Hausübung 2: Chat als verteilte Anwendung

Es soll eine einfache verteilte Chat-Anwendung realisiert werden. Sie besteht aus einem Server-Prozess und mehreren Client-Prozessen. Der Chat-Server empfängt Nachrichten von Chat-Clients, die sich zuvor bei ihm unter einem bestimmten Namen registriert haben. Der Server verbreitet diese Nachrichten an alle registrierten Clients. Die Clients zeigen sowohl die empfangenen Nachrichten als auch die Namen aller am Chat beteiligten Clients an.

Client und Server haben eigene GUIs. Die GUI des Clients soll Verbindungsaufbau, -abbau und Kommunikation nach dem unten beschriebenen *Chat-Protokoll* ermöglichen. Die GUI des Servers soll das Starten und Stoppen des Servers sowie Anzeige der Ereignisse (Verbindungsversuche, ein-/ausgehende Nachrichten etc.) gewährleisten.

Transportschicht: TCP/IP-Verbindung über Sockets

Die Prozesse sollen mittels TCP/IP-Verbindungen über Sockets miteinander kommunizieren. Zur Herstellung solcher Verbindungen erzeugt der Server ein Objekt der Klasse `java.net.ServerSocket`. Dabei wird ein Socket geöffnet und an die lokale IP-Adresse gebunden. Danach wartet der Server in der blockierenden Methode `accept()` auf Clients.

Der Client erzeugt ein Objekt der Klasse `java.net.Socket`. Dabei wird dieses mit dem Socket des Servers verbunden. Im Erfolgsfall ist jetzt eine TCP/IP-Verbindung zwischen Client und Server hergestellt: Serverseitig wird dabei für die Kommunikation mit dem Client ein zusätzliches Kommunikations-Socket erzeugt.

Anwendungsschicht: Chat-Protokoll

Server und Client tauschen jetzt Nachrichten über diese TCP/IP-Verbindung und folgen dabei dem hier beschriebenem Chat-Protokoll.

Jede Nachricht wird durch eine Zeichenkette dargestellt, die mit einem Zeilentrenner-Zeichen (*end line, '\n')* endet. Der String wird in ein Byte-Array umgewandelt (ggf. implizit durch `PrintWriter`) und über den Ausgabestrom des Sockets gesendet. Beim Empfang wird ebenfalls *zeilenweise* aus dem Eingabestrom des Sockets gelesen, wobei auch die Rückumwandlung des Byte-Stroms nach String stattfindet (ggf. implizit durch `BufferedReader`).

Jede Nachricht hat folgendes allgemeine Format (hier und weiter wird ein `'\n'` am Ende vorausgesetzt):

```
commando:rest
```

`commando` ist die Zeichenfolge bis zum ersten `':'`.

`rest` ist die Zeichenfolge nach dem ersten `':'` bis Ende der Zeile (exklusive `'\n'`). Sie kann auch leer sein.

Phase 1. Verbindungsaufbau

Ein Client sendet `connect:NAME`. Dabei ist `NAME` ist der gewünschte Name für die Chat-Kommunikation. Da das `':'`-Zeichen als Trennzeichen wirkt, darf der Name kein `':'` enthalten. Die maximale Länge des Namens beträgt 30 Zeichen.

Der Server anwortet im Normallfall mit `connect:ok` und sendet an alle Clients inklusive dem

gerade angemeldeten eine Liste mit allen derzeit registrierten Clients. Diese Liste hat die Form `namelist:NAME[:NAME],` d.h. die Namen der Clients werden mit `':'` voneinander getrennt. Der neue Client bestätigt den Empfang der Namensliste nicht. Alle Clients aktualisieren daraufhin ihre Anzeige der Chat-Teilnehmer.

Folgende Fehlerfälle sind zu berücksichtigen:

1. Bestehen bereits 3 Chat-Verbindungen, antwortet der Server mit `refused:too_many_users`
2. Ist der Name `NAME` bereits belegt ist, antwortet der Server mit `refused:name_in_use`
3. Enthält der Name unzulässige Zeichen oder ist er zu lang, antwortet der Server mit `refused:invalid_name`

In diesen Fehlerfällen baut der Server die TCP/IP-Verbindung mit dem Client ab, indem er das Kommunikations-Socket schließt.

Phase 2. Kommunikation

Clients senden Nachrichten der Form `message:MESSAGE` an den Server. `MESSAGE` ist eine beliebige Zeichenfolge, die keine Zeilentrenner enthält. Der Server bestätigt den Empfang nicht.

Der Server versendet eine von `NAME` empfangene Nachricht an alle registrierten Clients in der Form `message:NAME:MESSAGE`. Clients zeigen die empfangene Nachricht zusammen mit dem Namen des absendenden Clients an.

Phase 3. Verbindungsabbau

Der Client sendet `disconnect:`, der Server anwortet mit `disconnect:ok`, baut die TCP/IP-Verbindung mit dem Client ab, indem er das Kommunikations-Socket schließt. Die Verbindung wird auch dann abgebrochen, wenn der Client ein ungültiges Kommando sendet, der Server antwortet in diesem Fall - vor dem Abbau der TCP/IP Verbindung - mit `disconnect:invalid_command`

Danach sendet der Server an alle verbleibenden Clients eine Liste mit den noch registrierten Clients im gleichen Format wie beim Verbindungsaufbau. Alle Clients aktualisieren daraufhin ihre Anzeige der Chat-Teilnehmer.

Hinweise und Entwurfsvorgaben

1. Ein Team besteht immmer aus **zwei** Personen, die **eine** gemeinsame Lösung abgeben. Zwingend vorgeschrieben ist, dass **pro Team ein gemeinsames git-Repository** verwendet wird, während der gesamten Bearbeitungsdauer des Projektes.
2. Der Entwurf ist in zwei **Klassendiagrammen** (Server und Client) zu dokumentieren. Darüber hinaus muss der Quelltext vollständig mit sinnvollen **Dokumentationskommentaren** (für Klassen, Schnittstellen, Methoden und Attribute) erläutert sein. Die API-Beschreibungen müssen mit `javadoc` erzeugt werden.
3. Sehen Sie auf jeden Fall zwei Pakete vor: `pis.hue2.client`, `pis.hue2.server` mit Klassen `LaunchClient` und `LaunchServer`. Sinnvoll ist ein drittes Paket `pis.hue2.common`, in dem genau die Klassen vorkommen, die sowohl vom Server als auch vom Client verwendet werden. Dies könnten die Klassen der Objekte sein, die die ausgetauschten Nachrichten repräsentieren.

4. Der Server soll jeden Client in einem eigenen Thread bedienen.
5. Die GUI des Client sollte aus einem nicht editierbarem (`J)TextArea`-Objekt für die eingehenden Nachrichten der Chat-Teilnehmer, einem (`J)List`-Objekt für die Liste der Namen der Chat-Teilnehmer und einem (`J)TextField`-Objekt zur Eingabe eigener Chat-Beiträge bestehen. Als Klassenbibliothek für die GUI können Sie Swing oder JavaFX wählen.
6. Clients und Server müssen nicht unbedingt auf verschiedenen Rechnern laufen. Zum Testen empfiehlt es sich, alle Prozesse auf dem gleichen Rechner zu starten. IP-Adresse des lokalen Rechners ist 127.0.0.1.
7. Ein Chat-Client mit grafischer Oberfläche ist für Testzwecke nicht unbedingt erforderlich. Mit einem gewöhnlichen `telnet`-Client lässt sich ein Chat Server und das vorgegebene Protokoll ebenfalls gut testen.
8. Das oben beschriebene Protokoll ist *genau so* zu implementieren. Ein guter Test ist es, Client- und Server-Implementierungen aus verschiedenen Lösungen zu mischen.
9. Der Server verwaltet in einer Liste die Daten (Name, IP-Adressse, ...) zu allen Chat-Teilnehmern. Definieren Sie diese Liste als eine **threadsichere** Klasse `TeilnehmerListe`, die **alle** benötigten Zugriffe als Methoden anbietet und deren Attribute versteckt bleiben. Diese Liste soll threadsicher sein, weil sie von mehreren Threads des Servers gemeinsam genutzt wird. Identifizieren Sie die kritischen Abschnitte der Zugriffsmethoden und garantieren Sie wechselseitigen Ausschluss mit Java-eigenen Mitteln. D.h. Sie kennzeichnen die entsprechenden Methoden mit dem Schlüsselwort **`synchronized`**. Außerhalb der Klasse `TeilnehmerListe` sind Sperren (also Verwendungen des Schlüsselwortes **`synchronized`**) überflüssig und damit falsch. Dokumentieren Sie die **Klasseninvariante** sowie den Kontrakt jeder öffentlichen Methode!
10. Wenn ein Chat-Client eine Nachricht empfängt, wird dies in einem eigenen Empfänger-Thread geschehen. Die empfangene Nachricht muss in der GUI angezeigt werden, es muss also eine **nicht threadsichere** Swing (oder JavaFX-) Datenstruktur geändert werden. Dies darf bekanntlich nicht der Empfanger-Thread tun, sondern er muss den Event-Dispatch-Thread (Swing) bzw. den JavaFX-Application-Thread mit dieser Aufgabe beauftragen. Dies kann mit den Methoden `SwingUtilities.invokeLater(…)` bzw. `Platform.runLater(..)`) erreicht werden.
11. Im Client entkoppelt man die Empfangskomponente von der GUI am besten durch eine Schnittstelle namens `Ausgabe` mit allen Methoden, die den Zustand der GUI verändern: Also z.B. `zeigeNachricht(...)`, `zeigeListe(...)` , u.a. .

Codebeispiele und Literatur (alle online verfügbar)

(1) Client/Server-Programmierung in Netzwerken ,
Ratz, Dietmar, Scheffler, Jens, Seese, Detlef, and Wiesenberger, Jan
In: Grundkurs Programmieren in Java (über die THM-Bibliothek)

(2) Netzwerkprogrammierung mit Sockets, Kap. 24 in
Heinisch, Cornelia, Müller-Hofmann, frank Goll, Joachim
In: Java als erste Programmiersprache, 6.Aufl. (In späteren Auflagen fehlt dieses Kapitel)

(3) Netzwerkprogrammierung, Kap 11 in Java 7 - Mehr als eine Insel :
http://openbook.rheinwerk-verlag.de/java7/ (Aufl. 7 reicht für diese HÜ.)

(4) im Trail Custom Networking, Kapitel All About Sockets