BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

# INTERNET OF THINGS DEVICE BASED ON ZIGBEE AND 6LOWPAN

INTERNET OF THINGS ZAŘÍZENÍ S PODPOROU ZIGBEE A 6LOWPAN

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR Bc. DÁVID HALÁSZ
AUTOR PRÁCE

SUPERVISOR Ing. PETR MUSIL
VEDOUCÍ PRÁCE

BRNO 2016

*Zadání diplomové práce/18445/2015/xhalas06*

**Vysoké učení technické v Brně - Fakulta informačních technologií**

Ústav počítačové grafiky a multimédií                    Akademický rok 2015/2016

# Zadání diplomové práce

| | |
|---|---|
| Řešitel: | **Halász Dávid, Bc.** |
| Obor: | Počítačové a vestavěné systémy |
| Téma: | **Internet of Things zařízení s podporou ZigBee a 6LoWPAN** **Internet of Things Device Based on ZigBee and 6LoWPAN** |
| Kategorie: | Vestavěné systémy |

Pokyny:

1. Prostudujte dostupnou literaturu týkájící se fenomenu Internetu věcí (Internet of Things - IoT).
2. Seznamte se s používanými komunikačnimi rozhraními a protokoly využitými v IoT. Zaměřte se na rozhraní ZigBee a protokol 6LoWPAN.
3. Seznamte se s tematikou Cloud computingu
4. Navrhněte zařízení spadající do kategorie IoT.
5. Zařízení realizujte a otestujte.
6. Vytvořte webovu aplikaci v cloud prostředí pro správu IoT zařízení.
7. Zhodnoťte výsledky práce a diskutujte případné pokračování práce.

Literatura:

- Dle pokynů vedoucího

Při obhajobě semestrální části projektu je požadováno:

- Body 1 až 3 zadání

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

| | |
|---|---|
| Vedoucí: | **Musil Petr, Ing.**, UPGM FIT VUT |
| Datum zadání: | 1. listopadu 2015 |
| Datum odevzdání: | 25. května 2016 |

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačové grafiky a multimédií
612 66 Brno, Božetěchova 2

[illegible]

doc. Dr. Ing. Jan Černocký
*vedoucí ústavu*

## Abstract

Internet of Things is the latest phenomenon in the computing industry. Even if it has not been completely defined yet, we are already surrounded by various devices connected to the Internet. This thesis project focuses on low cost and low-power wireless solutions and on the on-line backend behind the architecture. At the same time the present work also deals with Cloud Computing which can provide a highly scalable runtime environment for this backend without building an infrastructure. To handle the huge amount of data collected by billions of devices, BigData services could be used in the same cloud space. The project is a collection of the theoretical background of the Internet of Things; so as a result, it provides the reader with an overview of the concept. It also provides a walktrough of the design, implementation and testing process of a complex agricultural Internet of Things solution.

## Abstrakt

Internet věcí je nejnovější trend v počítačovém průmyslu. I když ještě nebyl zcela jasně definován, jsme již obklopeni různými zařízeními připojenými k internetu. Tato diplomová práce se zaměřuje na nízkonákladová a úsporná bezdrátová řešení a na on-line backend v pozadí této architektury. Zároveň se tato práce zabývá Cloud Computingem, který je schopen poskytnout vysoce škálovatelné prostředí pro běh tohoto backendu bez budování infrastruktury. Aby bylo možné zvládnout obrovská množství dat poskytnutých miliardami zařízeními, dalo by se využít služeb BigData v tom stejném prostředí cloudu. Projekt shrnuje teoretické pozadí konceptu Internetu věcí na základě dostupných materiálů. Výsledkem výzkumu je přehled konceptu, který poskytuje popis procesu návrhu, implementace a testování komplexního zemědělského řešení pro internet věcí.

## Keywords

Internet of Things, Wireless Low Power Networking, Bluetooth Low Energy, ZigBee, 6LoWPAN, CoAP, MQTT, Cloud Computing, Big Data, SensorTag, Arduino, Raspberry Pi, Sensors, Containers, Web Applications

## Klíčová slova

Internet věcí, Bezdrátové úsporné sítě, Bluetooth Low Energy, ZigBee, 6LoWPAN, CoAP, MQTT, Cloud Computing, Big Data, SensorTag, Arduino, Raspberry Pi, Senzory, Kontejnery, Webové aplikace

## Reference

HALÁSZ, Dávid. *Internet of Things Device Based on ZigBee and 6LoWPAN.* Brno, 2016. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Musil Petr.

# Internet of Things Device Based on ZigBee and 6LoWPAN

## Declaration

Hereby I declare, that this thesis has been written by myself under the supervision of Ing. Petr Musil. I have included all the sources and publications that were used to write this thesis.

. . . . . . . . . . . . . . . . . . . . . . .
Dávid Halász
May 23, 2016

# Contents

# Chapter 1

# Introduction

The concept of the Internet of Things (or shortly IoT) was first drawn up by Kevin Ashton. He defined it as “a global network of RFID and other sensors” [5]. However, the concept did not have any technological background at that time. After the rapid growth of the smartphone industry, the hardware manufacturers have started to invest in developing low-power microprocessors and wireless network adapters to maximize battery life. These technologies have made a solid ground not just for smartphones with higher computation power, but for various other devices too. Even though an IoT architecture has not been standardized yet, various manufacturers are already developing their own product palettes.

The thesis tries to summarize the main aspects of an IoT ecosystem. Based on these aspects, it provides an overview of the design and the implementation process of a production-ready sensor network. The main goal of this design was to provide a scalable backend infrastructure for a theoretically unlimited number of sensing devices.

The concept of the IoT as well as an overview of a possible architecture are discussed in the second chapter. The third chapter introduces the protocols that can be used to achieve wireless connectivity among devices. Higher-level protocols for data collection and transmission are also presented in this chapter. The fourth chapter is going to present the server-side, i.e. the on-line infrastructure behind the IoT. In the fifth chapter, the design and the implementation process of a complete IoT system is presented. The testing and verification of this design is described in the sixth chapter. The last part will summarize the findings and provide a conclusion and a reflection.

# Chapter 2

# Internet of Things

This chapter introduces the reader to the concept of the Internet of Things. It describes a mapping between the physical and the virtual devices and specifies a reference architecture.

## 2.1 The IoT concept

The term Internet of Things (IoT) is more of a concept than a well specified set of technologies. "It can be perceived as a far-reaching vision with technological and societal implications. From the perspective of technical standardization, the IoT can be viewed as a global infrastructure for the information society, enabling advanced services by interconnecting things based on existing and evolving interoperable information and communication technologies. Through the exploitation of identification, data capture, processing and communication capabilities, the IoT makes full use of "things" to offer services to all kinds of applications, whilst ensuring that security and privacy requirements are fulfilled." [13] It opens a new dimension into the information and the communication technologies by allowing any object from the physical world to communicate through a virtual communication channel (e.g. the Internet).

According to the specification published by the International Telecommunication Union, a "thing" can be any physical or virtual object possessing static or dynamic information. These objects "are capable of being identified and integrated into communication networks" [13]. Devices existing in the physical world "are capable of being sensed, actuated and connected" [13], while virtual objects "are capable of being stored, processed and accessed" [13].

## 2.2 Technical overview

"A physical thing may be represented in the information world via one or more virtual things (mapping), but a virtual thing can also exist without any associated physical thing." [13] As it has already been described above, a physical object is any kind of object existing in the physical world. However, it requires an interface which is capable to communicate with other devices. It can optionally have additional abilities, such as sensing, actuation, data capture, data storage and data processing. This device can collect information and forward it to other devices via its communication interface or it can receive information from other devices. Based on this information, it can execute various operations. The communication between the devices can be realized in three ways or with their combinations. These modes include either direct communication or through a network with or without a gateway.

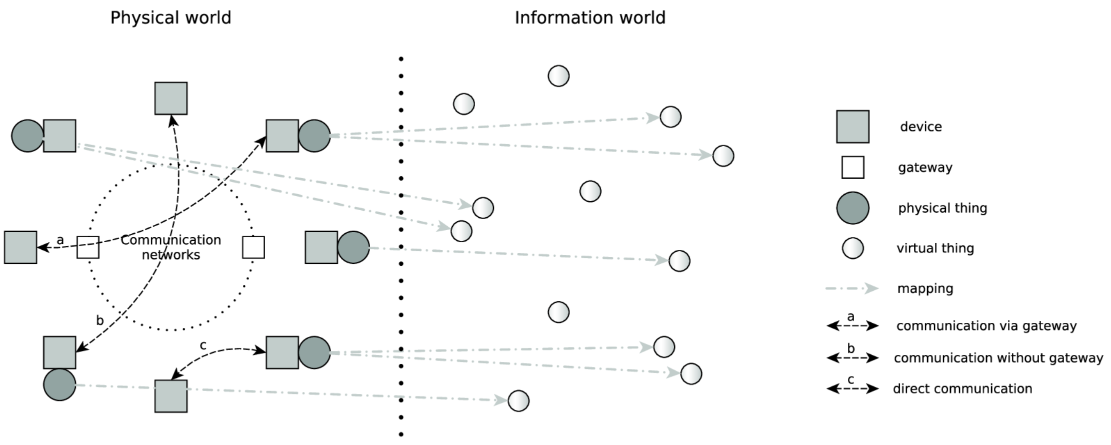

Figure 2.1: Technical overview of the IoT [13]

“The IoT applications include various kinds of applications, e.g., “intelligent transportation systems”, “smart grid”, “e-health” or “smart home”. The applications can be based on proprietary application platforms, but can also be built upon common service/application support platform(s) providing generic enabling capabilities, such as authentication, device management, charging and accounting.” [13]

Based on the behavior of the device, it can be categorized as data-carrying, data-capturing, sensing and actuating or general device. If a device provides indirect connection between a physical thing and the network, it is a data-carrying device. A data-capturing device can get into contact with physical things through a data-carrying device or via a data carrier. Detecting, measuring and digitalizing information can be done through a sensing device. This device can also be an actuating device, that is able to convert the recieved information into operations. “General devices include equipment and appliances for different IoT application domains, such as industrial machines, home electrical appliances and smartphones.” [13]

## 2.3 Reference architecture

Altough the IoT has not been precisely defined yet, there are multiple reference architectures available. Most of these architectures were designed by profit-oriented companies to promote their products. The defined models may be different in details, however, the main concepts are very similar.

Devices capable of communication can form a communication network that can provide interconnectivity among the devices. Each device requires to be uniqely identified in this interconnected network. The used identification (addressing) method has to tolerate a huge amount of devices. Adding new devices to the network has to be autonomous similarly to the reconfiguration of the network if a device disconnects. Even though most of the networking does not require any human interaction, a management interface to customize the network is necessary. The network should also provide a gateway service that is responsible

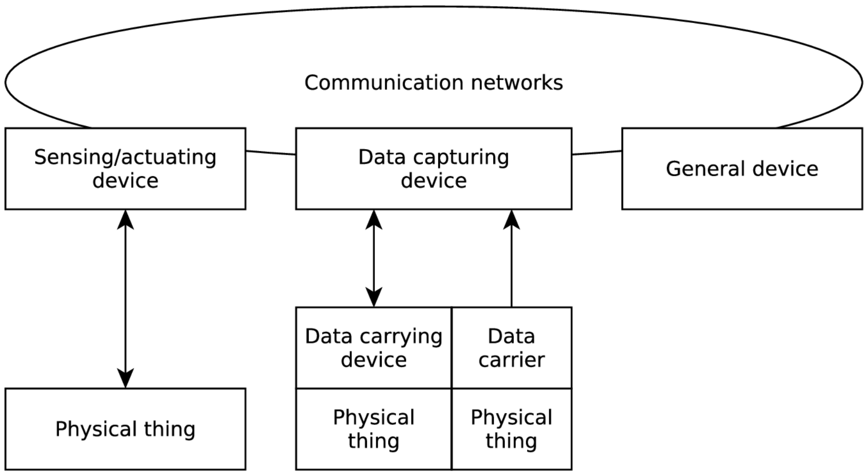

Figure 2.2: Types of devices and their relationship with physical things [13]

for the connectivity to the Internet. The higher-level protocols used for the data transfer should be optimized for transferring small amounts of information. This can mean a classical request-response communication model as well as a more specialized publish-subscribe model. Transferring data to the Internet can be done through these protocols, however, it is also possible by using a specialized API that is implemented on both sides.

The received information requires storage and analytical services on the server-side. This can be managed by a middleware implementing the same API or a high-level communication protocol as on the client-side. The information sent back from the Internet to the devices is generated by a backend service. User interaction is possible through a web-based interface which can provide access to the stored data and an ability to configure the reactions of the backend service. Other clients, e.g. mobile phones or tablets may gain access to this interface too if the backend provides an API.

The entire architecture requires some security considerations. On the level of networking, it is necessary to secure the link among devices by encrypting the data transmission and verifying the integrity of any data received. The gateway service requires authentication and authorization of both sides while keeping the transferred data in an encrypted form. The server-side services require similar features to prevent an unauthorized use.

# Chapter 3

# Connectivity

To realize the mapping of a physical device to the virtual world, it is necessary to have some kind of connection. It is possible to use conventional computer networking technologies, however, this is not suitable for use with low-power devices. This chapter will introduce wireless networking solutions and specialized communication protocols which have been designed for such devices.

## 3.1 Bluetooth Low Energy

In order to provide a wireless alternative to the RS-232 cable connection, Bluetooth was developed by Ericsson in 1994. [7] It operates in the unlicensed 2.4 GHz ISM band and uses master-slave communication with a packet based protocol.

In 2006 Nokia introduced [15] a low-power wireless technology called Wibree with a minimized amount of differences from Bluetooth. In 2010 Wibree was intruduced by the Bluetooth 4.0 specification as Bluetooth Low Energy [14] (BLE, but also marketed as Bluetooth Smart). It is an optimized version of the Bluetooth for low cost and low-power solutions. The smartphone and the tablet industry adopted the technology very quickly and they invested a lot into the software support in all their available platforms.

Due to the fact that it is a completely different implementation, it is not compatible with the classical (BR/EDR) Bluetooth. It comes in two versions [14]:

- Bluetooth Smart - it can communicate with BLE devices only
- Bluetooth Smart Ready - it can communicate with both BR/EDR and BLE devices

#### Protocol stack

The **Physical Layer (PHY)** is responsible for the data modulation/demodulation on the level of electromagnetic waves. It divides the 2.4 GHz ISM band into 40 channels. 3 channels out of those 40 are dedicated advertising channels to set up connections and broadcasting. It uses a technique called *frequency hopping spread spectrum* to switch between channels for each connection. The maximum throughput is limited by the 1 Mb/s fixed modulation rate in the used *Gaussian Frequency Shift Keying.*

The **Link Layer (LL)** handles all the timing, addressing, encryption, checksum generation and verification. To avoid software stack overloading with the most computationally expensive tasks, the LL is usually implemented in the hardware. The only required software

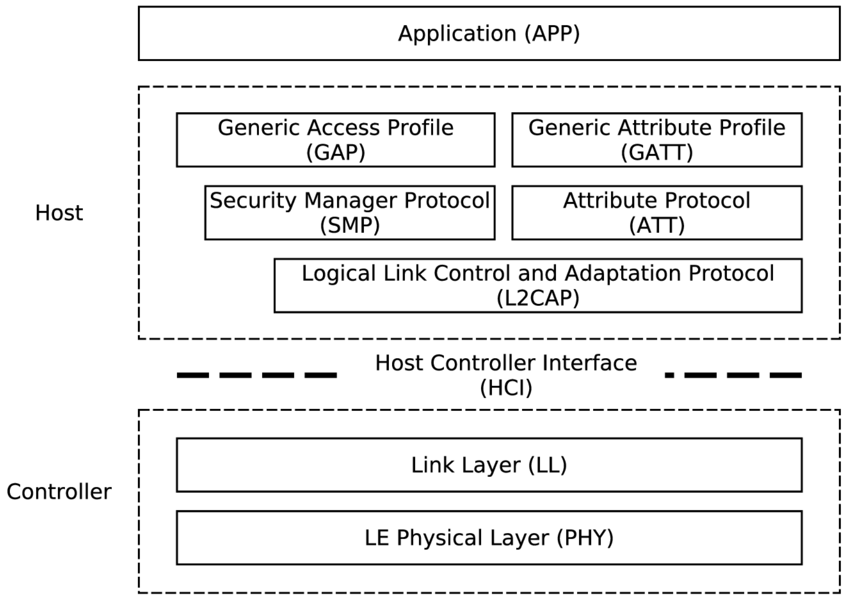
Application (APP)
Host
Generic Access Profile (GAP)
Generic Attribute Profile (GATT)
Security Manager Protocol (SMP)
Attribute Protocol (ATT)
Logical Link Control and Adaptation Protocol (L2CAP)
Host Controller Interface (HCI)
Controller
Link Layer (LL)
LE Physical Layer (PHY)

Figure 3.1: The protocol stack of Bluetooth Low Energy [14]

implemented part is the link state that can be master and slave or advertiser and scanner. If an active connection is not present, a smaller device can advertise itself to a scanner that usually possesses more computational power. When a connection is established, the scanner device switches itself to the master and the advertiser to the slave mode. This asymmetrical implementation allows the microcontroller-based low-power devices to connect to devices with higher computational capacity.

The connectivity between the host device and the Bluetooth controller is ensured by the serial **Host Controller Interface (HCI)**. The existence of the interface allows different configurations for devices with different computational capabilities and power requirements. Practically, this means USB Bluetooth adapters, adapters directly connected to a microcontroller with a serial interface, but it is also possible to integrate an adapter into a single package and produce a System-on-Chip (SoC).

Packet encapsulation, multiplexing, fragmentation and recombination is provided by the **Logical Link Control and Adaptation Protocol (L2CAP)** similarly to TCP. It is responsible for routing the **Attribute Protocol (ATT)** that is a client-server protocol for data exchange. Moreover, it is responsible for the **Security Manager Protocol (SMP)** that can provide encryption and pairing services. Since Bluetooth version 4.1, it also provides user-defined channels to transfer bigger amounts of data with low-latency.

**Generic Access Profile** is used for device advertising, discovery and connection initiation. It implements the master-slave analogy defined on lower levels. The analogy is extended with different roles, operation modes, procedures, security features and additional GAP data formats. A discoverable device periodically sends out advertising data until it receives a response from an observing device. If both devices agree on the connection parameters, a dedicated bidirectional connection can be established. On the other hand, an observing device listens for broadcasts. If it the announcement from the desired device, it will answer correspondingly. This feature can also be used for broadcasting smaller amounts of useful data to multiple devices.

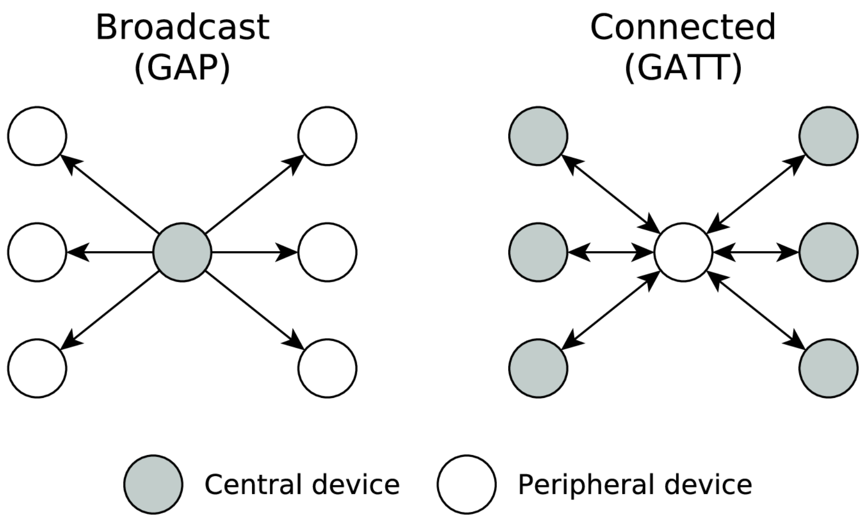

Figure 3.2: Realisable topologies by using Bluetooth Low Energy [14]

After the dedicated bidirectional connection is established, the **Generic Attribute Profile (GATT)** can be used for data transfer. It defines a hierarchical structure of **Services** and **Characteristics** and this feature is inherited in all higher-level GATT-based profiles. A service is practically a group of various characteristics identified by a unique identificator (UUID). A single characteristic can be analogically interpreted as a data structure known from structural programming languages.

Mesh networking is not supported directly, but it can be implemented on a higher level by connecting multiple central devices together. A peripheral device can connect to a central one and the data can be passed through multiple central devices, i.e. extending the operational range and increasing the number of the connected devices. There are various proof-of-concept solutions implementing this scheme and the Bluetooth Smart Mesh study group is currently specifying a standard.

The theoretical maximum of the throughput is limited by the fixed 1 Mb/s modulation rate, but the real throughput is significantly lower. Low cost microcontrollers can only process or generate a few data packets at the same time. Different power saving considerations can also limit the available bandwidth. As in any other wireless solutions, the operating range depends on different circumstances. However, the protocol was designed to operate on a very short-range. In a best-case scenario the transmission speed is around 10 kB/s and the operating range is between 2 and 5 meters.

### Gateway device

Connecting Bluetooth Low Energy devices to a classical computer network requires a gateway device that has two types of interface. Because of the popularity of the Bluetooth protocol in different consumer electronic devices, even a smartphone or a tablet can provide this interconnection. This can really simplify the interoperability of various devices in a household and it also allows to control these devices via the Internet.

## 3.2 IEEE 802.15.4

The IEEE 802.15.4 standard [2] specifies the Physical Layer and Media Access Control of the low-rate wireless personal area network (LR-WPAN). It was defined in 2003 and it is currently being maintained by the IEEE 802.15 working group. It is not a complete networking specification, but it is used as a basis by higher-level standards, e.g. ZigBee, MiWi, WirelessHART and 6LoWPAN.

As the table below shows, 802.15.4 can operate in multiple frequency domains. As a result of the chosen modulation technique, each domain has a different transmission speed. The number of the available channels is limited by the 5 MHz interval among the carrier frequencies, however, a channel has only 2 MHz bandwidth. The same frequency usage of multiple devices is coordinated by Carrier Sense Multiple Access with Collision Detection (CSMA-CD) which does not allow a device to transmit when the line is busy.

| PHY (MHz) | Frequency band (MHz) | Modulation | Bit rate (kb/s) | Channels |
|---|---|---|---|---|
| 868 | 868 - 868.6 | BPSK | 20 | 1 |
| 915 | 902 - 928 | BPSK | 40 | 10 |
| 2450 | 2400 - 2483,5 | O-QPSK | 250 | 16 |

Table 3.1: Available frequencies in IEEE 802.15.4 [2]

Devices implementing the 802.15.4 standard can communicate with each other in two different topologies: star and peer-to-peer. The star topology means multiple devices connecting to one central device (coordinator). This allows asymmetrical protocol implementation by reducing the functionality of the non-central devices. While the latter one allows a device to communicate with any other accessible device, peer-to-peer topology can be used on a higher level as the basis of the mesh networking.

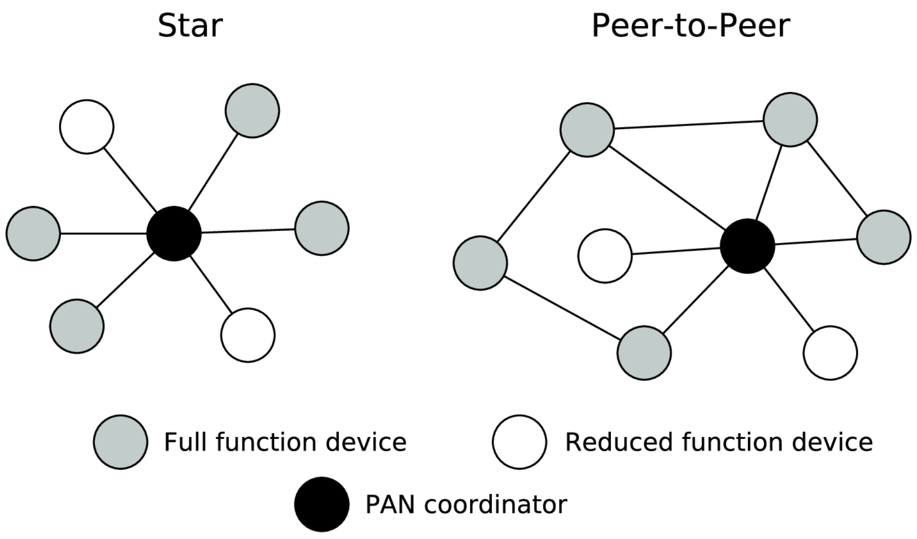

Figure 3.3: Available topologies in 802.15.4 [2]

The MAC layer defines frames as its basic transmission unit with a maximum length of 127 bytes. Addressing can be done by using short 16-bit PAN-specific addresses or long 64-bit globally unique addresses. Those might be mixed in a MAC header. Four types of frames are available to keep the protocol complexity as low as possible: beacon, data, acknowledgement and MAC command. The coordinator can optionally allow the use of superframes which consist of 16 equally divided slots bounded by beacon frames. The beacon frames are used for synchronization, PAN identification and the structural description of a given superframe. At the end of the frame the last seven slots might be dedicated for specific application.

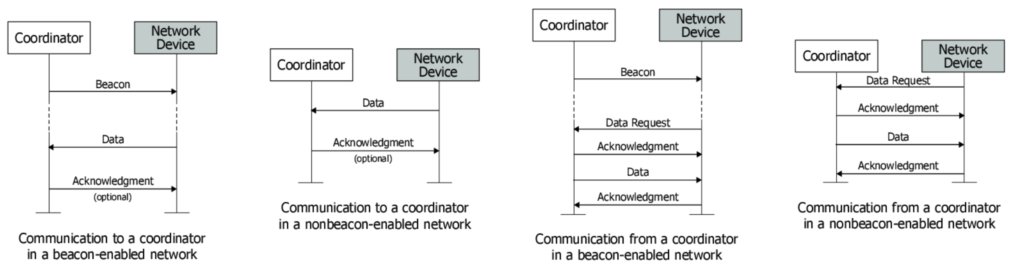

Figure 3.4: Communication modes in IEEE 802.15.4 [2]

The protocol provides some reliability by using acknowledgement frames after receiving a data frame or a MAC command frame. If an acknowledgement frame is not sent back, the sender can choose either to retry or to terminate the transmission. The integrity of each frame is protected by a 16-bit Cyclic Redundancy Check (CRC). If a bit error is detected, the receiver can force the sender to retry by not returning an acknowledgement frame. Basic security functionalities are also implemented in the form of Access Control Lists (ACL) and symmetric key cryptography. However, these require additional support from higher layers which are not part of this standard.

### 3.2.1 ZigBee

ZigBee [18] is a low-power, short-range wireless LR-WPAN standard published by the ZigBee Alliance. It was formed by hundreds of member companies in 2002. It was designed and optimized for a battery-powered application where a device spends most of its time in power-saving mode. It provides tools to connect thousands of devices together by using various networking topologies. This means, it is ideal for distributed sensor networks or home automation.

The ZigBee protocol is based on the IEEE 802.15.4 standard, i.e. the PHY and MAC layers are identical with the layers alredy described in the previous chapter. However, it uses a slightly different terminology for the device roles:

| ZigBee name | 802.15.4 name | Full function device | Reduced function device |
|---|---|---|---|
| Coordinator | PAN Coordinator | yes | no |
| Router | Coordinator | yes | no |
| End Device | Device | yes | yes |

Table 3.2: The terminological differences between ZigBee and IEEE 802.15.4 device roles

On the top of the previously mentioned PHY and MAC layers, the Network Layer is responsible for the data transmission. It also provides network management functionalities for the PAN coordinator, e.g. connecting and disconnecting devices, assigning network addresses and organizing the topology. The star topology was inherited from the 802.15.4, but on the top of the peer-to-peer topology new ones were implemented: tree, clustered tree and mesh. Each of the three new topologies can extend the operating range by allowing data to travel through multiple devices up to the destination. The coordinator can add or remove devices from the network at any time without supervision, i.e. the network is self-forming and self-healing. [18]

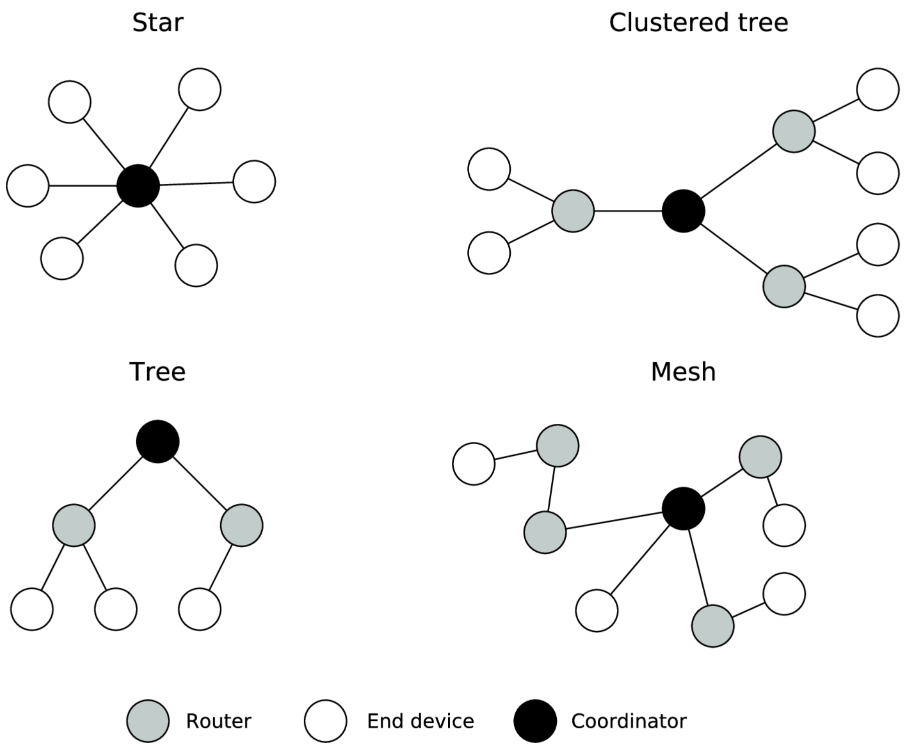

Figure 3.5: Available topologies in ZigBee

Device management is done by ZigBee Device Objects (ZDO), which are responsible for device discovery, operating mode settings and security services. The Application Support Sublayer (APS) provides advanced networking functionalities such as address mapping, fragmentation and reliable data transport. Vendor-specific services can be implemented by using the function primitives from the Application Framework. These three sublayers form the Application Layer (APL) of the protocol.

### ZigBee Gateway

To connect a ZigBee network to a different type of a network (e.g. to the Internet), a ZigBee Gateway is required. By using this device, it is possible to convert ZigBee packets to IP packets and forward them through traditional computer networks. It is also possible to connect two non-overlapping ZigBee networks by using a gateway in each network and connect them together with an Ethernet cable.

### 3.2.2 6LoWPAN

6LoWPAN is a standard that allows transmission of IPv6 packets through IEEE 802.15.4 low-power wireless personal area networks. It is maintained by the Internet Engineering Task Force (IETF). The concept has originated from the idea that "the Internet Protocol should and could be applied even to the smallest devices" [16].

Due to the fact that IPv6 packets have a maximum length of 1280 bytes and the 802.15.4 MAC frames can be only 127 bytes long, an adaptation layer has been implemented. To minimize the packet size, it compresses IPv6 and UDP headers. This would be impossible in case of using IPv4. Header compression is not enough to fit an entire packet into the frame, therefore, low-level fragmentation and reassembly has been implemented in this layer. There is also an optional support for mesh addressing that can be used when a packet has to travel through multiple devices in a mesh network.

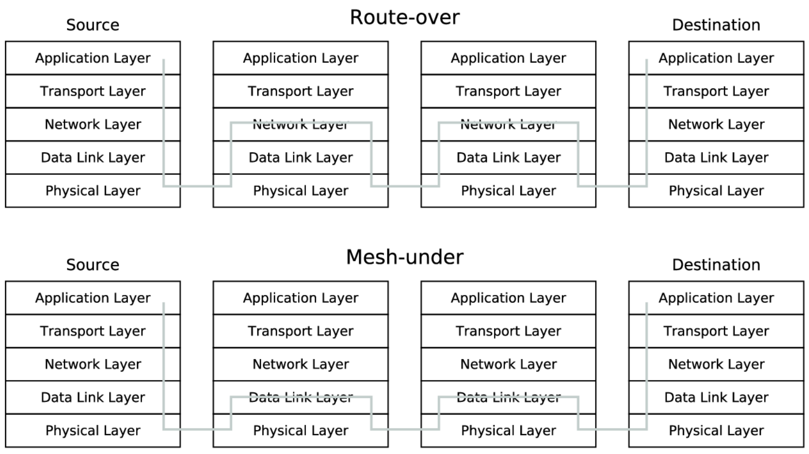

Figure 3.6: Routing modes in 6LoWPAN

Routing in 6LoWPAN networks can be done in two ways: either on the adaptation layer level by using the mesh-under or on the IPv6 level by using the route-over approach. In case of using mesh-under routing, fragmentation will be preserved during hops and the packets will be reassembled only at their final destination. This can result in a higher chance of packet loss. But on the other hand, it allows faster packet forwarding. If the routing is left to the IPv6 layer, all the fragmented frames will be reassembled at each hop and fragmented again before the transmission. This requires more computational capacity from all of the intermediate devices and the use of a higher-level mesh routing protocol. When the mesh-under routing is not used, it can be omitted from the stack. This way, the code complexity can be singnificantly reduced.

#### Border router

In order to connect a 6LoWPAN network to an other IP-based network, an intermediate device (i.e. a bridge) is required. This device is called the Border or Edge Router. It is also responsible for the Neighbor Discovery (ND), address resolution and duplicate address detection. The number of border routers in a network is not limited to one. Therefore, it is possible to increase the redundancy by connecting more of them into the same 6LoWPAN network.

### 3.2.3 Thread

According to its specification, "the Thread stack is an open standard for reliable, cost-effective, low-power, wireless D2D (device-to-device) communication. It is designed specifically for Connected Home applications where IP-based networking is desired and a variety of application layers can be used on the stack". It provides a simple but robust solution to operate a highly scalable network of battery-powered embedded devices. It is implemented on the top of 6LoWPAN and IEEE 802.15.4. [11]

The mesh-under routing provided by 6LoWPAN is implicitly disabled and the protocol uses the Routing Information Protocol (RIP) in router-over mode. Even though it uses different naming convention for mesh nodes, the role of each device remains the same. The end devices are called Sleepy End devices and the PAN coordinators are Routers or Router-eligible End Devices if not acting as a router. Network discovery is carried out by using extended Mesh Link Establishment (MLE) messages, which are sent among nodes with single-hop unicast or multicast. Unfortunately, the Thread is a proprietary solution and its detailed specification is available only to the members of the Thread Group.

## 3.3 Messaging protocols

### 3.3.1 CoAP

The Constrained Application Protocol is a UDP-based application layer machine to machine transfer protocol defined in RFC 7252. It was specifically created to be used in low-power microcontroller-based devices. These devices usually communicate through wireless personal area networks. The main idea behind the protocol was to provide HTTP-like features while keeping the message overhead of the protocol as small as possible. This packet size reduction allows limiting the use of packet fragmentation. As the Hypertext Transmission Protocol (HTTP), CoAP also has a request-response architecture with sending different types (GET, POST, PUT, DELETE) of requests to a Uniform Resource Identifier (URI). It supports the content-type header known from HTTP, which allows the interchange of the information by using popular formats such as JSON or XML. These similarities between the two protocols allow the mapping of CoAP packets into HTTP and vice versa.[3]

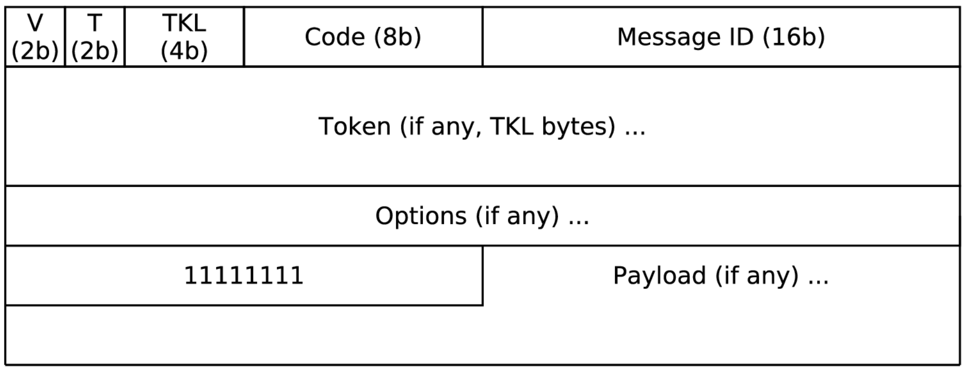

Figure 3.7: CoAP packet structure

The communication model is reversed compared to the most common protocols. The server is running on the end-devices and the client usually has more computational power. This kind of solution allows embedded devices to consume less electricity, by letting them sleep until a CoAP request arrives. Because all the HTTP request methods are available, it is possible to implement a REST-like behavior on a microcontroller. The CoRe specification describes a linking format, which can be used by clients for service discovery. The server side has to implement a response to GET requests on the */.well-known/core* URI, where all the available resources should be listed. Resource filtering is also possible by appending keywords to the URI similarly to a GET query in HTTP. Discovering services on multiple devices at the same time can be done using multicast. [1]

### 3.3.2 MQTT

IBM's MQ Telemetry Transport (MQTT) protocol is a TCP-based lightweight publish-subscribe protocol. It was designed for collecting and distributing small amounts of data among multiple devices. It promises small code footprint which allows to operate in environments with limited network bandwidth. The reliability of the protocol is guaranteed by the TCP, but MQTT also provides three different layers of Quality of Service.

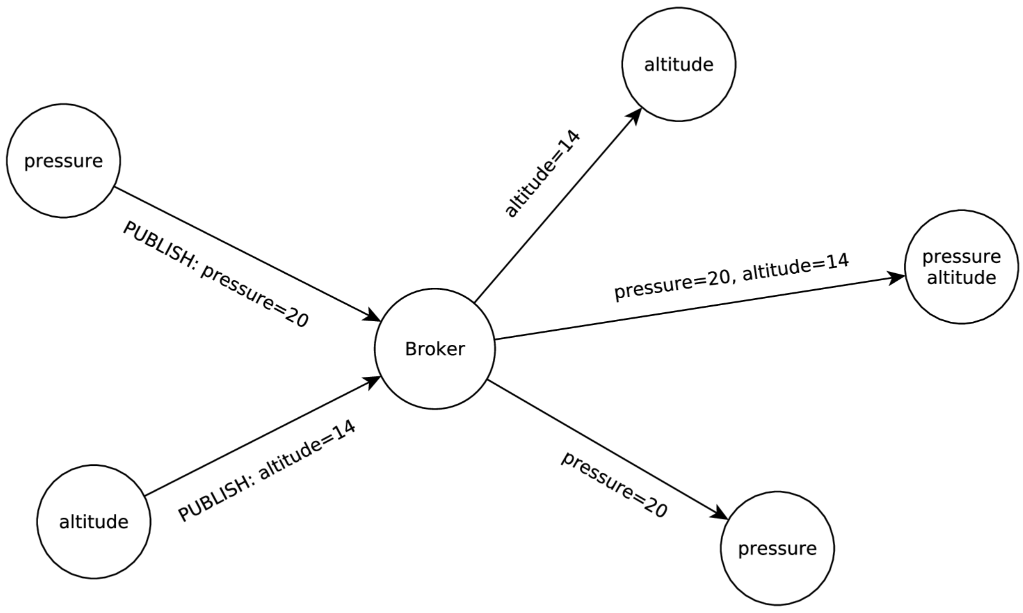

Figure 3.8: MQTT communication model

The central unit of the protocol is a *broker* device which allows multiple clients to subscribe for topics. These clients can also push messages about a given topic to the broker. Later the broker will distribute the information to the subscribers of the given topic. The topics can be organized hierarchically. By subscribing to a higher-level topic, the client will also receive all the messages from all the subtopics. If a broker receives a message which is set as retained, it will store the message in its memory and resend it to any newly connected client who has subscribed to the topic of the message. Clients can also set messages as *wills* which are stored similarly to the retained messages. However, they will be sent in that case the given client disconnects abnormally. [6]

### 3.3.3 HTTP/2

HTTP was not designed for use in constrained systems. However, since version 2.0, it is not a text-based protocol anymore. Thanks to the header compression support, the probability of packet fragmentation has been minimized. These features have made HTTP/2 into a suitable alternative in the Internet of Things to the previously described protocols. It is still using the request-response communication model, however, now it is possible to achieve bidirectional communication while opening just a single socket. [4] The popularity of the protocol has resulted in a lot of community-driven IoT solutions.

As an extension to the previous version of the protocol, the Representational State Transfer (REST) specification has become the architectural standard of any Application Programming Interface available through a network. Its main concept is that web applications can be interpreted as state machines where requests and responses are the transitions between states. When a client is using a web application, it usually executes operations on an element or on a collection of elements of the same type. The operations can be categorized based on the future changes of the collection or the element. These categories are analogous to HTTP request types. [10] For example a to retrieve a collection of users, a client should send a *GET* request to the */users* URL. To update the phone number of the second user, a *PUT* request containing the new phone number has to be made with the */users/2* URL. The architecture is not a standard, it is just a set of suggestions how to design web applications.

### 3.3.4 WebSocket

WebSockets are capable of bidirectional communication while using a single TCP connection. It is described in the RFC 6455 and standardized by W3C. The intention behind the protocol was to allow the web browser to send and receive small amounts of information without continuously opening new connections to a web server. A WebSocket connection is initiated with a handshake, which has to begin with an HTTP request with the *Upgrade: websocket* and *Connection: Upgrade* headers. These headers were originally implemented to support upgrading to a newer version of HTTP without opening a new connection. If the WebSocket is supported by the server, it responds with the same headers and the communication channel can switch to bidirectional mode. The communication itself is realized by using data frames. These frames have small overhead, therefore, they can be used to periodically send small amounts of data.

# Chapter 4

# Server side

Because of the limited resources, the information gained from embedded devices needs to be stored and processed using external services. This chapter describes a modern way of provisioning a backend required for these services and introduces modern data storing and processing technologies.

## 4.1 Cloud Computing

Cloud Computing is a way of providing services rather than a technology. It can also be referred to as on-demand computing. It is an approach to provide computing, storage, networking and other services on-demand from a shared pool of resources. These resources can be distributed among multiple customers. Based on the service utilization of the customer's request, it is also possible to dynamically redistribute them. The whole implementation is hidden on the customer's side and it only receives a simple management screen with the minimum number of options. Practically, similarly to electricity, Cloud Computing is a *pay as you use* model for computing services.

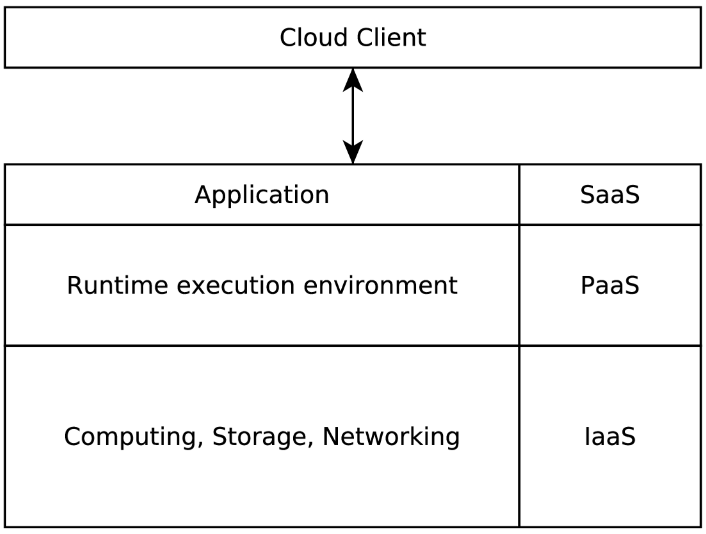

Figure 4.1: Cloud computing layers

Depending on the availability, Cloud Computing has three different deployment models. Firstly, a **private** cloud is restricted to use within a single organization for security reasons. However, it can be hosted both internally or externally outside the organization. Secondly, **public** clouds are available for multiple customers through a public infrastructure (i.e. the Internet). And lastly, a **hybrid** cloud can allow the use of both the public and the private clouds at the same time as a compromise, e.g. when a company wants to allow their customers to use a part of its internal cloud-based services. Cloud Computing can be also divided into three different service models: infrastructure, platform and software.

### 4.1.1 Infrastructure as a Service

Infrastructure as a Service (IaaS) offers an abstraction over the basic cloud services such as computers, storage and networking. This abstraction completely hides the infrastructural details, so the customer does not have to care about the security, the backups, the location and other details. Computers are usually offered as virtual machines through multiple hypervisors connected to a pool, i.e. thousands of machines can be provisioned in seconds with a single click on a user interface. Software defined virtual networks can provide connection among those machines. The networks provide virtual networking devices such as load balancers, firewalls, gateways or VPN services. Storage services can be both file or object storage, but it is also possible to provide raw block storage as additional disks for existing virtual machines.

### 4.1.2 Platform as a Service

Platform as a Service (PaaS) offers development and runtime environment to the application developers. The platform relies on an infrastructure that is not necessarily provided by the same vendor and is completely hidden from the customer. The runtime environment can support multiple languages by providing a standard interface. On the other hand, it can also provide a container-based runtime environment that is able to simplify the deployment of an application on both sides. Containers offer a virtualization technology on the level of the operating system by bundling the application and its runtime dependencies into a single image file. Finally, the deployed application can be handled by a user interface where the customer can set scaling parameters and order additional higher-level services.

### 4.1.3 Software as a Service

The Software as a Service (SaaS) model offers end-user applications running in the cloud environment. It usually runs on the top of the PaaS model which can be outsourced to an external provider. The palette of the available applications can vary by the given provider, for example databases, data analytics services, e-mail and collaboration, customer relationship management, content management and others.

## 4.2 Big Data

Nowadays more and more information is being collected from the world around us. Due to this, the need to store and process more and more information is rapidly growing. “Much of this data explosion is the result of a dramatic increase in devices located at the periphery of the network including embedded sensors, smartphones, and tablet computers. All of this data creates new opportunities to “extract more value” in human genomics, healthcare, oil and gas, search, surveillance, finance, and many other areas." [17] This huge amount of data has already exceeded the limits of the relational database management systems (RDBMS) and it requires special care. The term Big Data refers to a set of technologies which can collect, store and process huge amounts of data. Compared with conventional databases, these technologies are based on massive parallelism and they are highly scalable. This allows them to run distributed on hundreds or thousands of machines.

| Data Sources | Content Format | Data Stores | Data Staging | Data Processing |
|---|---|---|---|---|
| Web & Social | Structured | Document-oriented | Cleaning | Batch |
| Machine | Semi-structured | Column-oriented | Normalization | Real time |
| Sensing | Unstructured | Graph-based | Transform | |
| Transactions | | Key-Value | | |
| IoT | | | | |

Table 4.1: Big Data classification [17]

### 4.2.1 MapReduce

MapReduce is a programming model published by Google in 2004. It has been designed for massively parallel data processing on large clusters. It was inspired by the **map** and **reduce** functions known from functional programming languages. "Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key." [8]

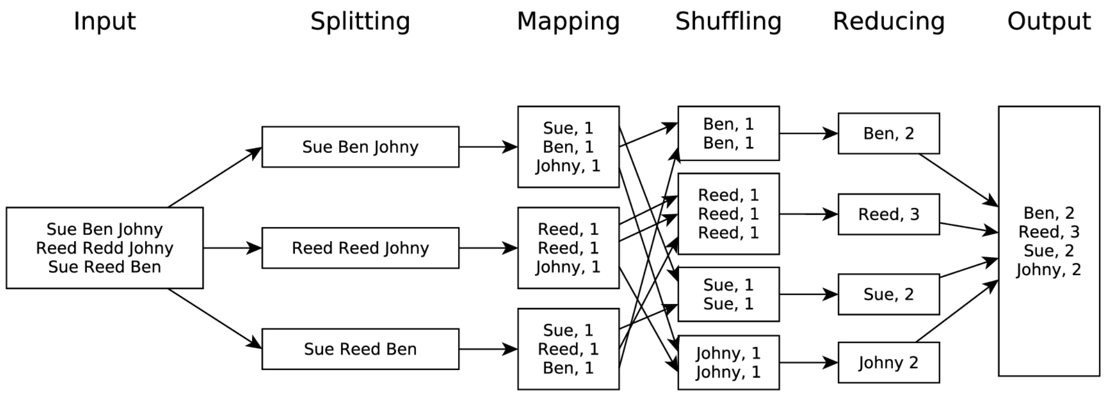

Figure 4.2: An example of MapReduce execution with word count calculation

The execution model can vary according to the implementation. However, it can be divided into a few basic steps. First, the input is split to N pieces where the size of a piece can be specified as an optional parameter. Those pieces are sent as inputs to the map function running on the cluster's nodes. The next step is called shuffling and it generates a set of lists grouped by keys. Those keys are passed to the reduce function which results in a new key-value pair based on the input. Lastly, the results are collected and returned as output. [8]

### 4.2.2 NoSQL

NoSQL is an umbrella term covering database management systems that are not based on the relational algebra. These systems do not usually use SQL as their query language and do not provide ACID (Atomicity, Consistency, Isolation, Durability) behavior. Their significance is reflected in Big Data applications where the usage of traditional database management systems is not suitable. To handle huge amounts of data, NoSQL databases are usually highly scalable and easily distributable among multiple machines.

The simplest NoSQL database type is the **Key-Value store**. Technically, it is an associative array. These systems usually store their data in memory, so the query time can be really fast. A more complex database is the **document store**, an indexed collection of documents which can be implemented as separated Key-Value stores. Their implementation is very close to the relational databases. However, instead of using foreign keys and joins, the data is usually duplicated. **Object databases** are very similar to document stores, but they are suitable for storing objects known from the Object Oriented Programming (OOP). **Graph databases** are based on the graph theory and they are suitable when the relationship among data is the most important aspect. They can be interpreted as an optimized way of storing multiple many-to-many relations known from the relational databases. In that case, if it is not enough to store the data in one type of database and to prevent the usage of multiple database systems together, **Multimodel databases** are available. Multimodel databases can combine the advantages of various database systems in one application, e.g. a graph database on the top of a document store.

### 4.2.3 Time series database

Time series databases are suitable for storing huge amounts of numerical values, such as stock prices, energy consumption or temperature. Because of the nature of this data, this approach requires completely different data processing and storing techniques. The incoming data is arriving in short time periods, therefore it is necessary to have an advanced buffering. Processing the data requires much simpler operations than in other databases, but the amount of data to be processed in a single request is much bigger. The time series are stored with a multidimensional Key-Value strategy and they are indexed with timestamps. Scalability has to be solved with high precision timing and clock synchronization among multiple nodes of a database cluster.

# Chapter 5

# Design and Implementation

The original intention of the thesis was to create an agricultural sensor network. Data such as soil moisture, temperature and luminosity are possible to measure with various sensors and these sensors can be simply integrated into a single device. Such a device has to operate without any human intervention for a whole growing season for example on a wheat field. In order to cover a bigger area, it is necessary to have multiple devices that are able to communicate wirelessly with each other. This implicates that the devices should be based on a battery-powered microcontroller with a low-power wireless networking interface. The information gained from the sensors needs to be forwarded to the Internet by using one or more gateway devices. As it has been already mentioned earlier in the work, a gateway device has to provide two types of networking interfaces. This particular case would require a 3G USB modem connected to a single board computer.

On the server-side, the incoming data has to be stored and processed. Depending on the frequency, the type and the purpose of the data, it has to always be stored in the most optimal way. The numerical values measured by the sensors will form the majority of the traffic. The information will arrive in even intervals, which makes it ideal for time series databases. Information about the state, the layout and the health of the sensor network should also be stored. However, this kind of information does not change very often and it structured in a more complex way. Based on the number of nodes, and on the used wireless technology it can either be stored in a relational database or in a graph-document multi-model store. The entry point to the server-side infrastructure has to distinguish between the two types of data and save them into the corresponding database. In agriculture there is no need for real-time data analytics, however, the received data still needs to be analysed. Based on the results, future decisions can be made. This might be easily done by using a job queue with background workers. Furthermore, a web application and/or REST API is required to allow user interaction.

The whole system has to be scalable and redundant while keeping it cost-effective. Having such a complex backend just for one sensor network is economically not viable. However, it can be provided as a Software as a Service solution for multiple customers. All the required services are available through cloud computing providers, which makes the service ideal for the cloud ecosystem.

## 5.1 Embedded device

All bigger semiconductor manufacturing companies provide devices with low-power wireless networking capabilities. The 6LoWPAN protocol has been chosen as the communication basis for the thesis. The reason is that it is an open standard. Moreover it provides true Internet connection for the devices by using IPv6. The ARM Cortex-M3 based CC2650 from Texas Instruments has been selected as the target platform for the thesis. It was designed for low-power sensor networks. Moreover, it supports both 802.15.4-based wireless networking and Bluetooth Low Energy. Texas Instruments also distributes this chip in an evaluation kit named **SimpleLink™ Bluetooth Smart®/Multi-Standard SensorTag**. This device already has 10 integrated sensors and a wireless antenna; it runs from a 3V coin-cell battery. It provides a SKIN connector as well, which allows the hardware developers to connect additional devices to the unused pins of the microcontroller.

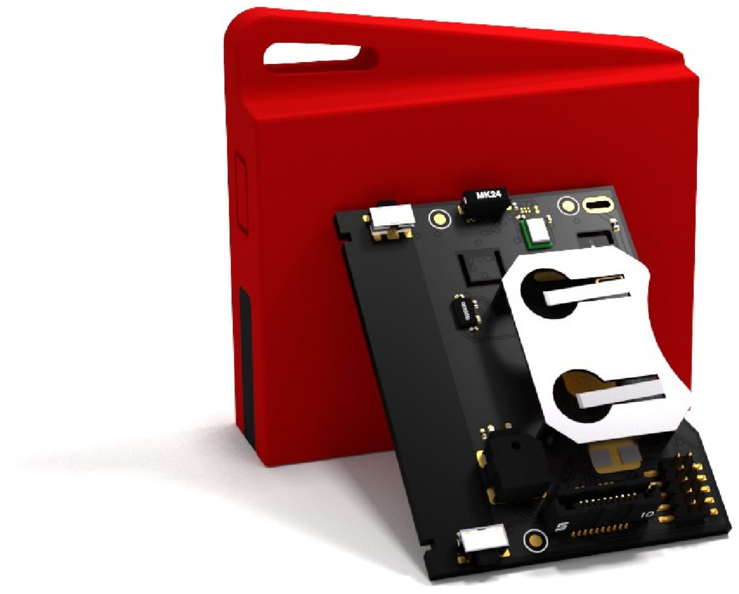

Figure 5.1: SimpleLink™ Bluetooth Smart®/Multi-Standard SensorTag [12]

The SensorTag supports measuring luminosity and temperature without any hardware or software modifications. On the other hand, measuring the soil moisture requires some external components. The SKIN connector provides access to GPIO pins with AD/DA converters and 1.2V between the VDD and GND. A transistor-based analog soil moisture sensor, however, requires at least 3.3V to operate. This is possible only by connecting the VCC of the moisture sensor to an external battery. In high moisture environments, e.g. on rice fields, this solution can drain the battery very quickly. To reduce the unnecessary power consumption caused by the sensor, the VCC pin could be controlled by a transistor connected to a second GPIO pin. This way the microcontroller can turn on or off the sensor on demand. As a proof of concept, the moisture sensor was interfaced through the pin headers of the **Debugger DevPack** connected to the SKIN connector of the SensorTag. As a result, this eliminated the need of a custom printed circuit at the very beginning of the development phase.

Based on the manufacturer recommendation, the Contiki operating system has been chosen as the base software running on CC2650. The whole codebase is available for downloading and modification via GitHub and it also contains a few of examples dedicated to the SensorTag. The **cc26xx-web-demo** implements a CoAP server, and an MQTT client exposes all the available sensors through a 6LoWPAN network. This was an ideal starting point to carry out an experiment with both the hardware and the software. To reduce the amount of transmitted data, CoAP has been chosen as the default messaging protocol for the whole design. 6LoWPAN provides compression only for UDP headers, which makes the TCP-based MQTT highly vulnerable to fragmentation. After removing the unnecessary parts from the demo code, the reading of the analog pin was implemented as a new protothread. The code was compiled with GCC and successfully flashed onto the device.

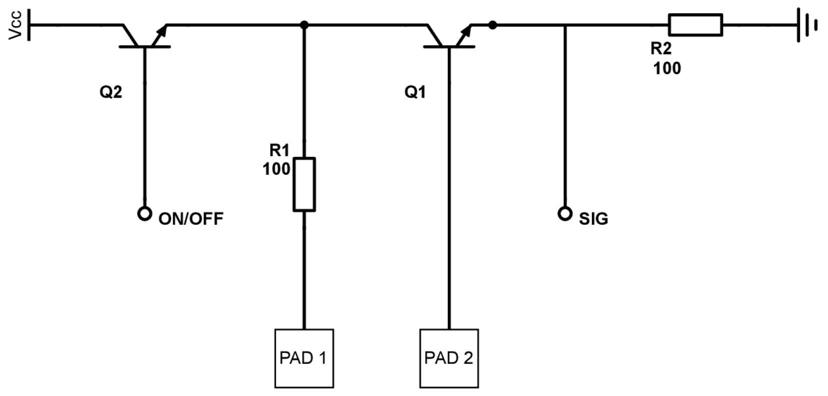

Figure 5.2: Schematic of the soil moisture sensor with the Q2 switching transistor

The border router was planned to be a Raspberry Pi 3 with an external 802.15.4 adapter. In the first experiment, the adapter was an Atmel AT86RF233-based transceiver connected through SPI. There is software support for the transceiver, but the given kernel module needs to be activated in the Linux device tree configuration. After activating the device as a standard Linux networking interface, the packet sniffer was able to catch some packets arriving from the SensorTag. Unfortunately, the networking subsystem of the kernel has no support for the 802.15.4 coordinator mode, which would be necessary for an border router.

The second experiment was based on the documentation available for the SensorTag. A USB dongle based on CC2531 low-power wireless microcontroller was connected to the Raspberry Pi 3. This device can operate in multiple modes based on the used firmware. It has been programmed to provide a serial-to-radio interface for the thesis. The firmware was available directly from the manufacturer and in binary format only. On the software side, the **6lbr** project of Cetic was used for routing and neighbor discovery. 6lbr can connect a serial device to a networking interface in bridge, router or gateway mode. It is also an open source project supported by Contiki. A web interface is available in order to ease the monitoring and configuration process. It was planned to extend the web interface with a REST API for more robust configuration through the Internet; however, the CC2531 had a hardware issue and it was not suitable for data transmission. On the online forums of Texas Instruments, multiple users reported the same problem and the only solution was to replace the dongle. Because of the slow replacement process provided by the manufacturer, the 6LoWPAN as a wireless communication solution was discarded.

Because of the lack of the functioning hardware, none of the IEEE 802.15.4-based solutions were realisable in time. Therefore, it was necessary to discard some of the aims of the thesis and pay more attention on the parts which are not dependent on the used wireless solution. The mesh networking part of the design was abandoned. Instead of 6LoWPAN, Bluetooth Low Energy has been chosen to substitute the wireless networking. The SensorTag also supports Bluetooth Low Energy and ZigBee, however, the software stack for both protocols is compatible with the compiler provided by the manufacturer. Unfortunately, the freely available version of this compiler has a 32 KB code size limit.

Therefore, the design was transferred to an Arduino 101 instead of using the SensorTag further. This board has an Intel Curie x86-compatible core and a Bluetooth Low Energy radio with integrated antenna. The Curie is capable of running the Zephyr IoT operating system, but it does not have support for Bluetooth Low Energy. On the other hand, the Arduino has its own high level hybrid C/C++ programming language with all the required (well documented) libraries. To overcome the lack of the integrated battery, a USB powerbank was temporarily used to power the device.

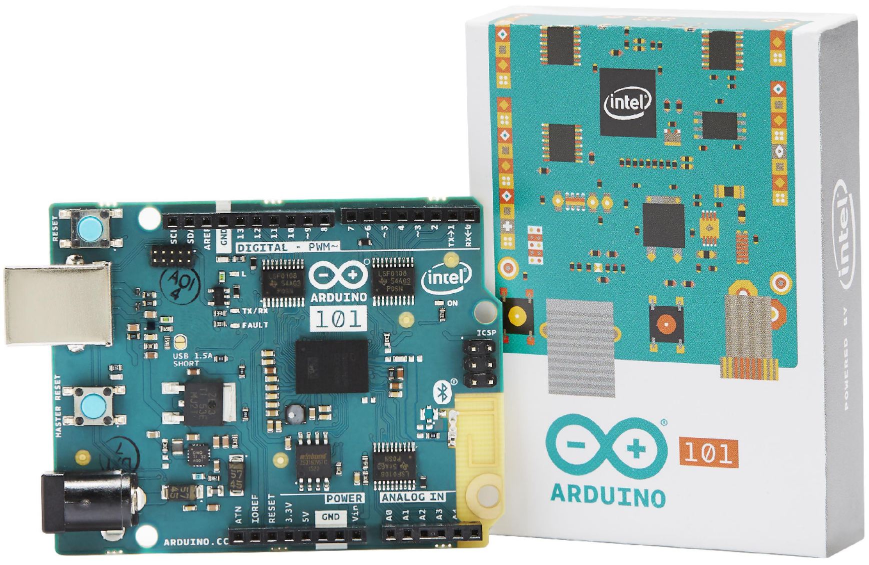

Figure 5.3: Arduino 101 [9]

Due to the fact that Arduino 101 runs on 3.3V, the soil moisture sensor can be powered directly from an analog pin. To measure the temperature and luminosity, an LM35 analog temperature sensor and a photodiode was also connected to the board. The above mentioned high level programming language provides an easy access to these analog pins. Even though there are 6 analog input pins available on the board, they share only a single analog-to-digital converter. To prevent the distortions among the three measurements, the documentation of the board has been analyzed and a wrapper has been created around the *analogRead()* function.

```
int multipleAnalogRead(int pin) {
  // Charge up the capacitor in the ADC
  int value = analogRead(pin);

  // Wait for the capacitor
  delay(10);

  // Read the desired value
  value = analogRead(pin);

  // Wait for the capacitor
  delay(10);

  return value;
}
```

Listing 5.1: The wrapper function used for reading the sensor values

The Bluetooth Low Energy library implements a scoreboard, where the services and characteristics can be attached using the *addAttribute()* method of the *BLEPeripheral* data structure. To make the measured sensor values available wirelessly, it is enough to write the measured values periodically to the characteristics on the scoreboard. A Temperature characteristic is available in the Environmental Sensing Service, however, the other two sensors are not compatible with this profile. To keep the design as simple as possible, the moisture was mapped into the Humidity and the luminosity into the Pressure characteristic in the same profile.

## 5.2 Border router

As the design of the end device was altered, the roles of the border router changed along with it. Moreover, the routing and discovery features required by the 6LoWPAN specification had become obsolete. The only requirement from the border router was to read the data from the available Bluetooth devices and forward them to the backend. This solution, however, would have changed the entire design and this was undesirable. In order to avoid any changes in the backend, it was necessary to implement a compatibility layer. This layer should be responsible for simulating an IPv6 network of sensor modules accessible by the backend.

The Raspberry Pi 3 provides an integrated Bluetooth Low Energy radio with all the required software support. Multiple experiments with various programming libraries were conducted. The goal of these experiments was to find a suitable Bluetooth Low Energy and a CoAP library that are compatible with each other. Due to the fact that the Bluetooth Low Energy is a relatively new technology, it was assumed that most of the high-level libraries have not adopted it yet. The problem with CoAP is that the server part is usually implemented in microcontrollers and most of the libraries are optimized for low-level use only. However, a high level solution was more desirable. The best results were achieved with the two **Node.JS** libraries: **noble** and **node-coap**.

The compatibility layer was designed to run as a daemon and it basically behaves as a proxy between the end devices and the backend. Firstly, it is responsible for providing information about itself and the connected devices. This feature is realized by using a CoAP server which responds to the **GET /device** request. The response is encoded as a JSON object and it contains information about the router and the discovered sensors along with the topology of the network. Getting information about the sensors is also possible by using multicast, however, processing multicast responses with a distributed backend architecture is a complicated task. Secondly, it is constantly polling for available Bluetooth Low Energy devices. If a new one appears, the compatibility layer tries to initiate a connection.

After a new device gets connected, the compatibility layer will request for the Environmental Sensing Service and for the Temperature, Humidity and Pressure characteristics described earlier. If the desired service or the characteristics are not available, it will retry the discovery two more times. If the third retry also fails, the deamon blacklists the MAC address of the given device and it will ignore it until the program gets restarted. This blacklisting has been implemented in order to prevent a connection to Bluetooth Low Energy devices other than the designed sensors. However, this is not a security feature and it is not enough to prevent the connection of harmful devices to the border router. On the other hand, if the services and the characteristics of the device meet the criteria, the daemon will start the CoAP emulation.

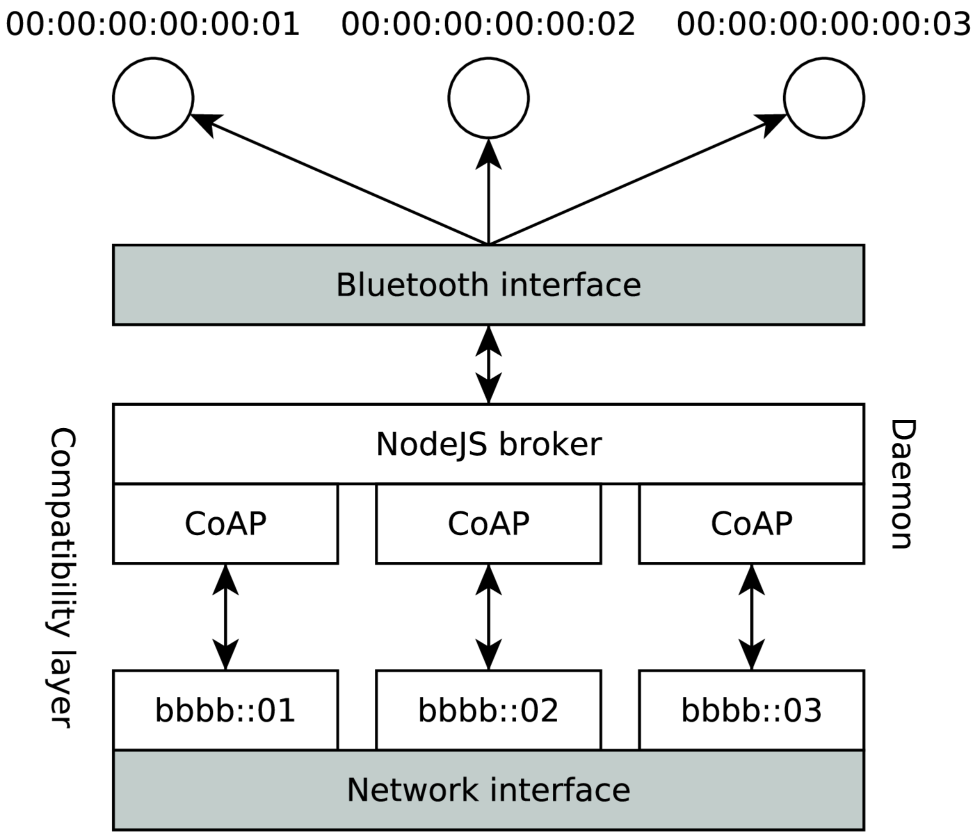

Figure 5.4: The border router daemon between the network interfaces

The emulation starts with generating a new IPv6 address for the network interface, which is reachable by the backend. This address is generated similarly to the link-local IPv6 addresses, but it is based on the Bluetooth MAC address of the device. On the newly created address, a CoAP server is started on the default UDP port (i.e. 5683). This single port solution is sustainable even if multiple devices are available because each one is listening on a different addresses. The uniqueness of the MAC addresses guarantees that two CoAP servers would never listen on the same address. If the device disconnects, the CoAP server is stopped and the IPv6 address is released.

As it has been mentioned earlier, the CoAP server was implemented by using the **node-coap** library. The implementation is just one callback function, which is called when a request has been made. Only GET requests are allowed, i.e. in case of any other type of request the server will return a *404 Not Found* error. The same error is returned if the client tries to access an unavailable URI. A client can read a measured value by sending a GET with the corresponding URI. To keep the payload size as low as possible, the response is not preconverted to ASCII. The library does not implement the CoRe specification required for service discovery. Therefore, to have the highest compatibility with the original design, the response for the **GET /.well-known/core** and **GET /sensors** requests was hardcoded into the server. This workaround, however, does not support query filtering. It is not expected that the clients will need to use this service bacuse of the low number of resources.

| URI | Description | Content Format | Encoding |
|---|---|---|---|
| / | Banner | text/plain | ASCII |
| /.well-known/core | List of all services | application/link-format | ASCII |
| /device | Device information | application/json | ASCII |
| /sensors | List of all sensors | application/link-format | ASCII |
| /sernsor/temperature | Temperature | N/A | Binary |
| /sernsor/moisture | Soil moisture | N/A | Binary |
| /sernsor/luminosity | Luminosity | N/A | Binary |

Table 5.1: Valid URIs in the CoAP server with their specifications

## 5.3 Backend

It is obvious that the backend and the border router(s) require the ability to communicate with each other. For security considerations, it is not recommended to connect a border router to the Internet with a public IP address. However, the backend requires the ability to initiate a CoAP connection as a client. This requires a tunneled link between the two endpoints, which can be realised with various Virtual Private Networking (VPN) solutions. The most popular Infrastructure as a Service providers offer VPN functionality in their Software Defined Networking stack. It is also possible to run a VPN server on a virtual machine, however, in this case scalability might be a problem.

### 5.3.1 Device discovery

To collect sensor data from a network, it is necessary to discover the available border routers first. One possible solution is to use IPv6 multicast, however, this is not a scalable solution and it would be not optimal for a larger number of routers. Software defined networking solutions usually provide a high-level API to retrieve a list of all available devices. Unfortunately this API is not always available and it is not standardized, i.e. each vendor implements it differently. The best way to overcome this problem is to temporarily establish a TCP connection between a newly connected border router and the backend. The backend needs to listen on a TCP socket for incoming connections and the border router should initiate a connection after it is started. Thanks to the three way handshake implemented in TCP, the connection can be immediately closed after it is established.

```ruby
require 'sinatra/base'

class Broker < Sinatra::Base
  post '/' do
    token = request.env['HTTP_X_AUTH_TOKEN']
    ip = request.env['REMOTE_ADDR']

    # The find_by! returns with an error if not found
    network = Network.find_by!(:token => token)
    # Do nothing if the router is already in the database
    network.routers.find_or_create_by!(:address => ip)

    # The border router expects this response
    'OK'
  end

  # Allow running only from Rack
  run! if app_file == $0
end
```

Listing 5.2: The source code of the HTTP server

This service has been implemented using the **Sinatra** library as an HTTP server. It listens on IPv6 only, i.e. it immediately detects the remote address of the connecting border router. The server answers only to POST requests containing a token in the *X_AUTH_TOKEN* header. This token is used to identify the network which the border router belongs to. If an incoming request contains a valid token, the server will try to save it into the database. To prevent possible duplicate entries, a test for an existing router has been implemented. The database abstraction has been realized by using the **ActiveRecord** library. This library also provides callback functions, which can be initiated upon various database events. In this case, after a new router is saved, an *after_create* callback is responsible to initiate the service discovery process.

### 5.3.2 Service discovery

As mentioned earlier, the border router provides a service for retrieving the network topology. This has been implemented by using a CoAP server, therefore, the backend requires a CoAP client to discover sensing devices. Because of the limitations of the border router, a received topology will be just a one-dimensional list of addresses, i.e. a star topology. As the architecture allows the dynamic connection and disconnection of sensing devices, the discovery process needs to be repeated periodically. It is also desirable to have control over the frequency of the repetition. To support an unlimited number of border routers, the service needs to be implemented in a scalable way.

The service discovery has been implemented in Ruby language using the **coap** gem. After a successfull request, the retrieved JSON is parsed into an array of addresses. This array is iterated over and each unstored address is saved into the database. For scalability support, the discovery has been implemented as a worker process using the **Sidekiq** gem. This gem implements a thread pool controlled by an asyncronous job queue. This queue is realized in a database, therefore, it might be available to multiple Sidekiq instances running on different machines. New tasks are created by the device discovery service and the periodical repetition has been realized by scheduling a job recursively from itself. If a worker fails for some reasons, it is automatically restarted after a given amount of time. After a given number of restarts, the job is moved to the dead job queue and the database record for the given router is removed. This technique allows the removal of temporarily unavailable routers and all the sensors connected to it. If the router becames available again, the device discovery service automatically recreates its database record and schedules a new task for its service discovery.

```
class DiscoveryWorker < ApplicationWorker
  def perform(id)
    router = Router.find(id)
    result = client.get('/device', router.address)
    raise unless result.mcode == [2, 5] # Wrong status code

    JSON.parse(result.payload).each do |address|
      router.sensors.find_or_create_by!(:address => address)
    end

    # Reschedule the job based on the frequency
    self.class.perform_in(router.frequency.minutes, router.id)
  end

  # Remove unavailable routers
  sidekiq_retries_exhausted do |job, _|
    Router.destroy(job['args'].first)
  end
end
```

Listing 5.3: The source code of the service discovery worker

### 5.3.3 Data collection

In agriculture, sensor networks are usually placed in uninhabited areas. While building their infrastructure, Internet Service Providers did not handle the coverage of such areas with the highest priority. This means that an Internet connection on the fields will not always guarantee high speed and low latency. These networks are usually cellular, which means that an increased number of border routers with separate Internet connections would still share the same cell. Collecting data by using the multicast feature of the CoAP would put a high load on the Internet connection in even intervals. On the other hand, as mentioned before, it is problematic to implement a scalable multicast client. To prevent high loads on a limited Internet connection, the best solution is to request the data from the sensors one by one in an iterative way.

The data collection service has been implemented by using the same **Sidekiq** and **coap** gems as it has already been done for the service discovery. There are many similarities between the two workers, however, this one is started with three arguments instead of a table row identifier. The three arguments are the address of the sensing device, the network identifier and the update frequency. The service first requires to retrieve the list of sensors attached to a connected device. This has been realized by sending a *GET /sensors* CoAP request to the device. The result is encoded in the link format described by the CoRE specification and it is parsed into a Key-Value pair of sensors and URIs. By sending GET requests to these URIs, the sensing device returns with the actually measured value of the corresponding sensor. These returned values first require a binary conversion and after that they can be stored in the database. To prevent data loss, the measured sensor values are indexed by the address of the sensing device. This way if the sensing device is deleted from the database, its measured values will be not lost. If the device reconnects to the network, the device discovery will recreate its database record. Because the address of the device is not changing, the measured sensor values will be immediately available for this previously deleted device. For periodical task repetition and to remove unavailable sensing devices, the same strategy has been used as the one described at the service discovery.

```ruby
def parse_link(address, link)
  # Iterate through each record in the link format
  CoRE::Link.parse_multiple(link).map do |measurement|
    key = measurement.rt.to_sym
    result = client.get(measurement.uri, address)

    return [key, -1] unless result.mcode == [2, 5]

    # Conversion from binary to 2-bit little-endian
    value = result.payload.unpack('S<').first
    [key, value]
  end.to_h
end
```

Listing 5.4: Parsing the data received from a sensing device

### 5.3.4 Data storage

Based on the previously designed services, it is clear that one type of database is not suitable for storing all the required information. Therefore, the data needs to distributed among multiple database engines based on various factors. To select the best engine for each data type, a simple analysis based on three factors has been done. The first factor was the frequency of the incoming data. Secondly the dynamicality of the data has been analyzed, i.e. how often the stored data will have to be changed. The third factor was the size and the complexity.

The stored user data, i.e. e-mail addresses, passwords and user settings are not changing very often. Network-specific information (together with the list of the available routers and/or the sensing devices) are also rather static than dynamic. The best solution for this kind of data is to use a traditional SQL database. By using an object-relational mapping library, the database connection can be completely hidden from the backend services. Therefore, any kind of database system can be used until it meets the SQL specification and it is supported by the object-relational mapper. Even if the scalability of a relational database is not ideal, cloud-based database services overcame this issue.

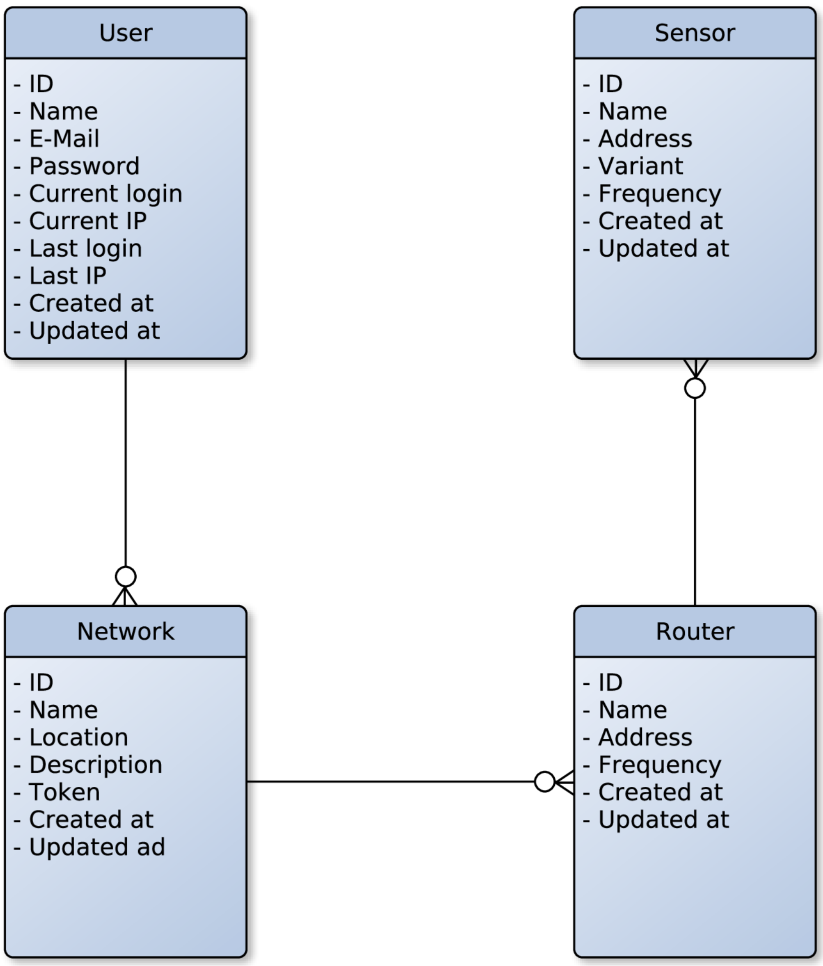

Figure 5.5: Entity-relationship diagram for the data stored in SQL

The content of the job queue is constantly changing and the periodically arriving tasks have just a short lifetime. To prevent multiple workers from executing the same task, it is necessary to provide atomic operations on the queue. Because of the scalability of the worker-based architecture, the database needs to handle a lot of input/output operations. An in-memory Key-Value database is ideal for this kind of data. The Sidekiq and sidetiq gems providing the job queue system have support only for Redis. Because no other data needs to be stored this way, there was no reason to use a different solution.

Values measured by the sensing devices are numerical and they arrive periodically. Except of some kind of system failure, there is no need for data deletion. They can be stored in a relational store in a small volume, however, the ideal solution is to use a time series database. After analysing multiple products, InfluxDB has been chosen as the database for the thesis. The reason behind this is the support for continuous queries. A continuous query is a caching tool for downsampling the incoming data in real time. It is running in the backround on a set of data and its output is a smaller dataset. The query language of the InfluxDB is very similar to SQL. It is accessible through a REST API, therefore, any programming language can interface with it. The stored data is organized in measurements, which have points indexed by timestamps. The points store the measured values in fields and additional metadata in tags, which are always indexed.

```
> SELECT * FROM sensordata WHERE luminosity > 50

name: sensordata
-----------------
time                 luminosity  moisture  network  address  temperature
1462561728602991028          53        35       10  bbbb::2           10
1462561728613079196          98        25        3  bbbb::1           17
1462561728638329865          90        29       10  bbbb::2            6
1462561728648150175          88        25       10  bbbb::2           23
1462561728672268086          65        31        3  bbbb::1            1
1462561728689305491          91        58       10  bbbb::2            1
1462561728719109120          66        13        5  bbbb::8            6
1462561728750523264          83        33        5  bbbb::6           23
1462561728764148585          96        88       10  bbbb::2            1
1462561728770810814          72        33        5  bbbb::6            9
1462561728792137920          84         2       10  bbbb::2            6
1462561728798299518          83        51        5  bbbb::8           11
1462561728805453284          87        81        5  bbbb::6           23
1462561728805453394          78        40       10  bbbb::2            5
```

Listing 5.5: InfluxDB query showing points with luminosity higher than 50

The received information from all the sensing devices, arriving from different networks, can be stored in a single measurement. A single point can store all the three measured values in three different fields. Because the database is schema-free, it is possible to append new fields to the points in the future. To distinguish among sensing devices, the address of each sensing device in has to be stored as a tag. To facilitate the queries for sensor values in a whole network, the network identifier has also been stored in the same way. To cache the hourly, daily and weekly statistics; continuous queries have been used. After filtering the measurements generated by a continuous query, the web interface can immediately visualize data without putting a high load on the database.

It is not important to store network topologies in a database, however, doing so can be useful for future optimizations. Based on the stored topology, the load on the critical parts of a mesh network can be reduced. This can be done either by adjusting the scheduling of the data collection tasks or by balancing the CoAP requests among multiple border routers. A network topology is technically a graph structure, therefore, the best way to store it is to use a graph database. Because the sensors are already stored in another database, for a node it is enough to save the identifier of each sensor pointing to this database. If the data from the relational database needs to be transferred to a document store, it is more sufficient to use a multimodel graph-document database.

Because of the chosen wireless technology among sensing devices, it was not possible to establish mesh-like topologies. The hop count between a sensing device and a border router is constantly one. Due to the fact that the list of the devices in a network is already stored in an SQL database, it is inevitable to have information about which sensing device belongs to which border router. In the special case of the flat topology provided by the Bluetooth Low Energy, an one-to-many relationship can loslessly describe the topology. Therefore, it has been chosen to not use a graph database. However, it is possible to add it any time in the future.

| Data | Frequency | Dynamicality | Structural complexity | Optimal database |
|---|---|---|---|---|
| User | Low | Almost static | Complex | SQL |
| Network | Low | Almost static | Complex | SQL or Document |
| Sensor | Low | Almost static | Complex | SQL or Document |
| Measurement | High | Static | Simple | Time series |
| Tasks | Medium | Dynamic | Complex | Key-Value |
| Topology | Low | Almost static | Graph | Graph |

Table 5.2: Analysis of the data to be stored

## 5.4 Web application

Unlike the previously described backend services, the web application needs to be publicly available. Securing the availability by a private network can increase the security, however, it is not a comfortable solution from the perspective of the user. Using the same private network for the sensors and the web application might cause security issues. The web has become an application platform because of its high availability and its universal nature. The HTTPS protocol provides asymmetric encryption and host validation features. Together with an authentication system it is as strong as the private network solution.

To increase the availability of the application, it is possible to export its services in the form of a REST API. This way it is possible to implement native frontends in the future. Desktop computers, tablets and smartphones can have their own user interface locally available and only the data would be requested through the network. It is also possible to implement the web application as a client of the REST API. This can be done by using a client-side model-view-viewmodel (MVVM) framework, such as AngularJS or React. Doing so would require the browser to load the frontend just once and after that it would behave as a previously described native client. These distributed implementations would be ideal for the service, however, they are not covered in this thesis.

The web application requires to support the following core features:

- Authenticate users based on the entered name and password
- Authorize users to access networks and other resources
- List, add, edit and delete sensor networks
- Display the list of border routers connected to a network
- Display the list of sensing devices connected to a border router or to a network
- Visualize the topology of a network or a single border router
- Provide various views of the data measured by sensors

It is obvious that the backend requires the ability to retrieve data from multiple databases at the same time. The data stored in the SQL can be mapped into objects using an object-relational mapper library. This library can be easily extended with a connection to other databases. Because the data in the time series database from the perspective of the web application is read only, there is no need for using locks or any other synchronization techniques. However, the database used for background jobs is writable by both the workers and the web application. Fortunately, it already implements an advanced locking system for concurrent data access.

The implementation of the web application has been done using the Ruby on Rails framework. This framework provides all the required tools to create web-based applications by following the model-view-controller design pattern. It is well documented and uses various generators, that help the new developers to adopt the framework easier. Its modular design allows to partially share the codebase of an application with other modules. To reduce code duplicity, the previously described broker and backend workers were implemented in the same codebase sharing all the required dependencies. Starting multiple services from a single codebase will guarantee their compatibility.

The model part is realized by the same **ActiveRecord** object-relational mapper as in the one used in the backend. Thanks to the model and migration generators, the database schema has been generated automatically. Accessing the time series measured by the sensing devices required some additions both to the *Sensor* model and to its migration. The connection to the time series database has been realized by using the **influxdb-rails** rubygem, which is an abstraction over a REST API client. First it was necessary to have a process for setting up the database in InfluxDB. No other model than the *Sensor* is using this database, therefore, the database creation and the continuous query setup has been added to its migration. Because migrations in Rails are two-way executable, it was also necessary to define the deletion of this InfluxDB database. Storing new time series data has been implemented by using the *write_point* method of the *InfluxDB::Rails.client* object. This method needs to be called with a measurement name, and with a hash of tags and values. The name of the measurement is a constant string and it has been hardcoded into the model. The tags are based on the current sensor, i.e. they can be generated from the instance variables of the given record. Retrieving data from the InfluxDB required to create a query builder similar to the one used in ActiveRecord. This query builder is parametrizable by a hash, which is converted to a *WHERE* cause during runtime.

User authentication has been implemented by using the **devise** rubygem. The *User* model and migration has been created by using a predefined generator provided by this gem. To add the custom fields to the model, it was necessary to edit the generated migration before synchronizing it with the database schema. Devise has support among others for external authentication providers, token-based authentication and password reset service. If there is a need for it in the application, these services can be turned on in the future. After a user is authenticated, its context is stored as a session variable. By default, Rails serializes session variables in an encrypted form into cookies. This solution is not an ideal one, however, the latest version of Rails does not have support for session variables stored in a database yet. If this issue will be fixed in the future, the **redis-rails** gem can be used with **Redis** as session store. To prevent collisions with the job queue used by Sidekiq, it is not necessary to use a different database, just a different namespace.

User authorization does not require any special implementation, because only one user can have access to a network. Therefore, it is safe to rely on the *has_many* and *belongs_to* helpers when retrieving data for a single user. However, this can be changed in the future by adding a new table for many-to-many relationship handling between the networks and the users. This architecture would require an advanced role based access control solution using access control lists (ACL).

| Method | URI Pattern | Controller | Action |
|---|---|---|---|
| GET | /networks | Networks | index |
| POST | /networks | Networks | create |
| GET | /networks/:id | Networks | show |
| PATCH | /networks/:id | Networks | update |
| DELETE | /networks/:id | Networks | destroy |
| GET | /networks/:id/routers | Routers | index |
| GET | /networks/:id/routers/:id | Routers | show |
| PATCH | /networks/:id/routers/:id | Routers | update |
| DELETE | /networks/:id/routers/:id | Routers | destroy |
| GET | /networks/:id/sensors | Sensors | index |
| GET | /networks/:id/sensors/:id | Sensors | show |
| PATCH | /networks/:id/sensors/:id | Sensors | update |
| DELETE | /networks/:id/sensors/:id | Sensors | destroy |
| GET | /networks/:id/routers/:id/sensors | Sensors | index |
| GET | /networks/:id/routers/:id/sensors/:id | Sensors | show |
| PATCH | /networks/:id/routers/:id/sensors/:id | Sensors | update |
| DELETE | /networks/:id/routers/:id/sensors/:id | Sensors | destroy |

Table 5.3: RESTful routes implemented in the application

An incoming HTTP request is being processed by a chain of function calls, the Rack middleware. At the end of this middleware, an instance method of a controller is called. The controller and the method selection are done by a router, which stores its configuration in *config/routes.rb*. It supports a special feature called resourceful routing, which specifies Create, Read, Update and Delete (CRUD) routes for a given controller. Based on the specifications of the web application, all the routes can be specified as resourceful. After creating the required routes, the controllers have been generated using a generator. To respond to all specified routes, the controller methods have been adjusted to handle nested routes as well.

The frontend part has been implemented by using *ActionView* templates and partials. All the templates have been written in the *slim* HTML templating language. This language simlifies the code readability by using indentation instead of opening and closing tags in the HTML. As any other ActionView-compatible template engine, slim also allows to directly inject ruby code into the templates. Because all the required data had already been generated by the controllers, only displaying the data was necessary to be implemented in Ruby.

To speed up the page load, all the CSS an JavaScript files have been concatenated into a single file using the Rails Asset Pipeline. The concatenated file in production is automatically minified by shortening variable names and removing whitespace. This way while a web browser is loading the application, only two extra HTTP connections are established.

Because a web browser window is dependent on the screen resolution of the given devices, it was necessary to design the user interface to support multiple screen widths. This can be realized by scripting the layout rendering or by using a responsive stylesheet. In CSS, the responsiveness can be implemented by encapsulating style definitions into *@media* queries. These queries can tell the browser to use or discard their encapsulated content based on their parameters, such as screen width or aspect ratio. These queries can be written manually based on the requirements. However, it is easier to use a predefined grid system.

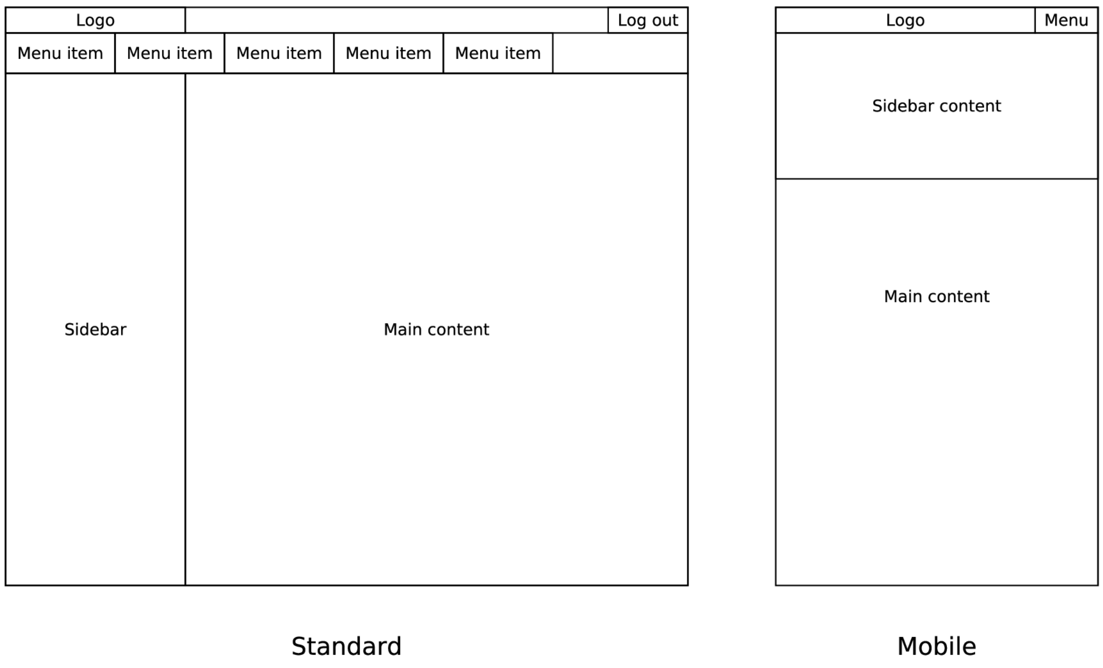

Figure 5.6: User interface mockup for standard and mobile layout

For a unified look and feel on the user interface, the PatternFly framework has been used. It is a customized version of the Bootstrap framework extended with a collection of user experience design patterns. These patterns have been used to construct the frontend based on the mockup. For reusability, the common elements, such as navigation bar and sidebar have been implemented in separate templates. The top level layout is responsible for calling those templates and rendering them into a single page.

After a successfull login, a user is able to access all of his available resources. Displaying the list of available networks has been implemented as a responsive table where the user can see the basic properties of his networks. The details of each network are available by clicking on the corresponding button, as well as the screen for editing or deleting it. There is also an option to add new networks to the list by submitting a form. The routers available in a network are accessible by selecting a particular network in the sidebar on routers page. Similarly to networks, routers can be displayed in tables. But because they are generated by the workers, there is no option for adding new routers. The list of sensors can be displayed both for a whole network or for a given router. To be able to select a router or a network, the accordions design pattern has been implemented into the sidebar. The sensors can be displayed in a table view and also in a topology graph implemented by using the **d3** library.

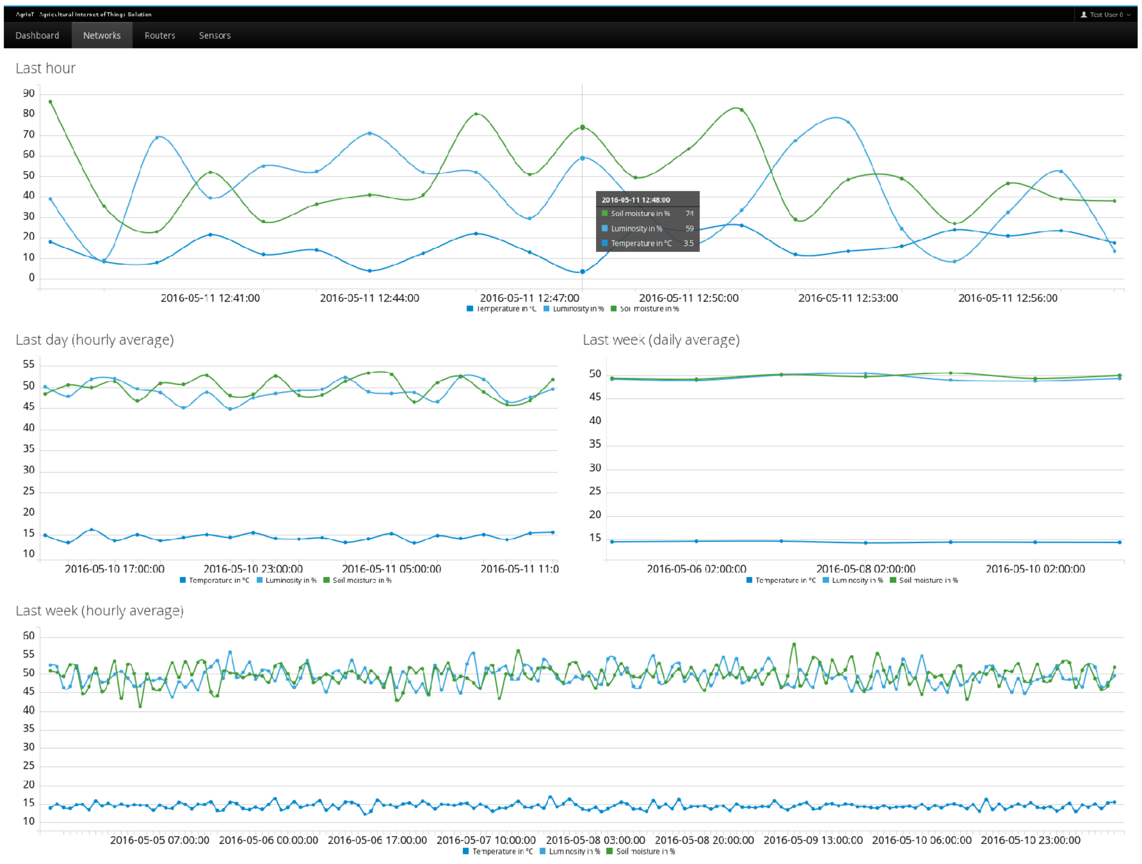

Figure 5.7: Data visualization charts displaying randomly generated data

For visualizing time series data, PatternFly suggests the **c3.js** chart rendering library. It is based on d3 and also has its own stylesheet definitions available in PatternFly. It can visualize data from a JSON object in different chart types. However, from the aspect of the thesis, the line chart with time series is the most relevant. The stored time series could be visualized for any sensing device, for multiple devices connected to the same border router and for all the sensors connected to a network. Thanks to the continuous queries provided by InfluxDB, it is possible to render daily, weekly and monthly views for the measured values.

## 5.5 Deployment

To ensure that all the dependencies are available, it has been decided to build the backend applications as containers. This way the only required dependency to be able to run a service is the runtime for the chosen container format. As the currently most popular container management system is Docker, it has been chosen to be used for deployment. Because all the backend services share a single codebase, it was obvious to build them into a single container image. It is not recommended to start multiple services from a single container, however; by using different parameters, a single image can be started as a different container. This way running a container requires an extra argument, which is the specific command for starting the given service. Because the bundler rubygem has been used for dependency management, it is also possible to deploy the services to a Platform as a Service solution or to start it locally.

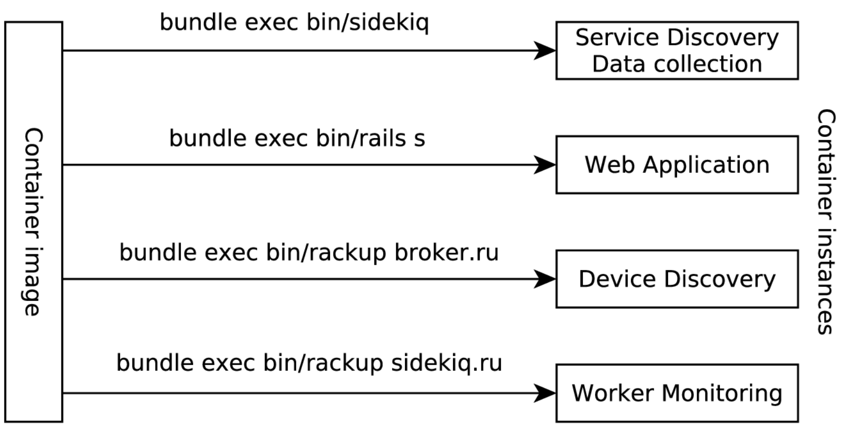

Figure 5.8: The container image with all the possible instances

The configuration parameters of each service can be specified by using environment variables in both the container and the non-container mode. Because of the large number of configuration parameters, it is recommended to use a container orchestration tool, such as *docker-compose* or *Kubernetes*. To scale the worker service, it is enough to increase the number of running containers of that type. Containers running a web server, however, require a load balancing reverse proxy which is responsible for distributing the requests among the instances. The databases also can be started from a container environment, however, it is better to rely on external cloud-based databases. As described earlier, these databases are automatically scaling on demand. Most of the cloud providers offer private networking solutions with the possibility of connecting external devices. This is usually realized by VPN, however, it can be different for each provider. Therefore, the realization of the private network between a border router and the backend varies by platform.

# Chapter 6

# Testing and verification

To verify the functionality of the designed system, multiple tests have been conducted. This chapter describes the testing process and evaluates its results.

## 6.1 Automated tests

The backend code has been developed by using the Test Driven Development (TDD) methodology. Before implementing any kind of functionality, tests had been written based on the specifications. After creating the tests, the implementation was done in small iterative steps. These steps always ended with running the corresponding tests. By doing so there is a guarantee that if the tests are correct then the code will meet the specifications. To facilitate the testing process, the **RSpec** framework has been used for test automation. This framework provides a Domain Specific Language (DSL) for a behavior-driven way of writing tests. This means that all the test cases and their generated report are easily readable. To make sure that the tests cover all the implemented features, the **simplecov** code coverage analyzer was used.

```
context 'new address' do
  let(:sensor) { Sensor.find_by(:address => address) }

  it 'saves the new database record' do
    expect do
      subject.perform(router.id)
    end.to change(Router, :count).by(1)
  end

  it 'stores the address in the new record' do
    subject.perform(router.id)
    expect(sensor.persisted?).to be_truthy
  end
end
```

Listing 6.1: RSpec test for creating new records in the service discovery

Tests have been added to the methods and callbacks defined in models, except the *Users* model which had no methods and its functionality is implemented in an external gem. Rails has its own tests for ActiveRecord functionalities, therefore, testing the validations and the relationship declarations were not necessary. Most of the controller tests have been automatically generated with the controllers. However, these tests were not prepared to handle nested routes and therefore required some modifications. Since the codebase is small enough to easily go through all the renderable pages with every possible combination of the environment, the views have not been tested with automated code. Backend services, i.e. the device discovery server and the background workers have been also covered with tests. The only exception from these are the callbacks for handling the exhausted retries in Sidekiq. Testing these callbacks would have caused an extreme slow-down of the testing process. Altogether there are 99 tests available and at the end all of them have passed. The coverage analyzer analyzed 848 lines of code, from which 835 have been covered with tests. Therefore the coverage of all the tests against the codebase is 98.47 %.

## 6.2 Functional tests

It is not possible to completely cover all the implemented services with automated tests and the sensing devices are not even testable this way. Therefore, a functional testing environment has been created to verify the whole realized product. Two Arduino 101 devices were interfaced with soil moisture, temperature and luminosity sensors and programmed as sensing devices. The sensors have been placed around a flower pot containing an African violet except for the moisture sensors, which were pushed into the soil. The border router software was installed on a Raspberry Pi 3 and as a control group it was also started from a virtual machine. This way it was possible to simulate a network containing two border routers and two sensing devices with three-three sensors.

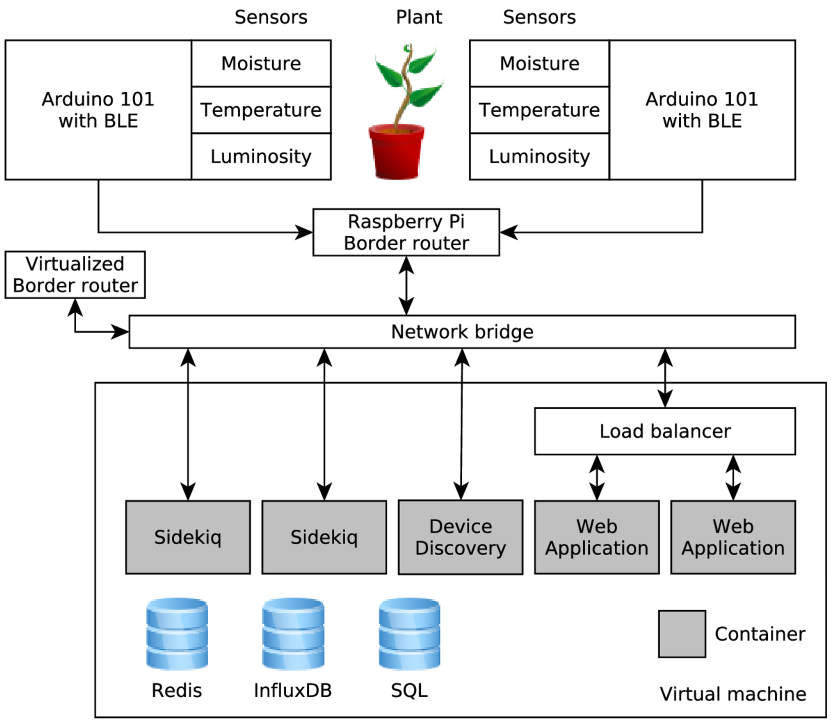

Figure 6.1: Testing environment

The backend was deployed as a set of containers on a virtual machine providing the Docker runtime. To automatize the start of all the required containers, the **docker-compose** container orchestration tool was used. To test the scalability of some services, two-two instances of the Rails application and the Sidekiq were started. External services which are usually available through cloud providers were not started as containers. These services include all the three database systems and the load balancing reverse proxy. They were set up on the virtual machine by using the default package manager. The private network which allows the backend to connect to the border routers was simulated by bridging all physical and virtual network interfaces into a single local network.

This system was up and running for around **33 hours** without any intervention. The frequency of the data collection was set to **60 seconds** and each task ran for **150 milliseconds** in average. Therefore, the estimated number of stored records was around **3450** for the two sensing devices. As the actual number of records in the database reached **3500**, it could be stated that the testing of data collection was successful. All the data has been exported into a CSV file that is available on the attached CD.

To verify the fault-tolerance of the border router, both sensing devices have been randomly restarted multiple times. By analyzing the generated logs, it has been verified that upon a reconnection, the CoAP simulation had been restarted. This fault tolerance has also been verified in the backend. First the device was disconnected for a longer time than the measurement period which caused Sidekiq to retry the data collection task related to this device. This functionality has been verified by analyzing the database and the graph provided by the Sidekiq Web UI. A similar experiment has been conducted with the border router, which has been verified with the same technique.

The verification of the sensors has been done by comparing the measured values with reference data. There were some slight differences between the values measured by the two sensing devices, this is because of the quality of the used sensors. The luminosity values have been compared with the sunset and the sunrise time related to the geographic location of the system. Unfortunately, no external data about the soil moisture was available. On the other hand, comparing the measured data with the watering time made it possible to verify the correctness of the soil moisture sensors. Using the available room thermostat, the temperature around the plant has been configured to 20 °C. In order to not measure just a static value, the window above the plant was opened three times. This caused a change in the time series data according to the outdoor temperature.

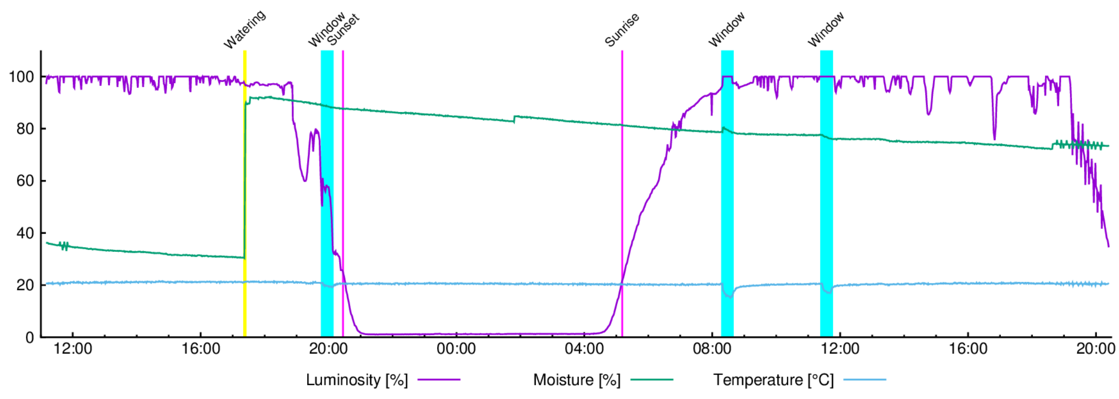

Figure 6.2: Graph of measured values with the external events

# Chapter 7

# Conclusion

The goal of this thesis was to summarize the theory behind the terms *Internet of Things* and *Cloud Computing* and based on this summary to design and implement a system of sensing devices with a backend infrastructure. These goals have been fulfilled and verified by both an automated and a functional testing.

Because the IoT has not been standardized yet, the palette of IoT devices may vary according to the vendor. Generally, an IoT device is an embedded device with communication capabilities. Despite the fact that the used communication technologies can be different, the pioneers are the short-range solutions that can provide wireless personal area networks. Thanks to Cloud Computing, it is not necessary to care about the details of the on-line infrastructure. The backend application and the user interface can be deployed to a PaaS service and the service will take care of all the infrastructural matters. The huge amount of data collected by IoT devices can be processed by using Big Data techniques.

Due to the fact that the original mesh-network design was not functional, it had to be altered during the implementation phase. The used wireless protocol was changed and the border router has become responsible for the simulation of the originally designed mesh network. Thanks to this simulation, the wireless network behind the border router has not changed from the point of view of the backend. This means that no changes had to be made in the backend. Each backend service was designed to be scalable by using techniques like load balancing and asynchronous task queueing. Even though the design was tested only on a small scale, it should work similarly with a larger number of devices.

In the future there are some cases that would be worth research. Firstly, it would be good to find an alternative wireless communication protocol that supports mesh networking. A precisely installed grid of sensing devices could also serve as a navigation system for a harvesting and planting machine or a sprayer. Another step to improve the project might be a system that is able to make decisions based on the collected values, control the watering or execute other agricultural processes. Since this time the access to the tools and the length of the thesis was limited, it was not possible to implement all the ideas. However, in the future it would be feasible to use new sensor types to measure for example soil nutrition levels or barometric pressure. It is also possible to achieve a higher measurement precision by automatically comparing the values provided by multiple neighboring devices. An other aspect of some future developments might be the use of more advanced data visualization techniques, such as heatmaps or 3D histograms.

# Bibliography

[1] Constrained RESTful Environments (CoRE) Link Format. RFC 6690, RFC Editor, August 2012.

[2] Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (LR-WPANs), October 2013. IEEE 802.15.4.

[3] The Constrained Application Protocol (CoAP). RFC 7252, RFC Editor, June 2014.

[4] Hypertext Transfer Protocol Version 2 (HTTP/2). RFC 7540, RFC Editor, May 2015.

[5] Kevin Ashton. That 'Internet of Things' Thing. *RFID Journal*, June 2009.

[6] Andrew Banks and Rahul Gupta. MQTT Version 3.1.1. http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html, 4 2012.

[7] Bluetooth SIG. Bluetooth. https://www.bluetooth.com/what-is-bluetooth-technology/bluetooth. [cit. 2016-05-13].

[8] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[9] Sigma Electronica. Arduino 101. [online] http://www.sigmaelectronica.net/arduino-p-2486.html. [cit. 2016-05-12].

[10] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[11] Thread Group. Thread Stack Fundamentals. http://threadgroup.org/Portals/0/documents/whitepapers/Thread%20Stack%20Fundamentals_v2_public.pdf, July 2015.

[12] Texas Instruments. SimpleLink™ multi-standard CC2650 SensorTag™ kit reference design. [online] http://www.ti.com/tool/TIDC-CC2650STK-SENSORTAG. [cit. 2016-05-12].

[13] ITU-T. Overview of the Internet of things. http://handle.itu.int/11.1002/1000/11559, June 2012.

[14] Akiba & Robert Davidson Kevin Townsend, Carles Cufí. *Getting Started with Bluetooth Low Energy.* O'Reilly Media, 2014.

[15] John Kooker. Bluetooth, zigbee, and wibree: A comparison of wpan technologies, 2008.

[16] Geoff Mulligan. The 6LoWPAN Architecture. In *Proceedings of the 4th Workshop on Embedded Networked Sensors*, EmNets '07, pages 78–82, New York, NY, USA, 2007. ACM.

[17] Carl W. Olofson Richard L. Villars, Matthew Eastwood. Big Data: What is it and Why You Should Care. Technical report, June 2011.

[18] ZigBee Wireless Networks and Transceivers. *Shahin Farahani.* Newnes, 2008.

# Appendices

# List of Appendices

# Appendix A

# CD content

Directories:

- **csv** - measured values in CSV format
- **img** - photos of the testing environment
- **pdf** - PDF version of the thesis
- **src** - source code files
  - **arduino** - source files for the sensing device
  - **backend** - shared codebase for all backend services
  - **latex** - files required for building this document
  - **router** - software for the border router

# Appendix B

# Manual

### Arduino

For compiling the provided source code, the latest version (1.6.8) of the **Arduino IDE** is required. Because it is written in Java, it can run on any operating system which has support for the Java Virtual Machine. However, on Windows it is possible that the USB programming interface requires some extra drivers. When installing the IDE with administrative right using the included installer, these drivers are automatically installed. To have support for Intel Curie core on the Arduino 101, it is necessary to download the compiler using the *Board Manager* located under the *Tools* menu. Because of the required Bluetooth Low Energy support, the code can be flashed only to Arduino 101 and Genuino 101 devices.

### Border router

The border router requires a **Raspberry Pi 3** with the **Raspbian Jessie** operating system installed. The dependencies can be installed by running the following commands in the directory of the border router:

```
> sudo apt-get install -y curl
> curl -sL https://deb.nodesource.com/setup_4.x | sudo -E bash -
> sudo apt-get install -y nodejs libbluetooth-dev bluetooth \
  bluez libbluetooth-dev libudev-dev pi-bluetooth
> npm install --unsafe-perm
```

The router needs to started as root and it is necessary to set up an IPv6 connection. Both the backend and the router are required to have an IPv6 address from the *bbbb::/64* subnet. If the network is set up and a device discovery web server is available, the router can be started with the following command:

```
> sudo node index.js <interface> <URL> <token>
```

### Backend

Starting the backend requires a Linux-based operating system running on *x86_64* architecture. To start all the backend services, it is necessary to install the **Docker** (>= 1.11.1) and **docker-compose** (>= 1.6.2) packages. The installation of these packages is described on the website[1] of the project. The container can be built by running the *docker build .* command, however, it is necessary to set up a database environment. There is a *docker-compose.yml* file available, which allows the automated setup of the backend together with the required databases. It can be executed by running the *docker compose up* command from the directory of the backend. After the databases are up and running, it is necessary to create the tables and generate some random testing data. This can be initiated by running the *docker-compose run –rm web bundle exec rake db:migrate db:seed* command. If this command finishes, the services will be available on the following URLs:

- *http://localhost:3000* - Rails web application
- *http://localhost:4000* - device discovery
- *http://localhost:5000* - Sidekiq Web UI

To keep the generated data visible, the backend worker has been commented out in the compose file. This was necessary because the addresses of the routers and sensing devices are not accessible and Sidekiq would invalidate them in a short time. However, it can be re-enabled by uncommenting the marked lines in the compose file.

[1] https://docs.docker.com/