Hochschule
Kaiserslautern
University of
Applied Sciences

Hochschule Kaiserslautern

Entwicklung Verteilter Anwendungen mit Java

Multi Client Chat App

Leandre joel Tchefandjem & Mohamed Achref Boukhit

Unter Aufsicht von

Prof. Dr. Manuel Duque-Anton

Inhaltsverzeichnis

❖ Einleitung

Der Zweck dieser Anwendung besteht darin, Benutzern an unterschiedlichen Rechnern zu kommunitieren .Das ist *Client-Server-Konzept.*

Das Client-Server-Modell (auch *Client-Server-Konzept*, *-Architektur*, *-System* oder *-Prinzip* genannt) beschreibt eine Möglichkeit, Aufgaben und Dienstleistungen innerhalb eines Netzwerkes zu verteilen. Die Aufgaben werden von Programmen erledigt, die in Clients und Server unterteilt werden. Der Client kann auf Wunsch einen Dienst vom Server anfordern (z. B. ein Betriebsmittel). Der Server, der sich auf demselben oder einem anderen Rechner im Netzwerk befindet, beantwortet die Anforderung (das heißt, er stellt im Beispiel das Betriebsmittel bereit); üblicherweise kann ein Server gleichzeitig für mehrere Clients arbeiten.

❖ Server/Client

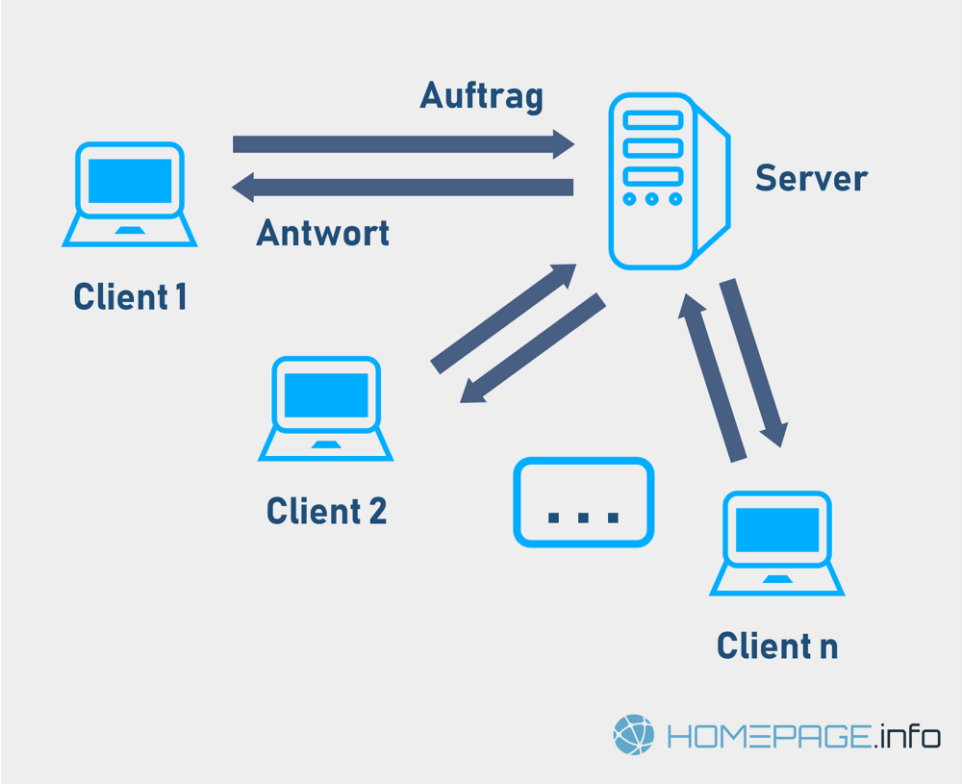
Wir schreiben einen Server und einen Client, die über Sockets miteinander kommunizieren? Ein Socket (engl. Sockel) ist eine bidirektionale Netzwerk-Kommunikationsschnittstelle, deren Verwaltung das Betriebssystem übernimmt.
Die Kommunikation findet zwischen einem Server und einem Client über einen definierten Port statt.Das Programme demonstriert das Versenden eines Strings durch einen Client über Port an einen Server auf die entschbrechende Adresse, das Rücksenden durch den Server und die Ausgabe wiederum durch den Client.

In main()-Methode nutzt der Client die Klasse java.net.Socket zur Einrichtung einer Socket-Verbindung über ein gegebene Port Methoden getInputStream() und getOutputStream() liefern Input-, bzw. OutputStreams, aus denen gelesen, bzw. in die geschrieben werden kann.
Zum Schreiben von Strings bieten wir ein PrintStream an, dessen Konstruktor das InputStream-Objekt und einen boolschen Wert für das Autoflushing entgegennimmt. Methode println() schreibt den gewünschten String und fügt automatisch einen Zeilenumbruch an.

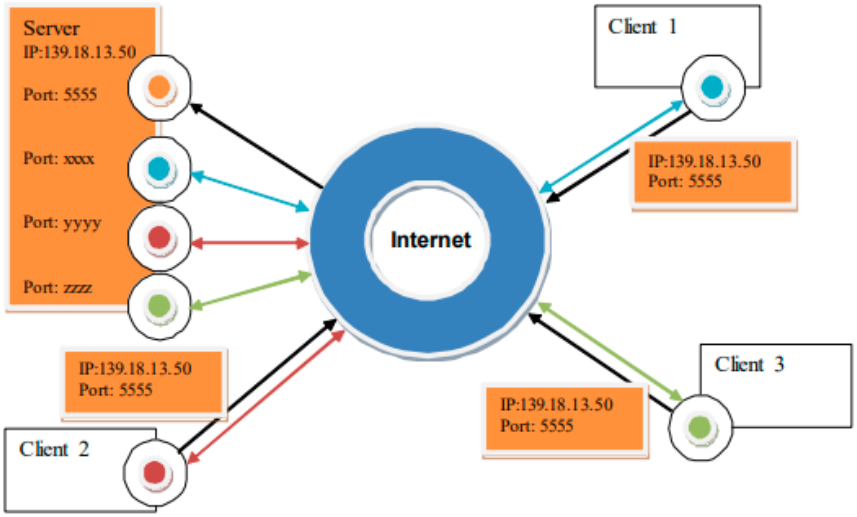
Das Lesen erfolgt zur Performance-Optimierung zunächst in einen BufferedReader, der ein Objekt eines InputStreamReaders im Konstrukor entgegennimmt. Sein Inhalt wird schließlich über eine Schleife Zeile für Zeile ausgelesen und auf die Konsole ausgegeben. Die Methode available() erlaubt bei Bedarf die Abfrage der im InputStream verfügbaren Bytes.
Probleme bei den Input-Output-Vorgängen, sowie ein falscher Host-Name können Exceptions werfen, die in einem try-catch-Block abgefangen werden müssen.

```
Socket socket = new Socket(adr.getText(), Integer.parseInt(port.getText()));

InputStream inputstream = socket.getInputStream();

OutputStream outputstream = socket.getOutputStream();

InputStreamReader instreamreader = new InputStreamReader(inputstream);

BufferedReader br = new BufferedReader(instreamreader);

pw = new PrintWriter(socket.getOutputStream(), true);
```

- ***Client-Server-Konzept***



In verteilten Systemen unterscheidet man fast immer zwischen „Client“ und „Server“. Ein Server ist ein Programm, das einen bestimmten „Dienst“ anbietet, den ein anderes Programm, der Client in Anspruch nimmt. Beispiel: Ein Webserver stellt im Internet Webseiten zur Verfügung, die von einem Webbrowser (Client) abgerufen werden können. In der Regel wartet ein Server auf eine eingehende „Anfrage“ von einem Client, bearbeitet diese und schickt dem Client dann eine „Antwort“. Der Client andererseits veranlasst den Verbindungsaufbau, schickt dem Server eine „Anfrage“ und wartet dann auf die „Antwort“. Häufig besteht die Kommunikation zwischen einem Client und einem Server aus einer Folge von Anfragen und Antworten, die nach vorher vereinbarten Regeln abläuft. Diese Kommunikationsregeln nennt man ein „Protokoll“.



➢ Hostadressen (IP)

IP-Adressen identifizieren einen Host

Während MAC-Adressen (siehe vorhergehender Abschnitt) an Hardware gebunden sind, arbeiten die höheren Schichten der Netzwerkkommunikation mit logischen Adressen, den sogenannten **IP-Adressen**. IP-Adressen gibt es heute in zwei Formaten, als IPv4- (32-stellige Binärzahlen) und als IPv6-Adressen (128-stellige Binärzahlen). IPv6 wird auf lange Sicht IPv4 verdrängen, jedoch bremst der nicht unerhebliche Aufwand zur Umstellung den Umstieg.

➢ Ports

Wollen zwei Prozesse miteinander kommunizieren, egal ob über das Netz oder Systemintern, dann identifizieren sich die Prozesse gegenüber TCP/IP mit einer *Port-Nummer*. Das ist eine 16bittige Zahl, es gibt also 65535 Ports für jedes Transport-Protokoll (UDP und TCP). Die Port-Nummer sagt also aus, an welchen Prozess ein bestimmtes Paket weitergereicht werden möchte. Die Prozesse müssen vorher für Klarheit gesorgt haben, welche Ports sie benutzen wollen. Teilweise geschieht das "on the fly", teilweise sind Portnummern fest vorgegeben: "*well known ports*". Diese werden immer von den entsprechenden "*well known services*" benutzt, also zB. portmap-Service, DNS, DHCP etc.

➢ Arten von Sockets

Internet-Sockets ermöglichen die Kommunikation mittels bestimmter Kommunikationsprotokolle. Generell kann man unterscheiden zwischen **Stream Sockets** und **Datagram Sockets**: Stream Sockets kommunizieren über einen Zeichen-Datenstrom; Datagramm Sockets über einzelne Nachrichten. In der Netzwerkkommunikation verwenden Stream Sockets meist TCP, Datagramm Sockets üblicherweise UDP. TCP ist verlässlicher, die Reihenfolge und Zustellung von Paketen werden garantiert (Prinzip: ganz oder gar nicht). UDP ist für bestimmte Aufgaben effizienter und flexibler, oft auch schneller (zeitlich) – Reihenfolge und Zustellung der Pakete werden jedoch nicht garantiert (weiterhin sind Duplikate möglich).

Während Stream Sockets und Datagramm Sockets den TCP- oder UDP-Header eines Datenpakets normalerweise verbergen und automatisch setzen, lassen **Raw Sockets** (*Raw*: roh) das Erstellen eigener TCP- und UDP-Header zu. Raw Sockets werden am ehesten in netzwerknahen Applikationen verwendet, z. B. für Router, Packet-Sniffer oder bei Packet-Injection.

Stream Sockets

Client-seitig:

1. Socket erstellen
2. erstellten Socket mit der Server-Adresse verbinden, von welcher Daten angefordert werden sollen
3. Senden und Empfangen von Daten
4. evtl. Socket herunterfahren (shutdown(), close())
5. Verbindung trennen, Socket schließen

Server-seitig:

6. Server-Socket erstellen
7. Binden des Sockets an eine Adresse (Port), über welche Anfragen akzeptiert werden
8. auf Anfragen warten
9. Anfrage akzeptieren und damit ein neues Socket-Paar für diesen Client erstellen
10. Bearbeiten der Client-Anfrage auf dem neuen Client-Socket
11. Client-Socket wieder schließen.

Datagram Sockets

Client-seitig:

12. Socket erstellen
13. An Adresse senden

Server-seitig:

14. Socket erstellen
15. Socket binden
16. warten auf Pakete

❖ Projektaufbau

➢ Socket-Programmierung

Socket-Programmierung wird verwendet, um eine Verbindung zwischen zwei Knoten herzustellen, nämlich Server und Client in einem Netzwerk. Auf diese Weise können wir eine bidirektionale Verbindung zwischen mehreren Knoten herstellen.

➢ Logik

1) Erstens verwenden wir Sockets, um eine Verbindung zwischen den Knoten anzufordern, indem wir die Portnummer übergeben und den Host als localhost beibehalten.

2) Sobald der Server die Verbindung akzeptiert, implementieren wir eine Runnable-Schnittstelle und überschreiben ihre Methoden, um die Nachrichten zwischen den Knoten anzuzeigen.

4) Wir werden Threads verwenden, um mehrere Nachrichten von Clients gleichzeitig zu verarbeiten.

5) Sobald den buton “logout” ausgedruck wird, stoppt unser Programm.

➢ **Server**

Ein Server ist ein Programm (Prozess), das mit einem anderen Programm (Prozess), dem Client kommuniziert, um ihm Zugang zu einem Dienst zu verschaffen. Hierbei muss abgrenzend beachtet werden, dass es sich bei „Server“ um eine Rolle handelt, nicht um einen Computer an sich. Ein Computer kann nämlich ein Server und Client zugleich sein

Probleme bereiten die blockierenden Methoden; sowohl die Methode `java.net.ServerSocket#accept()` als auch die Methode `read()` (oder seine Verwandten) warten solange, bis sich ein Client angemeldet hat bzw. bis die Gegenseite eine Nachricht gesendet hat.

Auf der Seite des Servers wird es deshalb in der Regel einen eigenen Thread geben, der auf Anmeldungen von Clients wartet. Für jeden Client, der sich angemeldet hat, wird dann wiederum ein neuer Thread gestartet, der auf Nachrichten des Clients wartet. Daneben wird man in aller Regel den Client in einer Liste speichern, damit es möglich ist, allen angemeldeten Clients eine Nachricht zu senden.

Auch im Client gibt es einen eigenen Thread, der auf Nachrichten des Servers wartet. Während man es sonst bei Methodenaufrufen gewohnt ist, dass man auf den Rückgabewert (und sei es `void`) der Funktion wartet (synchron), verläuft die Kommunikation hier asynchron.

- **Client**

Die Klasse `java.net.ServerSocket` dient in erster Linie zur Anmeldung von Clients. Sie hört nicht - wie man vielleicht erwarten könnte - auf die Nachrichten von Clients. Im Konstruktor wird der Port (im Beispiel 1234) übergeben (die IP-Adresse des Servers ist natürlich die IP-Adresse des Computers, auf dem der Server läuft). Die Methode `accept()` wartet so lange, bis sich ein Client verbunden hat. Dann gibt sie einen `java.net.Socket` zurück.

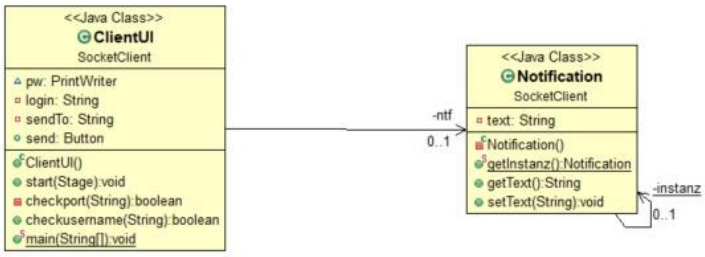
Die Klasse `java.net.Socket` hat zwei Funktionen:

im Server dient sie dazu, sich den angemeldeten Client zu merken, auf seine Nachrichten zu hören und ihm zu antworten. Erzeugt wird eine Instanz durch die Methode `accept()`, wenn sich ein Client angemeldet hat.

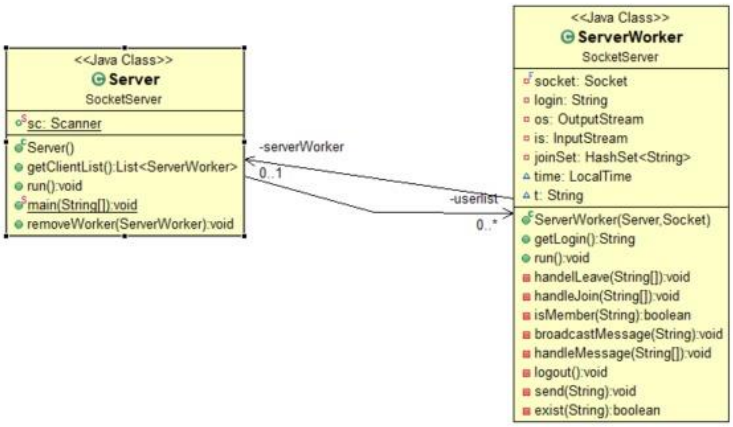
im Client erzeugt sie die Verbindung zum Server; dem Konstruktor wird die IP-Adresse des Servers und dessen Port übergeben. Mit dem Aufruf des Konstruktors im Client wird die Verbindung hergestellt - beim Server gibt `accept()` den verbundenen Client zurück.

Die Kommunikation läuft dann auf beiden Seiten gleich:
Mit `java.net.Socket#getInputStream()` bzw. `java.net.Socket#getOutputStream()` können die Nachrichten der anderen Seite gelesen werden bzw. dorthin gesendet werden. Die lesenden Methoden blockieren dabei so lange, bis eine Nachricht empfangen wurde.

SocketClient



SocketServer



❖ Interface

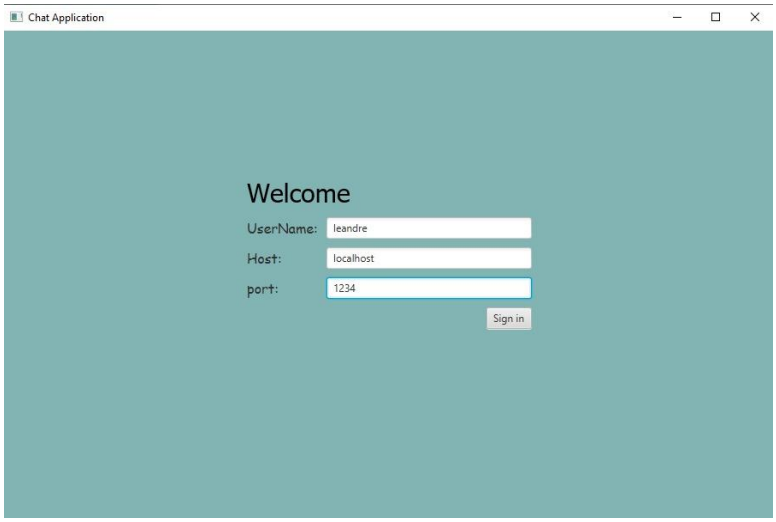
➢ Anmeldung

Hier wir den benutzer zuerst sich durch ein interface wie folge loggen :

- User Name eingeben (darf keine simbole enthalten)

-Host der Server schreiben , es wird localhost hier verwenden

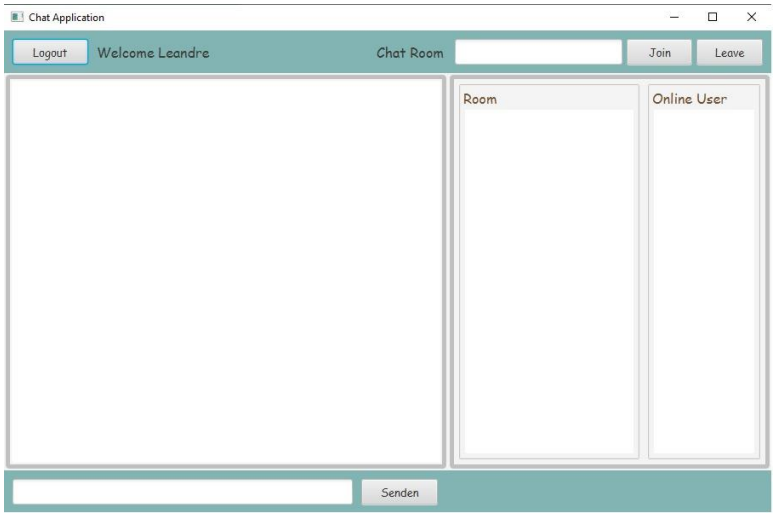
-Und Port von den Server eingeben (nur zahlen)



➢ Homepage

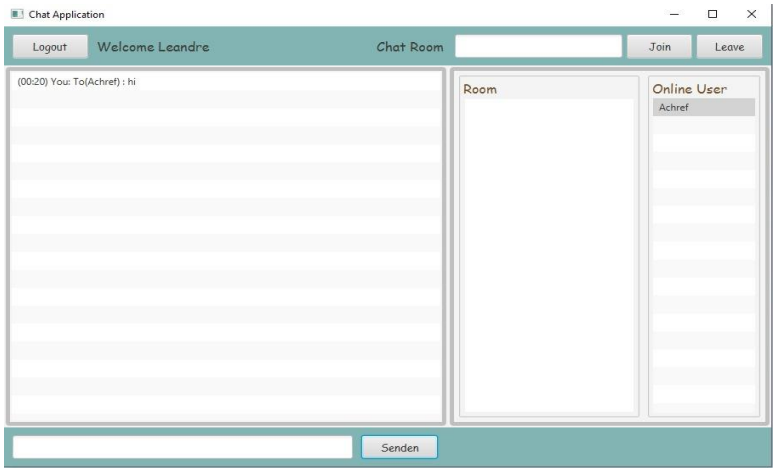
Nach dem er sich angemelden hat ,wird ein neue ferstern gezeigt werden ,nannlich der HomePage .man kann auf diser seite :

- Nachrichten schreiben und lesen
- Sich auf einen oder mehr Grüppen beitreten und verlassen
- Sehen wer ist auf den Server angemelden
- Sich auch abmelden



- Private Chat

Der User muss die name von anderem benutzer, mit dem er kommunitieren wird, auswählen.werden die nachrichten nur an dieser benutzer gechickt .



➢ Gruppenchat

- Join

Benutzer können Gruppen beitreten, indem sie auf den Gruppennamen im Feld “Room” klicken und zur Bestätigung auf "join". Von diesem Moment können Benutzer alle in der Gruppe gesendeten Nachrichten anzeigen Empfangen und verfassen Sie sogar Nachrichten an Gruppen.

- Leave

Benutzer können bei Bedarf auch zuvor beigetretene Gruppen verlassen Er muss den Namen der Gruppe in das Textfeld Room klicken und Beenden Sie ihre Mitgliedschaft, indem Sie auf „leave“ klicken.

➢ Logout

Der User kann sich durch ein Klick auf” Logout” abmelden .

❖ Fazit

Es beschreibt ein sehr einfaches Unidirektionales Client- und Server-Setup, bei dem ein Client eine Verbindung herstellt, Nachrichten an den Server sendet und der Server sie über eine Socketverbindung anzeigt. Es gibt eine Menge Low-Level-Dinge, die passieren müssen, damit diese Dinge funktionieren, aber das Java API-Netzwerkpaket (java.net) kümmert sich um all das, was die Netzwerkprogrammierung für Programmierer sehr einfach macht.

Ich habe Spaß an diesem Projekt zu arbeiten, teilweise richtig Komplexe Fragen des Client-to-Client-Layouts oder der Kommunikation Ein Server zum Lösen. Das Schreiben "größerer" Programme stellt dies sicher Verbessern Sie das allgemeine Verständnis der Programmiersprache "Java" und ihrer Funktionsweise PC besser kennenlernen. mit komplexen Problemen umgehen Die Kommunikation zwischen Computern stellt immer wieder spannende Aufgaben und neue Herausforderungen dar, die ich mit Großem interesse angehen kann.

Literatur

[1] Entwicklung verteilter Anwendungen: Mit Spring Boot & Co (erfolgreich studieren)

von Wolfgang Golubski | 19. Februar 2020 ISBN-10 **3658268131**

[2] Java Programmieren von Anfang an von reiner Backer ISBN 3-499-61203-8

[3] Dr. Michael Janneck. Ein einfacher Chat. 2009. url: https : / / http : / / www . janneck.de/pmwiki/uploads/01_Ein_einfacher_Chat_v02.pdf .