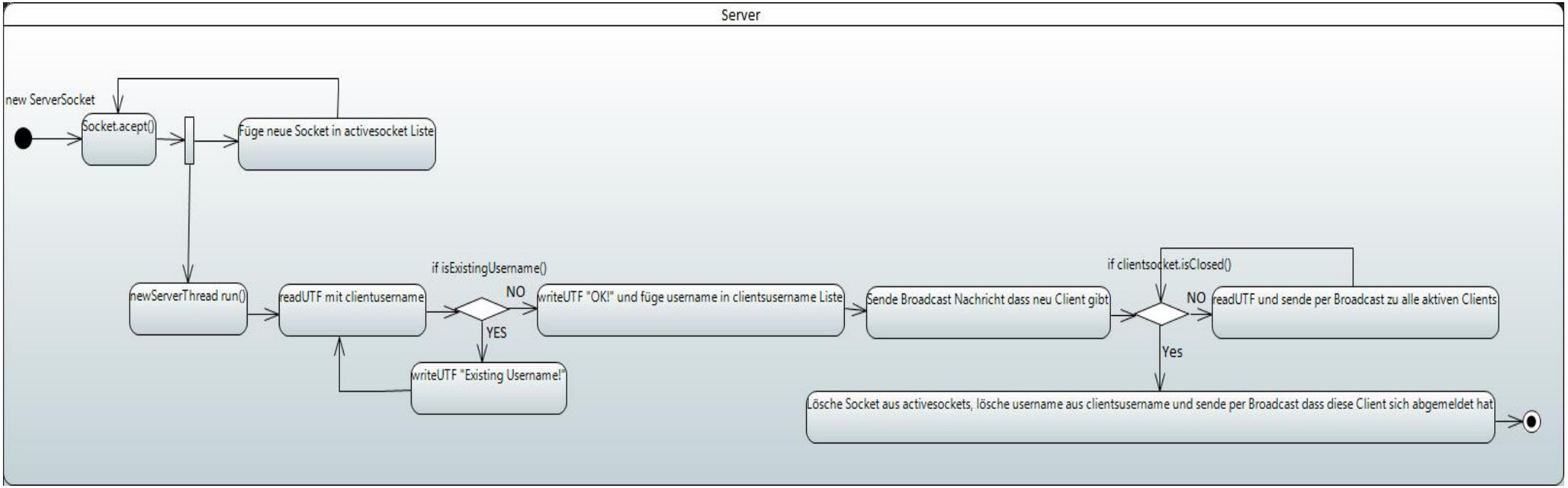
14057 - Savev David

Aufgabe nr. 2 – VERTEILTE SYSTEME

In meiner Lösung habe ich zwei Files, Server.java und Client.java, die 2 Haupt-Klassen implementieren: Server und Client; außerdem, nutzt der Server.java auch die Klasse ServerThread die Thread erstreckt, und Client.java die Klasse ClientThreadWrite (die auch den Thread erstreckt).

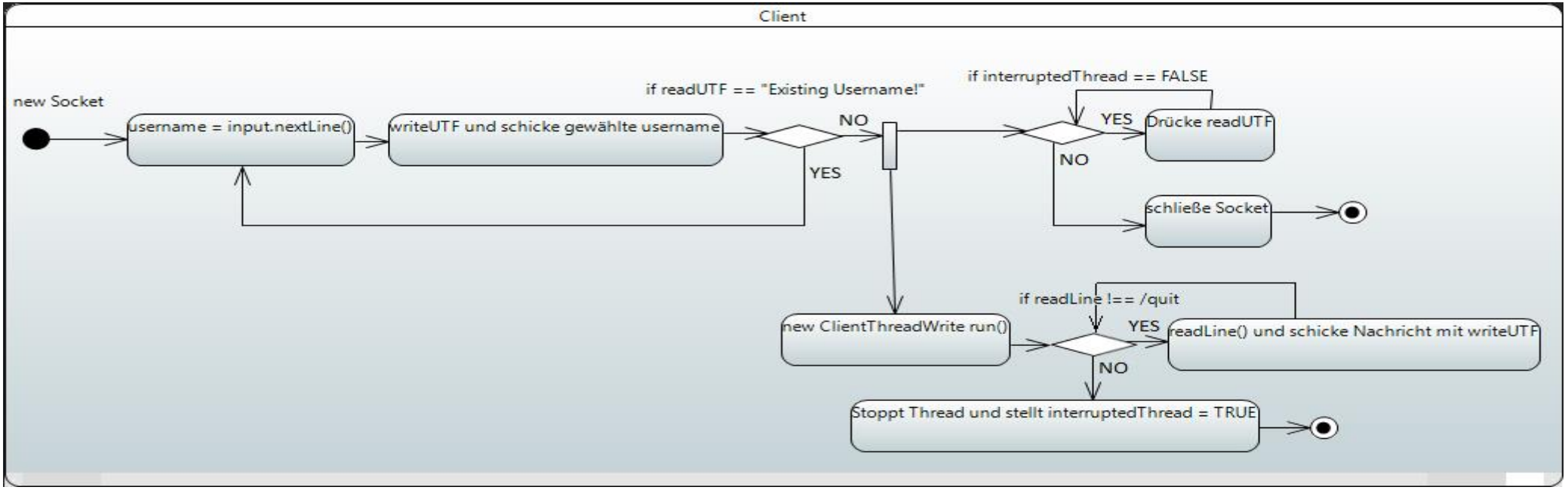
Die Klasse Server wartet in ihrem Thread main auf die neue Client-Verbindungen (sie hört und wartet auf TCP Port 10101) und ruft für jede neue Verbindung einen neuen Thread die UTF Nachrichten liest und die per Broadcast zu allen verbundenen Clients schickt (ich habe den Bedürfnis zwei Threads zu benutzen weil die Methode .accept() - vom Socket Klasse - blockierend ist).
Der Server benutzt auch zwei statische ArrayListen: activesockets und clientsusername.
Die erste enthält die aktive Sockets und das ist nützlich weil unseren Server die ganzen UTF Nachrichten per Broadcast zu allen aktive Clients schicken soll.
Die zweite enthält die aktive Usernames in den Chat: wenn ein Name schon von jemanden benutzt ist, dann ist die Verwendung des Namens verboten.
Immer wenn ein neuen Client sich auf den Chat verbindet/abmeldet, wird eine Nachricht per Broadcast zu allen aktive Clients geschickt, die informiert dass der Client mit dem Namen XXX sich verbunden/abgemeldet hat.

Der folgende Aktivitäts Diagramm zeigt wie der Server funktioniert:



Die Klasse Client benutzt ihre Thread main um sich mit den TCP Server auf Port 10101 mit einem gewissen Name zu verbinden, um UTF Nachrichten die vom Server kommen zu lesen, und um einen neuen Thread für UTF Nachrichten schreiben zu starten (sonst mit nur einem Thread kann man nicht in Parallel Nachrichten lesen und Nachrichten schicken, weil die Methode readLine() - vom Scanner Klasse - blockierend ist).
Der Client kann sich entweder mit „/quit“ abmelden oder er kann einfach den Command Line schließen.
Genauer gesagt, der ClientThreadWrite benutzt ein loop der kontrolliert wenn der User „/quit“ getippt hat, und wenn ja, wird der Thread blockiert und die booleanische Variable interruptedThread zu TRUE eingestellt; diese Variable stoppt auch den Thread main und schließt den Socket von Client-Seite.

Der folgende Aktivität Diagramm zeigt wie den Client funktioniert:



Logischerweise, sind die ganzen Exceptions verwaltet:

- Wenn den Client den Command Prompt schließt (ohne /quit zu schreiben) → der Server wird merken dass der Socket von Client Seite geschlossen ist, und der Client wird verwaltet als er sich abgemeldet hat;
- Wenn der Server nicht läuft und ein Client probiert die Verbindung zu öffnen oder wenn Clients im Chat interagieren, und der Server Probleme hätte → eine Exception wird abgerufen und der Client sieht dass Verbindungsprobleme gibt.

Eine alternative Lösung auf Grundlage des UDP Protokolls könnte mit Hilfe von DatagramSockets und DatagramPacket angeboten werden.

Die Server Klasse in UDP könnte so aussehen:

```
DatagramSocket socket = new DatagramSocket(10101);
while (true)
{
DatagramPacket packet = socket.receive();
DatagramPacket response = new DatagramPacket(data, data.length, packet.getAddress(), packet.getPort());
socket.send(response);
}
```

Wir können sofort bemerken, dass die Server Klasse in UDP einfacher wäre (in TCP müssen wir zwei Threads benutzen), und seit wir in UDP nur Packete und keine ACK schicken, wird der Austausch von Nachrichten viel schneller sein. Trotzdem, für Chats ist es besser den TCP Protokoll zu benutzen, weil:

- Seit UDP ein connection-less Protokoll ist, ist es schwierig aktive Clients zu erkennen;
- In TCP, wenn bestimmte ACK von der andere Seite nicht ankommen, wird der Packet nach eine gewisse Zeit wieder gesendet (UDP Datagramme können verloren gehen und werden nicht wiedergesendet);
- In TCP, mit Hilfe von Sequenz-Nummern, wird die Reihenfolge von den Packete kontrolliert um keine sinnlose Nachrichten zu entstehen (z.B. „BIN HALLO! ICH DAVID“ kann in eine UDP Kommunikation entstehen);
- Denn TCP der AIMD Algorithmus implementiert für die Staukontrolle und die Bandbreite wird zwischen Prozesse gleichmäßig aufgeteilt (in UDP kann sein dass ein sehr aktiven Client die ganze Bandbreite ausnutzt, und Nachrichten von andere Clients werden niemals bei Server ankommen).