

Verteilte Systeme und Kommunikationsnetze

Chat-System mit Java RMI

Praktikum 4

Fachhochschule Bielefeld
Campus Minden
Studiengang Informatik

Beteiligte Personen:

Name	Matrikelnummer
Michael Nickel	1120888
Jan Augstein	1119581

Aufgaben:

Aufgabe	Gelöst
Aufgabe 1	alle Teilpunkte

27. November 2017

Inhaltsverzeichnis

1	Aufgabe 1	3
1.1	Frage- bzw. Aufgabenstellung	3
1.2	Lösung	3
1.3	Ergebnis	7
2	Quellen	8

1 Aufgabe 1

[1, 2]

1.1 Frage- bzw. Aufgabenstellung

Sie sollen ein Chat-System unter Verwendung von Java RMI implementieren. Das System besteht aus einem Server, an den beliebig viele Clients angeschlossen werden können. Das System soll nach bekanntem Prinzip arbeiten: Nachrichten werden auf einem Client eingegeben und über den Server an alle angeschlossenen Clients weiter geleitet.

1.2 Lösung

Zunächst wurden die Interfaces für das ChatSystem realisiert. Eins wurde *ChatServerIF* genannt (siehe Zeile 1 bis 8), und das andere *ClientProxyIF* (siehe nächstes Listing, Zeile 1 bis 4).

```
1 public interface ChatServerIF extends Remote {
2     public void subscribeUser (ClientProxyIF handle) throws
      RemoteException;
3
4     public boolean unsubscribeUser (ClientProxyIF handle) throws
      RemoteException;
5
6     public void broadcast(String message) throws RemoteException;
7
8 }

1 public interface ClientProxyIF extends Remote {
2     public void receiveMessage (String message) throws RemoteException;
3
4 }
```

Danach wurde eine Klasse *ChatServerImpl* realisiert, die das Interface *ChatServerIF* implementiert. Hierzu wurde eine neue List *clients* deklariert, die Objekte von *ClientProxyIF* beinhalten kann. Diese List wird dann im Konstruktor der Klasse instanziiert (siehe Zeile 1 bis 11). Wichtig für diese, wie auch für die *ChatClient* Klasse ist, dass beide Klassen von *UnicastRemoteObject* erben, damit man Remote-Objects exportieren kann.

```
1 public class ChatServerImpl extends UnicastRemoteObject implements
   ChatServerIF {
2
3     /**
4      *
5      */
6     private static final long serialVersionUID = 1L;
7     private List<ClientProxyIF> clients;
8
9     public ChatServerImpl() throws RemoteException {
10         this.clients = new ArrayList<ClientProxyIF>();
11     }
```

Für die Methoden um einen neuen User zu erstellen oder einen vorhandenen User zu löschen, wurden nun die Methoden aus dem Interface *ChatServerIF* implementiert. Die

Methode *subscribeUser* bekommt als Parameter ein Objekt vom Typ *ClientProxyIF* und fügt der List *clients* einfach das übergebene Objekt hinzu. Bei der Methode *unsubscribeUser* funktioniert dies so ähnlich, nur das hier logischerweise das ClientProxyIF-Objekt aus der List entfernt wird (siehe Zeile 1 bis 9).

```

1 |      @Override
2 |      public void subscribeUser(ClientProxyIF handle) throws
      RemoteException {
3 |          this.clients.add(handle);
4 |      }
5 |
6 |      @Override
7 |      public boolean unsubscribeUser(ClientProxyIF handle) throws
      RemoteException {
8 |          return this.clients.remove(handle);
9 |      }

```

Schließlich wird noch die Methode *broadcast* implementiert, die dafür sorgt, dass jeder Client die eingegebene Nachricht bekommt. Hierfür wird mit einer for-Schleife die komplette clients-List iteriert, wobei jeder Client dann mithilfe der *receiveMessage-Methode* diese Nachricht empfängt (siehe Zeile 1 bis 8). Auf die *receiveMessage-Methode* wird noch später eingegangen.

```

1 |      @Override
2 |      public void broadcast(String message) throws RemoteException {
3 |          for (int i = 0; i < clients.size(); i++) {
4 |              clients.get(i).receiveMessage(message);
5 |          }
6 |      }
7 |

```

Zuletzt wird noch die *main* erzeugt, in der eine Referenz auf ein entferntes Objekt gespeichert wird. Dies wird mit der Klasse *Naming* und der Methode *rebind* erreicht (siehe Zeile 1 bis 5). Als Parameter werden ein Name und ein neues Objekt angegeben.

```

1 |      public static void main(String[] args) throws RemoteException,
      MalformedURLException {
2 |          Naming.rebind("RMICChatServer", new ChatServerImpl());
3 |      }
4 |
5 | }

```

Jetzt wird die *ChatClient* Klasse erstellt, die das Interface *ClientProxyIF* und *Runnable* implementiert, da die run-Methode überschrieben werden muss.

Die Klasse bekommt ein Objekt *chatServer* vom Typ *ChatServerIF*, welches dann am Ende, wie der Name schon sagt, den Server darstellen soll. Dann wird noch ein String *name* erstellt, der den Namen des Clients darstellt und es wird noch ein boolean gebraucht, welcher in der überschriebenen run-Methode zum Einsatz kommt (siehe Zeile 1 bis 10).

```

1 |
2 | public class ChatClient extends UnicastRemoteObject implements
      ClientProxyIF, Runnable {
3 |
4 |     /**
5 |     *

```

```

6 |         */
7 |         private static final long serialVersionUID = 1L;
8 |         private ChatServerIF chatServer;
9 |         private String name = null;
10 |        boolean chatExit = true;

```

Als nächstes wird ein Konstruktor erzeugt, in dem dann der *name* und der *chatServer* definiert werden. Zudem kommt hier dann auch direkt die Methode *subscribeUser* zum Einsatz um den Clienten für den Server zu registrieren (siehe Zeile 1 bis 5).

```

1 |         public ChatClient(String name, ChatServerIF chatServer) throws
2 |             RemoteException {
3 |             this.name = name;
4 |             this.chatServer = chatServer;
5 |             chatServer.subscribeUser(this);
6 |         }

```

Weiterhin wird die *receiveMessage*-Methode aus dem *ClientProxyIF* implementiert, welche einfach nur den mitgegebenen Parameter *message* ausgibt (siehe Zeile 1 bis 5).

```

1 |         @Override
2 |         public void receiveMessage(String message) throws RemoteException {
3 |             System.out.println(message);
4 |         }
5 |

```

Jetzt muss man noch die *run*-Methode überschreiben, damit man in der *main*-Methode dann ein neuen *Thread* erstellen kann. Die *run*-Methode ist recht simpel, da man am Anfang nur eine Informationszeile ausgibt und danach einen Scanner für den Input in der Kommandozeile erstellt. Wenn man *.LOGOUT* eingibt wird der Client entfernt, ansonsten wird die *broadcast*-Methode, die zuvor in der *ChatServerImpl*-Klasse erstellt wurde, aufgerufen. Hier wird dann der Clientname und die jeweilige Nachricht die er geschrieben hat für jeden anderen registrierten Clienten angezeigt (siehe Zeile 1 bis 25).

```

1 |         @Override
2 |         public void run() {
3 |             System.out.println("Geben Sie .LOGOUT ein um sich
4 |                 auszuloggen");
5 |             @SuppressWarnings("resource")
6 |             Scanner scanner = new Scanner(System.in);
7 |             String message;
8 |
9 |             while (chatExit) {
10 |                 message = scanner.nextLine();
11 |                 try {
12 |                     if (message.equals(".LOGOUT")) {
13 |                         chatServer.unsubscribeUser(this);
14 |                         System.out.println("Erfolgreich
15 |                             ausgemeldet");
16 |                         chatExit = false;
17 |                     } else {
18 |                         chatServer.broadcast(this.name + "
19 |                             : " + message);

```

```

20 |         } catch (RemoteException e) {
21 |             e.printStackTrace();
22 |         }
23 |     }
24 |
25 | }

```

In der main-Methode wird dann zunächst eine ChatServerURL als String gespeichert, die in der Form `//host:port/name` erscheinen muss. Diese URL wird dann dem ChatServer über die `lookup`-Methode der Klasse `Naming` übergeben. Als letztes wird dann noch ein neuer Thread erzeugt. Dieser enthält dann den ChatClient mit den Parametern `args[0]` und `chatServer`, wobei `args[0]` dann der Name des Clients ist (z.B John) und `chatServer` der Server (siehe Zeile 1 bis 6).

```

1 | public static void main(String[] args) throws RemoteException,
  |     MalformedURLException, NotBoundException {
2 |     String chatServerURL = "rmi://localhost/RMIChatServer";
3 |     ChatServerIF chatServer = (ChatServerIF)
  |         Naming.lookup(chatServerURL);
4 |     new Thread(new ChatClient(args[0], chatServer)).start();
5 |
6 | }

```

Nachdem man mit dem Programmieren fertig ist muss man noch in den Programmargumenten in der Konfiguration die folgende Zeile eingeben, da man ansonsten eine `UnknownHostException` bekommt.[3]

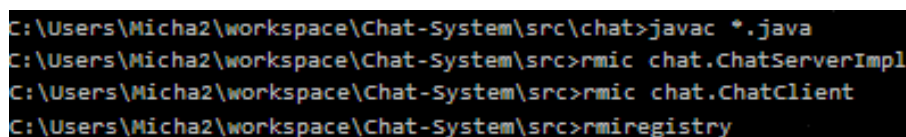
```

1 | -Djava.rmi.server.hostname=<1099> -Dremoting.bind_by_host=false

```

Nun kann man beispielsweise die Eingabeaufforderung von Windows öffnen und in den Ordner mit den ganzen .java-Dateien gehen. Hier kompiliert man zunächst einmal die ganzen Dateien mit `javac *.java`. Danach erstellt man die Remote-Klassen für `ChatServerImpl` und `ChatClient` und lässt zum Schluss die Registry laufen mit `rmiregistry`.

Dies sollte dann so aussehen:



```

C:\Users\Micha2\workspace\Chat-System\src\chat>javac *.java
C:\Users\Micha2\workspace\Chat-System\src>rmic chat.ChatServerImpl
C:\Users\Micha2\workspace\Chat-System\src>rmic chat.ChatClient
C:\Users\Micha2\workspace\Chat-System\src>rmiregistry

```

Abbildung 1: Registry laufen lassen

Jetzt öffnet man eine weitere Eingabeaufforderung und lässt den Server mit dem Befehl `java ChatServerImpl` laufen. Anschließend öffnet man beliebig viele weitere Eingabeaufforderungen, wo man dann die einzelnen Clients erstellen kann. Nun kann man mit mehreren Clients kommunizieren und sich am Ende mit `.LOGOUT` ausloggen"(siehe Abbildung 2).

```
C:\Users\Micha2\workspace\Chat-System\src>java chat.ChatServerImpl

Eingabeaufforderung - java chat.ChatClient Ben
(c) 2016 Microsoft Corporation. Alle Rechte vorbehalten.
C:\Users\Micha2>cd C:\Users\Micha2\workspace\Chat-System\src
C:\Users\Micha2\workspace\Chat-System\src>java chat.ChatClient Ben
Geben Sie .LOGOUT ein um sich auszuloggen
hallo
Ben : hallo
John : Hi
Wie gehts?
Ben : Wie gehts?
John : gut und dir?
bist du noch da?
Ben : bist du noch da?

Eingabeaufforderung - java chat.ChatClient John
(c) 2016 Microsoft Corporation. Alle Rechte vorbehalten.
C:\Users\Micha2>cd C:\Users\Micha2\workspace\Chat-System\src
C:\Users\Micha2\workspace\Chat-System\src>java chat.ChatClient John
Geben Sie .LOGOUT ein um sich auszuloggen
Hi
John : Hi
Ben : Wie gehts?
gut und dir?
John : gut und dir?
.LOGOUT
Erfolgreich ausgeloggt
```

Abbildung 2: Kommunikation mit mehreren Clients

1.3 Ergebnis

Es ist uns möglich ein einfaches Chat-System mit der Java RMI zu erstellen und dieses auch anzuwenden.

2 Quellen

Literatur

- [1] Nifal Nizar, SMTP - RMI Multiple Clients Chat Program
<https://easycodestuff.blogspot.de/2014/11/rmi-chat-program-using-java.html>, 27.11.2017
- [2] Oracle, Trail: RMI
<https://docs.oracle.com/javase/tutorial/rmi/index.html>, 27.11.2017
- [3] Kobi Ianko, java.rmi.UnknownHostException
https://developer.jboss.org/thread/152290?_sscc=t, 27.11.2017