocket* erzeugt. Anschließend *wartet* der Server *auf Clientanmeldungen*. Für *jeden* Client wird ein *neuer Socket* aufgebaut und über diesen ein neues Protokoll zur Nachrichtenübermittlung gestartet.

ChatServer.java

```

//ChatServer.java
import java.net.*; // ServerSocket, Socket
// MM 2015

/**
 * Server fuer Chatroom,
 * erzeugt einen ServerSocket mit angegebenen Port,
 * wartet auf Client, erzeugt Socket fuer Kommunikation
 * und startet Protokoll fuer Client.
 * Aufruf: java ChatServer <Port>
 */
class ChatServer
{
    public static void main( String[] args)
    {
        // Model
        ChatModel model = new ChatModel();
    }
}

```

```
    try
    {
// ServerSocket
        int port = Integer.parseInt( args[ 0] );
        ServerSocket server = new ServerSocket( port );

        System.out.println( "ChatServer laeuft" );

// Client
        while( true )
        {
            Socket s = server.accept();
            new ChatServerProtokoll( s, model );
        }
    } catch( ArrayIndexOutOfBoundsException e )
    {
        System.out.println
        ( "\nAufruf: java ChatServer <Port>\n" );
    } catch( Exception e )
    {
        System.out.println
        ( "\nServer konnte nicht gestartet werden\n " + e );
    }
}
```

Datentransfer

Für *jeden* angemeldeten Client wird ein neues Serverprotokoll gestartet. Über dieses läuft die Interaktion mit dem Client. Da auf einen Server *mehrere* Clients parallel zugreifen dürfen, müssen deren Serverprotokolle *nebenläufig* zur Verfügung stehen. Deshalb wird jedes Serverprotokoll als *Thread* angelegt.

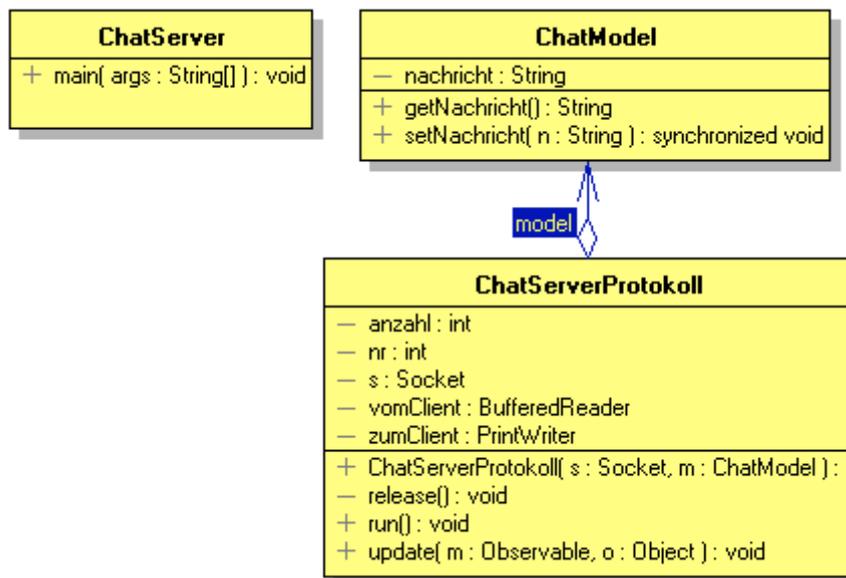
Ein Serverprotokoll eines Client soll auf alle Nachrichten reagieren. Deshalb *überwacht* jedes Serverprotokoll Nachrichtenänderungen im **ChatModel**.

Zur *Interaktion* zwischen Server und Client werden über den erzeugten Socket *Datenströme* aufgebaut. Die **Thread**-Methode **run** liest über einen *Eingabestrom* Nachrichten *vom Client* und schickt sie zum Modell. Die **Observer**-Methode **update** reagiert auf Änderungen im **ChatModel**, indem sie eingegangene Nachrichten über einen *Ausgabestrom zum Client* sendet.

Ein Clientzähler zählt die angemeldeten Clients.

Verbindungsabbau

Eine Methode **release** meldet ein Serverprotokoll beim observierten Modell ab, schließt den Socket einschließlich der Clientdatenströme.

**ChatServerProtokoll.java**

```

//ChatServerProtokoll1.java
import java.util.*;
import java.io.*;
import java.net.*;

/**
 * Steuert Interaktion zwischen Server und Client
 * ueberwacht ChatModel (Original).
 */
class ChatServerProtokoll
    extends Thread
    implements Observer
{
    /**
     * Model, Original serverseitig.
     */
    private ChatModel model;

    /**
     * Clientzaehler
     */
    private static int anzahl = 0;

    /**
     * Nummer des Client
     */
    private int nr;

    /**
     * Socket fuer Clientverbindung
     */

```

MM 2015

// Observer

// Reader, Writer

// Socket

```
    private Socket s;

/**
 * Eingabestrom vom Client
 */
    private BufferedReader vomClient;

/**
 * Ausgabestrom zum Client
 */
    private PrintWriter zumClient;

/**
 * Konstruktor,
 * baut Ueberwachungsmechanismus
 * und Datenstroeme zum/vom Client auf.
 * @param s Socket fuer Clientverbindung
 * @param m Modell auf dem Server
 */
    public ChatServerProtokoll( Socket s, ChatModel m)
    {
// Socket
        this.s = s;

// Modell, Ueberwachungsmechanismus
        model = m;
        model.addObserver( this);

        nr = ++anzahl;

        try
        {
// Datenstroeme
            vomClient = new BufferedReader
                ( new InputStreamReader( s.getInputStream()));

            zumClient = new PrintWriter
                ( s.getOutputStream(), true);

// Anfangszustand
            model.setNachricht
                ( "\tClient " + nr + " betritt den Chatroom," +
                  " Personen im Chatroom: " +
                  model.countObservers());

// Thread
            start();
        }
        catch( Exception e)
        {
            System.out.println
                ( "\nProtokoll fuer Client " + nr
```

```
        + " konnte nicht gestartet\n " + e);
    }
}

/**
 * Interaktion,
 * liest und verarbeitet Nachrichten vom Client.
 */
public void run()
{
    System.out.println
        ( "Protokoll fuer Client " + nr + " gestartet");

    try
    {
        String nachricht;

        do
        {
            nachricht = vomClient.readLine();

            if( nachricht.equalsIgnoreCase( "quit"))
            {
                model.setNachricht
                    ( "\tClient " + nr +
                      " verlaesst den Chatroom," +
                      " Personen im Chatroom: " +
                      ( model.countObservers() - 1));

                release();
                return;
            }
            else
            {
                System.out.println
                    ( "Client " + nr + " schreibt: " + nachricht);
                model.setNachricht
                    ( "\tClient " + nr + " schreibt: " + nachricht);
            }
        } while( true);
    }
    catch( Exception e)
    {
        System.out.println
            ( "\nVerbindung zum Client unterbrochen\n " + e);
        release();
    }
}

/**
 * Schliesst Verbindung zum Client,
 * setzt Model zurueck.
 */
```

```

*/
private void release()
{
    System.out.println
    ( "Protokoll fuer Client " + nr + " beendet");
    try
    {
        s.close();
        model.deleteObserver( this);
        model = null;
    }
    catch( Exception e)
    {
        System.out.println
        ( "\nFehler beim Schliessen des Clientprotokoll\n"
        + e);
    }
}

/**
 * Ueberschreibt Interfacemethode update des Observer,
 * teilt Client Aenderung im Modell mit.
 * @param m Modell, welches Aenderungen meldet
 * @param o geaendertes Objekt
 */
public void update( Observable m, Object o)
{
    if( model != m) return;

    System.out.println( "Datentransfer Client " + nr);
    zumClient.println( o);
}
}

```

Ein Server wird z.B. mittels **java ChatServer 5555** gestartet und wartet dann auf Clientanmeldungen. Ein Abbruch ist durch **^C** möglich.

- **Starten des Servers, wartet auf Anmeldungen:** **java ChatServer <Port>**
- **Abbruch:** **^C**

16.3.3 Client

Verbindungsaufbau

Möchte ein Client die Dienste eines Servers in Anspruch nehmen, so muss er sich mit diesem über *IP-Adresse* des Rechners und *Portnummer* des Dienstes in Verbindung setzen. Diese werden als Kommandozeilenparameter übergeben. Clientseitig wird ein *Socket* unter Verwendung der Anschlusswerte erzeugt. Ein *Clientprotokoll* übernimmt den *Datenaustausch*.

ChatClient.java

```
//ChatClient.java
import java.net.*;

/**
 * Client fuer Chatroom,
 * erzeugt Client-Socket fuer Kommunikation
 * mit angegebenen Server über angegebenen Port
 * und wickelt Protokoll ab.
 * Aufruf: java ChatClient <Host> <Port>
 */
class ChatClient
{
    public static void main( String[] args)
    {
        try
        {
            // Serververbindung
            String hostName = args[ 0];
            int port = Integer.parseInt( args[ 1]);
            Socket c = new Socket( hostName, port);

            System.out.println( "ChatClient laeft");

            // Protokoll
            new ChatClientProtokoll( c).action();
        }
        catch( ArrayIndexOutOfBoundsException e)
        {
            System.out.println
            ( "\nAufruf: java ChatClient <Host> <Port>\n");
        }
        catch( UnknownHostException e)
        {
            System.out.println
            ( "\nKein DNS-Eintrag fuer " + args[ 0] + "\n");
        }
        catch( Exception e)
        {
            System.out.println
            ( "\nVerbindung zum Server fehlgeschlagen\n " + e);
        }
    }
}
```

MM 2015

// Socket

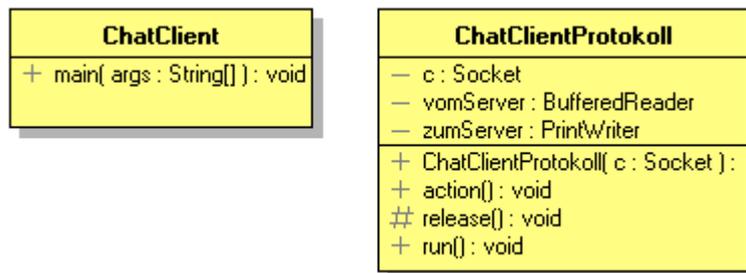
Datentransfer

Ein Clientprotokoll schickt Nachrichten zum Server und liest Nachrichten vom Server. Beide Aktionen müssen parallel zur Verfügung stehen. Deshalb wird der Datentransfer als Thread angelegt.

Zur Interaktion zwischen Client und Server werden über den erzeugten Socket Datenströme aufgebaut. Die **Thread**-Methode **run** liest über einen Eingabestrom Nachrichten vom Server und zeigt diese auf der Konsole an. Eine Methode **action** erlaubt, Nachrichten über Tastatur einzugeben und diese über einen Ausgabestrom zum Server zu senden.

Verbindungsabbau

Eine Methode **release** gestattet dem Client, sich zu jeder Zeit aus dem **Chatroom** zu verabschieden. Er meldet sich beim Server ab, schließt den Socket einschließlich den Serverdatenströmen.



ChatClientProtokoll.java

```
//ChatClientProtokoll.java
import java.io.*;
import java.net.*;

/**
 * Protokoll eines Clients,
 * zeigt Nachrichten vom Server an
 * und schickt Nachrichten zum Server
 */
class ChatClientProtokoll
  extends Thread
{
  /**
   * Socket fuer Serververbindung
   */
  private Socket c;

  /**
   * Eingabestrom vom Server
   */
  private BufferedReader vomServer;

  /**
   * Ausgabestrom zum Server
   */
  private PrintWriter zumServer;
}
```

MM 2015

// Reader, Writer
// Socket

```
/**
 * Konstruktor,
 * baut Datenstroeme zum/vom Server auf.
 * @param c Socket fuer Serververbindung
 */
public ChatClientProtokoll( Socket c)
{
    try
    {
// Socket, Datenstroeme
        this.c = c;

        zumServer = new PrintWriter
            ( c.getOutputStream(), true);

        vomServer = new BufferedReader
            ( new InputStreamReader( c.getInputStream()));

// Thread
        start();
    }
    catch( Exception e)
    {
        System.out.println
            ( "\nProtokoll konnte nicht gestartet werden\n " + e);
    }
}

/**
 * Interaktion,
 * liest und verarbeitet Nachrichten vom Server.
 */
public void run()
{
    try
    {
        String nachricht;

        while( true)
        {
            nachricht = vomServer.readLine();
            if( nachricht != null)
                System.out.println( nachricht);
        }
    }
    catch( Exception e)
    {
        if( !isInterrupted())
        {
            System.out.println
                ( "\nVerbindung zum Server unterbrochen\n " + e);
            release();
        }
    }
}
```

```
    }
  }
}

/**
 * Nutzereingaben,
 * Eingabe und Versenden von Nachrichten.
 */
public void action()
{
    BufferedReader vonTastatur = new BufferedReader
        (new InputStreamReader( System.in));

    System.out.println
        ( "Dialog gestartet (Dialog mit QUIT beenden)");
    try
    {
        String nachricht;

        do
        {
            nachricht = vonTastatur.readLine();

            if( nachricht.equalsIgnoreCase( "quit"))
            {
                release();
                return;
            }
            else zumServer.println( nachricht);

        } while( true);
    }
    catch( Exception e)
    {
        System.out.println
            ( "\nVerbindung zum Server unterbrochen\n " + e);
        release();
    }
}

/**
 * Beendet ChatClient,
 * meldet Abbruch dem Server,
 * bricht Verbindung zum Server ab,
 * bricht Programm ab.
 */
protected void release()
{
    // Thread beenden
    interrupt();

    try
```

```
{
// Server informieren
    System.out.println( "Dialog beendet");
    zumServer.println( "quit");

// Clientsocket schliessen
    c.close();
    System.out.println( "Client abgemeldet");
}
catch( Exception e)
{
    System.out.println
    ( "\nFehler beim Schliessen des Clients\n " + e);
}

// Programm beenden
    System.exit( 0);
}
}
```

Einen Client auf demselben Rechner, auf dem der Server läuft, kann man z.B. mittels **java ChatClient localhost 5555** anmelden. Das Abmelden erfolgt hier programmspezifisch mittels der Eingabe „**quit**“.

- **Anmelden:** `java ChatClient <Hostname> <Port>`
- **Server auf dem lokalen Rechner:** `java ChatClient localhost <Port>`
- **Abmelden:** `quit`