technische universität
dortmund

Fakultät für Informatik
Lehrstuhl 4

Dr. Jan Kriege
Dipl.-Inf. Horst Schirmeier

Wintersemester 2015/16

Rechnernetze und verteilte Systeme
Übungsblatt P

Ausgabe: 09.12.2015, **Abgabe:** 10.01.2016 (23:59)

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.

1 Aufgabenstellung

1.1 Projektinhalt

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

Die gesamte Infrastruktur besteht aus dem genannten Server und Clients. Eine Implementierung der Clients wird von uns zur Verfügung gestellt. Clients können sich beim Server anmelden und werden dann in eine Liste der verfügbaren Clients aufgenommen. Diese Liste kann wiederum von den Clients abgefragt werden, um Chatpartner zu finden. Als Nachrichten an andere Clients sind ausschließlich einfache Textnachrichten vorgesehen. Sämtliche Kommunikation soll über den Server laufen.

Zu Testzwecken wird neben dem Chat-Client auch ein Server als Beispiel zur Verfügung gestellt, der eine Orientierungshilfe über die gewünschte Funktionalität bietet.

1.2 Funktionsumfang

Der zu entwickelnde Chat-Server muss folgende Funktionalität bieten.

Eingaben/Konfiguration

Da es sich um einen Server handelt, sind kaum Nutzereingaben zu verarbeiten. Lediglich die Nummer des Ports, auf dem der Server auf eingehende Verbindungen wartet, sollte konfigurierbar sein und als Kommandozeilenparameter übergeben werden können. Zusätzlich sollte sich der Server per Befehl beenden lassen.

Anmeldung/Verwaltung der Nutzer

Eine der beiden Hauptaufgaben des Servers ist die Verwaltung der Nutzer. Clients sollen sich beim Server an- und abmelden können. Dazu müssen jeweils die entsprechenden Nachrichten des Protokolls abgearbeitet werden. Zusätzlich sollte der Server eine Liste mit angemeldeten Nutzern pflegen, die von den angemeldeten Clients abgefragt werden kann.

Nachrichtenaustausch

Die zweite Hauptaufgabe des Servers ist das Weiterleiten von Textnachrichten zwischen den Clients. Die Kommunikation zwischen Clients soll ausschließlich über den Server laufen, d.h. der Server erhält von den Clients eine Textnachricht und den gewünschten Empfänger und muss diese weiterleiten. Auch hierzu sind die entsprechenden Teile des Protokolls abzuarbeiten.

Das Protokoll, das das Nachrichtenformat für die Kommunikation zwischen Client und Server festlegt, ist in Abschnitt 2 aufgeführt.

1.3 Abgabe

Die Abgabe findet über AsSESS statt. Die Abgabe kann bis zum Abgabetermin beliebig oft durchgeführt werden. Es wird dann die zuletzt abgegebene Version gewertet.
Abzugeben sind die folgenden Dateien:

- ein Archiv (.zip) mit allen *.java-Dateien
- readme.txt: Hinweise, wie sich Ihr Programm in der Kommandozeile (mit `javac`) kompilieren und starten lässt.
- nochmals einzeln alle zu Ihrer Applikation gehörenden *.java-Quelldateien

1.4 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.

1.5 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/Kommandozeilenparameter dokumentieren, um das spätere Testen während der Korrektur zu ermöglichen. Dabei ist es auch zulässig, die Befehle beim Starten der Applikation zu erläutern.

1.6 Fehlerbehandlung

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

1.7 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.

2 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. Es gilt grundsätzlich, dass jeder Befehl mit einem Zeilenumbruch abzuschließen ist.

Der Server soll folgende Kommandos verstehen:

n ***name*** Meldet den Benutzer mit dem Namen *name* an. Der Server muss sicherstellen, dass jeder Name nur einmal vergeben wird. Es dürfen also nicht gleichzeitig zwei Nutzer mit dem selben Namen angemeldet sein. 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! Solange ein Client sich nicht erfolgreich angemeldet hat, antwortet der Server auf alle anderen Nachrichten mit einer Fehlermeldung. Die Verbindung eines Clients zum Server sollte nach der Anmeldung bis zum Logout dauerhaft aufrecht erhalten werden und für die gesamte Kommunikation zwischen Client und Server genutzt werden.

m ***name message*** Ein Client möchte die Nachricht *message* an Nutzer *name* senden. Falls der Nutzer existiert, leitet der Server die Nachricht weiter, andernfalls schickt er eine Fehlermeldung an den Absender der Nachricht.

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

x Beendet die Verbindung zum Server. Der Server sendet keine Antwort auf die Nachricht, wird aber die Verbindung von seiner Seite aus schließen und den Client aus der Nutzerliste löschen.

Der Server sendet folgende Kommandos:

s Bestätigt eine erfolgreiche Anmeldung am Server (Antwort auf **n** *name*).

m ***name message*** Nachricht *message* von Nutzer *name*.
Falls Nutzer Alice eine Nachricht an Nutzer Bob mit dem Text 'Test' senden möchte, findet also folgender Nachrichtenaustausch statt:

- Nachricht 1 von Alice an Server: `m Bob Test`
- Nachricht 2 von Server an Bob: `m Alice Test`

t ***Namensliste*** Das Kommando **t** wird gesendet, wenn der Client zuvor ein **t**-Kommando gesendet hat und wird gefolgt von einem Leerzeichen und der *Namensliste*. Die *Namensliste* enthält die Namen aller angemeldeten Nutzer, die jeweils ebenfalls durch ein Leerzeichen getrennt werden.

x ***text*** Kündigt das Schließen der Verbindung an. Der Server erwartet keine Antwort auf diese Nachricht, sondern wird die Verbindung von seiner Seite aus schließen und den Client aus der Nutzerliste löschen. *text* sollte einen Hinweis darauf enthalten, warum der Server die Verbindung beendet hat.

e ***Fehlermeldung*** Wird gesendet, wenn der Server einen Befehl erhalten hat, den er nicht kennt oder die Anfrage nicht bearbeitet werden kann. Die Fehlermeldung sollte einen sinnvollen Hinweis darauf geben, warum die Anfrage nicht bearbeitet werden konnte.

3 Materialien

Neben dieser Aufgabenstellung befinden sich auf der Internetseite der Veranstaltung noch zwei Java-Programme. Zum einen ist dies eine Implementierung des Clients, der sich mit Ihrem Server verbinden soll. Zum anderen befindet sich dort als Beispiel auch die Implementierung eines Servers, den Sie als Implementierungshilfe für Ihren Server nutzen können.

3.1 Client

Laden Sie sich von der Webseite der Veranstaltung das Java-Programm *ChatClient* herunter. Es kann mit dem Befehl `java -jar ChatClient.jar <IP> <Port> [debug]` gestartet werden, wobei `<IP>` und `<Port>` durch die IP-Adresse und Port-Nummer ersetzt werden müssen, unter der der Server zu erreichen ist.
Tippen Sie nach dem Starten des Clients `#help`, um eine Übersicht über alle Kommandos zu erhalten, die der Client beherrscht.
Über den optionalen Parameter *debug* kann der Client im Debug-Modus gestartet werden. In diesem Modus führt der Client keine Kommandos aus, sondern interpretiert alle Eingaben als Protokoll-Nachrichten, d.h. alle Eingaben werden unverändert an den Server weitergeleitet.

3.2 Server

Ebenfalls auf der Webseite finden Sie die Beispiel-Implementierung eines ChatServers. Laden Sie das Java-Programm *ChatServer* herunter und starten Sie es mit dem Befehl `java -jar ChatServer.jar [<Port>]`, wobei `<Port>` durch die Port-Nummer zu ersetzen ist, unter der der Server erreichbar sein soll. Die Angabe des Ports ist optional; falls kein Port angegeben ist, wählt der Server selbst einen freien Port aus. Der Server kann durch Eingabe von 'q' beendet werden, lässt darüber hinaus aber keine Interaktion zu.

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 parallel 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 https://ess.cs.tu-dortmund.de/ASSESS/.

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