

Informatikprojekt „Smart Home“



Von: Marvin Herhaus
Matrikelnummer: 11132049
Betreuer: Alexander Dobrynin

5.Semester
Allgemeine Informatik

Technology
Arts Sciences
TH Köln

Inhaltsverzeichnis

Das Ziel	3
Warum möchte ich das machen? Und der Weg zur Idee	3
Grobe Vorgehensweise	3
Verwendete Technoligen und verwendete Hardware.....	3
<i>Raspberry Pi:</i>	3
<i>ZigBee:</i>	4
<i>ZigBee Stick:</i>	4
<i>Message Queuing Telemetry Transport (MQTT):</i>	4
<i>ZigBee2Mqtt:</i>	5
<i>iBeacon:</i>	5
Detaillierte Vorgehensweise.....	5
1. <i>Installation von Zigbee2Mqtt</i>	5
2. <i>Verbindung zur Birne</i>	6
3. <i>App mit MQTT Befehlen</i>	6
4. <i>iBeacon Signal vom Raspberry Pi erzeugen</i>	8
5. <i>App mit dem iBeacon Protokoll</i>	9
6. <i>Zusammenführung der beiden Komponenten</i>	11
Probleme	11
Fazit	12
Nachbearbeitung.....	12
Links.....	13
Quellen	14

Das Ziel

Das Ziel ist es eine App für iOS zu programmieren, die eine Birne anschalten kann. Zusätzlich soll die Birne gedimmt werden können. Dazu soll nicht der Touchscreen verwendet werden, sondern das soll interaktiv geschehen. Wenn der User das Handy im Raum bewegt, sollen sich die Werte anpassen. Die Entfernung soll mit einem Beacon gemessen werden.

Warum möchte ich das machen? Und der Weg zur Idee

Im letzten Jahr habe ich angefangen, mich mit Android Studio zu beschäftigen und weil mir das gefallen hat, wollte ich gerne eine App programmieren.

Herr Dobrynin bietet ein WPF für iOS an und weil ich auch iPhone User bin, dachte ich mir, dass es ja auch eine iOS App sein kann.

Das Gebiet (Programm und Programmiersprache, Xcode und Swift) war mir völlig neu, somit startete ich bei null und hatte auch noch keine endgültige Idee.

In Absprache startete ich mit einem Video Kurs auf YouTube (Q13), der mir die Grundlagen beibrachte.

Weil ich mich auch sehr für das Thema Smart Home interessiere wollte, ich gerne eine App entwickeln, die meine Lampen ansteuert. Ich dachte erst über WLAN nach, aber besser ist Zigbee.

Die Idee mit dem iBeacon für die Helligkeit kam dann von Herrn Dobrynin. Er zeigte mir dann auch die Technologien, mit denen ich mein Projekt realisieren kann. Nachdem ich mich damit beschäftigt hatte, startete die Recherche zur Umsetzung.

Grobe Vorgehensweise

1. Lernvideos für die Grundlagen
2. Installation von Zigbee2Mqtt, erst auf dem Raspberry Pi dann auf dem Mac
3. Verbindung zur Hue-Birne
4. App mit Mqtt-Befehlen
5. iBeacon Signal vom Raspberry Pi erzeugen
6. App mit iBeacon Funktionalität
7. Zusammenführung der beiden Komponenten

Verwendete Technoligen und verwendete Hardware

Raspberry Pi:

Der Raspberry Pi ist ein Computer mit einer Platine und einem Chip. Er ist kostengünstig und damit gut für Bastler und Einsteiger in die Programmierwelt geeignet. Ursprünglich sollte auf dem Raspberry Pi der MQTT Broker laufen, der Befehle von der App empfängt und über den Zigbee Stick dann an die Birne weiterleitet. Aufgrund von Schwierigkeiten muss der Broker allerdings auf dem MacBook laufen und der Raspberry Pi fungiert nur als Beacon, indem er regelmäßig ein Signal sendet. Das macht das eingebaute Bluetooth Modul möglich.

Xcode ist die Entwicklungsumgebung für die iOS-App, die in Swift geschrieben ist.

ZigBee:

Zigbee ist ein Kommunikationsprotokoll, vergleichbar mit Bluetooth und WLAN, ein Funkstandard, den Geräte zur Kommunikation nutzen. Er wird vor allem in der Smart Home Branche verwendet, wenn es darum geht Sensoren oder Lampen anzusteuern. Der Vorteil von Zigbee ist seine Energieeffizienz. Da Handys oder ähnliche Geräte diesen Funkstandard nicht unterstützen, wird eine Bridge/ Hub oder Gateway benötigt, die ein Signal über WLAN oder Bluetooth empfangen, und dann ein Zigbee Signal weiterleiten.

„Das Funksignal wird in den lizenzfreien ISM-Bändern mit 868 MHz, 915 MHz und 2,4 GHz übertragen.“(Q2) Die Datenübertragungsrate ist mit 250 Kilobit pro Sekunde geringer als beim aktuellen WLAN Standard, reicht für die Anwendungszwecke aber aus.

„Die Übertragung des Funkprotokolls ist gemäß AES-CCM-Algorithmus mit 128 Bit verschlüsselt. Damit ist ZigBee abhörsicher.“ (Q1)

Zigbee kann auch auf ein Mesh System zurückgreifen, indem alle Geräte miteinander verbunden sind und jedes Signal an alle Geräte weiterleiten. Damit lässt sich eine größere Reichweite als 10-20 Meter erreichen, je nach Anzahl der Geräte.

Das Zigbee Protokoll weist vier Layer auf.

1. Der Physical Layer zum Senden und Empfangen.
2. Der Medium Acces Control Layer, der den Medienzugriff steuert und Netzwerkprotokolle implementiert hat.
3. Der Network Layer, für die Kommunikation zwischen Netzwerken.
4. Der Application Layer, die Anwendungsschicht. Letztere ist auch anpassbar.

ZigBee Stick:

In meinem Fall habe ich den ITSTUFF CC2531 ZigBee USB-Stick mit Antenne verwendet, dieser hat eine gute Reichweite und wurde mit Firmware geliefert, somit brauchte ich keine weitere Hardware zum flashen der Firmware. Dieser Stick wird benötigt, um das Zigbee Signal zu senden.

Message Queuing Telemetry Transport (MQTT):

MQTT ist ein Netzwerkprotokoll für Machine-to-Machine Kommunikation. Es ist das aktuell bekannteste Kommunikationsprotokoll für das Internet of Things.

Eine Eigenschaft, dass es für ressourcenarme Geräte ist, weil es ein sehr schlankes Protokoll hat (Q4). Auch dieses Protokoll gehört zum Standard in der Heimautomatisierung.

MQTT implementiert das Publish-Subscribe Pattern. Dabei werden Nachrichten mit einem *Topic* versehen. Um Nachrichten mit diesem Topic zu erhalten, muss der Empfänger sie abonnieren. Die Kommunikation findet über einen MQTT Message Broker statt. Clients senden mit Topic und Clients empfangen mit Topic. Dadurch kennt der Sender den Empfänger nicht. Im Idealfall ist die Kommunikation zum Server immer gewährleistet (über TCP), ist dies nicht der Fall, werden Nachrichten aber auch gespeichert und später zugestellt. Dadurch fungiert der Broker auch als Zustands-Datenbank.

Es wird auf den Port 1883 oder 8883 zurückgegriffen.

Zum Start benötigt man Broker und eine Client Bibliothek, mir wurde Mosquitto empfohlen, das habe ich dann auch benutzt.

ZigBee2Mqtt:

Zigbee2Mqtt kann man verwenden, wenn man Geräte von unterschiedlichen Herstellern auf einer Plattform zusammenbringen möchte, da alle denselben Funkstandard besitzen, aber die gegebene Software des Herstellers oftmals keine Kommunikation mit anderen Geräten zulässt.

Zigbee2MQTT ist das Bindeglied zwischen MQTT und Zigbee, wie in der folgenden Abbildung zu erkennen:

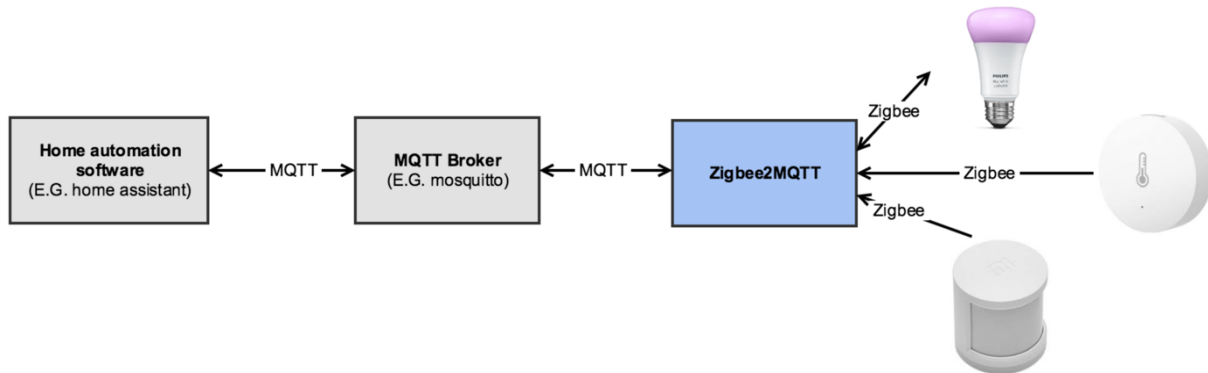


Abbildung 1: Zigbee2Mqtt Architektur (Q11)

iBeacon:

iBeacon wurde 2013 von Apple eingeführt. Es wird zur Lokalisierung genutzt und basiert auf Bluetooth Low Energy, welches eine Funktechnik für die Vernetzung innerhalb von 10 Metern ist und einen geringeren Stromverbrauch als Bluetooth hat.

Beacons können somit den Standort eines Gerätes erkennen. Wenn eine entsprechende App vorhanden ist und das Handy in Reichweite eines Senders kommt, wird die UUID des Senders identifiziert und die Signalstärke gemessen. Es wird zwischen drei verschiedenen Entfernungen entschieden:

Far ist über drei Meter Entfernung

Near ist zwischen einem und drei Meter Entfernung

Here bei weniger als einem Meter Entfernung

Mit vier Beacons lässt sich der genaue Standort in einem dreidimensionalen Raum ermitteln.

Detaillierte Vorgehensweise

1. Installation von Zigbee2Mqtt

Die Installation erfolgte in drei Schritten. Es musste zuerst der Standort des Zigbee Adapters festgestellt werden. Der Standard Ort ist: /dev/ttyACM0.

Das war der Fall für den Raspberry, da ich später auf das Macbook wechseln musste, war es dort ein anderer Speicherort (siehe Abbildung 2: Ausschnitt aus Configuration.yaml am MacBook, unter „serial: port:“ zu sehen).

Als nächstes werden die benötigten Programme installiert. Dazu wird das Node.js Repository und das Zigbee2Mqtt Repository geklont und installiert. Node.js ist notwendig, um JavaScript Code auszuführen und Zigbee2Mqtt enthält die benötigten Protokolle für Zigbee und MQTT.

Im dritten Schritt muss noch eine Änderung in der configuration.yaml Datei vorgenommen werden, die Server-Einstellung muss angepasst werden (wenn er nicht automatisch korrekt ist).

Im dritten Schritt wird im Terminal Zigbee2MQTT gestartet, mittels „npm start“, wenn man sich im Ordner /opt/zigbee2mqtt befindet.

Wenn man einen dauerhaften Einsatz mit dem Programm plant, kann Zigbee2MQTT als Daemon ausgeführt werden, dann startet es automatisch beim Booten, das habe ich aber nicht getan.

2. Verbindung zur Birne

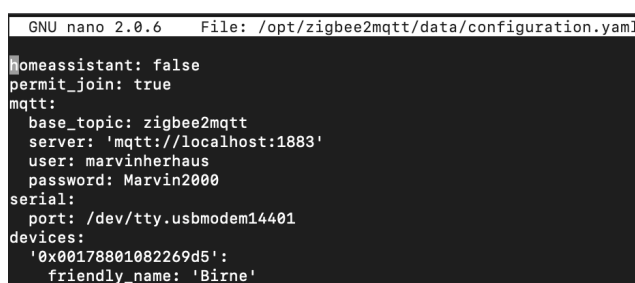
Dazu muss die Birne entweder neu sein oder auf die Werkseinstellungen zurückgesetzt werden. Dann verbindet sich das Programm beim Starten automatisch mit der Birne.

Ich verwende eine Philips Hue Birne mit der Modellnummer 9290018216, die Kopplung hat ohne Probleme funktioniert.

Zum Testen habe ich ein neues Terminal geöffnet und eine Nachricht an den MQTT Server gesendet (mosquitto_pub -h localhost -t zigbee2mqtt/0x001e5e0902146174/set -m "ON")

Die unterstrichene Nummer findet man nach der Kopplung im Verlauf des Programms, indem gekoppelte Geräte aufgelistet werden, nach dem Codewort friendly_name.

Außerdem findet sich diese Nummer in der bereits angesprochenen configuration.yaml Datei, dort lässt sich für eine bessere Identifizierung und Ansprechbarkeit auch dieser Name ändern, in meinem Fall habe ich das Gerät Birne genannt. Dieses Vorgehen folgt den Quellen Q6 und Q12.



```
GNU nano 2.0.6 File: /opt/zigbee2mqtt/data/configuration.yaml
homeassistant: false
permit_join: true
mqtt:
  base_topic: zigbee2mqtt
  server: 'mqtt://localhost:1883'
  user: marvinherhaus
  password: Marvin2000
serial:
  port: /dev/tty.usbmodem14401
devices:
  '0x00178801082269d5':
    friendly_name: 'Birne'
```

Abbildung 2: Ausschnitt aus Configuration.yaml am MacBook

3. App mit MQTT Befehlen

Bevor es an die App selber geht, muss MQTT erstmal auf dem Mac installiert werden, dafür wird CocoaMQTT verwendet. Und um diese Bibliothek zu installieren muss CocoaPods verwendet werden, was auch vorher noch installiert werden muss. Die konkreten Befehle finden sich in Q7 im Abschnitt „Creating an App on iOS using Swift“.

Als nächstes wird das Projekt erstellt und über das Terminal wird das Podfile mit der CocoaMQTT Bibliothek hinzugefügt.

Nun konnte ich das Layout erstellen, welches auch mit der Zeit gewachsen ist. Das Ergebnis ist Folgendes:

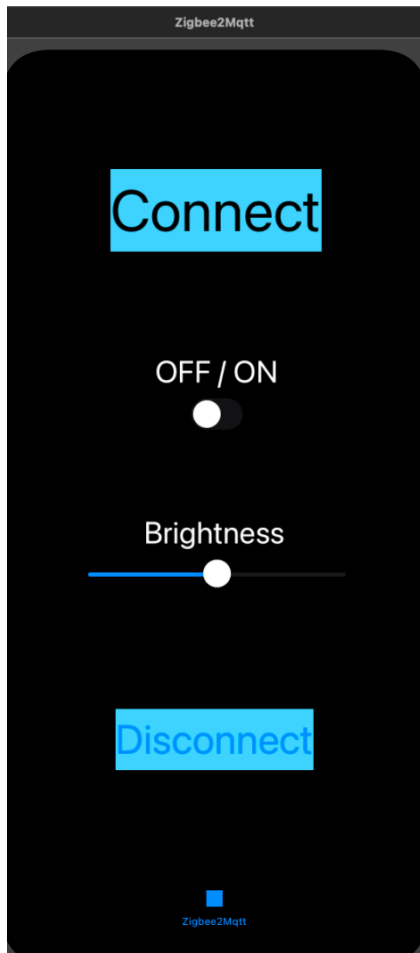


Abbildung 4: Controller 1

Connect Button: Wenn dieser Gedrückt wird, wird die Verbindung zum MQTT Server aufgebaut. Dies erfolgt über die Funktion `connect()`

UI Label: zeigt den Zustand ON und OFF nochmals beschriftet an. Zur Verdeutlichung des Schalters.

Switch: Um die Birne an und auszuschalten. Die Funktion verwendet eine if/else Verzweigung, die je nach Switch On oder OFF Zustand einen MQTT Befehl sendet mit der Nachricht „ON“ oder „OFF“. Das verwendete Topic ist in dem Fall „zigbee2mqtt/Birne/set“, wie auch bei den Befehlen über das Terminal.

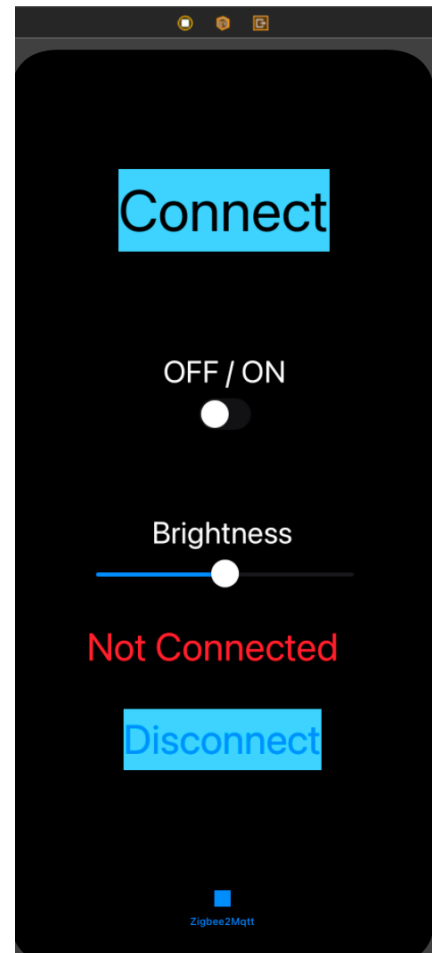


Abbildung 3: Controller 1.1

```
@IBAction func onOFF(_ sender: UISwitch) {
    if(mqttClient.connect()){
        fehlerMeldung.textColor = .red
    }else{
        fehlerMeldung.textColor = .black
        if sender.isOn {
            mqttClient.publish("zigbee2mqtt/Birne/set", withString: "ON")
        }else {
            mqttClient.publish("zigbee2mqtt/Birne/set", withString: "OFF")
        }
    }
}
```

Abbildung 5: Programmcode vom Switch

Slider: In der Funktion wird ein Wert zwischen 0 und 254 berechnet, um je nach Stand des Sliders die Helligkeit einzustellen. Die Funktion sendet dann eine Nachricht an den MQTT Server mit dem Helligkeitswert. Der errechnete Wert setzt sich aus der Position des Sliders (die zwischen 0 und 4) liegt multipliziert mit 63,5 (254/4) zusammen, so können die Werte 0 bis 254 herauskommen.


```

@IBAction func slider(_ sender: UISlider) {
    if(mqttClient.connect()){
        fehlerMeldung.textColor = .red
    }else{
        fehlerMeldung.textColor = .black
        let value = sender.value
        let wert = value*63.5
        print(wert)
        let Brightness = "{\"brightness\":\(wert)}"
        mqttClient.publish("zigbee2mqtt/Birne/set", withString: (Brightness))
    }
}

```

Abbildung 6: Programmcode des Sliders

Not Connected Label: Dieses Label hat immer dann die Textfarbe Rot (vorher schwarz), wenn der Switch oder der Slider bedient werden, ohne dass eine Verbindung zum Server hergestellt ist. Durch die Abfrage wird dann aber in der Regel eine Verbindung hergestellt - sofern der Server läuft - da die Abfrage der Connection durch dieselbe Funktion umgesetzt ist, die auch die Connection herstellt, sodass beim erneuten Betätigen das Label wieder „unsichtbar“ wird.

Disconnect Button: Dieser Button entfernt die Verbindung zum MQTT Server über die *disconnect()* Funktion.

Alle vorhandenen Funktionen hängen mit Objekten der Controller zusammen.

```

import UIKit
import CocoaMQTT

class ViewController: UIViewController{
    ...
    let mqttClient = CocoaMQTT(clientID: "iOS", host: "127.0.0.1", port: 1883)
    ...
}

```

Abbildung 7: Code Ausschnitt ViewController Klasse

Am Anfang der Klasse wird noch ein Objekt vom Typ CocoaMQTT, mit den Werten der Client ID, Host ID und dem verwendeten Port erstellt, um im Falle des *Connect()* zu wissen, zu welchem Host die Verbindung hergestellt werden soll. Die Client ID dient zur Identifizierung der verbundenen Geräte.

4. iBeacon Signal vom Raspberry Pi erzeugen

Die Installation für das Senden des BLE (Bluetooth Low Energy) Signals auf dem Raspberry Pi ist relativ simpel, nach der Installation einiger Bibliotheken (libglib2.0, libudev, libical, libreadline) kann Bluetooth gestartet werden.

Der Befehl „sudo hciconfig hci0 up“ startet Bluetooth und “sudo hciconfig hci0 leadv 3” versetzt den Modus auf Low Energy Advertising.

Im dritten Schritt muss dann noch eine UUID erstellt werden, die gesendet werden soll und auf welche dann auch die App hört. Das Mac-Terminal erstellt mittels *uuidgen* eine neue UUID, die dann verwendet werden kann.

Der Raspberry Pi muss dann noch folgenden Befehl ausführen:

```

„sudo hcitool -i hci0 cmd 0x08 0x0008 1E 02 01 1A 1A FF 4C 00 02 15 24 2E D10 9C F7 D4
6B A9 8C 46 1C F3 37 99 A4 D 00 00 00 00 C8“

```


Der Befehl setzt sich wie folgt zusammen: „sudo hcitool -i hci0 cmd 0x08 0x0008 1E 02 01 1A 1A FF 4C 00 02 15 *Hier steht die generierte UUID* 00 00 00 00 C8“ und schon sendet der Raspberry regelmäßig ein Signal.

5. App mit dem iBeacon Protokoll

In der bereits erstellten App habe ich nun ein neues Swift File und im Storyboard einen neuen ViewController erstellt. Das Layout ist in [Abbildung 11: Screenshot der Datei Main.Storyboard] zu sehen.

Als erstes muss *CoreLocation* importiert werden. CoreLocation bietet Dienste zur Standortbestimmung relativ zu einem Beacon an. Im weiteren Verlauf wird ein LocationManager Objekt und ein Objekt, welches die letzte Distanz speichert, benötigt. Dass die App den Standort abfragen darf, muss vom User erst bestätigt werden, dazu muss in der Info.plist Datei ein Key (*Privacy - Location When In Use Usage Description*) hinzugefügt werden. Dies sorgt für das Pop-Up Fenster beim Starten der App, welches der User dann bestätigt.

Bevor die App nun anfängt, nach Bluetooth Devices zu suchen, wird überprüft, ob sie autorisiert ist, die Standortdienste zu nutzen, ob das Gerät Beacons unterstützt die das iBeacon-Protokoll verwenden und ob das

```
func locationManager(_ manager: CLLocationManager, didChangeAuthorization status:
    CLAuthorizationStatus) {
    if status == .authorizedWhenInUse {
        if CLLocationManager.isRangingAvailable() {
            if CLLocationManager.isMonitoringAvailable(for: CLBeaconRegion.self) {
                // it can start
                startScanning()
            }
        }
    }
}
```

Abbildung 8: Codeausschnitt BluetoothViewController Klasse 1

Gerät die Regionsüberwachung unterstützt. Die letzten beiden Bedingungen sind an vorhandene Hardware geknüpft.

Wenn alle Bedingungen zutreffen, wird die *startScanning()* Funktion aufgerufen. Diese enthält die UUID, die der Beacon sendet und ein Objekt des Typen *CLBeaconIdentityConstraint*, welches die Merkmale des Beacon speichert. Zusätzlich gehört zur UUID noch der Major und Minor Wert. Major und Minor wird in meinem Fall aber nicht benötigt, da ich nur ein Beacon habe. Diese Werte werden nämlich dazu genutzt, zwischen Beacons zu unterscheiden, die dieselbe UUID verwenden, beispielsweise wenn ein Nutzer mit mehreren Beacons arbeitet.

Das dritte Objekt ist vom Typ *CLBeaconRegion*, welches eine Region definiert, in der die Beacons erkannt werden.

Nun wird die Methode *startMonitoring* gestartet, die nach Beacons sucht die in der Region sind. Und als nächstes die *startRangingBeacons* Methode, die die Relative Entfernung zu dem Beacon bestimmt.

Die nächste Funktion (siehe Abbildung 9: Codeausschnitt BluetoothViewController Klasse 2) teilt mit, dass ein Beacon gefunden wurde, der die Anforderungen erfüllt und ruft dann für diesen Beacon die nächste Funktion (*update*) auf.

```
func locationManager(_ manager: CLLocationManager, didRangeBeacons: [CLBeacon],
    satisfying beaconConstraint: CLBeaconIdentityConstraint) {
    if let beacon = beacons.first {
        update(distance: beacon.proximity)
        print("Ranging \(beacon.uuid)")
    } else {
        update(distance: .unknown)
    }
}
```

Abbildung 9: Codeausschnitt BluetoothViewController Klasse 2

Zusätzlich gibt er noch die UUID zurück, mit der man die Korrektheit dann nochmals überprüfen kann und eine Info bekommt, mit welchem Beacon nun die Entfernung ausgetauscht wird.

Die *Update()* Funktion speichert nun die letzte Distanz. Diese wird auch dauernd aktualisiert und führt je nach Distanz einen bestimmten Code aus. Erstens wird die Distanz auf der Konsole zurückgegeben und in der App wird die Distanz schriftlich ausgegeben und das Label (welches zur Simulation eingefügt wurde) ändert seine Farbe.

Orientiert wurde sich an dem Vorgehen aus Q8.

```
func update(distance: CLProximity) {
    lastdistance = distance
    if distance == .far {
        print("Far") //Über 3 Meter
        distanceLabel.text = "Distance: Far"
        lightLabel.backgroundColor = .yellow
    }
    if distance == .near {
        print("Near") //1-3 Meter
        distanceLabel.text = "Distance: Near"
        lightLabel.backgroundColor = .orange
    }
    if distance == .immediate {
        print("HERE") //Ganz nah
        distanceLabel.text = "Distance: Here"
        lightLabel.backgroundColor = .red
    }
    if distance == .unknown {
        print("Unknown") //zu weit weg
        distanceLabel.text = "Distance: Unknown"
        lightLabel.backgroundColor = .black
    }
}
```

Abbildung 10: Codeausschnitt BluetoothViewController Klasse 3

6. Zusammenführung der beiden Komponenten

Um die beiden Oberflächen leicht ansteuerbar zu machen, habe ich mich für den Tab Bar Controller entschieden.

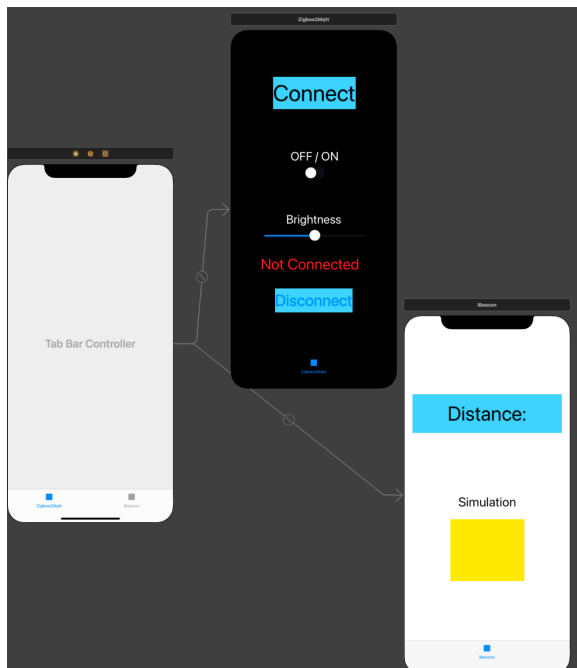


Abbildung 11: Screenshot der Datei Main.Storyboard

Ohne die aufgetretenen Probleme wäre es nur ein Controller geworden.

Probleme

Es konnte kein Signal vom iPhone und MacBook (Simulator) an den Raspberry Pi gesendet werden: Die App soll ein Signal an den MQTT Server senden, dieser lief auf dem Raspberry Pi. Es konnte allerdings keine Verbindung aufgebaut werden, die Funktion `connect()` hat nie ein `True` zurückgegeben.

Dann habe ich versucht, den Server auf dem Macbook laufen zu lassen und die Simulation lief auf dem Handy, auch so konnte keine Verbindung hergestellt werden. Nur wenn die Simulation auf dem MacBook lief, konnte eine Verbindung zum Localhost hergestellt werden. Aber auch das funktionierte nur, wenn die angegebene IP-Adresse wirklich 127.0.0.1 war. Die öffentliche IP-Adresse hat auch nicht funktioniert.

Die Recherche für den richtigen Code für die App, die den iBeacon sucht, vor allem für die Bestimmung der Distanz, gestaltete sich als schwierig. Es gibt viele Lösungen im Internet, die mit Beacons arbeiten, in den seltensten Fällen wird allerdings die Distanz abgefragt bzw. verwendet, sondern es wird bei Reichweite eine URL oder ähnliches an das Smartphone gesendet. Somit hat sich die Recherche nach der korrekten Lösung etwas in die Länge gezogen.

Ebenfalls die Recherche für die Steuerung der Helligkeit, bzw. der Gedanke, dass es über Json geht, war etwas länger.

Eine Nachricht mit „ON“ und „OFF“ zu senden, war simpel, aber eine Nachricht mit dem Format „*Brightness*“:100 zu verschicken, war für mich lange unerklärlich, weil das nicht in einem String zu verwirklichen war.

Nach erneutem Lesen der Dokumentation von Zigbee2MQTT bin ich darauf aufmerksam geworden, dass die Nachricht im JSON Format gesendet wird. Auch hier gibt es im Internet viele Lösungen, die mit einer URL und mit dem Encoder und Decoder arbeiten. Das war aber nicht das, was ich suchte. Schlussendlich bin ich dann darauf gekommen einfach, selber ein passendes Objekt zu erstellen und das hat dann funktioniert.

Das Problem mit dem Slider tritt erst beim Ausführen der App auf. Wenn der Slider langsam bedient wird, errechnet er für jede minimale Veränderung einen neuen Wert. Dieser wird dann sofort an den MQTT Server gesendet. So kommen in Millisekunden sehr viele Werte und Nachrichten zustande, sodass die Funktion mit den jeweils neuen Werten sehr oft ausgeführt wird, das führt zu einer verzögerten Übertragung. Das Problem hat man nicht, wenn der Slider schnell bedient wird.

Um das zu beheben habe ich die Sleep Funktion verwendet. Nach jeder Werte änderung/ neu gesendetem Wert wurde sie einmal ausgeführt. Dadurch hing das Programm bzw. der Slider und hat sich selbst auf *sleep(1)* nur sehr langsam bewegt, was auch zu keiner guten Nutzungserlebnis geführt hat.

Fazit

Es war ein gelungenes Projekt. Auch wenn die endgültige Funktionalität fehlt, habe ich viel gelernt und meine persönlichen Ziele erreicht. Ich kann mit Xcode umgehen und habe erste Erfahrungen mit Swift gesammelt. Dazu habe ich noch mit Kommunikationsprotokollen gearbeitet und werde auch in Zukunft eigene Programme entwickeln, um meine Smart Home Geräte zu steuern und gute Automationen zu haben.

Nachbearbeitung

Hier nenne ich noch ein paar Ideen und Verbesserungen die als nächstes noch vorgenommen werden sollten, für die im Rahmen des Projekts aber die Zeit fehlte.

- Der Switch für ON/OFF sollte den Status abfragen und in die richtige Position gehen, sodass, wenn die Lampe zu Beginn an, ist auch der Schalter auf „an“ steht und nicht auf „aus“. Aktuell ist er standardmäßig auf „aus“, da in der Regel die Birne aus ist, wenn man mit der App beginnt.
- Das Design kann auch noch überarbeitet werden, damit die App noch benutzerfreundlicher wird und auch ein besseres Design hat. Buttons können beispielsweise dreidimensional gestaltet werden, das verdeutlicht die Tatsache, dass es Buttons sind, die gedrückt werden können.
- Dann sollte natürlich das Zusammenspiel von Beacon Entfernung und Helligkeit zusammengeführt werden. Und Zigbee2MQTT sollte auf einem Raspberry laufen.
- Ergänzen, dass man den Wert zwischen 0 und 254 für die Helligkeit manuell angeben kann.

Links

Programmcode: <https://github.com/Marvin0103/Informatikprojekt-Smart-Home->

„Produktvideo“: https://www.youtube.com/watch?v=tg_hWos0jTE

Quellen

- [Q1] <https://blog.deinhandy.de/was-ist-zigbee-der-funkstandard-kurz-erklart> [07.02.21, 11:20Uhr]
- [Q2] <https://www.conrad.de/de/ratgeber/technik-einfach-erklart/zigbee-standard.html> [08.02.2020, 12:40]
- [Q3] <https://de.wikipedia.org/wiki/MQTT> [08.02.21, 12:50]
- [Q4] <https://www.embedded-software-engineering.de/was-ist-mqtt-a-725485/> [08.02.22, 13:05]
- [Q5] <https://de.wikipedia.org/wiki/IBeacon> [08.02.2020, 20:20]
- [Q6] <https://www.zigbee2mqtt.io>
- [Q6.1] https://www.zigbee2mqtt.io/getting_started/running_zigbee2mqtt.html [regelmäßig zwischen 11.12.20 und 15.01.21]
- [Q7] <https://www.raspberrypi.org/forums/viewtopic.php?t=196010> [regelmäßig zwischen 11.12.20 und 15.01.21]
- [Q8] <https://www.youtube.com/watch?v=ICNpEaZiKqU> [regelmäßig zwischen 10.01.21 und 2.2.21]
- [Q9] <https://encrypted-tbn0.gstatic.com/images?q=tbn:ANd9GcShPOaBXgOxgflByhDplJfXfJdoz2qBXpR15g&usqp=CAU> [08.02.21 13:08]
- [Q10] <https://developer.apple.com> und verschiedene Unterseiten, die Teile aus dem Programmcode erklären [Januar und Februar 2021]
- [Q11] <https://raw.githubusercontent.com/Koenkk/zigbee2mqtt/master/images/architecture.png> [12.02.21 13:36]
- [Q12] <https://www.rs-online.com/designspark/rounding-up-unruly-zigbees-with-zigbee2mqtt-de> [regelmäßig zwischen 11.12.20 und 15.01.21]
- [Q13] https://www.youtube.com/playlist?list=PL3d_SFOiG7_8ofjyKzX6Nl1wZehbdiZC [1.11.20 bis 15.12.20]