BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

INTEGRATION OF SENSORS AND ACTORS INTO THE BEEEON SYSTEM

INTEGRACE SENZORŮ A AKTORŮ DO SYSTÉMU BEEEON

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **LUKÁŠ ŠIŠMIŠ**
AUTOR PRÁCE

SUPERVISOR **Ing. MARTIN SAKIN**
VEDOUCÍ PRÁCE

BRNO 2018

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2018/2019

Zadání bakalářské práce

Student: **Šišmiš Lukáš**
Program: Informační technologie
Název: **Integrace senzorů a aktorů do systému BeeeOn**
Integration of Sensors and Actors into the BeeeOn System
Kategorie: Vestavěné systémy

Zadání:

1. Nastudujte dodané senzory nebo aktory určené pro automatizaci domácnosti podporující technologii Wi-Fi.
2. Seznamte se s architekturou brány systému BeeeOn.
3. Navrhněte způsob integrace nastudovaných technologií do systému BeeeOn.
4. Implementujte navržené řešení a ověřte funkčnost v domácnosti.
5. Diskutujte dosažené výsledky a možnosti pokračování projektu.

Literatura:

- Dle pokynů vedoucího, zejména dokumentace k vybraným senzorům/aktorům a dokumentace systému BeeeOn.

Pro udělení zápočtu za první semestr je požadováno:

- Splnění bodů 1 až 3 zadání.

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Sakin Martin, Ing.**
Konzultant: Viktorin Jan, Ing., FIT VUT
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 15. května 2019
Datum schválení: 1. listopadu 2018

Abstract

The goal of this thesis was to extend BeeeOn system with a Vektiva module and to do a research on possibilities of Smarwi implementation, which not only included studying different communication protocols that the device uses, but also comparing them and choosing the most suitable protocol for the implementation. In the beginning of the thesis, BeeeOn system and its' components are described followed by the description of implementation and testing process of the Vektiva module and the Smarwi emulator.

Abstrakt

Cieľom tejto práce bolo rozšíriť systém BeeeOn o modul Vektiva a preskúmať možnosti implementovania produktu Smarwi do systému BeeeOn, čo zahŕňalo nielen štúdium komunikačných protokolov, ktoré zariadenie používa ale aj ich porovnanie a výber vhodného protokolu pre implementáciu. Na začiatku práce je popísaný BeeeOn systém a jeho komponenty, na ktorý následne nadväzuje popis procesu implementácie a testovania Vektiva modulu a aj Smarwi emulátoru.

Keywords

Home Automation, Smart Home, Vektiva, Smarwi, BeeeOn, Gateway, Window Ventilation, Window Opener

Klíčová slova

Automatizácia Domácnosti, Inteligentná Domácnosť, Vektiva, Smarwi, BeeeOn, Gateway, Ventilácia Okien, Otvárač Okien

Reference

ŠIŠMIŠ, Lukáš. *Integration of Sensors and Actors into the BeeeOn System.* Brno, 2018. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Martin Sakin

Rozšířený abstrakt

V dnešnej dobe sa často stretávame s pojmoj IoT alebo Internet vecí. Aby sme si mohli vysvetliť tento pojem, musíme si predstaviť obrovské množstvo zariadení, či už sú to vozidlá, senzory, monitorovacie prostriedky či domáce spotrebiče všetky pripojené do siete Internet aby následne spolu mohli komunikovať. Výsledkom môže byť zvýšená životná kvalita a úroveň života, ušetrené náklady, vyššia bezpečnosť alebo zachránené ľudské životy. Konkrétny príklad môže byť zapnutie robotického vysávača ak sa nikto nenachádza v dome a je detegovaná špinavá podlaha alebo doručenie kamerového záznamu polícii po násilnom vlámaní do budovy pre rýchlejšiu identifikáciu zločinca.

IoT a automatizácia domácnosti je stále sa rozvíjajúca oblasť IT. Preto podlieha dynamickým zmenám, ktoré implementujú rôzni výrobcovia týchto zariadení inak, čo zapríčiňuje nekompatibilitu medzi jednotlivými zariadeniami od rozličných značiek.

Tejto príležitosti využili výzkumníci z Fakulty Informačných Technológií v Brne, ktorí navrhli a implementovali systém BeeeOn. Jeho hlavnou úlohou je zjednocovať zariadenia rôznych výrobcov do jedného bodu, z ktorého je možné kontrolovať všetky zariadenia súčasne. Týmto krokom eliminujú potrebu natívnych aplikácií od jednotlivých výrobcov a prispievajú k lepšej komunikácií medzi samostatnými zariadeniami.
Systém BeeeOn je open-hardware a open-source, čím sprístupňuje kód nielen na štúdium ale aj jeho upravenie. Pojem open-hardware znamená, že vnútorná architektúra primárne zvoleného počítaču je verejne dostupná a aj keď by výrobca ukončil masovú produkciu, nie je potrebné návrh matičnej dosky robiť opäť aby sa mohla produkcia obnoviť.
Samotný systém sa skladá z 4 vrstiev, kde okrajové vrstvy tvoria koncové zariadenia, či už sú to senzory/aktory alebo zariadenia užívateľov ako mobily alebo počítače. Vnútorné 2 vrstvy sa skladajú z BeeeOn aplikácií Gateway a Server. Všetky dôležité dáta sú uchované na serveri, pričom aplikácia Gateway slúži ako prostredník medzi senzormi/aktormi a serverom. Gateway buď zbiera údaje od inteligentných zariadení a následne ich posiela na server alebo prijíma príkazy zo serveru, akú akciu má vykonať.

Ako možnosti inteligetných zariadení rastú, nachádzajú sa pre ne aj nové spôsoby využitia a uplatnenia. Jedným z príkladov môžu byť diaľkovo ovládané otváranie okien, ktoré môže byť nainštalované na už zabudované okná. Môžu priniesť niekoľko výhod ako inteligentné vetranie, prevetranie miestností v ktorých sa momentálne nezdržiavame alebo otváranie okien v ťažko dostupných miestach. Čelia však niekoľkým problémom ako nákladná počiatočná investícia, hlučnosť zariadenia, rôzne typy otvárania okien alebo obmedzený priestor na inštaláciu takýchto zariadení. Viacero výrobcov už vytvorilo takéto zariadenia pričom tie najznámejšie typy používajú hriadeľové, reťazové alebo skladacie otvárače okien.

Česká firma Vektiva priniesla na trh svoj produkt Smarwi, inteligentný otvárač okien, ktorý v sebe obsahuje Wi-Fi prijímač a dokáže tak komunikovať s inteligentnými systémami ako Fibaro, IFTTT, OpenHAB a iné. Úlohou tejto práce bolo implementovať toto zariadenie do systému BeeeOn. Na rozdiel od už predstavených typov otváračov okien, Vektiva zvolila hrebeň, ktorý je pripevnený na pevnom ráme okna a ozubené koleso, ktoré je zabudované v Smarwi. Keďže Smarwi je nalepené na pohyblivej časti okna, pomocou ozubeného kolesa sa pohybuje po hrebeni.
Existujú 3 možnosti pre ovládanie zariadenia a to buď cez fyzické tlačítko, cez API pomocou HTTP alebo cez MQTT správy. Cez HTTP je možné Smarwi ovládať len po lokálnej sieti, kdežto MQTT správy dokážu meniť stav zariadenia aj mimo LAN. Vektiva má aj vlastný portál pre správu pripojených Smarwi vďaka čomu poskytuje API aj vo WAN. Po

spracovaní požiadavky Vektiva server kontaktuje Vektiva MQTT broker a ten vyšle danú Smarwi MQTT správu.

Po preštudovaní BeeeOn systému a komunikácie medzi Smarwi a klientom sa vytvoril návrh implementácie, v ktorom sa posudzovali protokoly HTTP a MQTT, navrhol sa diagram tried a zvolila vhodná metodika implementácie. Návrh zariadenia obsahuje 4 moduly z čoho jeden poskytuje iba informácie a 3 ostatné moduly dokážu zmeniť stav Smarwi. Počas samotnej práce na zapracovaní návrhu do systému sa pracovalo v iteráciach s priemernou dĺžkou 10 dní. Počas tohto času sa opravovali chyby, pridávali sa nové časti a testy na ne a konzultoval sa ďalší postup. Modul je implementovaný v jazyku C++ s knižnicami std a Poco. V správcovi zariadení sa nachádzajú 2 klienti MQTT, z čoho jeden manipuluje vždy iba s jedným vybraným zariadením Smarwi a druhý prijíma a analyzuje všetky správy prijaté na vybranú tému *ion/#*.
Pre zaistenie korektného fungovania modulu prebiehalo aj testovanie, ktoré zahŕňalo aj jednotkové aj manuálne testy. Jednotkové testy slúžia pre overenie správnosti spracovávania prijatých správ a ich vytváranie, manuálne testovanie prebiehalo po každej pridanej časti pre overenie celkovej funkčnosti.

Pre zjednodušenie ďalšej práce so Smarwi bol vytvorený emulátor, ktorý má za úlohu simulovať chovanie Smarwi na sieti pomocou MQTT správ. Emulátor je schopný simulovať viaceré Smarwi zariadenia súčasne. Vytvorené zariadenie je možné ovládať cez MQTT správy, API alebo grafické uživateľské rozhranie. Pre tento účel bolo vytvorené REST API, pomocou ktorého sa dajú ovládať všetky zariadenia, ktoré emulátor obsahuje. Všetky zmeny vykonané cez jeden z vymenovaných spôsobov sa propagujú pomocou MQTT správ. Pre vytvorenie emulátoru bol použitý Python3 s knižnicami Paho client, Threading a socketserver. Pre vytvorenie uživateľského rozhrania bolo použité HTML, CSS a jQuery.

Integration of Sensors and Actors into the BeeeOn System

Declaration

Hereby I declare that this bachelor's thesis was prepared as an original author's work under the supervision of Mr. Martin Sakin. The supplementary information was provided by Mr. David Bednařík. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

.....
Lukáš Šišmiš
July 2, 2019

Acknowledgements

Here I would like to express thanks to my supervisor Ing. Martin Sakin for the consultation he has provided to me while writing the thesis and to Bc. David Bednařík for the consultation related to the code.

Contents

1 Introduction **3**

2 System BeeeOn **5**
2.1 System architecture 5
2.2 User interface 6
2.3 Server 7
2.3.1 Application layer 7
2.3.2 Service layer 7
2.3.3 Database layer 7
2.4 Gateway 8
2.5 Devices 10
2.6 Testing center and virtual devices 11

3 Remote window opening options **12**
3.1 Problems and requirements 12
3.2 Opening options 13
3.3 Existing products 14

4 Smarwi - a window opener **16**
4.1 Vektiva 16
4.2 Control 16
4.3 Principle of Smarwi functionality 16
4.4 Communication 17
4.5 HTTP API 18
4.6 MQTT 19

5 Integration design of Vektiva module **24**
5.1 Device Manager 24
5.2 Vektiva Device Manager 24
5.3 Communication 27
5.3.1 Vektiva's broker 27
5.3.2 Gateway's broker 28
5.3.3 On connect 28
5.3.4 Proposal of communication design 28

6 Implementation into the BeeeOn system **29**
6.1 Implemented parts of the proposed plan 29
6.2 Parts changed 29

6.3 Implementation tests 30

7 Testing - Smarwi emulator **33**
7.1 Implementation 33
7.1.1 Routing 33
7.1.2 Smarwi handling 34
7.2 API endpoints 34
7.3 MQTT 36
7.4 User interface 36
7.5 Scheduling and fixing errors 37

8 Conclusion **40**

Bibliography **41**

A Smarwi MQTT communication **43**
A.1 MQTT messages 43
A.1.1 Open 55
A.1.2 Close 55
A.1.3 Stop 56
A.2 Smarwi Status codes 57
A.3 Smarwi Errors 58
A.4 Mosquitto examples 58

B CD contents **59**

Chapter 1

Introduction

We often hear of the term Internet of Things or IoT. It basically connects every device that is able to receive and transmit data without any need of human being. That means we can have billions of devices talking to each other, exchanging gathered information and based on that, it can either help us to do the right decision or even do the decision for us. It can go from simple things such as reminding us to take an umbrella when a thunderstorm is forecasted to more complex ones such as detecting unclean floor to start home vacuuming and mopping when we are not at home.

It also adds a security layer to our lives by analyzing who just entered home, whether it was a brute force attack or not. In case of an intruder, home owner and police can be notified with intruder's photos.

IoT appears everywhere we can think of. It has found its' use case at homes, businesses and among all different kinds of industries. Industry 4.0 is the name of the current era of letting machines do the work for us.

Home automation is a great market opportunity with many things far away from perfection. Because of this, manufacturers often just experiment with their products and are moving extremely fast. However, no organizations stand on top of the industry and therefore no common universal protocol is used. Because of this, different devices often can not communicate with each other. It seems like there can not be any protocol since we have many totally distinct devices that don't share any common traits. Devices such as a window opener and a thermostat regulator don't go well along.

After researchers at the Faculty of Information Technology at the University of Brno realized this, they created a project named BeeeOn which act as a unifier or in other words it unites devices from completely different vendors with different communication protocols. Manufacturers' APIs are implemented into BeeeOn with the goal of having one app to rule them all. BeeeOn is developed and maintained by a group of professors, students, and enthusiasts.

List of supported devices and vendors slowly but steadily expands. One such device that is not supported so far is called Smarwi from Vektiva. It is an almost universal window regulator especially suitable for tilt-turn windows. Having always fresh air when we need it is very valuable. While windows can be controlled with a click on a security box, its' main advantage is that it can be interconnected with platforms like OpenHAB, Fibaro, IFTTT or Stringify. The main goal of the thesis is to implement Smarwi into the BeeeOn Gateway which allows us to remotely regulate windows.

In the following chapter, the BeeeOn system is briefly explained. It shows the inner architecture of the system with parts such as the gateway, server, user devices and sensors

or actors and their relationships. In the third chapter, remote windows opening solutions are presented where you will learn about not only the theoretical part but also you will see real-world examples. With not only that, Smarwi as a device is thoroughly described as well. In the fourth chapter, a proposal for integrating a Vektiva module into the BeeeOn system is presented. A chapter about implementing a proposed plan and testing follows where parts of the plan are described with their outcome. The last part describes Smarwi emulator, presents the implementation and ways to use it.

Chapter 2

System BeeeOn

For an introduction system called BeeeOn[1] is briefly described. This system works with Internet of Things (IoT) or more precisely, it is focused on home automation. Its' main purpose is not to create something entirely new but to act more as a unifier. Since home automation is a relatively new and broad field, there hasn't been introduced a protocol with which one could control devices across all spectrums. Because of this, vendors that produce devices intended for home automation create protocols by themselves. This leads to incompatibility issues across devices even if they are designated to do the same things. As a result people are forced to have multiple apps, since every manufacturer has an app or a web interface to control solely his devices. To solve this problem, the BeeeOn system was created. It is an open-hardware and open-source initiative to have a single app that is able to communicate with all supported devices from many different vendors by implementing necessary components that are required to successfuly handle these devices.

2.1 System architecture

With a system of this aim, it is expected that the project will grow at a fast pace. Often times projects as they are developed, get so big it is very complex to modify parts of the code. To properly handle this scenario, project is designed as a modular kit and therefore getting to know the code can be easier. This is particularly important attribute in case of open-source code. Thanks to dependency injection, it also allows us to only have the components we want to be enabled.

As we can see on image 2.1 BeeeOn consists of 4 separate layers, each connecting with neighboring layer. Arrows show the flow of communication. Bidirectional arrows mean there are flows of data from one layer to the other and vice versa.

A layer, which can be found on the top, is intended for end users. It is an access point for a user to interact with the system.

Beneath, we can see a server layer. The server serves as a controller and a data collector for a user. It is connected to multiple gateways that work as independent units. Communication with both end-user devices and gateways is usually ensured on the Internet (WAN) network.

To reduce the load on the server, the third layer called Gateway was implemented. Its' main function is to listen to what the server says and to communicate with end devices. As a result, Gateway translates the message from a unified protocol that is between the

[1] http://www.beeeon.org

server and the Gateway to a protocol that selected end device uses and vice versa. It sends collected data and exchange commands with the server. It also reads data from sensors and sends instructions to actors. To be able to do that it needs to have a list of properly connected end devices. This way the server is connected to only one device and is able to send commands to all of the registered end devices. Gateway is further described in 2.4.

End devices are a general term for sensors/actors. Sensors measure things according to their intention in their surroundings and actors can also change its' state and therefore impact the environment. After their initial setup, they connect to the gateway and the rest of the communication is solely between the end device and the gateway. Section 2.5 provides more information related to sensors or actors.

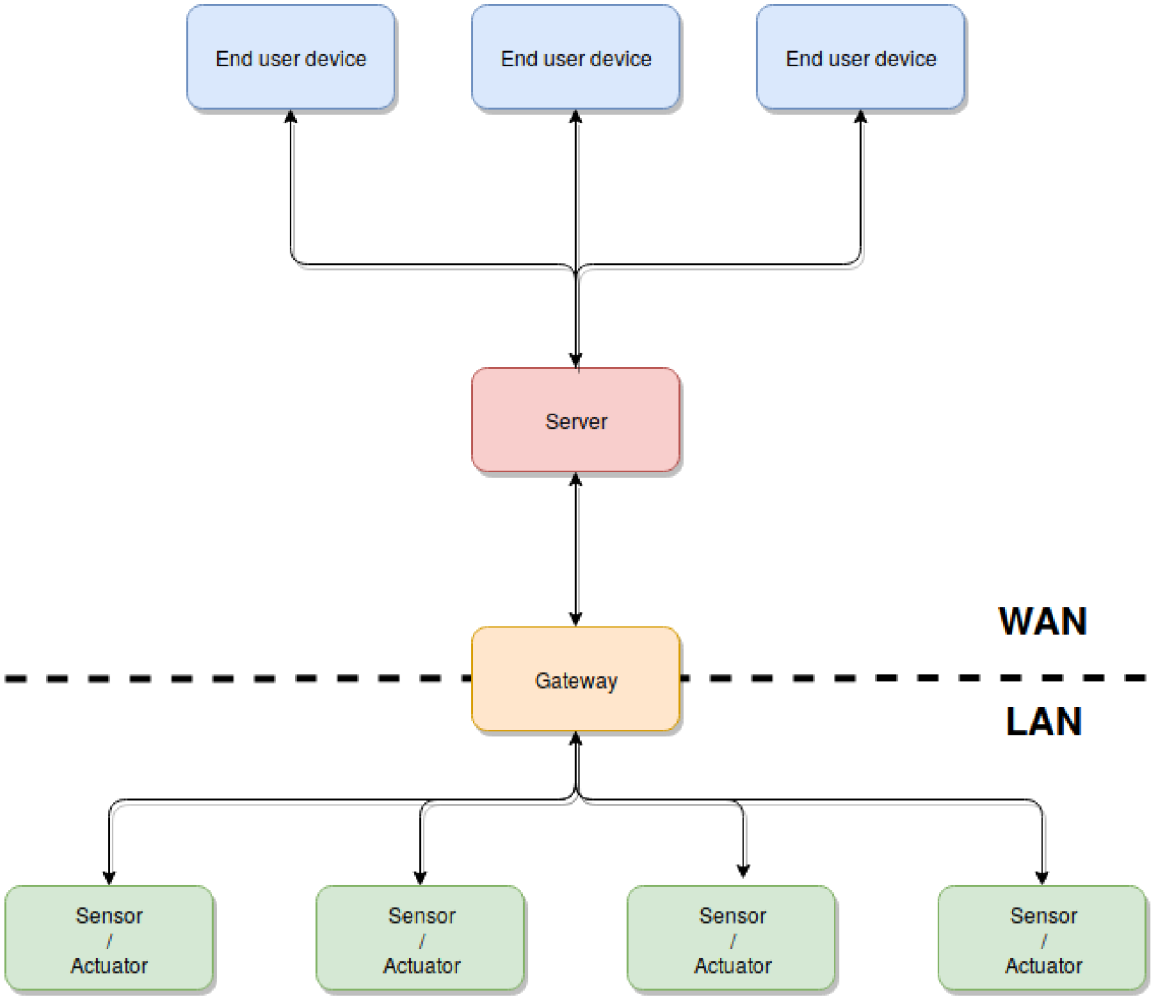

Figure 2.1: System architecture of the BeeeOn system

There are many other parts of the system which are not entirely important for this bachelor thesis so for this reason and for the sake of clarity they will not be described.

2.2 User interface

On the top layer, we find a user interface which consists of devices such as tablets, mobile phones or even laptops or personal computers. These are called end-user devices as they

serve as an access point for users to set up their intelligent devices or watch statistics gathered by them.

In the current state of the system, there are two ways of accessing the user interface of the BeeeOn system. An end user can use either a web application or an Android application. The latter option has not been maintained for a while so it is possible to be a little outdated. The web interface is a convenient way to interact with the system, thanks to the thin client that is a web browser that almost any device has.

2.3 Server

The layer just under the top one is called a server layer. The server is usually located at a remote location and is not maintained by a user. It serves as a middleware between a user and a gateway. User requests are sent to the server and after they are processed, the gateway is contacted with tasks to do. Once Gateway accomplishes given tasks it will respond to the server with their outcomes. The server can write results to a database that is usually located on the server and then notifies the user with the results. To further support extendability and manageability of the server code architecture, it was divided into three layers:

- Application layer
- Service layer
- Database layer

2.3.1 Application layer

The application layer provides a way how either users with end user devices or gateways communicate with the server. It deals with user registration or their requests and passes them to the service layer if needed. At the same time, it has a pool of connections with connected gateways, which is needed because gateways might not be available from outside of the local network. It periodically checks for the availability of gateways or sends tasks to them. In case of failure, it can promptly notify the user.

2.3.2 Service layer

The service layer implements the core logic of the server. It handles most of the user's or gateway's requests and responses. It works with both layers, especially with a database layer for easy and consistent access to the database.

2.3.3 Database layer

The database layer serves as an abstract layer to access the database. It also guarantees that server logic is not coupled to any database system. After the implementation of given interfaces, we can change database systems with ease and in a matter of seconds. In the current implementation, PostgreSQL is used.

2.4 Gateway

The main component that is important to the user is a gateway. It provides a point to which every smart device at home should connect to. Gateway then collects all the information it can from these sensors/actors and sends them to the server. If it receives any task to do, it contacts the required device and tries to fulfill the request. It periodically checks end devices for their availability and data and at the same time sends data to the server.

As an initiative to have a dedicated device that this system can run on, A10-OLinuXino-LIME[2] from Olimex was chosen. It is an open hardware ARM Linux computer that serves well enough to cover all needs BeeeOn Gateway requires. An example can be seen in a modified BeeeOn case on image 2.2 with the specifications such as 1GHz Allwinner A10 Cortex-A8, Mali 400 GPU, 512MB DDR3 RAM memory, 160 GPIOs on four GPIO rows of pins (0.05" step), 5V input power supply, noise immune design and PCB dimensions of 84x60 mm. More precise information can be found on the vendor's webpage[2].

Since it is an open hardware solution, the project is not closely dependent on Olimex. Even if A10-OLinuXino-LIME manufacturer stops mass-producing these boards, schemas and data sheets are always available and production can be renewed easily.

Figure 2.2: Photo of **Olimex board A10** on which the BeeeOn system runs [18]

.

To A10-OLinuXino-LIME, an extension board MRF89XA from Microchip is connected via GPIO pins, providing radio connectivity to the entire system.

Although this board is recommended for the gateway software, it can run on any other device with Linux distribution based on Debian. My personal experience was setting up the BeeeOn Gateway on Raspberry Pi Model 1B which has only an ARM1176JZF-S 700 MHz CPU and 256MB RAM. After an initial hassle, I was able to install the whole gateway system with an installation script. I did not try to connect the radio extension board. I successfully ran the system in testing mode.

[2] https://www.olimex.com/Products/OLinuXino/A10/A10-OLinuXino-LIME-n4GB/open-source-hardware

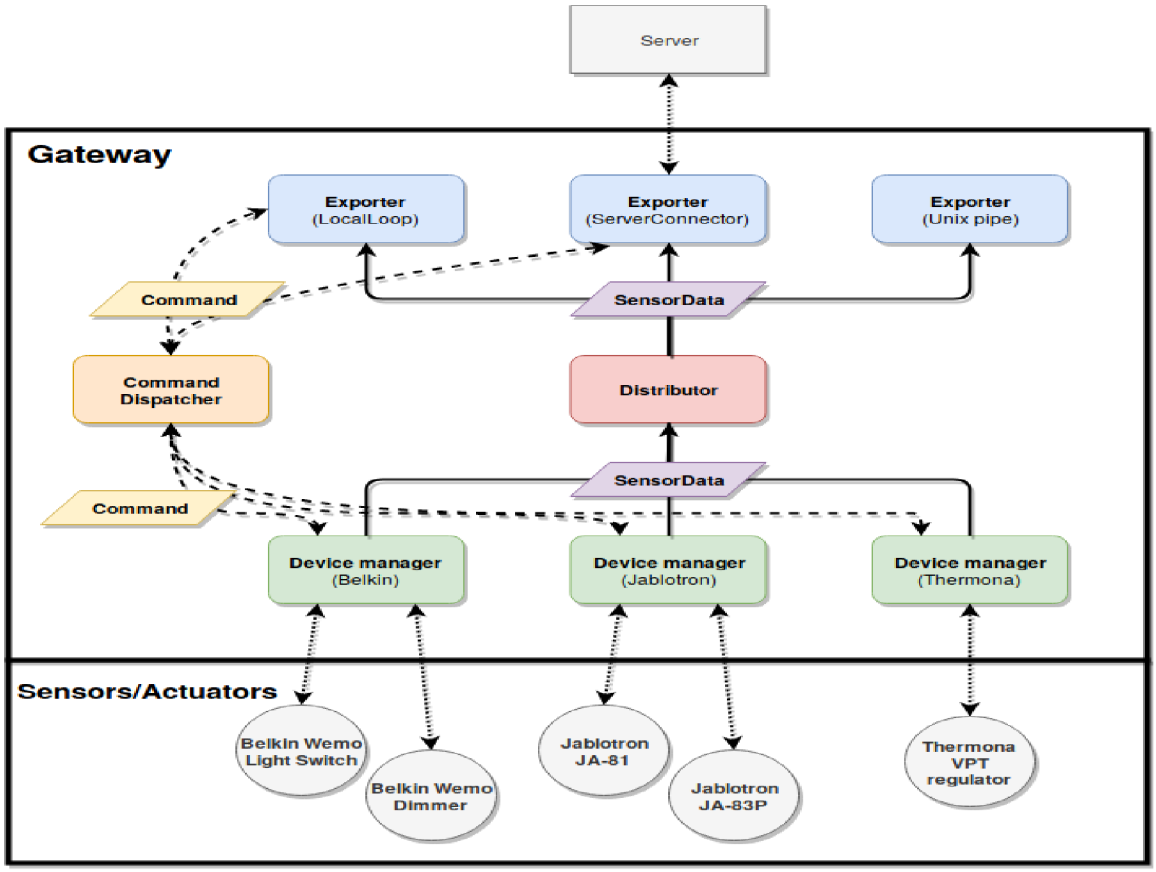

Figure 2.3: Gateway architecture

Internal gateway architecture

According to image 2.3, we see briefly described the way the gateway works.

Device managers are one of the most important parts of the gateway system. They communicate with end devices and gather their data. They act as a bridge between *Distributor* and end devices. Once they receive something from the end device they support, they immediately convert the data to a unified message format and send it to the *Distributor* and vice versa. Each device manager represents a group of devices with similar characteristics (usually the same vendor) and runs in a separated thread simultaneously with other device managers. Asynchronous processing allows us to run the gateway even in high traffic environments and it also optimizes and speeds up data handling. Each device manager is tailored to the needs of the vendor's devices or any other group of devices. Since home automation progress extremely fast and no universal and comprehensive protocol has been introduced, the BeeeOn system implements vendors' protocols.

For messages, a class *SensorData* is used. For a device manager to know to which device a message should be sent, we need some form of a unique ID. For this purpose an identification field in *SensorData* is created. It has space for a unique identifier that has 64 bits, where the first 8 bits are reserved for an ID of the *DeviceManager* and the rest is created in another way. It can be either random or set according to some other device's property usually a MAC address of the device is used.

When *Distributor* receives a *SensorData* message from the *DeviceManager* it can propagate the message to every registered *Exporter*. *Exporter* is an interface that when implemented, aims to dispatch every message that is meant to be exported to a platform or a place according to the implementation. Such class can be a *ServerConnector*, which sends messages to the specified server or a UnixPipe that sends output to the Unix pipeline.

CommandDispatcher runs in a separate thread, waiting for commands that come from a server or any other registered place. On received command, CommandDispatcher looks up a receiver of the given command and sends it directly to him. Any object can obtain a Command after implementing a *Command* interface and register itself to the *CommandDispatcher*.

2.5 Devices

Sensors/actors are the devices for which is this system designed. They are usually small, often easily overlooked but they are bringing the new era of home automation in the form of smarter, affordable technologies and more intelligent and resource-saving homes for us. There are many ways how they enhance our lives ranging from having automatically adjusted room's temperature according to the desires of people there or weather forecast throughout the day to efficiently manage all home resources without the need of us even thinking about it.
From the most abstract view point we can divide this category into two main groups:

- Sensors – devices, which are often unnoticeable, passive, serve as data collectors in a sense, their purpose is to measure its' surroundings with tools they have available. Gathered data are sent immediately after measurement is finished to the gateway they are connected to, where they are additionally processed. Instances of equipment that falls into this category might be sensors for CO detection, moisture detection, open window and door detection and many others.

- actors – devices that can actively alter the environment where they are installed. Regularly accompanied with a sensor or a collection of them, they try to adjust their surroundings to the desired state. Gateway may either ask for a state in which the device currently is or it may request to change it. The actor afterwards sends a response with an outcome. Devices that can be considered as members of this category might be a smart bulb, dimmer, thermostat or a smart window opener – Smarwi which is the main topic of this bachelor thesis.

Currently supported end devices are from vendors like:

- Jablotron
- Philips Hue 2.4
- Belkin 2.4
- Z-Wave
- and other devices supporting protocols such as Bluetooth or FIT Protocol [16]

The FIT protocol was designed by a student of Faculty of Information Technology at Brno University of Technology and is used for a communication with end devices developed at FIT VUT[16].

Figure 2.4: From left to right: Belkin Wemo Switch, Philips Hue [3] [4]

2.6 Testing center and virtual devices

Implementing new features to the system with always real sensors would be a hard and a time-consuming task, not only because it takes time for a sensor/actor to either measure data or change state but also some edge cases occur just occasionally.

Therefore, the testing center was developed along with the gateway which allows us to artificially emulate implemented sensors and actors for testing purposes. Once the device has been implemented for emulation, we can add it to our test cases with the parameters we set. Data can be either preset or randomly generated. Afterwards, device managers act as they would be in a regular mode.

Chapter 3

Remote window opening options

For centuries, humans strive to be more efficient and live more comfortably. Technology helps us to achieve greater things every day and allows us to be more productive. In return, it allows us to have extra spare time for relaxation. As described in the previous chapter, IoT and home automation can really make things easier for us.

This chapter will introduce problems and solutions for electric window openers controlled remotely. In the end solutions of different vendors are described.

3.1 Problems and requirements

Examples of common problems with windows that are eliminated with the use of smart windows:

- open windows that are in hard-to-reach places,
- be welcomed with fresh air when we enter a room,
- remotely create some airflow to currently unoccupied buildings,
- regulate room temperature with no need to waste resources either on the heater or A/C.

Smart or electric window openers were invented to help us with mentioned and many other problems. Here are the major requirements:

- windows open differently that means several mechanisms had to be invented,
- keep the initial cost low,
- not having to replace the whole window,
- quiet operation,
- installation in limited space,
- other problems with installation (e.g. can not drill, nonadhesive surface),
- easy to install.

3.2 Opening options

There were already many attempts to invent the most practical and universally fitting window regulator but it seems like there is no device to rule them all. There is a huge diversity of technologies for various use cases.

The most common form of solving this issue is using a window actuator, the main component to provide the force needed for a window movement. Window actuators are divided into three main types [12] and they are as follows.

Linear or spindle actuators

These devices apply a linear force to open windows usually upwards. They have great lifting force to open even heavy windows.

Such equipment has either a rack or a rod that can be extended or shortened according to user needs. The rack moves on a pinion that transforms rotational motion to linear. Linear actuators can be used in a wide variety of window openings such as louvers, dampers, awning windows, vents and external vertical louvers [13]. An example can be seen in figure 3.1.

Chain actuators

Both chain and linear actuators share the same fundamentals. A window is pushed upwards or to any other desired direction with a bar or in this case a chain. It is enclosed in a body of a window opener carefully winched over a pinion. As it rotates, chain segments are rotated at a 90-degree angle which afterwards form into a solid bar that can put a pressure on the window to open it up. Together with linear actuators, they are suitable for the majority of windows, however thanks to the characteristics of linear actuators, it is better to use the linear actuators for lifting heavy windows. On the contrary, thanks to the compact size of chain actuators they are widely used in the industry and at home applications. An example can be seen in figure 3.1.

Folding arm actuators

Instead of ejecting a rod straight they have an arm that pushes window sideways. They don't take as much space as other variants. They are usually meant for side hung windows as they don't have as much force as other types of actuators. An example can be seen in figure 3.1.

Figure 3.1: **Window actuators.** From left to right: linear actuator, chain actuator, folding arm actuator [7] [6] [5]

For comparison purposes, the data from Arens[1] are put into table 3.1 and are used to show differences in linear and chain actuators. We can clearly see, how linear actuators are much stronger but also have higher current requirements. On the other hand, compact size and flexibility of chain actuators allow them to have a much longer stroke. When the actuators with the best characteristics from each category are compared, we can see that the chain actuator can have a stroke length more than 3 times longer than the linear actuator. On the other hand, it can apply almost three times less force than the linear actuator.

Chain actuators	**Stroke[mm]**	**Force[N]**	**Current[A]**
Arens Compact	50-400	250	~0.6
UCS Vega Synchro	300-800	250	~0.7
UCS Quasar L	300-1000	300	~0.9
Linear actuators	**Stroke[mm]**	**Force[N]**	**Current[A]**
Arens Spindle	180, 300	650, 800	~0.8, ~1
UCS Ulysses	180, 300	650, 800	~0.8, ~1

Table 3.1: Comparison between the chain and linear actuators [13]

From table 3.1 we can observe chain actuators are suited for solutions where either long strokes have to be implemented or limited space restricts usage of linear actuators. In case the window requires more force than the actuator can provide, more actuators can be added to the same window thanks to synchronisation across all actuators. This way multiple actuators act as one stronger actuator but the pushing force is also better distributed on the window frame.

3.3 Existing products

As mentioned in the beginning, many companies developed various window actuators. Several examples were picked from each category, each with a unique solution. The main differences can be in the way and the direction it opens the window or in the power supply.

As the most classic example of windows actuators, the linear actuator developed by the company Teal Products [11] can be introduced. The description of linear actuators can be found in the previous section. Source of power is electricity provided by a cable leading to the actuator. They are provided with just a cable that can be attached to the controller of your choice. It can be either a simple switch or a more complex receiver connected to the network.

Another product is from a company named Solar Smart [10] that eliminates the need of cables with a set of batteries and a photovoltaic panel on the top of the actuator. Panel converts solar energy to electric power and recharges the batteries. As it is not dependent on power from the electrical grid, in case of a power outage, the window is not stuck in the position and can operate with no change. It is also equipped with a rain sensor, which can help to adjust the window to the proper position according to the weather.

Actuator from Fenestra [2] serves for a different type of windows as previously mentioned products. It allows to opening horizontally slid windows. Another benefit is a solar panel and a LiO battery that comes with the actuator to allow cable free installation. A position

[1] https://arens.com.au/electric-products/

of the window can be changed even when the Fenestra actuator has no power by manually moving the window.

All up to now presented products are solutions to already installed windows from which we want to make smart windows. The next generation of glass panels can look like Blickdomi[1]. It has built-in roller shades together with motors for controlling the window position and window sensors. It is also equipped with cameras and other sensors to detect a potentional intruder.

Chapter 4

Smarwi - a window opener

Another idea came from a small Czech innovator company called Vektiva and their Smarwi. It is a smart window opener which allows us to comfortably and remotely control window ventilation according to our needs or according to data measured by other sensors (e.g. CO2 sensor or temperature and humidity sensors). This way we no longer worry about an airflow at our homes because BeeeOn in combination with Smarwi can do that for us and we are always greeted at home with fresh air to breathe.

4.1 Vektiva

Vektiva is a small company located in the capital city of the Czech Republic. Started as a start-up company in 2015 with an idea of an inteligent device for controlling window ventilation. Fast forward to 2016 and they have several working prototypes available and started a platform vektiva.online[1] to allow users to remotely manage their Vektiva's devices.

Nowadays, they have Smarwi, as can be seen in the image 4.1, in serial production which slowly makes its' way to shops. They are also in the progress of making new sensors and other IoT products.

4.2 Control

Window regulation can be done either from remote locations or by a safety control box which is connected to Smarwi and acts as a switch for opening and closing window. For distant access, WebGUI is provided on a site vektiva.online where a user can register their Smarwis and manage them from any device that has a web browser and access to the Internet. After login, the site informs us about the status of registered devices and provides options to adjust settings to the window requirements. We can also add more devices if needed.

4.3 Principle of Smarwi functionality

The whole system that moves the window has two core elements and that is a ridge and Smarwi. Ridge is glued with double sided tape on top of the window frame (i.e. immovable part). One side of the ridge consists of cogs, the other is flat. It is attached with a hinge to

[1] https://vektiva.online

Figure 4.1: Smarwi and other accessories

properly fit in Smarwi. Smarwi is placed on top of the window (i.e. moveable part). Inside we can find a cogwheel that runs back and forth on the ridge.

Once everything is correctly installed and calibration is finished, Smarwi saves the calibrated distance between opened and closed positions of the window. As can be seen on image 4.2, Smarwi has 2 sensors. The ridge in/out sensor serves as a protective mechanism to ensure Smarwi stays on the window in case improper handling or conditions occur. The sensor pushes the ridge towards the cogwheel with a plastic slider lifted by a spring. If the cogwheel is locked and an extreme force is applied to the window, the spring is pushed downwards and Smarwi slides on the ridge. This mechanism ensures plastic parts don't break.

Smarwi checks pressure on the open/close sensor and when the sensor is slightly pushed down, Smarwi stops because it has reached the closed position.

4.4 Communication

Smarwi uses Wi-Fi to be able to communicate with other devices. It first creates an access point to which a user can connect to (default name prefix for Wi-Fi network (SSID) is `SWR-` followed by numbers and a default password is 12345678) and then access a user interface on the local IP address `192.168.1.1`. Since it's a website, it can be accessed by any device ranging from all kinds of mobile devices to desktop computers as long as it

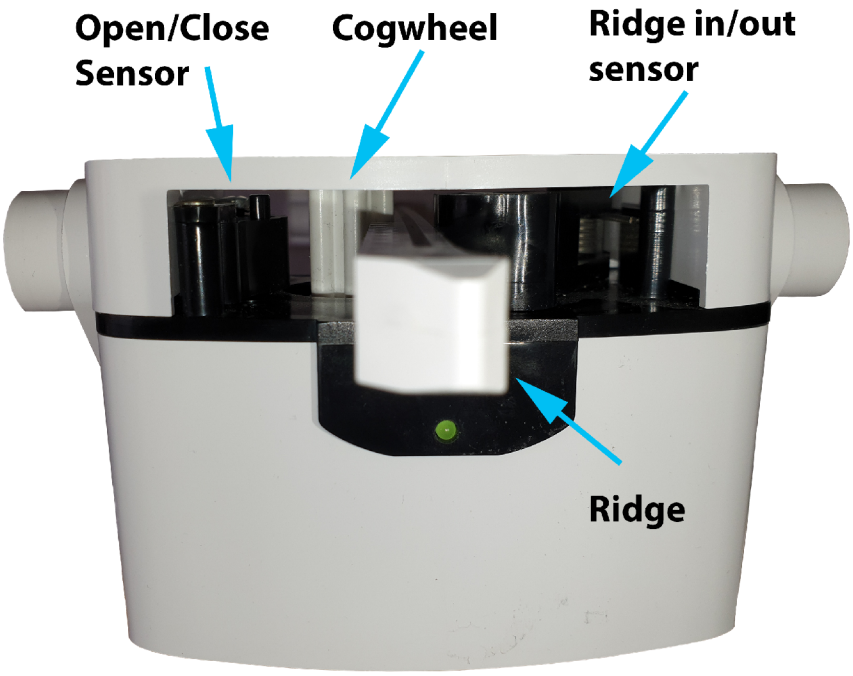

Figure 4.2: Front-facing view of Smarwi

has a web browser. Here users can properly configure credentials to successfully connect Smarwi to the Wi-Fi router, they want to and more importantly, they can set an IP address of the MQTT broker. They also have an option to supply RemoteID and RemoteKey to connect to the Vektiva portal, however, this action is not mandatory. Smarwi can fully work even with no connection to the Internet as Smarwi's state can be altered in three different ways and that is using an API (application programming interface) described in section 4.5, through MQTT messages to which Smarwi is subscribed described in 4.6 or through the physical button that should be placed next to the window where Smarwi is installed. Button not only serves as an option to open the window manually but also as a safety feature to stop Smarwi from currently performed action in case action is not desired.

4.5 HTTP API

Smarwi's application programming protocol communicates through the HTTP protocol which can be easily accessible with tools such as `curl` or simply a web browser. This is the easiest way to change the state of Smarwi. Communication uses basic request-response method and there are currently implemented two types of responses which have a simple text format. It can be either `OK` when an action is successful and `ERR` for other cases. Smarwi can be easily controlled via GET requests to following end-points:

- open – opens the window
- close – closes the window
- stop – stop any current action
- fix – fixes the window

- statusn – returns a status message
- lcfg – loads basic configuration
- lcfa – loads advanced configuration

The more advanced commands such as those that save configuration are POST requests encapsulated and then sent in MIME format to the Smarwi. These requests are more complicated to simulate and will not be discussed any more.

API can be accessed by both external and local network. Its' structure from **external network** is

`https://vektiva.online/api/<REMOTE_ID>/<API_KEY>/<DEVICE_ID>/<command>`
HTTP API call from an external network

where `REMOTE_ID`, `API_KEY`, `DEVICE_ID` are parameters that can be obtained from the Vektiva web interface.
Since WAN communication is implemented using MQTT messages, remote access through the Vektiva portal will not work if we change the default MQTT broker to a user defined one.
The command structure for a **local network** is:

`http://IP/cmd/<command>`
HTTP API call from a local network

where `IP` is an IP address of Smarwi. Parameter `command` is for controlling actuator's state. All possible options can be found in the product's documentation [14].

4.6 MQTT

MQTT [8] protocol or Message Queuing Telemetry Transport protocol is a messaging protocol that is designed to work even in unstable networks with limited bandwidth. Therefore, it fits well into an IoT field where devices send just small chunks of data periodically and a small delay isn't a problem. It works on publish-subscribe (`PUB/SUB`) model what means there is one MQTT broker which acts as a router and handles all messages sent to him. MQTT architecture is displayed on image 4.3.

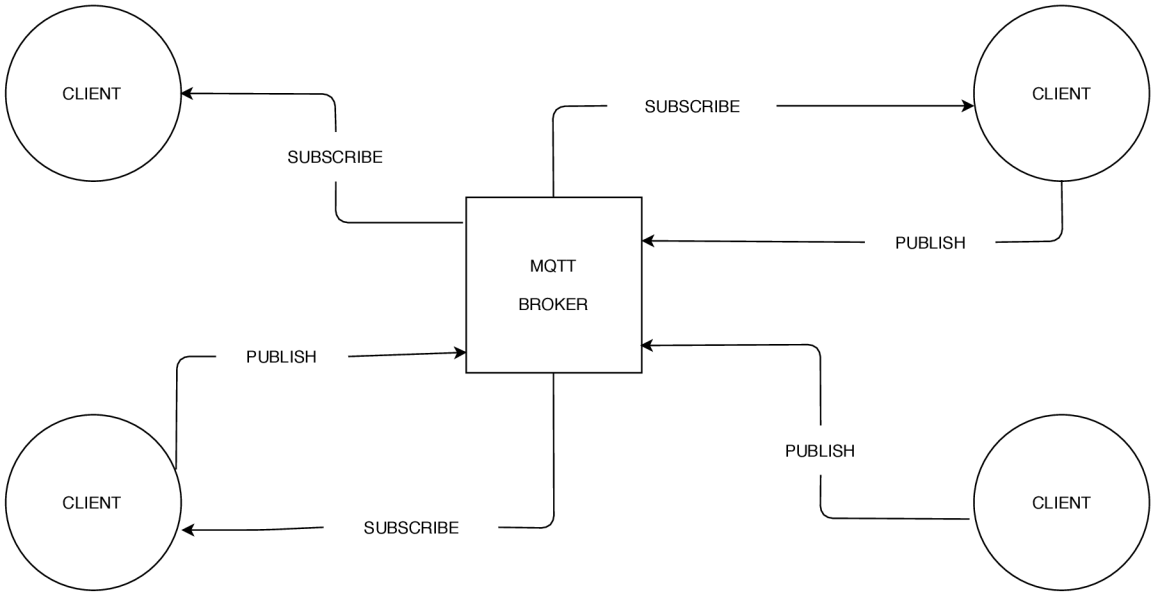

Figure 4.3: MQTT architecture

As we can see the architecture of MQTT is centralized to one broker and multiple clients connected to the broker. Clients can be either publishing or receiving messages or both of them simultaneously. Each client can be subscribed to a different topic or even multiple topics and therefore it receives only messages it intends to.

Topics are similar to what we are already used to in file systems and therefore it creates a sort of a path where the delimiter is a slash `'/'`. When subscribed to topics, wildcards `'#'` and `'+'` are available. Wildcard `'+'` can be used for single segment wildcard, whereas `'#'` can be used for all remaining segments and thus it is mandatory to only use it at the end of the subscribed topic. Several examples are provided to show how topic subscriptions work.

If a device publishes a message with topic `/ion/dowarogxby/%aabbccddeeff/status` and the client is subscribed to following topics, it:

- **#** – matches because it accepts every message it receives
- **ion/#** – matches because it accepts all topics that have the first segment equal to `ion`
- **ion/dowarogxby/+/status** – matches because it accepts all topics that start with `ion/dowarogxby/` and end with `/status`
- **ion/+/status** – doesn't match because the „+“ wildcard is only applicable to one segment
- **ion/** – doesn't match because it only matches topics that are only equal to ion
- **#/%aabbccddeeff** – invalid subscription

MQTT has a mechanism of retained messages and wills. Retained messages received by the broker stay stored in the broker storage and are sent to every subscription that matches the topic without the exception of the new subscriptions. One topic is allowed to

have one retained message, therefore if a new message that should be retained is received, the previous is replaced by the newer one. If a retained message needs to be deleted, an empty message should be sent to the intended topic. Wills are published by the broker when the client unexpectedly disconnects from the broker. For this mechanism, `PING` is implemented in the MQTT protocol. Will messages are specified when the client connects to the broker. Smarwi uses wills when is disconnected and the broker publishes a message to topics `online` and `status` to let other clients know it is not available anymore.

The communication as seen at 4.4 runs on the default port 1883 and Smarwi uses Vektiva's broker by default. Even though the port can not be altered, the user can define custom URL of the broker. Because of that, Smarwi can be easily controlled both locally and remotely via MQTT messages. Message's topic structure is:

`ion/<USER>/%<DEVICE_ID>`

Topic structure of Smarwi MQTT messages

where `USER` and `DEVICE_ID` can be found in Smarwi's WebGUI.

An instance of such communication is provided in the following lines. If a message is sent with topic „`ion/dowarogxby/%aabbccddeeff/cmd`“ and message „`status`“, Smarwi if connected to the MQTT broker and is not malfunctioned, responses with:

```
t:swr
s:250
e:0
ok:1
ro:0
pos:o
fix:0
a:-98
fw:3.4.1-15-g3d0f
mem:23704
up:8631362
ip:268446218
cid:xsismi01
rssi:-70
time:1550970087
wm:1
wp:1
wst:3
```

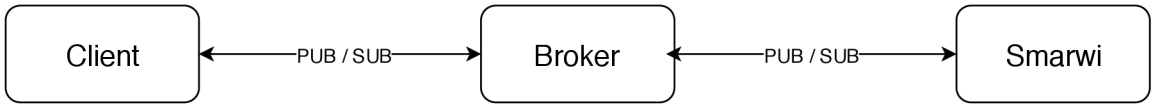

Figure 4.4: Communication between a client and Smarwi

Status message description

A brief explanation of the message can be found in the list below.

- t – device type (swr),
- s – state code,
- e – error code,
- ro – ridge out of the device (0 - in, 1 - out),
- ok – OK status → when OK == 1 then RO == 0 and vice versa,
- pos – window position (c - closed, o - open),
- fix – window position fixed (window fixed by device),
- ip – a 32bit number representing IP addres,
- cid – device name,
- rssi – signal strength,
- fw – device firmware version,
- time – device time in seconds,
- up – uptime of the device in milliseconds,
- wm – not documented,
- wp – not documented,
- wst – not documented,
- mem – not documented,
- a – not documented.

Smarwi Status codes

Smarwi produces a wide range of status codes which can be found in the status message in the „s“ field. In the end, we can find basic procedures of how are status codes in messages produced during some tasks.

- **200** – near frame opening,
- **210** – opening,
- **212** – closing but will open,
- **220** – closing,
- **230** – near frame closing,
- **232** – closing from the closed state, opens a little bit,
- **234** – closing from the closed state, closing,
- **250** – no action,

- **-1** – not calibrated not ready,
- **130** – closing window, finishing calibration,
- **10** – error has occurred.

When opening from the closed state, 200 and then 210 status codes are produced.
When opening from the open state, 212 and then 210 status codes are produced.
When closing from the open state, 220 and then 230 status codes are produced.
When closing from the closed state, 232, 234 and then 230 status codes are produced.
When changing states and outside button is pressed, the 250 status code is produced with OK and RO set to 0 and then OK set to 1.

Smarwi Errors

Smarwi also detects some errors which can occur during regular usage. Errors can be found in the status message in the „e“ field. Following errors were detected and found ways to reproduce them:

- **0** – no errors have occurred,
- **10** – window seems locked. To reproduce - press on ridge sensor, press on open/close sensor and send message open,
- **20** – movement timeout. To reproduce - set movement speed to 1 and let it open in long enough distance so the opening time will reach 30 seconds.

An overview of the complete MQTT communication is described in the appendix A. During configuration a domain name or an IP address of the broker can be entered but Smarwi uses a default port for MQTT that is 1883. In the current version QoS and secured MQTT transmission is not supported. Once connected Smarwi publishes retained status and online messages and last-will messages of the same types. Such messages can be seen in the example below. As mentioned earlier last-will messages are published by the broker when Smarwi does not respond to MQTT PING requests.

Topic: `'ion/dowarogxby/\%600194496fd2/online'`
Message: `'1'`
Topic: `'ion/dowarogxby/\%600194496fd2/status'`
Message:

```
t:swr
s:250
e:0
ok:1
ro:0
pos:o
fix:1
a:-98
...
```

Chapter 5

Integration design of Vektiva module

This chapter consists of the proposed implementation details that are planned for the next several months. Class diagram, design as well as other important parts of the BeeeOn Gateway are described here. At the beginning DeviceManager as the most important component is described with an explanation of the main functions that a device manager fulfill. Afterwards, a class diagram of the Vektiva module is described.

5.1 Device Manager

Each group of devices has to be implemented in a separate module for BeeeOn to be able to communicate with them. Each module has one main component and that is a DeviceManager. Each specific device manager act as a hub to receive commands and according to their meaning, it attempts to perform predefined implemented actions. For example, such actions can be changing device's state or collecting data. Gathered data are then shipped to exporters which are designed to handle sensor data accordingly. DeviceManager is an abstract class to provide a uniformed API following the principles of polymorphism. It implements basic functionalities such as command handling and calling appropriate methods that each module has to implement. For this reason, every module is based on the DeviceManager class.

5.2 Vektiva Device Manager

On image 5.1 we can see a class diagram of Vektiva module. *VektivaDeviceManager* inherits the abstract class of *DeviceManager* and implements all virtual methods needed to function properly such as *startSetValue*, *handleAccept* or *startUnpair*.

Part of *VektivaDeviceManager* is *VektivaSeeker* which is used to look for Vektiva devices such as Smarwi. When a new device is found, Gateway let the user know that he is able to pair the device. When a command for accepting the device is received, *VektivaDeviceManager* adds the device to the list of paired devices. Every time the Gateway starts up, it has an empty list of paired devices but once it connects to the server it retrieves all paired devices from the past. After obtaining the list of DeviceIDs, Vektiva module attempts to contact every paired device with a request for the status message. If the device responses in time, *VektivaSmarwi* class is instantiated and the device is added to the hash map of found

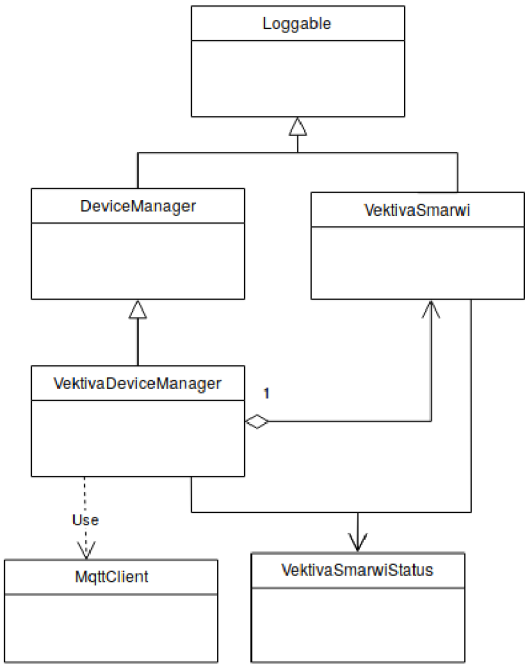

Figure 5.1: Proposed class diagram of the Vektiva module

		HTTP	**MQTT**
1	status changes propagated	✗	✓
2	LAN communication	✓	✓
3	WAN communication	✗	✓
4	easy to request basic command	✓	✓
5	easy to request advanced command	✗	✓
6	simple discovery	✗	✓
7	simple user configuration	✓	✗
8	small network overhead	✗	✓

Table 5.1: Comparison between HTTP and MQTT

devices. Hash map consists of key-value pairs where the key is a DeviceID and the value is an instance of the device. An instance of the device is then used throughout the lifetime of the Gateway app to manipulate with an actual Smarwi device. Apart from this initial process of finding already paired devices, the user can start the discovery of new devices at any time. The Gateway then requests status messages from all found devices and confirms their availability.

An instance of *MqttClient* provides an easy way to interact with Smarwi. *VektivaSmarwi* is a class which represents standalone Smarwi device and therefore it allows to manipulate with the actual device. *VektivaSmarwiStatus* is a class which reproduce a status message that is parsed and therefore is more accessible and is easier to manipulate with. As mentioned in section 4.4, Smarwi listens to commands received by HTTP and MQTT messages. Comparison is shown in table 5.1 and further explaination of every aspect included in the table is provided in the following list.

Plain TCP session	Outcoming bytes	Incoming bytes	Number of packets
HTTP	675	431	10
MQTT [Wi-Fi]	615	352	11
MQTT [Ethernet]	601	342	11

Table 5.2: MQTT vs HTTP performance tests from Flespi [15]

1. There are three ways how to change the state of Smarwi as described in 4.4. When a state change occurs (e.g. opening the window), Smarwi publishes an MQTT message no matter which way was used to change the state. As HTTP is a stateless protocol, it is impossible for Smarwi to let interested clients know that a change has happened. For this reason, to keep up-to-date information in which state Smarwi is, it is needed to periodically check the status by sending requests to Smarwi. Since the HTTP protocol defines one-to-one communication, every client that is interested in this information needs to contact Smarwi individually, which even more clutters the network. As a result, MQTT seems as a much better solution since it works on one-to-many basis and publishes a message after each status change.

2. Ability to communicate on a LAN network.

3. Ability to communicate on a WAN network. While it would be possible for HTTP (via port mapping in the router's settings), it's not straightforward to configure for an ordinary user.

4. Basic commands include: `open`, `close`, `stop`, `statusn`.

5. Advanced commands include `scfg`, `acfg`, `acfa` or others.

6. As previously discussed HTTP is a stateless protocol and thus when Smarwi connects to the network, it is not aware to whom it should send a message that it has connected. To find new devices, a scan of the whole network would be required which can be resource expensive. In the case of MQTT communication, as soon as the device connects to the network, it publishes *online* and *status* messages to let interested clients know, it is available.

7. When configuring Smarwi initially, the user has to input the network SSID and the password to be able to connect to the desired network. As soon as Smarwi is connected to the network, it can be controlled via HTTP API. To set up MQTT communication and connect Smarwi to the BeeeOn Gateway, the user has to change the MQTT broker address to the IP address of the device on which Gateway runs.

8. According to performance and power profiling tests done by Flespi [15] and Stephen Nicholas [17] MQTT requires 10 % less traffic than HTTP. That means it not only saves the bandwidth but also it allows bigger message throughput. Tests also show MQTT uses less power than HTTP client however, in case of Smarwi this is negligible. Examples of tables can be seen at 5.2 and 5.3.

Receiving	HTTPS	MQTT
Messages / Hour	3628	263314
Messages Received	524 / 1024	1024 / 1024
Sending	**HTTPS**	**MQTT**
Messages / Hour	5229	23184

Table 5.3: HTTPS vs MQTT comparison from Stephen Nicholas [17]

5.3 Communication

To better understand the way Smarwi works, reverse research engineering was conducted to reveal communication between a client and Smarwi and data that were captured, were written down and described in the best manner possible. Communication is almost completely described in the appendix A with little information missing that couldn't be deciphered. Reverse engineering was done by a tool named `Wireshark` and being in the middle of the communication between Vektiva broker and Smarwi. In other words, a computer with `Wireshark` created an access point to which Smarwi was connected to. After that, every possible option in the web interface was executed to reveal all messages that are involved in the communication. To have a better idea of how the communication worked please have a look at 5.2. The dashed arrows show a relationship between the two elements while solid bidirectional arrows show the flow of communication.

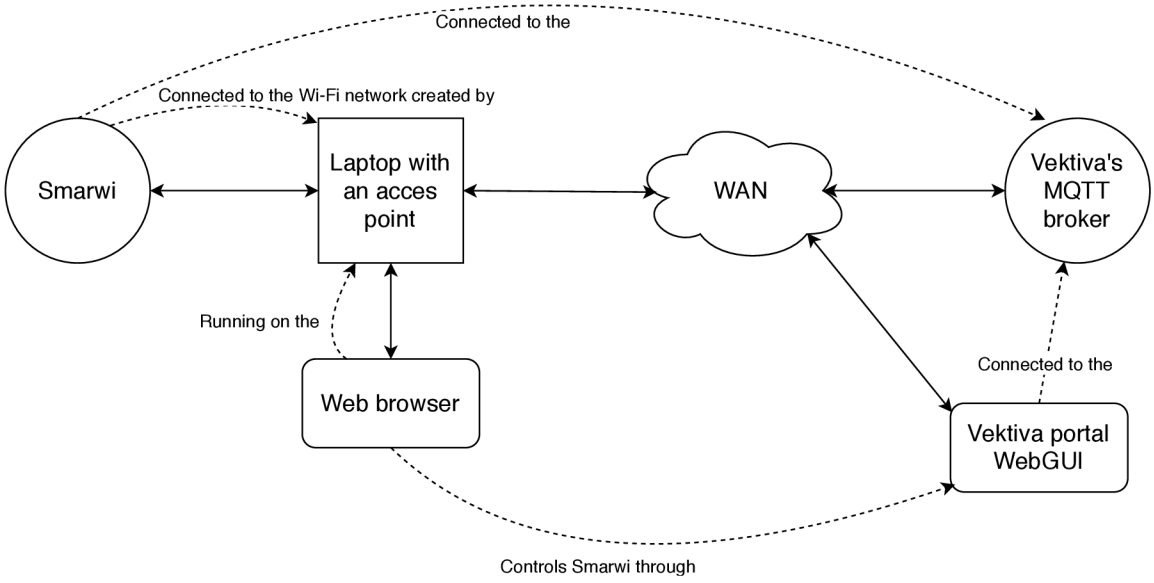

Figure 5.2: Communication setup during the process of reverse engineering

5.3.1 Vektiva's broker

As stated earlier in 4.6, communication is done on port 1883 and in case of Vektiva's default broker server[1] an authentication is required to be able to conect to their broker. As 1883

[1] `https://broker.vektiva.com`

port is the default for unencrypted MQTT communication, getting a username (*RemoteID*) and a password *RemoteKey* can be easily obtained if MITM attack is performed since they are transferred in a clear text form. With credentials a hacker can connect to the broker, however, the broker after successfull connection only allows to control devices or subscribe to topics that are assigned to the same RemoteID. That means if RemoteID is equal to „dowarogxby“, it only grants us access to the topic of *ion/dowarogxby/#*. This leads to exploiting all devices that the user has registered to the broker and a possible threat of altering windows to an undesired position.

An example of publishing a message to the Vektiva's broker is provided.

```
mosquitto_pub -h "broker.vektiva.com" -t "ion/REMOTEID/%MACADDRESS/LAST_SEGMENT"
 -m "MESSAGE" -u "REMOTEID" -P "REMOTEKEY" -r
```

5.3.2 Gateway's broker

Since the main purpose of the Gateway is to search for devices on a local network and control them, no authentication is required to connect to the broker. Communication is done on port 1883.

5.3.3 On connect

When Smarwi is attempting to connect to the specified broker, together with credentials it is sending the last-will message which is published by the broker, when Smarwi does not respond to the PING requests for an extended period of time. Once Smarwi successfully connects to the broker, it publishes retained `online` and `status` messages.

5.3.4 Proposal of communication design

As mentioned in 4.4, Smarwi is able to communicate both on local and wide-area networks, however in regards to network, pure LAN communication between the device and the Gateway is chosen as the best option since the network is not only less cluttered but also the communication is more reliable and to a greater extent more secure.

When deciding what protocol for communication to use table 5.1 was created to clearly presents the advantages of each protocol. According to the criteria presented in the table, MQTT is chosen as a way of communication.

Inside the `VektivaSmarwi` class there are 4 modules, one which is only an output module and the other 3 are controllable. The list of the modules with description is below where they are ordered respectively to their module ID starting from 0.

- `open/close` module – to either open or close the window completely,
- `open to` module – to open the window measured in percentage of the calibrated distance,
- `un/fix` module – to un/fix or to un/lock the window by Smarwi,
- `rssi` module – output module to provide information about the strength of the Wi-Fi signal.

Chapter 6

Implementation into the BeeeOn system

The chapter presents the results of the gradual progress of the final implementation and the testing that was conducted in each stage of the development. Several not so fortunate decisions were unveiled in the proposed plan as the development progressed but thanks to well timed code evaluations, they were corrected. As no code design is fault-proof, the proposed implementation plan was no exception although in many regards it was correct.

Development was conducted in series of iterations, where in the end of each iteration code review took a place. This process allowed effective communication and minimum misunderstandings. On average each iteration was long about 9–10 days. In the first half of iteration, time was reserved for fixing the code and implementing goals defined in the previous iteration. When work was done and tested, code was commited to the repository. The rest of the time was reserved for code review and additional discussion.

Several libraries has helped to avoid code repetition and speeded up the whole development process. The most used ones were *std* and *Poco* [9] C++ libraries. *Poco* consists of several modules that range from string manipulation through XML, JSON or ZIP manipulation to network and cryptographic operations.

6.1 Implemented parts of the proposed plan

As described in 5.2 the main function of *VektivaDeviceManager* is to control all paired Smarwis and search for other available devices. This remained unchanged together with the class diagram which can be seen at 5.1. For communication MQTT protocol was chosen as a more reliable and versatile option compared to the HTTP. Additionally, dependency diagram generated from the code at 6.1 can confirm class diagram has not changed.

6.2 Parts changed

During the development, several decisions were made where the actual implementation started to differ from the proposed plan.

VektivaDeviceManager has 2 instances of MQTT clients, one of which is constantly passively analyzing incoming messages and the second one controls individual Smarwis. Since MQTT messages in the current implementation are impossible to sort through and can only be buffered until they are returned from the client, it was clear that 2 instances are

needed. In case of one instance the problems such as finding a device while waiting to finish the operation might have occurred. To be more elaborate, it takes at least 3 seconds for the Gateway to receive the final message of the operation. Its' aim is the status message which signalises an idling state from the specific Smarwi. While waiting, several less important messages are thrown away and possibly with them a message about the new device in the network. As MQTT has no way of device discovery, the device restart would be necessary.

To further explain the first instance of MQTT client, if an online message of unknown Smarwi is received, it creates a *VektivaSmarwi* instance and adds it to the list of found devices. In another case, if a status message of paired Smarwi is received and the status code equals to the idling state (250), the Gateway ships the data to the exporters. The instance is only used for analyzing and never for publishing any messages. The process of analyzing the received MQTT message can be seen at diagram 6.2.

The second instance is solely used for manipulation with other Smarwis. To ensure one to one communication, the second instance is protected by mutex and topics and messages are filtered by a regular expression. Before any modification, a message buffer is cleared to make sure only the newest messages are analyzed. Afterwards, a command is sent to the specified device and the Gateway awaits for the confirmation message. In case message delivery timed out or a change led to failure, exceptions are thrown.

As mentioned in 5.2 part of the *VektivaDeviceManager* class is a seeker to add new devices. When the Gateway receives *command listen*, all found devices are contacted with a message requesting for their status response. `NewDeviceCommand` is then dispatched for every device that responds in time. This strategy can be also beneficial to determine the last time paired, infrequently used device was active.

6.3 Implementation tests

Most of the testing was manually performed as it's very hard to simulate network traffic and device behaviour in automated tests. Until the modified Gateway was able to communicate with the server, tests were conducted thanks to *TestingCenter* and an application `netcat`. *TestingCenter* as mentioned in the 2.6 act as a server for developers to verify newly implemented features. After communication with the server was established, the testing was done with the web browser. To ensure that message parsing is correct, several unit tests were written together with tests for MQTT message building and for validating MQTT message topic. These tests were implemented with respect to other tests in the testing framework `CppUnit`.

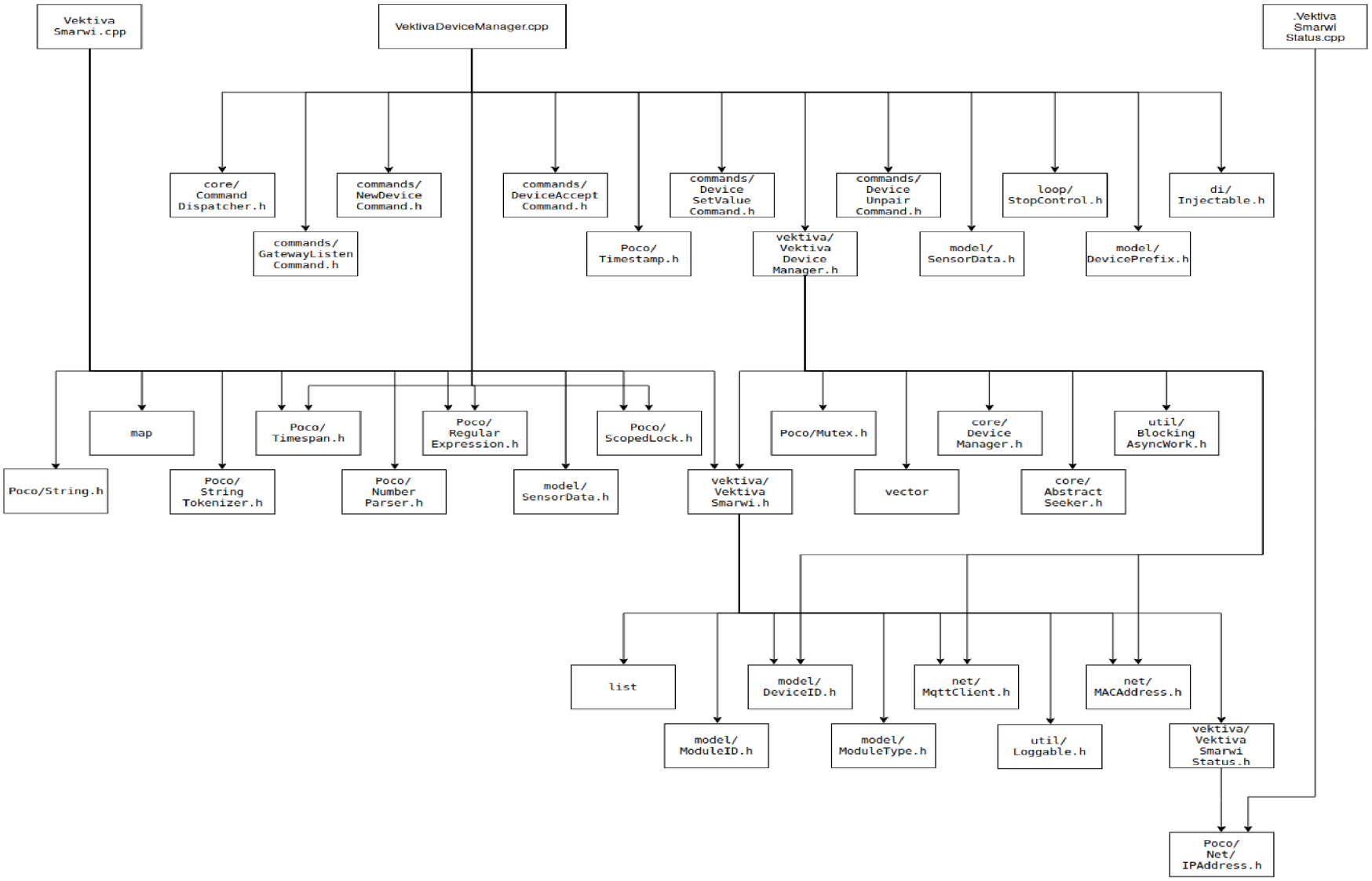

Figure 6.1: Dependency diagram of Vektiva module

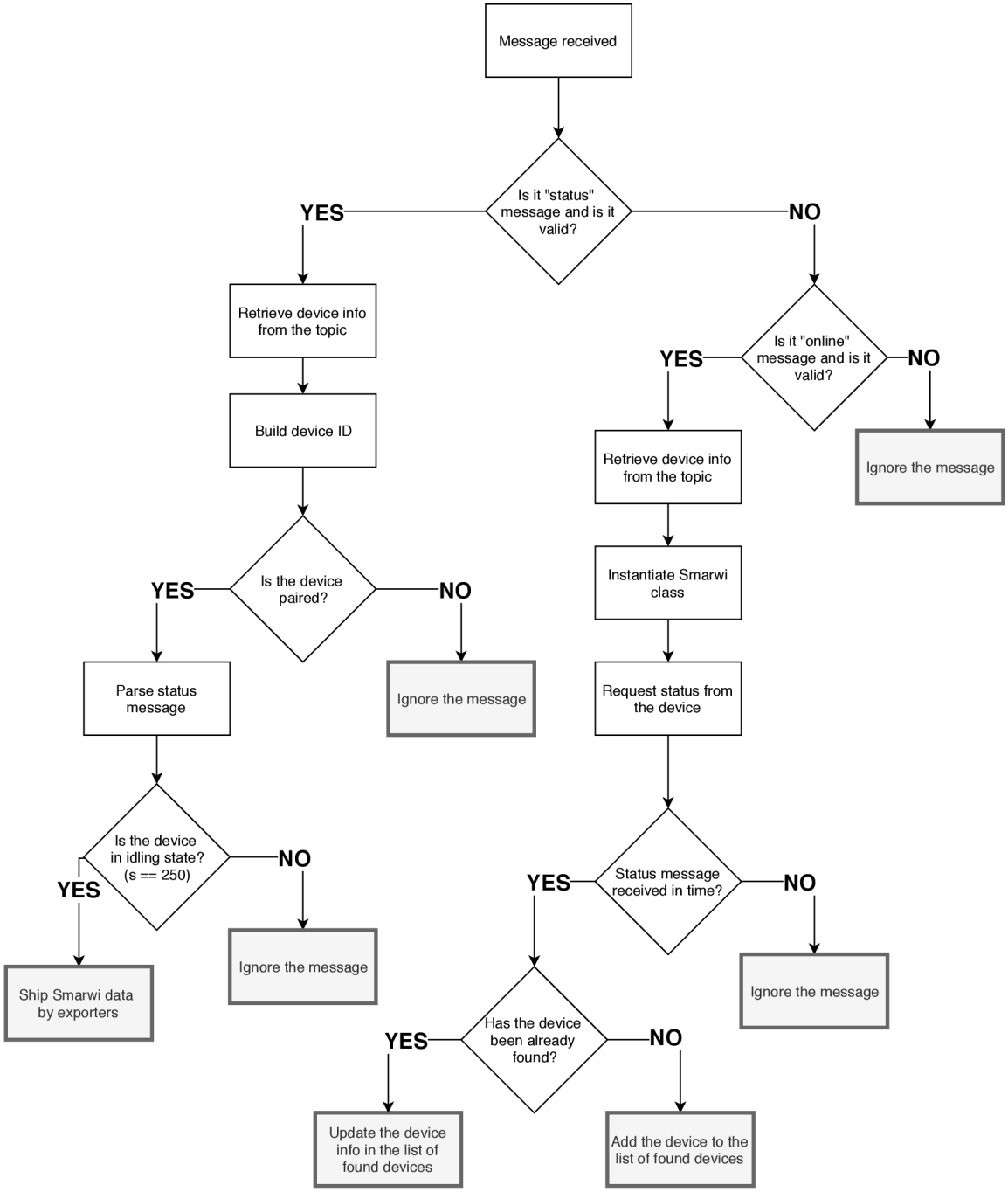

Figure 6.2: Decision tree of analyzing the received message

Chapter 7

Testing - Smarwi emulator

Often times, we can find ourselves in a situation when our work depends on something we don't possess at the time and therefore we are unable to continue, until we obtain it. It can range from credentials, through a file to a tangible thing. It can be possible to store all electronic information we need to small hard-drives but it can be usually challenging to always carry around bulky IoT devices. For this reason, emulators exists. They substitue devices that they are specifically designed to, by simulating their normal operation as close to reality as possible. Additionally, since running multiple instances of the emulator is generally cheaper than buying dozens of devices, they can be used in stress or load testing to prove the tested system is reliable and does not contain bugs. For this reason an emulator was built that is easy to control, has a wide variety of options and at the same time is easy enough to run on any platform. Vektiva Smarwi emulator provides a user interface from which a user is able to manage multiple instances, dynamically change the number of Smarwis and at the same time, they all act as standalone devices which communicate correctly with other programs similarly as Smarwi.

7.1 Implementation

Vektiva Smarwi emulator was implemented in Python 3 as a multithreaded application, providing a REST API to interact with the emulator. On top of that we can find a graphical user interface provided on a page hosted on the local server. For the user interface HTML5 together with CSS and jQuery as a Javascript framework. In the backend, Python 3 was used together with a set of libraries that helped to speed up the development process. From the main ones Paho client, Threading and socketserver can be mentioned.

7.1.1 Routing

To be able to recognize what action user wants from the emulator, it needs some decision making process, in this case it is implemented as a router. Since the whole application runs on a light HTTP server, it can be controlled via specified end points. Once the request is received by the HTTP server, the URL is parsed and the path is passed to the class Router. The path is divided into segments where the delimiter is a slash. If the first segment is correct, it is passed to the class *VektivaSmarwiHandler* for further processing. In case the first segment is not recognized by the Router, the main page is returned.

To have a better idea of how the emulator works, diagrams of decision trees can be seen at 7.2 and 7.3. Figure 7.2 shows the internal structure of the whole emulator. At the

beginning we can see HTTP request received by the server, which calls *Router* to further analyze the request. Following conditional control statements decide what is the outcome of the received request. In case the first segment of the URL contains a valid MAC address, the request is passed to the *VektivaSmarwiHandler* where a correct action is executed according to the request attributes.
Figure 7.3 shows the decision tree of one of the internal methods in the *Router*. It can represent the node "Return method output" in the 7.2. It shows the endpoint */devices* which have a REST interface to enable easy manipulation with Smarwis. In the first three conditional control statements, we are distinguishing between GET, POST and DELETE request methods. GET method returns an array of Smarwis serialized in JSON, POST request passes a request to the *VektivaSmarwiHandler* which validates data received in the body of the request and in case of success a new Smarwi is added to the list of running Smarwis. DELETE method removes the Smarwi from the list of existing Smarwis if a valid MAC address is in the second segment of the URL.

7.1.2 Smarwi handling

After the parsed path has been passed to the *VektivaSmarwiHandler*, according to other segments of the path, the correct action is performed. The class' main goal is to keep the manipulation and storing Smarwis separated from the routing process. It holds a list of created Smarwi instances and performs operations on either the whole list or a specific Smarwi.
General operations over Smarwi instances include:

- **adding** – creates a new Smarwi instance and adds it to the list,
- **deleting** – deletes the specified instance from the list,
- **retrieving** – retrieves the list of Smarwis with their status.

Specific Smarwi operations include:

- **open/close** – opening or closing the window,
- **on/off** – turning on or off Smarwi,
- **stop** – provides an immediate stop and unfixing the window,
- **fix** – fixing the Smarwi, which locks the window and can't be moved mechanically,
- **status** – provides the status of the Smarwi,
- **error** – schedules an error. This action does not throw an error state immediatelly but just after opening or closing begins as the real behaviour of Smarwi.

Following sections present a short explanation of how to use the respective parts of the emulator. Even though the emulator was designed to be as intuitive as possible, following sections will try to guide you when in doubt.

7.2 API endpoints

API provides an easy way to programmatically control the emulator. It is built in REST architectural style following the design guidelines. All CRUD operations can be found in table 7.1.

Request type	**Route**	**Description**
GET	/devices	lists all devices available in the emulator
POST	/devices	with correct JSON in the body creates new device
DELETE	/devices/<MAC>	deletes the device with the MAC address provided in the MAC parameter
GET	/<MAC>/open	opens the device specified in the MAC parameter
GET	/<MAC>/close	closes the device specified in the MAC parameter
GET	/<MAC>/on	turns on the device specified in the MAC parameter
GET	/<MAC>/off	turns off the device specified in the MAC parameter
GET	/<MAC>/error/<ERRNO>	schedule error on the device specified in the MAC parameter with the error number specified in the NUMBER parameter
GET	/<MAC>/stop	stops the current action carried out on the device specified in the MAC parameter and unfixes the window
GET	/<MAC>/fix	fixes the window by the device specified in the MAC parameter
GET	/<MAC>/status	publishes the status message of the device specified in the MAC parameter

Table 7.1: Documentation of an API of Vektiva Smarwi emulator

7.3 MQTT

Part of Vektiva Smarwi emulator is an MQTT client and it works by default on localhost however, this can be easily changed by providing a host name and a port to connect to. This enables using the emulator on the WAN network through MQTT messages. Adding new devices is however required to do locally either in the user interface or through API. Messages sent by the emulator are described in A. Each device can be controlled the same as Smarwi or similarly to the API commands. As an example, we can find a command to schedule an error on Smarwi.

```
mosquitto_pub -t "ion/dowaroxby/%aabbcceeddff/cmd" -m "error/20"
```

Scheduling an error over MQTT

7.4 User interface

As mentioned earlier, user interface built in HTML, CSS and a Javascript framework jQuery, provides an easy access to the main to control all emulated Smarwis. It is possible to work with the graphical interface even with no connection to the Internet, however as it is using jQuery it needs to be either loaded from a CDN or a local file. By default the CDN is used, however a user can put the jQuery file into the same folder as the emulator and name it accordingly to the HTML page.

In the image 7.1 in the top left corner we can see a *Reload* button to retrieve the status of all devices in the emulator. Next, we can see *Generate MAC:* checkbox which allows us to automatically generate MAC address when creating a new device. If we wish to input a specific MAC address we can use the text field next to the *MAC:* field. The last element is a *Create* button that creates a new device if the correct MAC address is provided. Below, we can find a list of all instantiated Smarwis that we can control.

- **online** – availability of Smarwi
  - **green** – online
  - **orange** – error occurred
  - **red** – offline
- **position** – position of Smarwi
  - **open**
  - **close**
- **mac address** – MAC address of Samrwi
- **actions** – available actions to Smarwi
  - **open** – opens the specific Smarwi
  - **close** – closes the specific Smarwi
  - **fix** – fixes the window by the specific Smarwi
  - **stop** – stops the current action the specific Smarwi and unfixes the window
  - **on** – turns on the specific Smarwi

- **off** – turns off the specific Smarwi
- **throw error** – schedules error on the specific Smarwi and throws error immediately as open/close action is performed
- **error schd** – notices that error is scheduled on the specific Smarwi, can be canceled by stop action
- **delete** – deletes the specific Smarwi

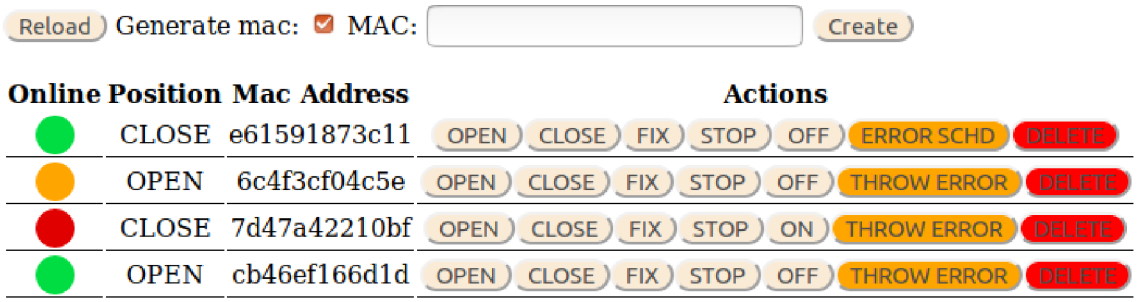

Figure 7.1: User interface of Smarwi emulator

7.5 Scheduling and fixing errors

As stated several times before, errors can not be thrown immediately after receiving a command to do so because it would not simulate real life operation of Smarwi. Instead, we can schedule an error and it is thrown as soon as emulator attempts to manipulate with the window with the specific Smarwi.
To follow Smarwi operation as close as possible, an error can be fixed by sending a `stop` command. It both fixes an existing error state and unschedule any error that is supposed to be thrown.

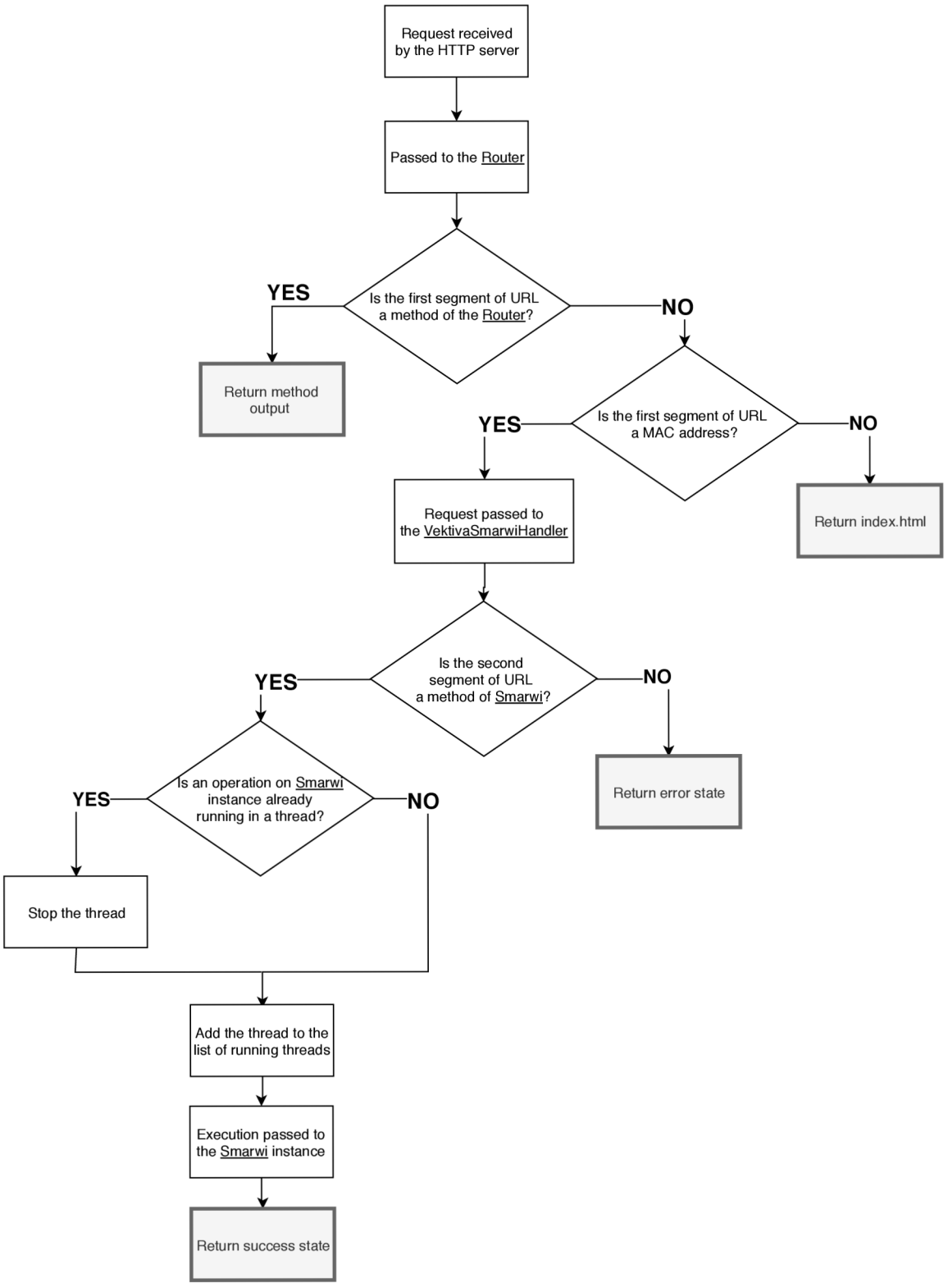

Figure 7.2: Decision tree of the emulator

Figure 7.3: Decision tree of the emulator's "/devices" endpoint

Chapter 8

Conclusion

In this thesis a successful implementation of a new Vektiva module was described which allowed a new device called Smarwi to be controlled from the BeeeOn interface. Smarwi, a name of the implemented device, is a smart window opener connected to the network with Wi-Fi and communicating with HTTP and MQTT.

BeeeOn system was the first thing that was needed to study to know, how it works and what parts it consists of. Parts of the system related to the implementation of new modules were a primary focus afterwards. After that, the possibilities of Smarwi implementation were researched and with gathered data a decision about the best strategy for Vektiva module implementation has been made. The module was designed to follow patterns of object-oriented programming and a set of standards established by the BeeeOn community. While MQTT and HTTP are available options to control Smarwi with, the whole communication with the BeeeOn Gateway uses only MQTT protocol as it had more advantages compared to HTTP. After reading the documentation of Smarwi and making it more comprehensive, usage of the MQTT client in the BeeeOn Gateway was explored to further understand possibilities of implementation.

Implementation was conducted in a series of iterations with an average duration of 10 days. Implemented parts were regularly tested either with manual testing or unit tests. Manual testing included work with the device over the network while unit tests were mostly testing message parsing and message creation.

To further verify the correctness of the implementation and to allow Vektiva Smarwi development even with no presence of an actual Smarwi, an emulator was developed with an intention to substitue Smarwi as best as possible. In the emulator multiple instances can be managed to fully simulate a network with Smarwis. It has proven the implementation of Vektiva module in Gateway works with multiple instances of Smarwi at the same time.

Bibliography

[1] *BlickDomi actuator*. [Online; Accessed 15.01.2019].
Retrieved from: https://www.blickdomi.com/blickdomi-compact.html

[2] *Horizontally sliding actuator*. [Online; Accessed 15.01.2019].
Retrieved from: http://www.smartfenestra.com/products/

[3] *Image of the arm-folding actuator*. [Online; Accessed 19.05.2019].
Retrieved from: https://www.belkin.com/uk/p/P-F7C027/

[4] *Image of the arm-folding actuator*. [Online; Accessed 19.05.2019].
Retrieved from:
https://www.aihome.my/product/philips-hue-single-bulb-a60-e27/

[5] *Image of the arm-folding actuator*. [Online; Accessed 06.03.2019].
Retrieved from:
https://cellcode.us/quotes/openers-window-electric-casement.html

[6] *Image of the chain actuator*. [Online; Accessed 05.03.2019].
Retrieved from:
https://www.window-openers.com/ack5-electric-chain-actuator/

[7] *Image of the linear actuator*. [Online; Accessed 05.03.2019].
Retrieved from: http://coral-home.over-blog.com/article-how-to-remote-control-skylight-window-116201643.html

[8] *Message Queuing Telemetry Transport*. [Online; Accessed 22.03.2019].
Retrieved from: http://mqtt.org/documentation

[9] *Poco, C++ library*. [Online; Accessed 22.03.2019].
Retrieved from: https://pocoproject.org/

[10] *Solar smart linear actuator*. [Online; Accessed 13.01.2019].
Retrieved from: https://www.solarsmartopener.com/store/

[11] *Teal products linear actuator*. [Online; Accessed 14.01.2019].
Retrieved from: https://www.tealproducts.com/product/actuators/linear-type/rack-pinion-actuators/mingardi-s1-rack-actuator

[12] *Types of electrical window actuators*. [Online; Accessed 01.05.2019].
Retrieved from: https://www.tealproducts.com/products/actuators

[13] *Use of linear actuators and comparison between linear and chain actuators.* [Online; Accessed 04.05.2019].
Retrieved from: https://arens.com.au/electric-products/

[14] *Vektiva API documentation.* [Online; Accessed 19.01.2019].
Retrieved from: https://vektiva.gitlab.io/vektivadocs/api/api.html

[15] Bartnitsky, J.: *MQTT vs HTTP performance tests.* Jan 2018. [Online; Accessed 24.02.2019].
Retrieved from: https://flespi.com/blog/http-vs-mqtt-performance-tests

[16] Nečasová, K.: *Extension of wireless sensor protocol.* May 2017. [Online; Accessed 28.12.2018].
Retrieved from: https://www.vutbr.cz/www_base/zav_prace_soubor_verejne.php?file_id=159153

[17] Nicholas, S.: *MQTT vs HTTP comparison.* May 2012. [Online; Accessed 22.02.2019].
Retrieved from:
http://stephendnicholas.com/posts/power-profiling-mqtt-vs-https

[18] Čížek, J.: *Na brněnské FIT se rodí BeeeOn. Univerzální chytrá domácnost.* February 2016. [Online; Accessed 28.11.2018].
Retrieved from: https://www.zive.cz/clanky/na-brnenske-fit-se-rodi-beeeon-univerzalni-chytra-domacnost/sc-3-a-181423/default.aspx

Appendix A

Smarwi MQTT communication

A.1 MQTT messages

Settings->Info

Command

`stat`

Description

Status of Smarwi, which consists of multiple fields is returned.

- t – device type (swr)
- s – state code
- e – error code
- ro – ridge out of the device (0 - in, 1 - out)
- ok – OK status → when OK == 1 then RO == 0 and vice versa.
- pos – window position (c - closed, o - open)
- fix – window position fixed (window fixed by device)
- ip – 32bit number representing IP addres
- cid – device name
- rssi – signal strength
- fw – device firmware version
- time – device time in seconds
- up – uptime of the device in milliseconds
- wm – unknown
- wp – unknown

- wst – unknown
- mem – unknown
- a – unknown

Sample return

Topic: `'ion/dowarogxby/\%600194496fd2/status'`
Message:

```
t:swr
s:250
e:0
ok:1
ro:0
pos:c
fix:1
a:-98
fw:3.4.1-15-g3d0f
mem:25416
up:477890
ip:268446218
cid:xsismi01
rssi:-58
time:1550968238
wm:1
wp:1
wst:3
```

Settings->Basic

Command

`lcfg`

Description

Loads a basic configuration.

- ssid – name of the Wi-Fi network Smarwi is connected to
- pass – password used to authenticate when connecting to Wi-Fi network
- ssidap – name of the Wi-Fi network that Smarwi creates when set to AP mode
- passap – password for connecting to the Wi-Fi network created by Smarwi
- mode – current mode in which Smarwi is (cli - client, ap - access point)
- dst – Daylight saving mode (more described in Settings->Time)
- zone – Time zone in which Smarwi is (more described in Settings->Time)

- wsleep – hours in which Wi-Fi network (that Smarwi is connected to) is planned to be turned off. (more described in Settings->Advanced)
- mqttsvr – MQTT broker for communicating with Smarwi
- mqttuser – known as RemoteID, user ID according to which Smarwis can be assigned to one user
- mqttpass – RemoteKEY
- mqttport – port to which Smarwi connects to when communicating with MQTT broker
- swrname – device name
- lat – latitude (more described in Settings->Advanced)
- lon – longitude (more described in Settings->Advanced)
- phym – WiFi PHY Mode (more described in Settings->Advanced)
- unst – unknown
- sunalgo – unknown
- mqttka – unknown
- rsetup – unknown
- rasetup – unknown

Sample return

Topic: `'ion/dowarogxby/\%600194496fd2/config/basic'`
Message:

```
ssid:luksPC
pass:A0CqjsTF
ssidap:SWR-496fd2
passap:12345678
mode:cli
dst:1
zone:60
rsetup:1
rasetup:1
wsleep:0
mqttsvr:10.42.0.1
mqttuser:dowarogxby
mqttpass:53214716
mqttport:1883
mqttka:30
swrname:xsismi01
lat:50.088001
```

```
lon:14.420000
sunalgo:0
phym:b
unst:0
```

Settings->Basic->Select Wifi Network

Command

```
scan
```

Description

Once clicked on dropdown named „Select Wifi network“, Smarwi scans for available networks in his surrounding area and shows it appropriately in a dropdown menu.

Sample return

Topic: `'ion/dowarogxby/\%600194496fd2/wlist'`
Message:

```
-30|Vodafone-4AF188|9
-45|luksPC|1
-76|MEO-WiFi|6
-78|MEO-18CF7D|6
-83|Vodafone-23280A|3
-85|Vodafone-DE58E8|1
-89|MEO-A2F510|1
-89|MEO-WiFi|1
-89|JCM|8
-89|Vodafone-CF6DB0|5
-90|Vodafone-25F158|3
```

Settings->Basic->Save

Command

```
scfg01/1|swrname:xsismi01
ssid:luksPC
ssidap:SWR-496fd2
mode:cli
mqttuser:dowarogxby
mqttpass:53214716
```

Description

After button is clicked, it sends an actual state of configuration to Smarwi to save.

Sample return

Topic: `'ion/dowarogxby/\%600194496fd2/online'`
Message:

```
0
```

Settings->Basic->Apply

Command

```
acfg01/1|swrname:xsismi01
ssid:luksPC
ssidap:SWR-496fd2
mode:ap
mqttuser:dowarogxby
mqttpass:53214716
```

Description

Applies currently set settings to Smarwi without saving them. When „mode“ is set to „ap“, Smarwi switches to access point mode. If password is not specified, Smarwi uses default „12345678“

Sample return

No message is returned.

Settings->Basic->Reboot

Command

```
boot
```

Description

Smarwi is rebooted.

Sample return

No message is returned.

Settings->Advanced

Command

```
lcfg
```

Description

Loads configuration of Smarwi.

Sample return

Same as Settings->Basic

Settings->Advanced->Apply

Command

```
acfg01/1|mqttsvr:broker.vektiva.com
wsleep:2113665
lat:50.088002
lon:14.420001
phym:g
```

Description

Applies settings currently set on configuration „Advanced“ page.

- wsleep states for Wi-Fi off hours, where hours in which Wi-Fi will be off can be selected. They are in binary order where 0 o'clock is an least significant bit. 2113665 as a decimal or 10000010000000010000001 in binary therefore selects 0, 7, 14, 21 hours as Wi-Fi off hours.
- Phym states for WiFi PHY Mode. 3 options are available and that is:
-
  - 802.11b (set as b)
  - 802.11g (set as g)
  - 802.11n (set as n)
- lat states for latitude (N/S) in degrees
- lon states for longitude (E/W) in degrees

Sample return

No message is returned.

Settings->Advanced->Save

Command

```
scfg01/1|mqttsvr:broker.vektiva.com
wsleep:2113665
lat:50.088002
lon:14.420001
phym:b
```

Description

Saves settings from „Advanced“ tab to Smarwi.

Sample return

No message is returned.

Settings->Finetune

Command

`lcfa`

Description

Loads a finetune configuration of Smarwi. In this configuration changes related to movement speed, power or window position can be made.

- vpct – Maximum open position
- ospd – Movement speed
- ofspd – Near frame speed
- orpwr – Movement power
- ofpwr – Near frame power
- ohcpwr – Closed holding power
- ohopwr – Open holding power
- hdist – Window closed position finetune
- lwid – Window locked error trigger
- cfdist – Calibrated distance
- cvdist – unknown

Sample return

Topic: `'ion/dowarogxby/\%600194496fd2/config/advanced'`
Message:

```
vpct:97
ospd:85
ofspd:3
orpwr:77
ofpwr:5
ohcpwr:46
ohopwr:41
hdist:-2
lwid:23
cfdist:2560
cvdist:79488
```

Settings->Finetune->Apply

Command

```
acfg01/1|ospd:87
ofspd:4
orpwr:79
ofpwr:7
ohcpwr:47
ohopwr:43
hdist:-3
lwid:24
vpct:95
cfdist:2560
```

Description

Applies finetune settings to Smarwi configuration.

Sample return

No message is returned.

Settings->Finetune->Save

Command

```
scfa01/1|ospd:87
ofspd:4
orpwr:79
ofpwr:7
ohcpwr:47
ohopwr:43
hdist:-3
lwid:24
vpct:95
cfdist:2560
```

Description

Saves finetune settings in Smarwi. Settings are described in A.1.

Sample return

No message is returned.

Settings->Finetune->Reset to defaults

Command

```
rcfa
```

Description

Resets finetune settings to defaults except field named Calibrated Distance. Therefore recalibration is not needed.

Sample return

Topic: `'ion/dowarogxby/\%600194496fd2/config/advanced'`
Message:

```
vpct:100
ospd:40
ofspd:40
orpwr:50
ofpwr:60
ohcpwr:60
ohopwr:30
hdist:0
lwid:20
cfdist:2560
cvdist:79488
```

Settings->Calibration->Step1

Command

`stab`

Description

Window position is saved in its' closest position to window frame while we push it towards a window frame.
Return message described in A.1.

Sample return

Topic: `'ion/dowarogxby/\%600194496fd2/status'`
Message:

```
t:swr
s:250
e:0
ok:1
ro:0
pos:c
fix:1
a:-98
fw:3.4.1-15-g3d0f
mem:25240
up:6041397
ip:268446218
```

```
cid:xsismi01
rssi:-59
time:1550957092
wm:1
wp:3
wst:3
```

Settings->Calibration->Step2

Command

```
cstart
```

Description

Window position is recorded in its' loose state or a state in which window stays relatively closed without applying any force to it.
Return message described in A.1.

Sample return

Topic: `'ion/dowarogxby/\%600194496fd2/status'`
Message:

```
t:swr
s:110
e:0
ok:0
ro:0
pos:c
fix:1
a:-98
fw:3.4.1-15-g3d0f
mem:25240
up:6121679
ip:268446218
cid:xsismi01
rssi:-57
time:1550957172
wm:1
wp:3
wst:3
```

Settings->Calibration->Step3

Command

```
cend
```

Description

We open window to a desired open state. Smarwi measure the maximum open position. Return message described in A.1.

Sample return

Topic: `'ion/dowarogxby/\%600194496fd2/status'`
Message:

```
t:swr
s:130
e:0
ok:0
ro:0
pos:o
fix:1
a:-98
fw:3.4.1-15-g3d0f
mem:25240
up:6197250
ip:268446218
cid:xsismi01
rssi:-55
time:1550957248
wm:1
wp:3
wst:3
```

Settings->Time->Save settings

Command

```
scfg01/1|dst:2
zone:210
ntp:pt.pool.ntp.org
```

Description

Saves settings related to time to Smarwi. „dst“ means Daylight saving mode. Possible values are:

- 0 – DST off
- 60 – DST on +1h
- 30 – DST on +30min
- 1 – Auto EU
- 2 – Auto USA

„zone“ means what time zone is selected. Value of 0 means time zone is set to UTC. According to this time zone value can be either more, equal or less than zero. Value in this field means how many minutes it is distant from UTC time zone. Values range from -720 to +765. Possible values are:

- 0 – UTC zone
- 60 – +1h
- 120 – +2h
- 180 – +3h
- 210 – +3h 30min
- -60 – -1h

„ntp“ is an NTP server.

Fields „dst“ and „zone“ are loaded from Smarwi config/basic(„lcfg“) message, while field „ntp“ is most likely an artificial field, which is not loaded from anywhere and is just added by Javascript.

Sample return

No message is returned.

Settings->Update->Update Firmware

Command

`updf`

Description

Proceeds to update firmware in Smarwi if any update is available.
Return message described in A.1.

Sample return

Topic: `'ion/dowarogxby/\%600194496fd2/status'`
Message:

```
t:swr
s:250
e:0
ok:1
ro:0
pos:c
fix:1
a:-98
fw:3.4.1-15-g3d0f
mem:25560
up:329119
```

```
ip:268446218
cid:xsismi01
rssi:-53
time:1550968090
wm:1
wp:1
wst:3
ur:2
```

A.1.1 Open

Command

```
open;100
```

Description

Opens window on which Smarwi is installed. Parameter after semicolon is optional and sets the number of percent Parameter should be in range from 0 to 100, otherwise unexpected behaviour happen.
Return message described in A.1.

Sample return

Topic: `'ion/dowarogxby/\%600194496fd2/status'`
Message:

```
t:swr
s:212
e:0
ok:1
ro:0
pos:o
fix:1
a:-98
fw:3.4.1-15-g3d0f
mem:23568
up:8257915
ip:268446218
cid:xsismi01
rssi:-72
time:1550969713
wm:1
wp:1
wst:3
```

A.1.2 Close

Command

```
close
```

Description

Attempts to close window on which Smarwi is installed.
Return message described in A.1.

Sample return

Topic: `'ion/dowarogxby/\%600194496fd2/status'`
Message:

```
s:220
e:0
ok:1
ro:0
pos:o
fix:1
a:-98
fw:3.4.1-15-g3d0f
mem:23680
up:8526574
ip:268446218
cid:xsismi01
rssi:-67
time:1550969982
wm:1
wp:1
wst:3
```

A.1.3 Stop

Command

`stop`

Description

Immediately stop the current process of opening or closing window.
Return message described in A.1.

Sample return

Topic: `'ion/dowarogxby/\%600194496fd2/status'`
Message:

```
t:swr
s:250
e:0
ok:1
ro:0
pos:o
fix:0
```

```
a:-98
fw:3.4.1-15-g3d0f
mem:23704
up:8631362
ip:268446218
cid:xsismi01
rssi:-70
time:1550970087
wm:1
wp:1
wst:3
```

A.2 Smarwi Status codes

Smarwi produces wide range of status codes. They can be found in the status message A.1 in the „s“ field. To better understand them, a list of them was created with a brief explanation. In the end we can find basic procedures of how are status codes in messages produced during some tasks.

- **200** – near frame opening
- **210** – opening
- **212** – closing but will open
- **220** – closing
- **230** – near frame closing
- **232** – closing from closed state, opens a little bit
- **234** – closing from closed state, closing
- **250** – no action
- **-1** – not calibrated not ready
- **130** – closing window, finishing calibration
- **10** – error

When opening from closed state, 200 and then 210 status codes are produced. When opening from open state, 212 and then 210 status codes are produced. When closing from open state, 220 and then 230 status codes are produced. When closing from closed state, 232, 234 and then 230 status codes are produced.

When changing states and outside button is pressed, 250 status code is produced with OK and RO set to 0 and then OK set to 1.

A.3 Smarwi Errors

Smarwi also detect some errors which can occur during regular usage. Errors can be found in the status message A.1 in the „e“ field. Following errors were detected and found ways to reproduce them:

- **0** – no errors
- **10** – window seems locked. To reproduce - press on ridge sensor, press on open/close sensor and send message open
- **20** – movement timeout. To reproduce - set movement speed to 1 and let it open in long enough distance so the opening time will reach 30 seconds.

A.4 Mosquitto examples

In following examples we will assume that mosquitto broker is hosted on an address `BrokerIP` and Smarwi listens to topic `ion/RemoteID/%SmarwiMAC/cmd`.
To change state of Smarwi we send a following command:

```
mosquitto_pub -h BrokerIP -t "ion/RemoteID/%SmarwiMAC/cmd" -m $'content'
```

As Smarwi will change states, it will send several messages one of them described in A.1
To save configuration to Smarwi following command will work:

```
mosquitto_pub -h 10.42.0.1 -t "ion/dowarogxby/%600194496fd2/cmd"
-m $'scfg01/1|ospd:90\nofspd:3\norpwr:77\nofpwr:5\nohcpwr:46\n
ohopwr:41\nhdist:-2\nlwid:23\nvpct:100\ncfdist:2560\n'
```

There is no return message for this command.
To see all messages that were exchanged between clients and Smarwi we can subscribe to the broker such as:

```
mosquitto_sub -h BrokerIP -t "ion/RemoteID/%SmarwiMAC/#" -v
```

Appendix B

CD contents

- Thesis in PDF and TEX format.
- BeeeOn Gateway app with implemented Vektiva module.
- Installation script.
- Smarwi Emulator.
- Video example of achieved results.