**Simona Simoni**

**Kommunikationsprotokoll**

In der Informatik und in der Telekommunikation ist ein Kommunikationsprotokoll eine Vereinbarung, nach der die Datenübertragung zwischen zwei oder mehreren Parteien abläuft. In seiner einfachsten Form kann ein Protokoll definiert werden als eine Menge von Regeln, Syntax, Semantik und Synchronisation der Kommunikation bestimmen. Kommunikationsprotokolle sind eine Grundlage des Internets und tragen wesentlich zu seiner Leistung und seinem Erfolg bei. Zu den wichtigsten zählen das Internet Protocol und das Transmission Control Protocol, zusammengefasst als TCP/IP.

Die Kommunikationsprotokolle sind für:

* Feststellen der Existenz des anderen Endpunkts der Verbindung
* Datenflusskontrolle
* Fehlerkorrekturverfahren
* Beendigung der Verbindung

wichtig.

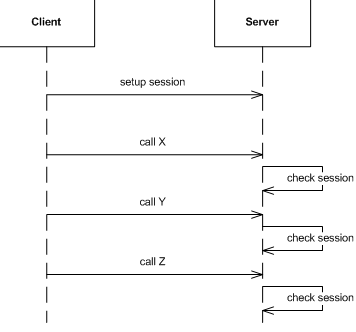


Abbildung 2: Beispiel für ein Kommunikationsprotokoll

Bei unserem Projekt gibt es zwei unterschiedliche Kommunikationsabläufe mit dem Server: die Kommunikation der Clients (Swing-Client und Web-Client) mit dem Server und die Kommunikation des Servers mit dem Arduino.

## Übersicht und beteiligte Komponenten

|  |  |
| --- | --- |
| **Lichter** | Ein LED Licht für jeder Raum. Die Lichter können ihren Status als Information an die Clients geben und Befehle von den Clients nehmen. |
| **Ventilator** | Ein Motor, der die Geschwindigkeit verändern kann und wie ein Ventilator ausschaut. Er kann Informationen über seinen Status geben und Befehle von den Clients nehmen. |
| **Heizungsgerät** | Das Heizungsgerät schaut nicht wie eine Klimaanlage aus. Es kann Informationen über seinen Status geben und Befehle von den Clients nehmen. |
| **Temperatursensor** | Dieser Sensor misst die Temperatur im Raum. Die Clients können seinen Status nur wissen, aber nicht ihn steuern. |
| **Garagentor** | Ein Elektromotor, der das Garagentor öffnet oder schließt. Er kann Informationen über seinen Status geben und Befehle von den Clients nehmen. |
| **Rollos** | Ein Elektromotor mit verschiedenen Stufen, der das Rollo auf- und zumacht, so viel die Clients möchten. Er kann die Clients über seine Lage informieren und Befehle nehmen. |
| **Wetter Sensor** | Dieser Sensor zeigt an, wenn es regnet. Er kann nur seinen Status als Information an die Clients geben. |

Tabelle 1: Die Bauteile

* **Jedes Gerät kann sowohl manuell als auch automatisch gesteuert werden.**

Die Verbindung zwischen dem Server und den Clients ist ähnlich mit der Verbindung zwischen dem Server und dem Arduino.

Der Verbindungsaufbau wird mit Hilfe von zwei Methoden gemacht, die ich updateState und getState genannt habe.

**Arduino (A) - Java Server (SA) Kommunikation**

**Fett**: Schlüsselwörter, *kursiv:* Text/Werte

### **Protokollübersicht:**

Gültige Befehle in Phase (I)

* **Login**

Hier gibt man den Benutzernamen und das Passwort. Der Benutzername und das Passwort befinden sich in der Datenbank und werden mit den eingegebenen Anmeldedaten verglichen.

* + *Dieser Befehl macht den Eingang im System möglich, damit man da Zugriff hat. Im Arduino wird das eingegebene Passwort überprüft, ob es mit dem in die Datenbank gespeicherten Passwort gleich ist. Die Clients geben das Passwort ein und schicken es an den Server.*

Gültige Befehle in Phase (II)

* **getState**
  + *Bei der* ***getState*** *Methode fragt der Arduino/fragen die Clients den aktuellen Status und bekommt eine Antwort. Der Server bekommt eine Frage und schickt eine Antwort an dem Arduino oder an den Clients.*
* **updateState**;temperature=numeric\_value\_float;door=[open/close];
  + *Bei der* ***updateState*** *Methode schickt der Arduino/schicken die Clients zweimal Anfragen an den Server.*

Gültige Befehle in Phase (III)

* **Logout**

Wenn die Verbindung nicht aufgebaut ist, wird man vom System ausgeloggt.

* *Die Verbindung funktioniert nicht, wenn man ein falsches Passwort oder einen falschen Benutzernamen eingibt.*

**Die Verbindung mit dem Arduino**

Arduino (A) - Java Server (SA) Kommunikation

Initialisierungsphase (I)

1. SA wird gestartet. Ein ServerSocket wartet auf eine TCP/IP Verbindungsanfrage auf Port 2340.
2. A wird gestartet.
   1. TCP/IP Socket auf A verbindet sich mit SA auf Port 2340.
   2. Verbindung wurde nicht hergestellt: Gehe zu 2.a und sieh, ob das Port richtig ist.
   3. Verbindung wurde hergestellt: Gehe zu 3.
3. A sendet “login;username;password” an SA.
4. Benutzername und Passwort werden von SA nicht akzeptiert: Gehe zu 2.
5. Benutzername und Passwort werden von SA akzeptiert: Gehe zu 4.
6. A bekommt die Nachricht “Login correct” und hat jetzt Zugriff im System.

Operative Phase (II)

1. SA bekommt die Nachricht von den Clients und schickt es zum A.
2. A schickt den Wert an der Simulation des Hauses.
3. A fragt immer wieder über den Status des Hauses nach.

Abschlussphase (III)

1. Der Wert wird im DataWrapper-Klasse um im Datenbankserver gespeichert.
2. Der Status des Hauses wird geändert.

**Die Verbindung mit den Clients**

Die Clients (Cs) - Java Server (SA) Kommunikation

Initialisierungsphase (I)

1. Im Server wird eine TCP-Socket-Verbindung auf dem Port 2190.
2. Mit diesem Port sind die beiden Clients und der Server verbunden.
3. Cs werden gestartet.
4. Wenn die Verbindung nicht funktioniert: Kontrolliere, ob der Port richtig ist (Geh zu 1).
5. Wenn die Verbindung funktioniert: Geh zu 4.
6. Cs geben das Passwort ein.
7. Das eingegebene Passwort wird mit dem Passwort geprüft, der im Datenbankserver gespeichert ist.
8. Wenn das eingegebene Passwort gleich mit dem Passwort ist, der im Datenbankserver gespeichert ist, dann wird man im System einloggen.
9. Wenn das eingegebene Passwort nicht gleich mit dem Passwort ist, der im Datenbankserver gespeichert ist, dann wird man kein Zugriff im System haben.

Operative Phase (II)

1. Cs geben das Passwort ein.
2. Das eingegebene Passwort wird mit dem Passwort geprüft, der im Datenbankserver gespeichert ist.
3. Wenn das eingegebene Passwort gleich mit dem Passwort ist, der im Datenbankserver gespeichert ist, dann wird man im System einloggen.
4. Wenn das eingegebene Passwort nicht gleich mit dem Passwort ist, der im Datenbankserver gespeichert ist, dann wird man kein Zugriff im System haben.

Abschlussphase (III)

1. Der Server nimmt den Wert und speichert es im Datawrapper-Klasse.

Unten wird die Funktionalität der updateState- und getState-Methode erklärt:

updateState

*Clients:* “Schalt das Licht1 aus”

*Clients:* “Schalt das Licht2 ein”

*Server:* \*schaltet das Licht1 aus\*

*Server:* \*schaltet das Licht2 ein\*

getState

*Arduino/Clients:* “Was ist der Status des 1.Lichtes?”

*Server:* “aus”

*Clients:* “Was ist der Status des Garagentors”

*Server:* „geschlossen“

Unser Kommunikationsprotokoll beschreibt deutlich wie die Verbindung zwischen den Clients und dem Server funktioniert. Das Protokoll ist derart spezifiziert, dass der Server auf allen Nachrichten der Clients reagieren kann und umgekehrt.

**Nachrichten, die die Clients an den Server schicken können:**

* Login: [username]:[password]\n

Wenn der Client eine Login-Nachricht schickt, entscheidet der Server, ob dieser Nachricht akzeptiert werden kann oder nicht. Deshalb folgt er diese Prozedur. Wenn die Login-Nachricht akzeptiert wird, schickt der Server eine Bestätigung an dem Client zurück. Diese Bestätigung haben wir login\_successfulbenannt. Falls die Login-Nachricht von dem Server nicht anerkannt wird, z.B, wenn die Benutzername oder das Passwort falsch sind, schickt der Server eine Ablehnung an dem Client zurück. Diese negative Antwort heißt login\_unsuccesful. Wenn der Client einmal eingeloggt ist, schickt der Server eine Anweisung, dass dieser Client einmal eingeloggt ist (already\_logged\_in).

* Logout\n

Wenn der Client die Logout-Nachricht sendet, muss der Server nachschauen, ob dieser Client ausloggt oder nicht. D.h., der Client muss warten, bis der Server eine Antwort zurück verschickt.

Diese Antwort kann eine negative oder positive Beantwortung sein. Die Bestätigungsnachricht ist logout\_succesful und die Abneigung ist logout\_unsuccesful.

* Update:led1:0:led2:1:led3:1:led4:0:led5:1:vnt:180:posF1:90:posF2:0:posG:180:posT:90:hg:1:ka:0:vnt:255\n

Falls der Client erfolgreich eingeloggt ist, kann er zu dem Server zwei Nachrichten schicken. Eine Nachricht ist die UpdateState-Nachricht. Mit dieser Nachricht beschließt der Client, welche Aktion er führen will. Wenn der Server dieser Nachricht bekommt und auch akzeptiert (update\_accepted), speichert er die Werte in der DataWrapper-Klasse, damit die die aktuellen Werte zwischen den Clients und dem Arduino geteilt werden können, und in der Datenbank. Es kann auch passieren, dass der Server diese Meldung nicht annimmt. In diesem Fall schickt er ein update\_denied zurück.

* Get:led1:0:led2:1:led3:1:led4:0:led5:1:vnt:180:posF1:90:posF2:0:posG:180:posT:90:hg:1:ka:0:vnt:255\n

Der Client kann auch nach dem momentanen Status des Hauses fragen. Diesen Status bekommt er einfach, wenn er eine Get-Nachricht schickt. Falls der Server das Get akzeptiert, bekommt der Client ein get\_accepted Nachricht. Falls der Server das Get nicht annimmt, bekommt der Client eine get\_denied Nachricht.

Wenn eine Nachricht ein ungültiges Format hat, dann schickt der Server die Antwort invalid\_request zurück.

**Threads**

*Was sind eigentlich Threads?*

Die sequentielle Abarbeitung von Anweisungen eines Programms wird als ein Prozess oder Task bezeichnet. Für jeden Prozess ist im Betriebssystem ein eigener Speicherbereich reserviert. Für die Verwaltung der Prozesse ist das Betriebssystem.

Auch innerhalb eines Prozesses können eigene Abläufe existieren. Ein Prozess wird in sogenannte Threads untergliedert. Das Ziel der Threads ist, dass sie Abläufe parallelisieren um Anweisungen schneller abzuarbeiten oder verschiedene Aufgaben parallel erledigen zu können. Das bedeutet, dass mit Threads sich Multithreading fähige Programme schreiben lassen, bei denen Threads „parallel“ in einem Prozess ablaufen können.

Der Unterschied zwischen Prozesse und Threads ist, dass die einzelnen Threads keinen eigenen Speicherbereich. Sie teilen den Speicherbereich des jeweiligen Prozesses, zu dem sie gehören und für die Verwaltung von Threads ist nicht das Betriebssystem zuständig, sondern das Anwendungsprogramm selbst.

**JDBC**

Java Database Connectivity ist die Standard-Schnittstelle für den Zugriff auf relationale Datenbanken mittels SQL. Der Kern von JDBC besteht aus einer Sammlung von Klassen und Interfaces, die im Paket java.sql zusammengefasst sind. Damit können Datenbankverbindungen aufgebaut, beliebige SQL-Anweisungen an die Datenbank geschickt und Abfrageergebnisse im Programm verarbeitet werden. JBDC stellt also eine datenbankneutrale Zugriffsschnittstelle bereit und ist eine Abstraktionsschicht zwischen Java-Programmen und SQL.

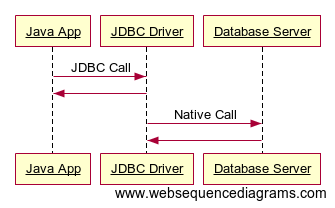


Abbildung 3: Verwendung von JDBC

**Synchronisation und Monitore**

Wenn in einem Programm mehrere Threads gestartet werden, kann es zur Situationen kommen, dass mehrere Threads auf die gleiche Ressource zugreifen möchten. Falls diese Situation passiert, kann es zu unerwünschten Ergebnissen kommen (Race Condition). In einem Programm können auch andere Fälle eintreten, wie z.B. Deadlock und Starvation. Race Condition passiert, wenn zwei oder mehrere Threads Zugriff auf bestimmten Daten haben und versuchen, diese Daten gleichzeitig aufzuteilen. Deadlock ist eine Situation, in dem zwei oder mehrere Threads blockiert sind und auf einander warten. Der Fall der Starvation tritt ein, wenn ein Thread keine CPU-Zeit mehr bekommt, weil ein anderes Thread diesen Letzen für sich selbst genommen hat.

Der Gebrauch nach Synchronisation wird aber nur für die Vermeidung von Race Conditions. In unserem Programm können nur Race Conditions passieren, weil der Thread für den Arduino und der Thread für die Clients haben Zugriff auf die gleichen Daten und versuchen sie gleichzeitig aufzuteilen. Die Deadlock-Situation kann nicht passieren, weil die Threads nie blockiert sind und müssen nicht auf einander warten.

Mit Hilfe vom Monitor-Konzept werden mehrere Threads miteinander synchronisiert. Mit einem Monitor wird ein kritischer Bereich für andere Threads gesperrt.

Um einen Monitor zu realisieren, wird das Schlüsselwort synchronized verwendet. Mit synchronized kann man entweder eine komplette Methode schützen oder einen Block innerhalb einer Methode.

**Server Programmierung**

Meine Aufgabe ist einen Server zu erstellen und programmieren, der für den Austausch von Nachrichten zwischen diesem Server und den Clients dient. Ich habe auch als Ziel, eine Verbindung zwischen dem Server und den Clients zu erstellen. Der Server muss mit einer Datenbank kommunizieren, in der gespeichert wird, alles, was die Clients zu dem Server schicken. Wir verwenden den Client/Server Modell, um eine bessere Verbindung zwischen den Komponenten zu haben.

**Was ist eigentlich ein Client/Server Modell?**

“Das Client/Server Modell ist vor allem ein Software-Architektur. Der Client formuliert Aufträge und schickt diese zur Ausführung an einen Server. Für jeden Server ist festgelegt, welche Arten von Diensten er anbietet.“

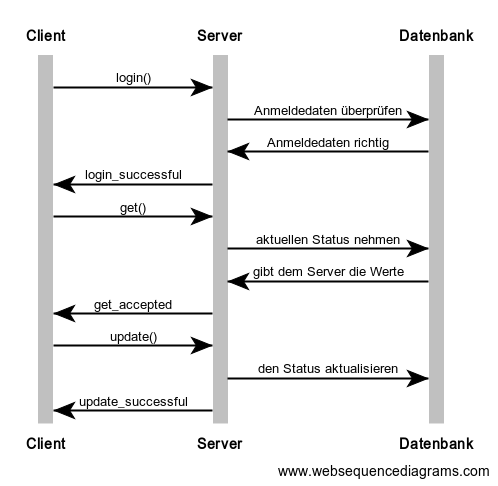


Abbildung 4: Die Kommunikation zwischen den Clients und dem Server

**Programmierungssprache und Entwicklungsumgebung**

Der Source Code dieses Servers wird mittels Programm NetBeans geschrieben. NetBeans ist ein Open Source Project, die komplett in der Programmiersprache Java geschrieben wurde und auch für diese Programmiersprache entwickelt wurde. Es wird auch als ein Werkzeug für Programmierer, um Programme zu schreiben. zu kompilieren, zu testen und bereitzustellen.

Für diese Programmierung wurde Java als Programmierungssprache verwendet. Diese Sprache ist sehr leicht zu verwenden und zu verstehen. Java ist eine Programmiersprache, die im Jahre 1995 von Firma Sun Microsystems veröffentlicht wurde und 2010 von Oracle gekauft wurde. Mit dieser Programmiersprache kann man Programme formulieren und festlegen.

**TCP/IP Sockets in Java**

Die Verbindung zwischen dem Server und den Clients wurde mit Hilfe eines Sockets ermöglicht worden. Dieses Socket ist ein sehr wichtiger Teil dieser Programmierung. Eine Socket ist eine Kombination zwischen IP-Adresse und Port-Nummer und stellt die einfache Verbindung zwischen einem Server und mehrere Clients im Netzwerk dar.

**Java Database Connectivity**

Mit dem Server ist auch eine Datenbank verbunden. In dieser Datenbank werden alle Werte und Zugriffe (Username, Passwort) gespeichert, die die Clients zu dem Server jedes Mal schicken. Da werden die Uhrzeit und das Datum für jeden Zugriff und Benutzer stehen. Der Server ist auch mit einem Arduino angebunden, der wieder das Haus steuern kann. Der Arduino fragt jede Sekunde für den Status des Hauses mittels einer Get-Nachricht.

Der Datenbankserver wurde mittels dem Programm MySQL Workbench erstellt. Die Verbindung zwischen dem Server und der Datenbank wurde mit Hilfe von Java Database Connectivity (JDBC) gemacht.

**Systemarchitektur und feines Design**

Um das System schöner und verstehbarer darzustellen, habe ich ein UML-Diagramm und ein ER-Diagramm erstellt. UML steht für Unified Modelling Language und diese Arten von Diagrammen werden für die visuelle Darstellung eines Softwareentwicklung-Codes verwendet. ER-Diagramm steht für Entity Relationship Diagram und dient für die Datenmodellierung eines Projektes der Informationssysteme.

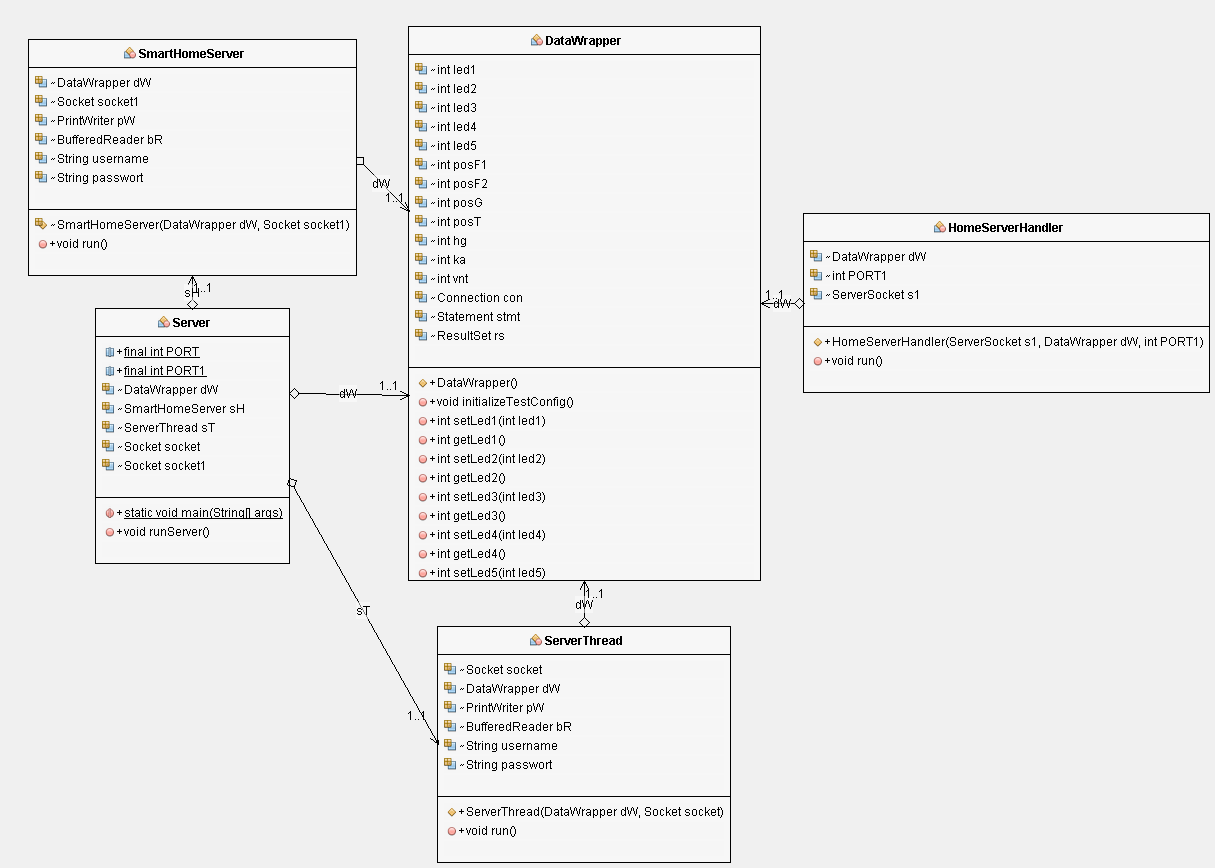


Abbildung 5: Das UML-Diagramm

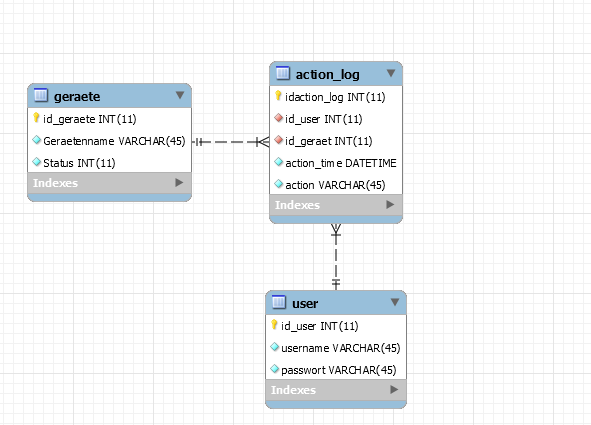


Abbildung 6: Das ER-Diagramm

**Implementierung**

Die Main-Klasse enthält den Einstiegspunkt für das Programm. Hier stehen die Ports des Servers, in denen alle Nachrichten der Clients kommen. Der Port für die Clients ist 50001, während das Port, wo der Server den Arduino zuhört ist 50004. Hier stehen auch die Variablen für die Sockets und anderen Klassen, wie die DataWrapper-Klasse, SmartHomeServer-Klasse und ServerThread-Klasse. Es gibt auch eine run-Methode, in der die Objekte des Sockets und der anderen Klassen stehen. Eine while-Schleife ermöglicht, dass der Socket der Web-Client oder SWING-Client auf dem Server akzeptiert wird. Der Thread, der für die Verbindung mit diesen beiden Clients zuständig ist, wird auch in der while-Schleife gestartet, wenn ein Client über einen Socket eine Verbindung aufbauen will. Um eine sicherere Verbindung zwischen dem Server und den Clients zu haben und zu beeinflussen, dass die Nachrichten nicht im Klartext getauscht werden, sondern in einem Hash gespeichert werden, habe ich SSLSocketFactory verwendet.

public class Server {

/\*\*

\* Hier werden die Ports geöffnet, die die Verbindung mit den Clients

\* ermöglichen 50001: für die SWING- und Web Clients

\*/

public static final int PORT = 50001;

public static final int PORT1 = 50004;

/\*\*

\* neue Instanzvariablen für jede Klasse und die Sockets erstellt

\*/

DataWrapper dW;

SmartHomeServer sH;

ServerThread sT;

Socket socket;

Socket socket1;

/\*\*

\* die Verbindugen mit den Threads

\*/

public void **runServer**() throws IOException {

SocketFactory newSF = SSLSocketFactory.getDefault();

dW = new DataWrapper();

ServerSocket s = new ServerSocket(PORT);

ServerSocket s1 = new ServerSocket(PORT1);

System.out.println("Der Server ist eingeschaltet und bereit für Verbindungen...");

HomeServerHandler hsh = new HomeServerHandler(s1, dW, PORT1);

while (true) {

Socket socket = s.accept();

new ServerThread(dW, socket).start();

}

}

}

Vom objektorientierten Standpunkt betrachtet, gibt es in Java eine Klasse mit dem Namen *Thread*. Um einen neuen Thread zu erzeugen, kann man eine Unterklasse der Klasse Thread ableiten. In dieser Klasse wird die Methode run() überschrieben werden und dann ein Exemplar für diese neue Klasse erzeugen. Wenn wir Threads mit den Prozessen vergleichen, verwenden die Threads keinen eigenen Speicherbereich. Threads teilen sich den Speicherbereich des jeweiligen Prozesses, zu dem sie gehören. Mit Threads lassen sich Multithreading fähige Programme schreiben, bei denen Threads „parallel“ in einem Prozess ablaufen können. Das ist auch das Ziel unserer Diplomarbeit; dass wir mehrere Clients haben, die in Form von Threads repräsentiert werden und gleichzeitig im Programm laufen können.

Die ServerThread-Klasse ist die Klasse, die mit dem Web-Client und dem SWING-Client verbunden ist. Diese Klasse leitet die Klasse Thread ab und dadurch auch alle Methoden dieser Klasse. Da stehen alle Methoden und Variablen, die mit diesen Clients zu tun haben.

public class **ServerThread** extends Thread {

Socket socket;

DataWrapper dW;

PrintWriter pW;

BufferedReader bR;

String username;

String passwort;

Hier werden Variablen für die Sockets, die DataWrapper-Klasse, Username und Passwort, PrintWriter und BufferedReader erstellt. Sie sind alle Instanzvariablen. Unter Instanzvariablen versteht man alle Variablen, die abhängig von einem Objekt existieren. Mit PrintWriter können Werte der einfachen Datentypen in einem Stream ausgegeben werden. Mit Hilfe dieser Methode kann man die Nachrichten, die für bestimmte Aktionen dienen, an den Clients verschicken. Im Gegensatz zu dem PrintWriter ist der BufferedReader nur ein Leser. Das heißt, er hat die typischen read()-Methoden. Intern aber, arbeitet der BufferedReader auf einem Speicherbereich, den er erst füllt, bevor seine read()-Methoden darauf arbeiten können. Der BufferedReader kann auf einmal größere Datenblöcke mit Hilfe einer readLine()-Methode lesen und somit verbessert sich die Lesegeschwindigkeit des Programms und der anderen Leser.

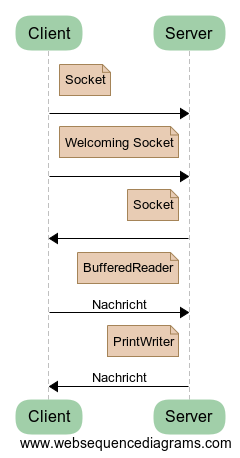


Abbildung 7: Skizze der Funktionalität des BufferedReaders und des PrintWriters

/\*\*

\*

\* @param socket, gibt das Socket

\*/

public **ServerThread**(DataWrapper dW, Socket socket) {

this.dW = dW;

this.socket = socket;

}

Das ist der Konstruktor mit den Parametern DataWrapper und Socket. Dieser Konstruktor wird als eine spezielle Methode verwendet, die automatisch beim Erzeugen eines Objektes ausgeführt wird. Die Aufgabe eines Konstruktors ist verschiedene Objekte zu erstellen und sie zu initialisieren. Da die Methode zwischen den beiden Variablen mit dem gleichen Namen unterscheiden muss, wird die Referenzvariable this verwendet.

/\*\*

\* Methode run(): der Thread ist im lauffähigen Zustand

\*/

@Override

public void **run**() {

try {

boolean login = false; //Variable für den Login-Prozess

boolean doWork = true; //Variable für den Ablauf des Prozesses

System.out.println(socket.toString());

pW = new PrintWriter(socket.getOutputStream());

bR = new BufferedReader(new InputStreamReader(socket.getInputStream()));

Die login-Variable stellt dar, ob die Clients eingeloggt sind, deshalb wird in einer boolean-Datentyp eingespeichert. Am Anfang sind keine Clients eingeloggt, daher ist mit dem Wert false gespeichert. Die doWork-Variable vertritt, ob die Clients Nachrichten zu dem Server schicken und Berichte von dem Server zurückbekommen. Der InputStreamReader bietet die Möglichkeit, InputStream-Klassen an die zeichenbasierten Reader-Klassen zu koppeln. Die OutputStream-Klasse ist eine abstrakte Klasse, die Zeichen oder Bytes ausgibt.

while (doWork) {

String m = bR.readLine(); //der Server kriegt eine Nachricht von den Clients

String[] split = m.split(":"); //die Nachrichten werden auf einem Array gespeichert

System.out.println("Der Client schickt jetzt eine Nachricht: " + m);

/\*\*

\* Die Clients möchten sich einloggen. Der Server entscheidet, ob er \* die übernommene Anfrage akzeptiert oder nicht.

\*/

if (split[0].matches(".\*login.\*"))

if (login == true) {

              pW.println("already\_logged\_in");

              pW.flush();

} else {

if (dW.isRegisteredUser(split[1], split[2]) == true) {

System.out.println("login test ausgabe: successful");

pW.println("login\_successful");

pW.flush();

login = true;

} else {

login = false;

pW.println("login\_unsuccessful");

pW.flush();

}

}

Die Clients öffnen die Webseite und möchten sich im System einloggen. Hier wird überprüft, ob die Clients erfolgreich anmelden können oder nicht. Bezüglich des Kommunikationprotokolls, wenn die Clients an den Server einloggen möchten, sollen sie einfach eine Login-Nachricht schicken. Wenn der Server die Nachricht „login“ bekommt, dann schaut der Server, ob die Clients den richtigen Username und das richtige Passwort geschickt haben. Der Username und dessen Passwort werden in der Tabelle „user“ in der Datenbank gespeichert werden. Wenn die login-Variable gleich true ist (d.h., dass schon existiert), dann schickt der Server already\_logged\_in zurück. Die Methode isRegisteredUser schaut nach, welchen Benutzername und welches Passwort der Client hat.

Wenn der Client die richtigen Aufforderungen im System eingegeben hat, dann schickt der Server login\_successful an den Clients zurück. Falls die Eingaben der Clients falsch sind, dann gibt der Server login\_unsuccessful zurück.

/\*\*

\* Die Clients möchten den Status des Hauses ändern. Der Server

\* entscheidet, ob er die übernommene Anfrage akzeptieren kann oder nicht.

\*/

else if (split[0].matches(".\*Update.\*")) {

if (login) {

pW.println("update\_accepted");

pW.flush();

dW.updateState(bR.readLine());

} else {

pW.println("update\_denied");

pW.flush();

doWork = false;

}

}

Im unseren System gibt es eine Methode, die die Clients hilft, um die Werte der Bauteile und Geräte im Haus zu ändern. Nachdem die Clients im System eingeloggt sind, schicken sie eine Update-Nachricht und danach entscheidet der Server, ob diese Nachricht angenommen werden kann oder nicht. Falls der Server die Nachricht akzeptiert, schickt er update\_accepted an die Clients. Aber, wenn das Gegenteil der vorherigen Situation passiert, sendet der Server update\_denied. D.h., es gibt einen Fehler mit der Verbindung zwischen den Clients und dem Server oder die Clients haben einen Fehler mit der Einstellung der Werte gemacht.

/\*\*

\* Die Clients möchten den aktuellen Status des Haues wissen. Der Server

\* entscheidet, ob er die übernommene Anfrage akzeptieren kann oder nicht.

\*/

else if (m.matches(".\*Get.\*")) {

  if (login) {

    pW.println("get\_accepted");

    pW.flush();

/\*\*

\* falls die Get-Nachricht akzeptiert wird, werden neue Variablen für

\* jedes Bauteil erstellt

\*/

    String strled1;

    String strled2;

String strled3

String strled4;

String strled5;

String strposF1;

String strposF2;

String strposG;

String strposT;

………………………………………

Wenn der Client eine Get-Nachricht schickt, kontrolliert der Server mit der Methode matches(), ob er diese Nachricht annehmen kann oder nicht. Die Nachricht, die die Clients zuerst schicken, muss gleich „Get“ sein. Wenn die Clients einen kleinen Tippfehler machen, wird die Eingabe vom Server nicht akzeptiert. Falls die Get-Anweisung eine Zusagung bekommt, dann werden verschiedene Variablen für jedes Bauteil erstellt werden.

/\*\*

\* Überprüfung der Werte der Bauteile

\*/

if (dW.getLed1() == 1) {

strled1 = "led1:1";

} else {

     strled1 = "led1:0";

     }

if (dW.getLed2() == 1) {

strled2 = "led2:1";

} else {

strled2 = "led2:0";

}

if (dW.getPosF1() == 90) {

strposF1 = "posF1:90";

} else if (dW.getPosF1() == 180) {

strposF1 = "posF1:180";

} else {

strposF1 = "posF1:0";

}

……………………………………………………………………

if (dW.getVentilator() == 180) {

strvnt = "vnt:180";

} else if (dW.getVentilator() == 255) {

strvnt = "vnt:255";

} else {

strvnt = "vnt:0";

}

Hier werden if-Anweisungen für jedes Bauteil des Hauses erstellt. Diese Anweisungen kontrollieren die Werte, die die Clients eingeben und an den Server schicken. Z.B. schaltet der Client das Licht 1 ein, bekommt der Server den Wert 1. Falls der Client keinen Wert eingibt, ist das Licht automatisch ausgeschaltet. Das gilt für die anderen Lichter und alle Bauteile, die nur zwei Werte (Garagentor, Tür, Heizungsgerät, Klimaanlage) haben können. Die Bauteile, die mehr als zwei Werte haben können (Fensterrollos, Ventilator), muss man anders einrichten. Die Clients müssen einen angemessenen Wert eingeben, damit der Server unterscheiden kann, ob das Bauteil eingeschaltet, normal oder ausgeschaltet sein soll.

/\*\*

\* die Werte der Bauteile speichern

\*/

String r = strled1 + ":" + strled2 + ":" + strled3 + ":" + strled4 + ":" + strled5 + ":" + strposF1 + ":" + strposF2 + ":" + strposG + ":" + strposT + ":" + strhg + ":" + strka + ":" + strvnt;

System.out.println(r);

pW.println(r);

pW.flush();

Hier wird eine Variable mit dem Namen „r“ (Abkürzung für Result) und dem Datentyp String initialisiert. In dieser Variable wird der aktuelle Status des Hauses gespeichert; d.h., hier finden die Clients die Werte, die sie vorher schon eingegeben haben. Um zu überprüfen, ob der Status des Hauses in Ordnung ist, wird er im Bildschirm des Servers ausgegeben werden.

„pW“ steht für PrintWriter und verwendet die Methode println(), um die Nachricht zu den Clients schicken zu können. Mit der Methode flush() macht man sicher, dass die Nachricht, die gesendet wurde, wirklich das Ziel erreicht hat.

// die Get-Nachricht wird nicht akzeptiert

} else {

pW.println("get\_denied");

pW.flush();

}

Hier wird eine else-Anweisung dargestellt. Wenn die Get-Anfrage der Clients vom Server nicht akzeptiert wird, dann bekommen die Clients get\_denied als Antwort zurück.

/\*\*

\* Die Clients möchten sich ausloggen. Die Clients werden erfolgreich

\* abgemeldet

\*/

} else if (split[0].matches(".\*logout.\*")) {

if (login) {

pW.println("logout\_successful");

System.out.println("Der Client wurde erfolgreich ausgeloggt");

login = false;

Nachdem die Clients mit dem Einstellen der Werte im Haus fertig sind, sollen sie sich ausloggen. Zuerst senden sie eine logout-Anfrage und der Server kontrolliert, ob diese Anfrage in Ordnung ist. Falls die logout-Befragung richtig ist, schickt der Server an den Clients logout\_successful zurück. Das bedeutet, dass die Clients erfolgreich ausgeloggt sind. Auf dem Bildschirm des Servers wird „Der Client wurde erfolgreich ausgeloggt“ angezeigt werden. Jetzt wird die login-Variable um gleich falseumgesetzt.

/\*\*

\* Die Clients können sich nicht abmelden. Ein Fehler ist aufgetreten

\*/

} else {

pW.println("logout\_unsuccessful");

}

Wenn der logout-Prozess der Clients nicht richtig funktioniert, dann kriegen die Clients logout\_unsuccessful ab.

pW.flush();

doWork = false;

}

Es wird mit der Methode flush() sichergestellt, dass die Clients die Nachricht des Servers bekommen und es wird sich nicht im Netzwerkverkehr verspäten oder verloren gehen. Der ganze Prozess ist jetzt abgeschlossen. Die Clients können nun nichts machen, ohne sich noch einmal anzumelden.

/\*\*

\* Die Clients haben eine fehlerhafte Anfrage geschickt.

\*/

else {

pW.println("invalid\_request");

pW.flush();

}

}

Falls die Clients eine sinnlose Anfrage an den Server schicken, bekommen die Clients invalid\_request als Meldung zurück.

pW.close();

bR.close();

socket.close();

Hier werden die Objekte von PrintWriter, BufferedReader und die Socket geschlossen. Die Verbindung zwischen dem Server und den Clients ist nun zu einem Ende gekommen.

} catch (IOException ex) {

Logger.getLogger(ServerThread.class.getName()).log(Level.SEVERE, null, ex);

} catch (SQLException ex) {

//weiteres Exception Handling

}

}

}

Falls es Input-, Output-, und SQL-Exceptions passieren, existiert eine Möglichkeit sie abzufangen. Diese Möglichkeit habe ich hier verwendet. Dieses Verfahren hilft dazu, dass Informationen über bestimmte Programmzustände an andere Programmebenen zur Weiterbehandlung weitergereicht werden können.

Die SmartHomeServer-Klasse ist für die Verbindung zwischen dem Server und dem Arduino verantwortlich. Da werden fast alle Methoden und Variablen wie in der ServerThread-Klasse, aber es gibt schon ein paar Änderungen. Diese Klasse leitet die Unterklasse Thread auch. Hier stehen alle Methoden und Variable, die mit dem Arduino zu tun haben.

public class **SmartHomeServer** extends Thread {

DataWrapper dW;

Socket socket1;

PrintWriter pW;

BufferedReader bR;

String username;

String passwort;

Diese Klasse hat die gleichen Instanzvariablen wie die ServerThread-Klasse. Der Unterschied ist, dass die Name der Instanzvariable Socket ist anders, sodass die Namen zu keinen Kollisionen führen.

/\*\*

\*

\* @param socket1, gibt das Socket

\*/

**SmartHomeServer**(DataWrapper dW, Socket socket1) throws IOException {

this.dW = dW;

this.socket1 = socket1;

}

Das ist der Konstruktor mit den Instanzvariablen DataWrapper und Socket als Parameter. Er wirft einen Input/Output-Parameter, damit es leichter ist, dass keine Fehler passieren, die der Programm nicht mehr durchschauen und herausfinden kann. Der Name des Konstruktors ist festgelegt und ist gleich dem Namen der Klasse, in der er festgelegt ist; in diesem Fall ist der Name ServerThread. Die Deklaration des Konstruktors ähnelt der einer Methode, aber der Unterschied ist, dass der Konstruktor keine Typangabe für einen Rückgabetyp hat. Der Konstruktor hat einen bestimmten Speicherplatz für zwei zu erzeugenden Instanzen, besser mit dem Namen Objekt bekannt, reserviert.

/\*\*

\* Die Clients möchten sich einloggen. Der Server entscheidet, ob er

\* die übernommene Anfrage akzeptiert oder nicht.

\*/

public void **run**() {

try {

System.out.println("hsh instance started");

boolean login = false;

boolean doWork = true;

System.out.println(socket1.toString());

pW = new PrintWriter(socket1.getOutputStream(), true);

bR = new BufferedReader(new InputStreamReader(socket1.getInputStream()));

Hier wird die Methode run() dargestellt. Wenn die Methode run() im Programm gestartet ist, dann ist der Thread im lauffähigen Zustand.

Es wird eine try-catch Exception-Handling erstellt. Hier wird überprüft, ob der Socket richtig läuft. Es werden neue Instanzen für den PrintWriter und den BufferedReader erzeugt. Am Anfang ist kein Arduino im System angemeldet, deshalb ist die login-Variable gleich false.

/\*\*

\* while-Schleife für den ganzen Prozess

\*/

while (doWork) {

String m1 = bR.readLine();

String[] split1 = m1.split(":");

System.out.println("Das Haus schickt jetzt eine Nachricht: " + m1);

Hier wird eine while-Schleife erstellt. Ein Array wird erzeugt und teilt dadurch die Nachricht nach Positionen. Wenn die Schleife beginnt, bekommt der Server eine Nachricht von dem Arduino und diese Nachricht wird im Bildschirm des Servers ausgegeben werden.

/\*\*

\* Die Clients möchten den aktuellen Status des Haues wissen. Der Server

\* entscheidet, ob er die übernommene Anfrage akzeptieren kann oder nicht.

\*/

if (split1[0].matches(".\*login.\*")) {

if (login == true) {

pW.println("already\_logged\_in");

pW.flush();

doWork = false;

} else {

if (split1[1].equals("Client") && split1[2].equals("1234")) {

System.out.println("login test ausgabe: successful");

pW.println("login\_successful");

pW.flush();

login = true;

} else {

pW.println("login\_unsuccessful");

pW.flush();

doWork = false;

}

}

}

Falls der Arduino einloggen will, muss er den Server fragen, ob die Befragung erfolgreich ist. Wenn die Anfrage des Clients vom Server akzeptiert ist, dann bekommt der Arduino login\_succesful zurück und die login-Variable wird auf true gesetzt. Wenn der Prozess einen anderen Weg nimmt, sendet der Server login\_unsuccessful zu dem Arduino.

/\*\*

\* Die Clients möchten den aktuellen Status des Haues wissen. Der Server

\* entscheidet, ob er die übernommene Anfrage akzeptieren kann oder nicht.

\*/

else if (m1.matches(".\*Get.\*")) {

if (login) {

pW.println("get\_accepted");

pW.flush();

Thread.sleep(100);

pW.println(dW.getLed1());

pW.flush();

Thread.sleep(100);

pW.println(dW.getLed2());

pW.flush();

Thread.sleep(100);

pW.println(dW.getPosF1());

pW.flush();

Thread.sleep(100);

}

else {

pW.println("get\_denied");

pW.flush();

}

Nachdem der Arduino im System angemeldet ist, fragt er immer nach dem Status des Hauses. Diese Anfrage wird durch die Get-Nachricht ermöglicht werden. Der Arduino schreibt „Get“ in seiner Eingabe ein und dann kann er einfach die Enter-Taste drücken. Falls diese Nachricht von dem Server bestätigt wird, kriegt der Arduino get\_accepted als Ausgabe zurück. Der Server stellt sicher, dass dieser Bescheid an dem Arduino geschickt wird.

Nach diesem Prozess, kann der Arduino den Status des Hauses gleichsehen. Die Werte der Bauteile werden von der DataWrapper-Klasse mit einer get-Methode genommen. Der Thread läuft für 100 Millisekunden (1 Sekunde) nicht, d.h., er ist in einem Sleep-Modus. Der Arduino akzeptiert die Split-Methoden nicht, deshalb wird der Thread für eine bestimmte Zeit nicht laufen. Das hat den Effekt, dass die Daten und Werte nicht miteinander gemischt werden und einander nicht verhindern, am Ziel zu kommen.

/\*\*

\* Die Clients möchten sich ausloggen. Die Clients werden erfolgreich

\* abgemeldet

\*/

} else if (split1[0].matches(".\*logout.\*")) {

if (login) {

pW.println("logout\_successful");

System.out.println("Der Client wurde erfolgreich ausgeloggt");

login = false;

/\*\*

\* Die Clients können sich nicht abmelden. Ein Fehler ist aufgetreten

\*/

} else {

pW.println("logout\_unsuccessful");

}

pW.flush();

doWork = false;

}

else {

pW.println("invalid\_request");

pW.flush();

}

}

Der Arduino muss am Ende des Prozesses sich ausloggen. Zuerst sendet er eine logout-Anfrage und der Server kontrolliert, ob diese Anfrage in Ordnung ist. Falls die logout-Befragung richtig ist, schickt der Server an dem Arduino logout\_successful zurück. Das bedeutet, dass der Arduino erfolgreich ausgeloggt ist. Auf dem Bildschirm des Servers wird „Der Client wurde erfolgreich ausgeloggt“ angezeigt werden. Jetzt wird die login-Variable um gleich falseumgesetzt. Wenn der logout-Prozess der Clients nicht richtig funktioniert, dann kriegen die Clients logout\_unsuccessful ab.

Es wird mit der Methode flush() sichergestellt, dass der Arduino die Nachricht des Servers bekommt und es wird sich nicht im Netzwerkverkehr verspäten oder verloren gehen. Der ganze Prozess ist jetzt abgeschlossen. Der Arduino kann nun nichts machen, ohne sich noch einmal anzumelden.

pW.close();

bR.close();

socket1.close();

} catch (IOException e) {

System.out.println(e);

} catch (InterruptedException ex) {

//weiteres Exception Handling

}

}

}

Hier werden die Objekte von PrintWriter, BufferedReader und die Socket geschlossen. Die Verbindung zwischen dem Server und den Clients ist nun zu einem Ende gekommen.

Falls der Programm Exceptions kriegen wird, werden sie mit einem catch() abgefangen werden. Das hilft, dass der Programm fehlerfrei läuft.

Die DataWrapper-Klasse ist ein sehr wichtiger Teil unseres Programmes. Da steht die Verbindung mit der Datenbank und alle Variablen mit den richtigen Namen für die Bauteile und Geräte des Hauses. Diese Klasse enthält Getter- und Setter-Methoden für jedes Bauteil, damit jede von denen einen bestimmten Wert nimmt. Diese Klasse hilft dem Server, dass er einen schnellen Zugriff in der Datenbank hat, wenn die Clients eine Anfrage an ihn schicken.

public class **DataWrapper** {

int led1;

int led2;

int posF1;

int posF2;

int posG;

int posT;

int ka;

int vnt;

…………………………………………

Connection con;

Statement stmt;

ResultSet rs;

Diese sind die Variablen für jedes Bauteil, die als globale Variablen deklariert werden. Die Variablen con, stmt und rs sind Instanzvariablen, die für die Verbindung mit der Datenbank dienen.

Class.forName("com.mysql.jdbc.Driver");

con = (Connection) DriverManager.getConnection("jdbc:mysql://localhost:3306/hsh","root","");

//hier HSH ist die Datenbank, root ist Username

System.out.println("Mit der Datenbank verbunden...");

}

Hier wird die Verbindung mit der Datenbank erstellt. Der JDBC-Driver ist das Tool, das diese Kommunikation möglich macht. Der Driver-Manager nimmt die verfügbare Verbindung der Datenbank, mit dem Namen und der Port Nummer, wie ich sie konfiguriert habe. Wenn die Verbindung aufgebaut ist und erfolgreich funktioniert, wird die Nachricht „Mit der Datenbank verbunden“ im Bildschirm des Servers angezeigt werden.

catch(Exception e){ System.out.println(e);

}

initializeTestConfig();

}

Damit keine Fehler passieren können, werden hier die Exceptions mit einem catch-Block angefangen. Die Test-Methode initializeTestConfig() wird hier gestartet, um zu überprüfen, ob der Status des Hauses richtig aktualisiert wird.

// Test-Methode für die Werte der Bauteile

public void **initializeTestConfig**() {

led1=0;

led2=0;

led3=1;

led4=1;

led5=0;

posF1=1;

posF2=0;

posG=1;

posT=0;

hg=1;

ka=0;

vnt=1;

}

Die Test-Methode initalizeConfig() enthält Testwerten und sie haben keinen Effekt in dem System, wenn die Clients diese Werte aktualisieren.

/\*\*

\*

\* @param led1, das Lichh1 einschalten oder ausschalten

\* @return den Wert von dem Licht (on/off)

\*/

public synchronized int **setLed1**(int led1){

return this.led1 = led1;

}

/\*\*

\*

\* @param led1, nimmt das Wert von dem Licht

\* @return den Wert von dem Licht

\*/

public synchronized int **getLed1**(){

return led1;

}

………………………………………………………………………

/\*\*

\*

\* @param vnt, den Ventilator einschalten oder ausschalten

\* @return den Wert des Ventilators (on/off)

\*/

public synchronized int **setVentilator**(int vnt){

return this.vnt = vnt;

}

/\*\*

\* @param vnt, nimmt den Wert von dem Ventilator

\* @return den Wert des Ventilators

\*/

public synchronized int **getVentilator**(){

return vnt;

}

Hier werden Getter- und Setter-Methoden, auch als Zugriffsmethoden bekannt, für jedes Bauteil des Hauses erstellt. Die Getter-Methoden, oder die Abfragemethoden, werden verwendet, um die Eigenschaft eines Objekts abzufragen. Hier fragen sie nach der Werte der Bauteile.

Die Setter-Methoden sind auch als Änderungsmethoden bekannt. Diese Methoden warten bis die Clients das Befehl „Update“ schicken, um den Status des Hauses zu ändern oder zu aktualisieren. Ein Vorteil der Änderungsmethode besteht darin, dass sie vor der Änderung den Wert auf Gültigkeit prüfen kann.

/\*\*

\* Diese Methode macht die Anmeldung der Clients möglich.

\*/

public synchronized boolean **isRegisteredUser**(String name, String password) throws SQLException {

//die Benutzernamen und die Passwörter werden in der Datenbank gespeichert

try {

System.out.println("Username: " + name);

System.out.println("Password: " + password);

String sql = "SELECT id\_user FROM user WHERE username ='"+ name + "' and passwort = '"+ password +"';";

Statement stmt = (Statement) con.createStatement();

ResultSet rs = stmt.executeQuery(sql);

if (rs.next()) {

System.out.println("Login erfolgreich");

return true;

}

else {

System.out.println("Error");

return false;

}

}

catch (NullPointerException e) {

System.out.println("Exception thrown");

return false;

}

catch (SQLException p){

}

}

Das ist die Methode isRegisteredUser. Sie ist zuständig nur für das Einloggen des Clients zuständig. Es kann passieren, dass Probleme oder Fehler mit der Interaktion der Daten in der Datenbank geben kann. Damit dieses Problem vermieden werden kann, wirft diese Methode einen SQL-Exception. Wenn die Clients sich im System einloggen, wird der Benutzername und das Passwort, die er verwendet hat, mit der Anmeldedaten, die sich in der Datenbank befinden, verglichen. Falls die Anmeldedaten zusammenpassen, werden die Clients sich erfolgreich im System einloggen. Falls das Prozess schiefläuft, wird das Programm einen Error ausgeben. Hier ist ein Objekt erforderlich, aber es kann auch ein NullPointerException passieren. Dieses Exception wird innerhalb dieser Methode abgefangen.

/\*\*

\* Diese Methode ändert den Status des Hauses, wenn die Clients das

\* erfordern

\*/

public boolean **updateState**(String m) throws SQLException {

System.out.println(m);

String[] s = m.split(":");

//String zum Integer umwandeln, damit der Arduino die Werte einlesen kann

int led1 = Integer.parseInt(s[1]);

int led2 = Integer.parseInt(s[3]);

int led3 = Integer.parseInt(s[5]);

int led4 = Integer.parseInt(s[7]);

int led5 = Integer.parseInt(s[9]);

int posF1 = Integer.parseInt(s[11]);

int posF2 = Integer.parseInt(s[13]);

int posG = Integer.parseInt(s[15]);

int posT = Integer.parseInt(s[17]);

int hg = Integer.parseInt(s[19]);

int ka = Integer.parseInt(s[21]);

int vnt = Integer.parseInt(s[23]);

Die updateState-Methode ändert oder aktualisiert den Status des Hauses, wenn die Clients diese Aktion erfordern. Die Nachricht, die der Server von den Clients bekommt, wird in Positionen in einem Array gespeichert und aufgeteilt. In einer Position des Arrays kommt der Name des Bauteils und in der anderen Position neben dem Namen steht der Wert oder der Status des Bauteils. Die Bauteile werden von dem Datentyp String zu einem Integer umgewandelt, damit im Arduino kein Fehler bei dem Speichern und dem Einlesen von Daten passiert.

setLed1(led1);

setLed2(led2);

setLed3(led3);

setLed4(led4);

setLed5(led5);

setPosF1(posF1);

setPosF2(posF2);

setPosG(posG);

……………………………………………………………………

Die Änderungsmethoden der Bauteile werden aufgerufen und der Status des Hauses wird aktualisiert, wie es der Server erfordert hat.

try {

String sql = "INSERT INTO action\_log (Geraetenname, Status) " + "VALUES(?, ?)";

stmt = (Statement) con.createStatement();

int result = stmt.executeUpdate(sql);

}

catch (NullPointerException e) {

System.out.println("Error");

return false;

}

return true;

}

Die geänderten Daten der Geräte werden in der Datenbank hineingefügt und da drinnen in einer Tabelle mit dem Namen „Geraete“ gespeichert. In dieser Tabelle stehen der Geräten Name und der Status als Attribute. In dem Attribut mit dem Namen **Geraetenname** wird der Name jenes Gerät gespeichert, dessen Wert geändert wurde. In dem Attribut **Status** steht der aktuelle Zustand des Gerätes.

/\*\*

\* Die Clients bekommen den aktuellen Status des Hauses

\*/

public boolean **getState**(String m) throws SQLException {

String[] s1 = m.split(":");

stmt = (Statement) con.createStatement();

stmt.executeQuery("SELECT \* FROM geraete ");

return true;

}

Mit Hilfe der getState-Methode bekommen die Clients den aktuellen Status des Hauses. Wenn die Clients eine Get-Nachricht schicken, soll der Server die Werte aus der Datenbank nehmen und danach an die Clients schicken.

public boolean **logoutUser**(String a) throws SQLException {

stmt = (Statement) con.createStatement();

stmt.executeQuery("SELECT username, password FROM user");

return true;

}

}

Die Clients wollen am Ende des Prozesses sich abmelden. Die logoutUser-Methode hilft, damit die Abmeldung der Clients fehlerfrei erfolgt. Der Server schaut in der Datenbank nach, wer sich ausloggen will und kontrolliert, ob die Abmeldung in Ordnung passieren kann.

Die Klasse HomeServerHandler ist ein Unterthread. Das bedeutet, dass sie ein Thread innerhalb eines anderen Threads ist. Die Clients und der Arduino haben Zugriff auf den gleichen Daten und versuchen sie gleichzeitig aufzuteilen. Hier kann das Problem von Race Conditions auftreten. Wenn der Arduino versucht hat, nach den Clients im System einzuloggen, musste er warten, bis die Clients sich abgemeldet haben. Das musste in unserem System nicht passieren. Aus diesem Grund habe ich die HomeServerHandler-Klasse erstellt. Diese Klasse enthält den Socket, der für die Verbindung mit dem Arduino zuständig ist. Der Prozess ist gleich wie es in der Main-Klasse mit dem Socket der Clients. Dieser Thread wird in der Main-Klasse implementiert und gestartet.

public class **HomeServerHandler** extends Thread {

DataWrapper dW;

int PORT1;

ServerSocket s1;

public **HomeServerHandler**(ServerSocket s1, DataWrapper dW, int PORT1) {

this.dW = dW;

this.PORT1 = PORT1;

this.s1 = s1;

}

public void **run**() {

while(true){

Socket socket1;

try {

socket1 = s1.accept();

new SmartHomeServer(dW,socket1).start();

} catch (IOException ex) {

Logger.getLogger(HomeServerHandler.class.getName()).log(Level.SEVERE, null, ex);

}

}

}

}