technische universität
dortmund

Fakultät für Informatik
Lehrstuhl 4

Dr. Jan Kriege
Dipl.-Inf. Christoph Borchert

Wintersemester 2014/15

Rechnernetze und verteilte Systeme
Übungsblatt P

Ausgabe: 17. November 2014, **Abgabe:** 11. Januar 2015 23.59 Uhr

Allgemeine Informationen:

- Im Rahmen der Übungen müssen folgende Leistungen für den Erhalt der Studienleistung erbracht werden:

  Bearbeitung von 40% der Übungsaufgaben (insgesamt wird es 30 Aufgaben geben)

  8 Programmieraufgabenpunkte müssen erzielt werden (in der Programmieraufgabe sind insgesamt max. 20 Punkte zu erreichen)
- Die Programmieraufgabe muss in Gruppen von zwei bis drei Studierenden bearbeitet werden. Die Gruppen dürfen sich aus verschiedenen Übungsgruppen zusammensetzen.
- Der Code ist sinnvoll zu kommentieren - Fehlende Kommentierung führt zu Punktabzug
- Der Abgabetermin ist eine Ausschlussfrist: Nach diesem Datum eingereichte Abgaben werden nicht anerkannt.
- Die Studienleistung RvS wird erworben, wenn alle o. g. Kriterien erfüllt werden
- Unter http://ls4-www.cs.tu-dortmund.de/phpbb3/viewforum.php?f=15 wurde ein Diskussionsforum eingerichtet

1 Aufgabenstellung

1.1 Projektinhalt

Im Rahmen dieser Programmieraufgabe soll ein Chat-Client in Java entwickelt werden.

Die gesamte Infrastruktur besteht aus einem Server, der von uns zur Verfügung gestellt wird und als *Adressbuch* dient, sowie mehreren Clients, die sich am Server anmelden können (Client-Server-Architektur), um Verbindungsinformationen über andere Chatteilnehmer abzurufen. Die eigentliche Kommunikation zwischen den Teilnehmern erfolgt dann Peer-To-Peer.

Zu Testzwecken wird neben dem Server auch ein Chat-Client zur Verfügung gestellt, mit dem die eigene Implementierung getestet werden kann und der eine Orientierungshilfe über die gewünschten Funktionalität bietet.

1.2 Funktionsumfang

Der zu entwickelnde Chat-Client besteht aus drei Teilen.

Eingabeschnittstelle

Der erste Teil ist die Verarbeitung von Benutzereingaben. Hierzu muss keine graphische Oberfläche erstellt werden, die Verarbeitung von Benutzereingaben über die Kommandozeile ist ausreichend. Der Nutzer soll unter anderem die Möglichkeit haben, sich bei dem Adressbuchserver anzumelden, sich die Liste der zur Verfügung stehenden Chat-Teilnehmer anzeigen zu lassen und Nachrichten an andere Teilnehmer zu senden.

Verbindung zum Adressbuchserver

Der zweite Teil ist die Kommunikation mit dem Adressbuchserver. Es soll eine Verbindung zum Adressbuchserver aufgebaut werden, das Protokoll zur Anmeldung abgearbeitet und schließlich eine Liste aller verfügbaren Teilnehmer abgefragt und gespeichert werden. Die Liste aller verfügbarer Teilnehmer soll dem Benutzer auf Anfrage dargestellt werden.

Nachrichtenaustausch mit anderen Clients

Der dritte Teil ist die eigentliche Kommunikation mit anderen Clients. Zunächst muss dazu eine Verbindung zu einem anderen Teilnehmer aufgebaut werden. Dazu gehört neben dem Verbindungsaufbau auch das Anmeldeprotokoll. Nach erfolgreicher Anmeldung können Textnachrichten ausgetauscht werden. Der Client soll sowohl in der Lage sein selbst Verbindungen zu anderen Teilnehmern zu initiieren, als auch Verbindungsanfragen anderer Clients entgegenzunehmen.

Die Protokolle, die das Nachrichtenformat für die Kommunikation mit dem Adressbuchserver und für die Kommunikation mit den anderen Clients festlegen, sind in Abschnitt 3 aufgeführt.

1.3 Vorgehen

Bevor Sie mit der Implementierung beginnen, machen Sie sich mit den nötigen Java-Klassen vertraut und überlegen Sie sich, wie Sie diese einsetzen können. Erstellen Sie dann ein Modell, das alle nötigen Elemente enthält, die für die Funktion der Applikation nötig sind. Erst nach der Modellierung beginnen Sie mit der Implementierung.

1.4 Kommentare und Dokumentation

Kommentieren Sie Ihren Quelltext ordentlich! Abgaben ohne ausreichende Kommentare werden mit Punktabzug bewertet. Zusätzlich sollten Sie die für die Bedienung der Applikation vorgesehenen Befehle dokumentieren, um das spätere Testen während der Korrektur zu ermöglichen. Dabei ist es auch zulässig, die Befehle beim Starten der Applikation zu erläutern.

1.5 Fehlerbehandlung

Im Programm sollten Exceptions, die beispielsweise beim Verbindungsaufbau oder Versenden von Nachrichten entstehen können, verarbeitet werden. Eine unzureichende Fehlerbehandlung führt zu Punktabzug.

1.6 Bibliotheken

Zur Bearbeitung der Aufgabe sind ausschließlich die Java-Bibliotheken aus `java.*`/`javax.*` zugelassen. Zusätzliche Bibliotheken und die Nutzung des Quelltexts anderer Fremdquellen ist daher nicht erlaubt.

1.7 Abgabe

Die Abgabe findet über das AsSESS statt. Packen Sie alle zu Ihrer Applikation gehörenden *.java Quelldateien in eine .zip Datei (alternativ .tar.gz) und laden Sie diese im AsSESS entsprechend hoch. Die Abgabe kann beliebig oft durchgeführt werden. Es wird dann die zuletzt abgegebene Version gewertet.

2 Materialien

Neben dieser Aufgabenstellung befinden sich auf der Veranstaltungswebseite noch zwei Java-Programme. Zum einen ist dies der Adressbuchserver, auf dem die Verbindungsinformationen gespeichert und abgerufen werden. Zum anderen befindet sich dort eine fertige Implementierung eines Clients, mit dem getestet werden kann.

2.1 Adressbuchserver

Laden Sie sich von der Internetseite der Veranstaltung das Java-Programm *ChatServer* herunter und starten Sie es mit dem Befehl `java -jar ChatServer.jar`. Der ChatServer ist unter der IP-Adresse des Rechners, auf dem er gestartet wurde, unter dem Port 2534 zu erreichen. Die Verbindung eines Clients zum Adressbuchserver sollte nach der Anmeldung bis zum Logout dauerhaft aufrecht erhalten werden, da der Teilnehmer beim Abbau der Verbindung aus der Teilnehmerliste gelöscht wird. In den Server ist bereits ein rudimentärer Client integriert. Dieser Client ist immer verfügbar und heißt *echo*. Falls Sie sich mit diesem Client verbinden und ihm eine Nachricht schicken, schickt er die Nachricht immer identisch zurück (daher echo). Das bedeutet insbesondere, dass er nicht auf gesendete Kommandos reagiert. Beim Starten gibt der Adressbuchserver eine Liste aller Kommandos, die über die Konsole des ChatServers ausgeführt werden können, aus. Zusätzlich kann der Server mit dem Kommandozeilenparameter *debug* im Debugmodus gestartet werden (`java -jar ChatServer.jar debug`).

2.2 Test Client

Jeder Client hat eine eigene Serverfunktionalität, indem er auf ankommende Verbindungen von anderen Clients wartet. Diese Server lauschen auf einem Port, den sie dem Adressbuchserver bei der Anmeldung mitteilen müssen. Auf der Internetseite der Veranstaltung können Sie sich einen fertigen Client herunterladen. Diesen können sie benutzen, um Ihre eigene Implementierung zu testen. Der Test Client wird mit dem Befehl `java -jar TestClient.jar <IP>` gestartet, wobei `<IP>` durch die IP-Adresse des Adressbuchservers ersetzt werden muss. Tippen Sie nach dem Starten des Clients `#help` um eine Übersicht über alle Kommandos zu erhalten, die der Client beherrscht.

3 Protokoll

Das Protokollformat ist textbasiert und besteht aus einem Kommando, das jeweils nur ein einzelner Buchstabe ist und von Argumenten gefolgt wird. Im folgenden steht **c** für ein beliebiges Kommando. Die Argumente variieren je nach Befehl.

c *Argument1 [Argument2 ...]*

Im folgenden wird das Protokoll im Detail erklärt. Je nachdem, ob mit dem Adressbuchserver oder anderen Clients kommuniziert wird, sind unterschiedliche Kommandos zulässig. Es gilt grundsätzlich, dass jeder Befehl mit einem Zeilenumbruch abzuschließen ist.

3.1 Protokoll Adressbuchserver

Der Adressbuchserver versteht folgende Kommandos:

n ***name port*** Meldet den Benutzer mit dem Namen *name* an und informiert den Server darüber, dass er über den Port *port* von anderen Clients erreicht werden kann. Bei erfolgreicher Anmeldung schickt der Server ein s zurück, bei nicht möglichem Login ein e mit einer entsprechenden Fehlermeldung. Der *name* darf keine Leerzeichen enthalten!

t Fragt den Server nach der Tabelle mit den Informationen über andere Nutzer. Siehe unten für die Antwort.

x *byebye* Beendet die Verbindung zum Server, der Server wird die Verbindung von seiner Seite aus schließen. *byebye* ist ein optionaler Abschiedsgruß, der auch ignoriert werden kann.

Der Adressbuchserver sendet folgende Kommandos:

s Bestätigt eine erfolgreiche Anmeldung am Adressbuchserver.

t *n Tabellenzeilen* Das Kommando **t** wird gesendet, wenn der Client zuvor ein **t**-Kommando gesendet hat. *n* sind die Anzahl der Tabellenzeilen, die folgen. *Tabellenzeilen* werden jeweils durch einen Zeilenumbruch getrennt. In einer Tabellenzeile steht zuerst der Name des anderen Users gefolgt von seiner IP-Adresse und dem Port (jeweils getrennt durch Leerzeichen).

x *byebye* Kündigt das Schließen der Verbindung an. Der Server wird die Verbindung von seiner Seite aus schließen. *byebye* ist ein optionaler Abschiedsgruß, der auch ignoriert werden kann.

e *Fehlermeldung* Wird gesendet, wenn der Server einen Befehl erhalten hat, den er nicht kennt oder die Anfrage nicht bearbeitet werden kann. Die Fehlermeldung gibt einen Hinweis darauf, was vom Server empfangen wurde.

3.2 Protokoll Clients

Clients senden und verstehen untereinander folgende Befehle:

n *name* Meldet den Benutzer mit dem Namen *name* beim anderen Client an. Auf Grund des Protokolls sollten Nutzernamen mit Leerzeichen vermieden werden.

m *message* Sendet *message* an den anderen Client.

x *byebye* Kündigt das Schließen der Verbindung an. Der Absender wird die Verbindung von seiner Seite aus schließen. *byebye* ist ein optionaler Abschiedsgruß, der auch ignoriert werden kann.

e *Fehlermeldung* Wird gesendet, wenn der Client einen Befehl erhalten hat, den er nicht kennt. Die Fehlermeldung gibt einen Hinweis darauf, was vom Kommunikationspartner empfangen wurde.

4 Implementierungshilfen

In diesem Kapitel werden einige Basisprogrammiertechniken vorgestellt, die für diese Programmieraufgabe nützlich sind.

4.1 Sockets

Um zu vermeiden, dass jeder Programmierer, der über eine Netzwerkschnittstelle kommunizieren will, alle Protokolle, die dafür nötig sind (z.B. Ethernet, IP, TCP, UDP, etc.) selbst implementieren muss, gibt es ein Software-Abstraktion mit dem Namen *Socket*.

In Java werden zwei unterschiedliche Arten von Sockets unterschieden: *Sockets* und *ServerSockets*. Beide verhalten sich unterschiedlich.

Sockets sind diejenigen Sockets, über die tatsächlich Nachrichten verschickt werden können. Um eine Verbindung aufzubauen dient folgender Code:

```
Socket mySocket = new Socket();
mySocket.connect(
        new InetSocketAddress( "123.234.111.100", 12345 )
        );
```

Dadurch wird eine Verbindung mit dem Rechner aufgebaut, der die Adresse 123.234.111.100 hat. Zur Adresse gehört noch ein Port. Da ein Rechner mehrere Verbindungen verwalten kann, kann mit Hilfe des Ports auf der Gegenstelle das zugehörige Programm ausgewählt werden.

Um textbasiert zu kommunizieren, werden in Java Helferobjekte benötigt. Das ist zum Senden der *PrintWriter* und für den Empfang der *BufferedReader*. Die *read-Funktionen* des *BufferedReader*s blockieren. Das heißt, dass sie warten, bis zu lesende Inhalte verfügbar sind. Dazu wieder ein Beispiel:

```
Socket mySocket = new Socket();
mySocket.connect(
        new InetSocketAddress( "123.234.111.100", 12345 )
        );
PrintWriter out = new PrintWriter(
                        mySocket.getOutputStream(), true
                        );
BufferedReader in = new BufferedReader(
                        new InputStreamReader(
                                mySocket.getInputStream()
                        ) );
```

ServerSockets dienen dazu auf ankommende Verbindungen zu warten. Ein ServerSocket muss an einen lokalen Port gebunden werden:

```
ServerSocket ssocket = new ServerSocket();
ssocket.bind( new InetSocketAddress( 12345 ) );
```

Die Methode *accept* ist eine blockierende Methode, die wartet, bis eine Verbindung aufgebaut wird. Wird eine Verbindung aufgebaut, so kehrt die Methode zurück und übergibt als Ergebnis einen neuen Socket. Mit Hilfe dieses neuen Sockets kann dann mit der Gegenstelle kommuniziert werden.

```
Socket client = ssocket.accept();
```

Da die Netzwerkschnittstelle ein Bauteil des Rechners ist, ist auch die Interaktion mit dem Betriebssystem nötig. Daher "lebt" jeder Socket im Betriebssystem und es muss dem Betriebssystem mitgeteilt werden, dass man die Verbindung nicht länger benötigt. Alle Sockets – also auch ServerSockets – müssen daher geschlossen werden. Ansonsten bleiben sie auch nach Ende des Programms offen. Ein erneutes Öffnen ist dann nicht mehr möglich – auch dann nicht, wenn das Programm neu gestartet wird. Denken sie also immer an

```
mySocket.close();
```

beziehungsweise

```
ssocket.close();
```

Weitere Informationen können Sie aus dem Buch "Java ist auch eine Insel" Kapitel 21 gewinnen. Kapitel 21.6 beschäftigt sich ausschließlich mit Sockets.

4.2 Threads

Threads sind parallele ablaufende Programmteile. Sie werden benötigt, wenn das Programm beispielsweise an einem Socket lauscht, ob Nachrichten eingehen oder auf eingehende Verbindungen wartet. Während gewartet wird, kann nichts anderes getan werden. Dieses Verhalten wird Blockieren genannt. Ein Thread kann nun Abhilfe schaffen, indem ein solches Warten *parallel* ausgeführt wird. Einen Thread erzeugt man in Java wie folgt.

```
public class MyThread extends Thread
{
        @Override
        public void run{
                /* do something useful */
        }
}
```

Die Methode run() ist dabei die "main"-Methode des Threads. Ansonsten verhalten sich Threads wie ganz normale Klassen. Der folgende Codeschnipsel erzeugt einen Thread und startet ihn.

```
public class Program
{
        public static void main( String[] args ){
                MyThread t;
                t = new MyThread();
                t.start();
        }
}
```

Beachten Sie, dass ein Thread gestartet werden muss, damit er mit seiner Arbeit beginnt. Eine weitere wichtige Erweiterung von "MyThread" ist das Einfügen einer Hauptschleife.

```
public class MyThread extends Thread
{

        private boolean _terminate;

        MyThread()
        {
                _terminate = false;
        }

        public void terminate()
        {
                _terminate = true;
        }

        @Override
        public void run
        {
                while ( !_terminate )
                {
                        /* do something useful */
                }
        }
}
```

Wichtig ist es zu beachten, dass der Zugriff auf den Thread aus einem anderen Thread heraus dazu führen kann, dass Daten inkonsistent sind. Beispielsweise könnte ein Thread einen String gerade ändern, während ein anderer Thread diese Variable gerade lesen möchte. In dem Fall könnte vom zweiten Thread ein unsinniges Wort gelesen werden. Dieses Verhalten nennt man *konkurrierend*, da zwei Threads um den Zugriff auf eine Variable konkurrieren. In den Java-Paketen java.util.concurrent.* finden Sie Datenstrukturen, die gegenüber solch konkurrierenden Zugriffen geschützt sind.

Für eine tiefer gehende Abhandlung lesen Sie bitte das Kapitel 14 in dem Buch "Java ist auch eine Insel".

Achtung!

Bitte nur selbst erstellte Lösungen abgeben. Mehrfache Abgaben derselben Lösung (Plagiate) werden mit 0 Punkten gewertet und führen zum Nichtbestehen der Studienleistung!

Die Abgabe der Aufgabe erfolgt über ASSESS unter http://ess.cs.uni-dortmund.de/ASSESS/.

Die Abgabe muss in 2er bis 3er Gruppen erfolgen.