Hochschule Kaiserslautern
Entwicklung verteilter Anwendungen in Java
Sommersemester 2015

Chat-Applikation auf Server- und Clientebene

868407
Dorian Weidler, Luca Hartmuth, Julian Meisel
25. Juni 2015

Inhaltsverzeichnis

1 Einleitung

Ziel dieses Teilprojekts war es einen Multiuser-Server zu programmieren, der fehlerfrei Verbindungen mit mehreren Nutzern aufbaut, diese verwaltet und sie auch wieder abbaut.

Eine weitere Anforderung war den Server so parallel zu gestalten wie möglich aber trotzdem so performant wie möglich. Resultat war, dass jedem User ein Thread zusteht, der beim Verbinden für ihn geöffnet wird. In diesem Thread werden all seine Aktionen (Login, Registration und Nachrichten, damit verbunden auch Befehle) abgearbeitet.

Des Weiteren haben wir die persönliche Anforderung an das Client Teilprojekt gestellt, dass möglichst wenige Updates für den Endnutzer bereitgestellt werden müssen. Aus diesem Grund haben wir uns für eine rudimentäre GUI entschieden, die ein großes Textfeld für die empfangen und ein kleines Textfeld für die zu sendenden Nachrichten enthält. Außerdem befinden sich noch ein kleines Label auf dem GUI, das anzeigt in welchem Channel man sich befindet und eine Userliste mit den aktuell im Channel befindlichen Usern.

Alle weiteren Aktionen, wie zum Beispiel das Ändern des aktuellen Channels, werden mittels Kommandos im Chat abgearbeitet. So sieht zum Beispiel der Befehl zum Ändern des aktuellen Channel wie folgt aus:

```
/join #channelname
```

Listing 1: Beispiel eines Befehls

Wir sahen darin einen mächtigen Vorteil, und zwar hat unserer Meinung nach der Serveradministrator mehr Kenntnisse bzgl. seines Computers als der Endanwender. Ihm sollte man die Updates zumuten können. Des Weiteren stellt sich das Verteilen der Updates auf eine Schar von Usern als deutlich schwerer heraus als des reine Verteilen auf die Server. Außerdem muss der Fall, dass ein Nutzer mit einer älteren Version mitchatten möchte, in aller Regel nicht beachtet werden. Sollte der Client doch einmal geupdated werden müssen, so muss dieser Fall natürlich gründlichst geprüft werden und ggf. durch weitere Änderungen im Serverquelltext ebenfalls abgedeckt werden.

In unserem aktuellen Bestreben ist das Updaten des Funktionsumfangs eine einseitige Angelegenheit für den Anbieter des Servers und damit seine/n Administrator/en.

2 Klassenerklärungen

- CMain: Das ist die Einstiegsklasse des Programms, sie liest mithilfe von CConfigParser die Konfigurationsdatei aus und gibt die nötigen Einstellungen an den Konstruktor der CServer Klasse weiter um ein solches Objekt zu erzeugen.

- CServer: Diese Klasse erbt von Thread und bekommt als Parameter im Konstruktor den Standard-Channelnamen und den Port auf dem der Chatserver horchen soll. Danach wird der Thread gestartet, indem ein Serversocket auf dem, in der Konfigurationsdatei angegebenen, Port öffnet, alle Channels aus der channels.lst mittels des CChannelParsers ausliest und auf eingehende Verbindungen wartet. Bei einer neuen Verbindung wird ein CClientHandler Objekt erzeugt und der Client zu den verbundenen Clients hinzugefügt. Daraufhin wartet der Server wieder auf neue Verbindungen.

- CClientHandler: Diese Klasse erbt ebenfalls von Thread. Im Konstruktor werden zunächst alle programmierten Befehle jeweils als CCommand Objekte angelegt, die den Befehl selbst und einen dazugehörigen Hilfetext beinhalten. Dann wird dem geöffnet Socket, d.h. der Verbindung zum Client, ein Timeout gesetzt, der dazu dient eine Exception zu werfen, wenn der Timeout ausläuft. Die Exception wird abgefangen und dabei geprüft ob die Verbindung noch besteht. Ist dem so, wird die Verbindung zum Client geschlossen und alle anderen Clients dementsprechend informiert. Der ClientHandler startet sich dann selbst als Thread und liest aus dem InputBuffer jeweils 256 Byte und prüft diese auf die Packet-Struktur. Stimmt sie wird das Paket entsprechend auf die nötige Länge beschnitten und ein CClientPacket Objekt damit erzeugt und darauf die handlePacket-Methode des Objekts aufgerufen. Dann liest er wieder aus.

- CClientPacket: Diese Klasse dient dazu, ein Packet zu prüfen, in seine Einzelteile zu zerlegen und dann entsprechend seines Typs zu behandeln. Zunächst werden dem Konstruktor das gesamte Packet als String, der aufrufende CClientHandler und die IP des verbundenen Clients. Aus dem gesamten Paket wird dann der Typ ausgelesen und als Objektattribut gesetzt. In der handlePacket()-Methode wird nun anhand des Typs geprüft was getan werden muss bzw. ob etwas getan werden muss. (Wird z.B. ein Login- oder Registerpacket von einem eingeloggten Client empfangen, wird dieses verworfen) In den entsprechenden Methoden processRegisterOrLoginPacket() (Behandelt beides, bekommt aber als Parameter ob es ein Login- oder Registerpacket ist,

weil der Anfang gleich ist. Am Ende wird nur unterschieden ob ein Login gemacht werden muss oder eine Registration) oder processMessage() werden die Pakete dann entgültig zerlegt und ihre Einzelteile ausgewertet.

- CClient: Dient als Repräsentation des Clients auf dem Server. Diese Klasse dient eigentlich nur als Datenhalter. Die einzige richtige Methode der Klasse ist die setCurrentChannel()-Methode, die den Channel wächselt, dem Client ein entsprechendes MessagePaket schickt mit allen im neuen Channel befindlichen Clients und ruft die leaveChannel Methode beim aktuellen Channel für den Client auf und die joinChannel Methode beim neuen Channel.

- CChannel: Dient als Repräsentation eines Channels und bietet die Methoden joinChannel und leaveChannel die jeweils die Clients über neue User im Channel bzw. verlassende User informiert.

- CChannelParser: Liest die channels.lst aus und gibt ein Array aller Channels aus.

- CConfigParser: Liest die config.cfg aus und speichert die darin befindlichen Informationen als Objektattribute des Parsers. Diese können mittels der Getter Methoden ausgelesen werden.

- CServerPacket: Dient als Bauplan für Pakete die vom Server zu einem Client gesendet werden. Darin ist auch die sendMessage Methode, die einen Empfänger und einen Sender als Attribut bekommt. So können z.B. Private Nachrichten realisiert werden. Um eine Nachricht an alle zu schicken müsste einfach durch alle verbundenen Clients iteriert werden. Nur für Nachrichten ist weiteres außer dem Packetheader nötig. Alle anderen Pakete (Server -> Client) bestehen nur aus dem Packetheader.
  Mehr dazu erfahren Sie im Packet Structure Abschnitt.

- CServerPacketHeaders: Beinhaltet nur eine Methode um den Packetheader eines Serverpackets auf Gültigkeit zu prüfen.

- CLogger: Ist eine Hilfsklasse im in die angegebene Datei zu schreiben. Wird zum Beispiel in der utils Klasse benutzt um die Servernachrichten in die log.txt Datei zu schreiben.

- utils: Beinhaltet die Ausgabemethoden debugMsg, errorMsg und infoMsg und eine isNumber Methode um zu prüfen ob es sich bei einem String um eine valide Zahl handelt oder nicht. Die Ausgabemethode debugMsg() prüft dabei

zunächst ob der Server aktuell im Debugmodus ausgeführt wird. Ist dem nicht der Fall, wird keine Ausgabe getätigt.

- Exceptions-Package: Enthält alle Exceptions.

3 Installation

Zur Installation muss lediglich das .zip Verzeichnis mit einem entsprechenden Programm entpackt werden und daraufhin die config.cfg nach eigenem Ermessen bearbeitet werden.

```
/join #channelnamePort = 1337
QueueLength = 1500
DefaultChannel = #default
```

Listing 2: Beispielinhalt der Konfigurationsdatei

Daraufhin muss noch die channels.lst bearbeitet werden. **Wichtig:** Der Default-Channel aus der config.cfg **muss** dort nochmal stehen! Eine Zeile pro Channel!

```
#default
#test
```

Listing 3: Beispielinhalt der Channeldatei

4 Hinzufügen eines neuen Befehls

Dazu muss die onMessageReceived()-Methode entsprechend modifiziert werden. Dort sind bereits einige Befehle angelegt, die essentiell für die Funktionalität sind.

Hier noch ein weiteres Beispiel, dass Sie theoretisch Copy&Paste integrieren können.

```
// Bei onMessageReceived in die If-Abfrage integrieren
else if(message.startsWith("/test")){
        /* Fall: /test als Beispiel */
        sendMessage("Testnachricht", mServer.getServerClient
            ());
}
/* Im Konstruktor den anderen registerCommand anhaengen. */
registerCommand("/test", "Dient als Beispiel.");
```

Listing 4: Beispielinhalt der Channeldatei

Im Beispiel wird ein Befehl "/test"eingefügt, der dem User nur eine Nachricht "Testnachricht"mit dem Serverclient schickt.