Connecting constrained devices to the cloud using Zephyr

A prototype with nRF Connect

Bachelor Thesis

Alex Nyffenegger

January 2020

Thesis supervisors:

Prof. Dr. Jacques Pasquier-Rocha
and
Arnaud Durand
Software Engineering Group

UNI FR
UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

Software Engineering Group
Department of Informatics
University of Fribourg (Switzerland)

Software Engineering Group

Acknowledgements

I want to thank my supervisor Arnaud Durand for guiding me through the project and helping me with any problems I encountered. He helped me resolve a lot of issues which I could not have overcome without his help.

Abstract

With the rise of the Internet of Things (IoT) new technologies are emerging. This work gives an insight on state of the art tools and their applications in the field of IoT and a prototype is implemented with the goal to minimize bandwidth. The lightweight IoT protocol CoAP and the serialization standard CBOR are used to reduce bandwidth. The prototype is built with the IoT framework nRF Connect SDK built on top of the Real Time Operating System (RTOS) Zephyr. As a prototyping platform a constrained device called Thingy:91 is used. The prototype is evaluated for its performance. Furthermore the used tools and technologies are evaluated and compared to other options. It was found that the nRF Connect SDK together with the other technologies used are still in the early phase of their development and not yet well established, but suited the purposes precisely. The CoAP protocol outperformed MQTT and HTTP in the chosen approach.

Keywords: IoT, Internet of Things, Zephyr, CBOR, CoAP, constrained devices, Cloud, aiohttp, aiocoap, prototype, Nordic Semiconductors, nRF Connect, Segger Studio, Thingy:91

Table of Contents

List of Figures

List of Tables

1 Introduction

1.1. Motivation and Goals

The Internet of Things (IoT) is taking off and new technologies to fit the requirements of the new web and its connected devices are emerging everywhere. Low power cellular networks like NB-IoT and LTE-M allow for communication with low power consumption, while protocols such as the Constrained Application Protocol (CoAP) and the Message Queuing Telemetry Transport (MQTT) make their use lightweight and efficient. It seems undeniable that those technologies will be an important part of our future. It is therefore significant to be informed and get in contact with the new capabilities those technologies make possible.

The goal of this thesis is to minimize bandwidth of IoT devices using state-of-the-art technology by implementing a prototype, compare the approach to other solutions and evaluate the used technologies. The focus lies on building a prototype with a constrained device and connect it to the cloud using the IoT protocol CoAP. The device runs on the RTOS Zephyr and the development environment nRF Connect Software Development Kit (SDK) is used.

The chosen solution is evaluated together with the technologies used by validating the prototype and comparing the approach with other options.

The prototype should be capable of reading sensors data while connecting to a cloud server and publishing its measurements over CoAP. Furthermore the prototype should be able to observe resources on the cloud server and act accordingly. The cloud server is implemented as well, including a small client interface to access the collected data. All data sent should be encoded with the Concise Binary Object Representation (CBOR) serialization standard. In the end the whole project is deployed and tested on the Swisscom LPN

network. The advantages and disadvantages of the approach, the used tools and the new technologies in comparison to others are evaluated.

1.2. Organization

This work is organized as follows:

Introduction The introduction contains goals and motivation of this work. A short overview over the thesis as well as some insights into the IoT and constrained devices are given.

Chapter 1: Technologies This chapter gives an overview on the technologies used in this thesis. It includes the Zephyr Operating System (OS) as well as tools provided by the Nordic nRF Connect infrastructure. Furthermore an introduction to the lightweight IoT protocol CoAP and the compression format CBOR will be given.

Chapter 4: Prototype implementation From a programmers view this chapter shows the architecture and design of the prototype, and explains in more detail the capabilities and possible usages of the prototype. It concludes with limitations and problems of the implementation.

Chapter 4: Evaluation This chapter aims to compare advantages and disadvantages of the prototype and evaluates the tools used. It will take a closer look at the differences in architecture, applied technologies and ease of use for new applications. Applied tools and technologies are compared to alternative options.

Chapter 5: Conclusion Contains a summary of what was the original idea and what was achieved in this thesis. Summarizes the evaluation and its contribution to the research community. It concludes with a statement of personal challenges.

Appendix Holds links to used hardware and results of this thesis including a web client and the code base.

1.3. Internet of Things

Today buzzwords like cloud computing, M2M, smart cities, connected devices, smart meters and many others surround the IoT, which has become a buzzword of its own. But what is the IoT?

The term IoT emerged with the rise of devices getting connected to the internet but is still discussed [7]. Smaller and smaller microcontrollers allow us to connect nearly any device, even the tiniest. A few years ago connected devices were just electronics, their purpose mainly to connect to the web. Today its not just the electronic devices that are connected, now people connect their "Things" [4]. Things include anything one could imagine. Washing machines, automatic windows, thermostats, weather stations, vacuum cleaners, watches and anything that is large enough to fit a small microcontroller. They are not designed to access the web for a user, but for the user to access their Thing or its data. By collecting and combining knowledge from all the Things connected, a user can gain a lot of information, making more informed decisions and allowing him to control his Things automatically rather than having to control everything himself.

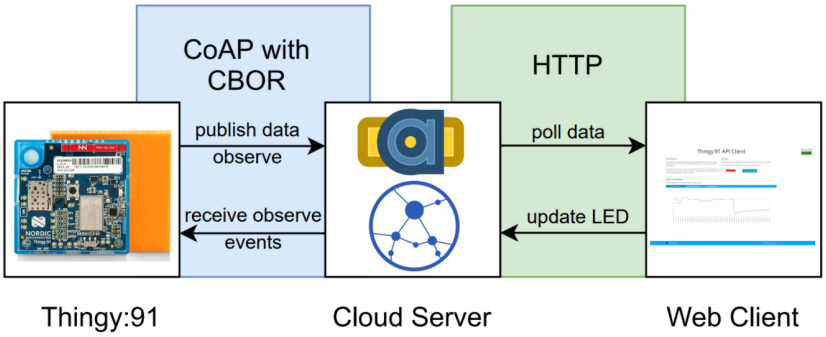

Fig. 1.1.: Project architecture overview with prototype, cloud server and web client

The car heats its seats when the owner is near, a house closes its shades if the weather forecast predicts a heatwave and the dishwasher at home can be started from the office.

With all those capabilities it is no wonder the IoT is taking off. With it new technologies emerge to satisfy the demand as more and more devices require a connection to the internet.

1.4. Constrained Devices

To connect all devices in a household, every single device needs a microcontroller which does the data collection, some thinking and most importantly, the publishing of data over a network. Computers, phones and other electronic devices are expensive and have a lot of computing power. For simple actions—like reading a sensor and publishing its data on the network or waiting for a command and start a small actuator—the computing resources of such devices are overpowered. Microcontrollers can do the same task with very limited resources. With just a few kilobytes of flash storage, an antenna, and a very simple chip set those controllers are able to achieve almost all small tasks. Their energy consumption is very low and they are cheap enough to be built into any kind of item.

The term constrained device comes from the controllers in devices which have very limited resources. Constrained devices are therefore just devices with very low computing capabilities while still able to interact trough the internet with simple interactions.

1.5. Contributions

In this work a prototype and surrounding infrastructure is implemented. The prototype runs on a constrained device. The device is called Thingy:91 and is fitted with several sensors, a microcontroller and a low-power antenna to connect to LTE-M and NB-IoT networks.

The surrounding infrastructure consists of a cloud server and a web client for users. The architecture is shown in Figure 1.1. The prototype collects data of its environmental sensors and sends them to the cloud server using the CoAP protocol. The data is stored on the server and provided to users of the web client. Furthermore users are able to change the color of the LED of the prototype. To achieve this the prototype observes the LED resource on the cloud server which can be mutated on the web client.

2 Technologies

2.1. Overview

In this project, many different technologies were used in several different fields. It is indispensable to understand the underlying technologies to find a good approach for the given project and apply fitting technology to it. This chapter gives an in-depth overview on all the relevant technologies used in this work.

2.2. Zephyr

Zephyr is a RTOS for constrained devices. It is open source and widely used and even supported by large companies such as google, Intel and Facebook. According to the Zephyr project team, it "strives to deliver the best-in-class RTOS for connected resource-constrained devices, built to be secure and safe."[31]

2.2.1. Features

The system is used for many projects as it supports a wide variety of hardware boards and architectures. With its many features it gives developers the freedom to focus efforts on their application instead of the operating system. Very high configurability and modular usage let users tune the system to their exact requirements. With many different example applications, developers can start from a template without doing the initial configuration themselves, saving a lot of time and effort. This holds true especially for the first project, as the initial knowledge required is high.

According to the Zephyr project page[1] the operating system supports cooperative and preemptive threading, as well as static allocation of memory to help systems to run for long time spans. To ensure good error handling it provides thread isolation, device driver permission tracking and memory protection using stack overflow protection [31].

Furthermore bluetooth, USB, filesystems, logging and firmware upgrades on running systems are supported. It also has a fully-featured networking stack natively integrated [31].

2.2.2. West Building Tool

To allow easy compilation, versioning and flashing of software to the desired boards, Zephyr provides a building and flashing tool called west [30]. To set up the development environment of Zephyr, many different repositories are required. Instead of installing each one separately and configure the correct version for each, west manages all requirements and their correct versions with a few simple commands. Additional libraries such as the ARM compile toolchain can be set in a configuration file as well.

west works very similar as Git or Docker in respect to its command line usage. It takes some basic command and options for this command. For example, to compile code for a board, the command ‘west build‘ can be used with the options for the desired target board and the project to be compiled. west will fetch all dependencies and call all necessary tools from the toolchain and output a compiled binary file.

Configurations for the compilation are written to configuration files by developers. This includes information for the board for which the code should be compiled, but also settings variables for the application. To easily configure said variables west allows developers to open the command line tool menuconfig, with which options can be edited in an interactive settings app. Chosen settings will be written to the configuration file, of which several versions can exist and be passed as option when building with west.

[1] `https://zephyrproject.org/`

Apart from dependency management and compilation, west can also flash the compiled application to the connected hardware. This requires the necessary connectors and drivers, but once they are installed a simple ‘west flash‘ will upload the most recent compilation to the configured board and start the application. With options, the parameters for the upload—such as the board or the desired application—can be changed.

2.2.3. QEMU

Developers may not always have the needed hardware available and can therefore not run and test their applications directly on the devices. The solution to this problem is QEMU. QEMU is a generic and open source machine emulator and virtualizer [28]. Developers can compile their applications to different architectures, among them the x86 architecture. This type of architecture can be run on QEMU [32]. This allows quick testing without requiring any hardware. There are however some limitations. Some modes and memory management features are not supported, and neither is I2C or supervisor mode execution protection (SMEP). Networking is possible but requires some more configuration to setup NAT (masquerading) and IP forwarding. Zephyr provides the tool set *net-tools* which helps to configure the necessary interfaces.

2.3. nRF Connect

nRF Connect is a software built by the Norwegian company Nordic Semiconductor. The company specializes in ultra-low power wireless systems. They build software and hardware for the IoT market and are a driving force in the field of IoT.

2.3.1. nRF Connect SDK

To make development easier, Nordic Semiconductors provides a cross-plattform software called nRF Connect. It allows to build, deploy and test software for constrained devices much easier. As most projects, nRF Connect is updated regularly and has a developer help forum. It builds up on Zephyr by using the same code base but expanding it with additional code, example applications and development features.

The tool is split into several components with different capabilities. There is an installation assistant which will setup all requirements and guide users through the required steps. Unlike west, this tool will also install the ARM toolchain and set path variables to be able to compile projects. The tool "Toolchain Manager" is included as well and allows to configure the installed toolchain.

Once everything is installed, a plethora of tools is available. The LTE Link Monitor provides, as the name suggests, a console to monitor the LTE link connection showing all current data of the LTE connection including the connected mast and position as well as signal strength. Additionally it has a AT command terminal through which the monitored device can be controlled and application output can be read.

Another tool used in this project was the "Programmer". It allows to flash not just the compiled application to the connected hardware, but also update its firmware.

The other tools included in the nRF Connect can help test applications with several devices by connecting them, control Bluetooth Low Energy applications, collect traces of applications and generate power profiles.

2.3.2. Segger Studio

To further simplify development of applications, Nordic Semiconductors teamed up with Segger [2] to expand the Segger Embedded Studio IDE with further tools to directly support Nordic products. The added features are very similiar to the ones provided by west as explained in section 2.2.2. The toolchain, required repositories and project settings can all be set directly in the IDE. It can automatically detect connected devices and can compile and flash hardware to them. A big advantage is the debugging mode, which allows to run the application on the device with breakpoints, memory inspection, function stepping and other typical debugging tools for low level languages.

A drawback is the poor editor capability, which is not very user friendly and responds very sluggish especially when resource intensive tasks such as compiling or flashing are executed.

2.3.3. Nordic Repositories

To provide developers with an easy start into their products Nordic Semiconductor provides two open source Github repositories, their main repository and a secondary called NordicPlayground. Together they provide nearly two hundred example applications, snippets and other tools to quickly get going. The projects go from simple tasks such as parsing some file formats to full development environments such as the sdk-nrf[3] used in this project.

2.3.4. Thingy:91

The device used in this project is the Thingy:91, which is a constrained IoT device built by Nordic Semiconductors. Its purpose is to serve as a playground and prototyping platform for companies to test out their IoT ideas for cheap within short timespans. The Thingy:91 provides a large amount of features to allow all kinds of different applications. The most important for IoT is most likely its LTE-M and NB-IOT (see section 2.6) capabilities to connect to the web on low power networks. The other very important components for IoT are its many sensors. It includes environmental sensors for temperature, humidity, air quality, air pressure as well as color and light sensors. Furthermore there is a low-power accelerometer and high-g accelerometer

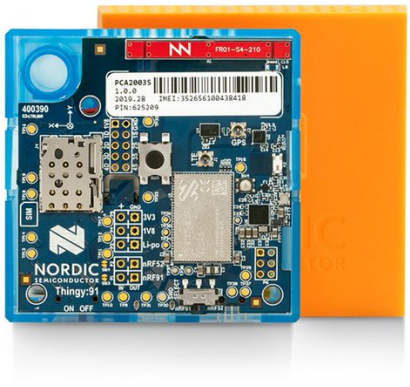

Fig. 2.1.: Thingy:91

[2]Segger Embedded Studios (`segger.com`)
[3]`https://github.com/nrfconnect/sdk-nrf`

built-in. It can also connect via Bluetooth and NFC (antennas) and has a GPS. To allow user interaction, a multi color LED, a button and a buzzer emitting sounds on different frequencies are provided as well.

The device offers a UART port for programming and debugging as well as a USB-mini port to charge its 1440 mAh li-Po battery and supervise the application output in a console. The 64MHz ARM Cortex-M33 processor with 1MB flash and 256KB RAM allows for relatively resource-intensive applications.

2.3.5. Segger J-Link

To program applications and firmware to the device used in this work a J-Link EDU[4] debugger was used. It allows to flash binaries to the device and can start debugging session which can be interactively controlled with Segger Studio. Breakpoints can be set, execution can be controlled on a step-by-step basis, and running routines, CPU cycles, current memory and registry entries can be precisely monitored and examined.

2.4. Constrained Application Protocol

The CoAP is an IoT protocol running on User Datagram Protocol (UDP). It is specialized in low-bandwidth networking for machine-to-machine (M2M) applications. Such networks are not always very reliable and generally have a low throughput. Devices using such low power networks often have very limited computing power. The CoAP protocol intends to solve this problem by allowing packets to be sent over UDP with much smaller headers and therefore reduced overall packet size while reducing fragmentation[5]. For simplicity and easy interoperability with the Hypertext Transfer Protocol (HTTP) the CoAP protocol supports a Representational State Transfer (REST) architecture. Other features include support for content-type, Uniform Resource Identifier (URI) and low parsing complexity for packets. As common with REST, auto discovery is available as well.

The messages used in CoAP are split in several types:

confirmable: A message that requires an acknowledgement from the receiver. Each such *confirmable* message results in a separate acknowledgement or reset message.

non-confirmable: Message which does not need confirmation upon arrival. Used for messages which are transfered but their correct arrival does not need to be guaranteed.

acknowledgement: A message to acknowledge that a specific *confirmable* message has arrived.

reset: Message type which can be sent to reset a message transfer if the received *confirmable* or *unconfirmable* message could not be interpreted or some required information was forgotten, for example if the recipient has rebooted. Empty messages (see empty response below) can be used to provoke a reset message to check the online status of an endpoint (CoAP ping).

[4] `https://www.segger.com/products/debug-probes/j-link/models/j-link-edu/`

[5] Fragmentation is the process of splitting data packets into several smaller packets to allow them to pass networks with constrained Maximum Transfer Unit (MTU)

piggyback: A response included in the acknowledgement message is called a piggyback response.

separate: If instead of a piggyback response a *confirmable* message is confirmed with an *acknowledgement* message, the response sent afterwards, for example because the response was not yet ready on the server, can be sent as a separate response.

empty: Messages sent without any content and code 0.00 are neither request nor response. It only contains the 4 byte header and can be used to execute a CoAP ping.

Using those types, a reliable communication can be executed on top of the unreliable UDP protocol. While with HTTP the Transmission Control Protocol (TCP) protocol handles reliable packet transfer, on UDP this is done directly on the CoAP layer. Using confirmable and acknowledgement messages reliable messaging is achieved. To keep track of which message a recipient responds to, tokens are used. Each message gets a token which identifies it for both the recipient and the sender. If for example the sender requests a resource but the recipient is not yet ready to serve it, the recipient sends an acknowledgement message for the request with the token, but does not yet include the requested data. Once the data is ready, it is sent with the same token included, allowing the sender to interpret the message as the response containing the data for the previously sent request. This separate response can be sent as *confirmable* or *non-confirmable*, depending on the importance of the message. The decision is normally based on the type of the initial request from the sender [18].

Like for the REST standard, the CoAP messages support the types GET, PUT, POST and DELETE in a similar way HTTP does [3]. For basic usage, the knowledge about the HTTP methods with the same names are enough to use CoAP, but according to the specification by Internet Engineering Task Force (IETF) [11] it is worthwhile to go into detail about the small differences between the two protocol's methods.

In HTTP devices who need an always up-to-date representation of an object on a server have to poll the server. This causes a lot of traffic and does not make sense for constrained devices for which CoAP is designed for. To resolve this problem, a standard [11] has been defined to implement the observer pattern. With the header option OBSERVE clients can register themselves to servers. The server holds a list of all subscribers and notifies them upon changes on the observed resource. This removes the need for polling. To stop observing clients can send a termination message and the server removes them from their observers list. Which resources on a server are observable is not defined by the protocol. Applications have to define this themselves.

A more special feature of CoAP is its multicast functionality. As it is based on UDP, recipients do not have to be defined as single entities. Currently secure communication is not possible with this feature, and great care has to be taken when using multicast as congestions can happen if it is not correctly implemented.

Problems of congestion can also happen if large messages are sent and fragmented. CoAP offers block communication to send large chunks of data, which can be split into several blocks to most efficiently use the MTU of UDP packets. The exact implementation of this is left to the developers. Some libraries might automatically switch to block communication if a message is too large for a single packet, others might trim the message.

CoAP offers many more capabilities, such as caching, security over Datagram Transport Layer Security (DTLS) [9], proxying, energy saving methods [6] and other features which

are not elaborated in this work. For further information consult the official specification by IETF[11].

2.5. Concise Binary Object Representation

The Concise Binary Object Representation (CBOR) is a compact binary data format to serialize data. It is based on JSON and similar to it it allows to serialize data into a specific format standard which allows data transfer between applications. Unlike JSON however, CBOR data embraces the binary format which reduces the overall size of the data at the cost of human readability. For some applications the binary format allows for faster processing for the conversion.
As with JSON, CBOR has defined tags to identify the different data types. Supported types are numbers, strings, arrays, maps and some values like false, true and null [2]. The format was published as official standard in 2013 and is regularly reviewed [2].

2.6. Low power networks

2.6.1. Narrow Band Internet of Things (NB-IoT)

Narrow Band Internet of Things is an open Low Power Wide Area Network (LPWAN) standard. It specifies in good connectivity in buildings and hard-to-reach places. Furthermore, it tries to reduce costs, helps with high connectivity density and is supposed to extend battery life of IoT devices using it [29],[1]. The standard is a substandard of LTE and is limited to a specific bandwidth of 200kHz and the leading standard in Europe. Even though the bandwidth is limited to a single frequency, the NB-IoT networks have a high capacity with more than 100'000 connected devices per sending mast [15]. It is well suited for IoT as modems for the network can cost less than 5 US dollars. A restriction which made it bad for other mobile applications is that it did not support hand-overs from cell tower to cell tower. This means a new connection had to be made every time a modem enters a new zone and is therefore inefficient and slow when moving a lot. This will be resolved with the newer version Cat NB2. Another point rendering the standard bad for fast mobile applications is its high latency.

2.6.2. Long Term Evolution for Machines (LTE-M)

LTE-M is similar to NB-IoT. Both are substandards of LTE and try to make IoT more affordable and easier to implement while covering indoor environments much better. In comparison to NB-IoT the LTE-M standard has a higher bandwidth and lower latency. The battery usage for devices increases accordingly. Unlike NB-IoT, LTE-M allows for mobile devices and will even support voice in the near future [20].

2.7. Other technologies

Python is a powerful scripting language and was used to implement several server components of the project. It offers simple syntax, a large selection of packages and allows for asynchronous applications using asyncio [14]. In the implementation of this project several Python libraries were used. aiohttp [13] and aiocoap [12] are used to build HTTP and CoAP web servers. They both run asynchronously using the asyncio [14] library. The cbor2 library was used to implement compatibility with CBOR [16].

Docker is a containerization software and was used to encapsulate components of the project allowing them to be set up quickly and run on different machines without prior setup [21].

MySQL is a relational database and available as an open source version [26].

Web Tools used in the project include the Cascading Style Sheet (CSS) and Javascript (JS) framework Materialize [24], the JS library jQuery for helper functions and Document Object Model (DOM) handling [23] and the JS plotting library Plotly [27]. All of them are open source and free to use.

Thingy:91 tools were included in the Zephyr or nRF-SDK. The most important ones are CoAP tinycbor. The library cJSON by Dave Gamble [17] was included from his Github and allowed to parse and create JavaScript Object Notation (JSON) strings.

3 Implementation

3.1. Architecture

The implementation architecture shown in Figure 3.1 consists of three parts: The Thingy:91 operates as a measuring device and communicates with the server. The server is the second part and runs a CoAP server and an HTTP server. The third part is the web client, which is served by the HTTP server part but will then operate over asynchronous requests without the client being reloaded.

The communication between the Thingy:91 and the server runs over the CoAP protocol on the internet. On this connection the collected data from the Thingy:91 is sent to the

server and the Thingy:91 observes the LED resource on the server to get updates for changing LED colors.

On the other side there is the connection between the server and the web client, which runs on HTTP. It is used to initiate the web client in a users browser and after the initial setup data from the server is periodically polled and the LED can be updated.

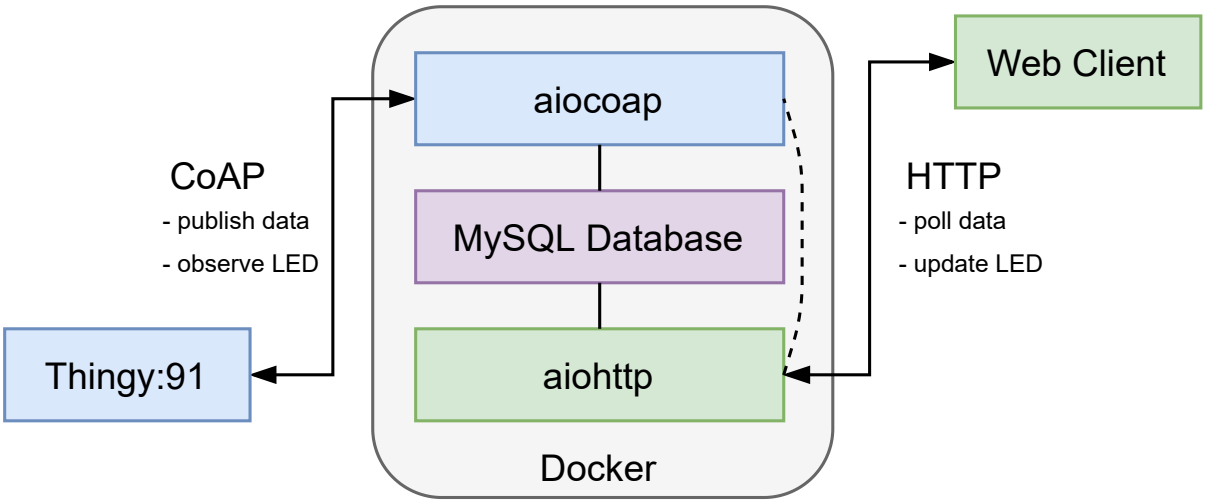

Fig. 3.1.: Project architecture

3.2. Communication Flow

3.2.1. Thingy:91 to Server

The communication flow for the Thingy:91 and the server is shown in Figure 3.2. It shows the two threads on the Thingy:91 explained in section 3.5.3 and 3.5.4. At startup the connection is checked. If it is successful, the observation thread sends its observation request, and the data transmission from the main thread starts to send data—as illustrated in the bracket "Data transmission". This transmission is triggered every 15 seconds by the main application.

If after some time the HTTP server receives an update for the LED (represented as yellow icon) from the web client, it triggers an event on the CoAP server component, which notifies the Thingy:91's observation thread.

3.2.2. Server to web client

The Figure 3.3 shows how the communication between the web client, server and its database work. Initially the web client does not exist and is loaded as index page from the HTTP server component. Like explained in section 3.4 the web client polls all required data regularly from the server. As can be seen in 3.3, this happens right after the web client has been initialized. Upon receiving a data request the server queries the database and returns the desired data to the web client. For a status update on the Thingy:91, the server reads directly from memory.

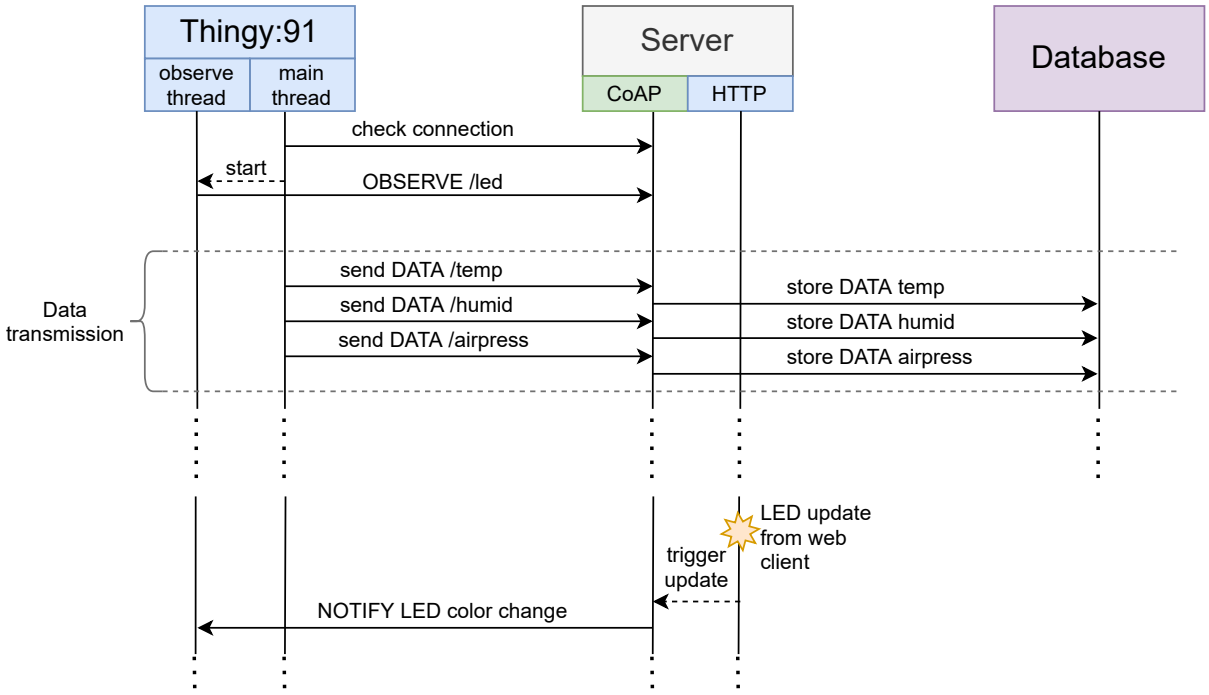

Fig. 3.2.: Communication flow between Thingy:91 and cloud server

As displayed, the LED change on the web client prompts a request to the server to update the LED with the given color from the web client request. The HTTP component triggers an update and the CoAP server component notifies all its observers, in this case the Thingy:91.

3.3. Server

The server of this project consists of three components. The two server components aiocoap and aiohttp run in the same process and exchange information directly over global variables. Those variables include the current number of observers on the observable LED resource as well as a reference to the LED resource to access its functionalities from both components. The third part is the MySQL database. It is accessed by both the aiohttp and the aiocoap components using a database helper class which handles query building and execution as well as error handling. All three components run in a Docker environment with the MySQL component in a container and the two other components in one. Communication between the two containers is internal to Docker over the defined ports for the applications. All three components expose a port to the host which can be accessed with the methods presented in the following sections.

Normally it would make sense to split the HTTP and the CoAP components into separate processes, but given the fact that both libraries as well as the database adapter work asynchronously there is no delay on actions and a single process suffices.

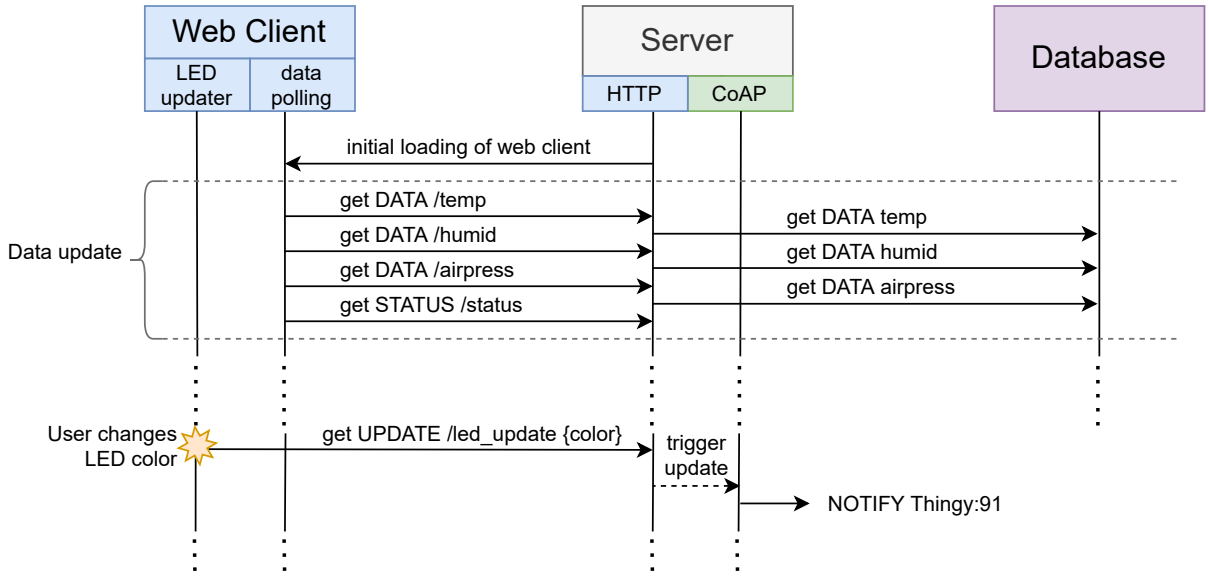

Fig. 3.3.: Communication flow between web client and cloud server

3.3.1. HTTP

The HTTP server component runs on the Python library aiohttp. It serves the routes presented in table 3.1. The base route / serves the index page with all the requirements for the web client to run autonomously. The route thingy_status allows to get the current online status of the thingy. The http server does not directly receive the status of the thingy, but reads the global observer count variable on the CoAP server component. If there is an observer the Thingy:91 must be online, as no other devices connect to server.

The route sensors allows to get all kinds of measurements stored on the server. Currently implemented are the data for temperature, humidity and air pressure. The structure is prepared to quickly add new data points. The request triggers a request to the database helper class which retrieves the requested measurements from the MySQL database. There is no restriction on the time-span, which means that all stored data points are retrieved. This can become a problem if the project is ran over long time periods because the retrieved data is sent to the web client in JSON format. With thousands of data points the client might reach a limit or the HTTP packet size gets too large.

For the led route requests from the client specify the desired color in hexadecimal as a body parameter in the POST request. As further explained in section 3.2.2 the HTTP component triggers an update event on the CoAP component to notify its observers about the change.

Method	URI
GET	`/`
GET	`/thingy_status`
GET	`/sensors/{measurement}`
POST	`/led`

Tab. 3.1.: Routes for the HTTP server component

3.3.2. CoAP

The CoAP server component runs using the Python library aiocoap [12]. It offers three routes shown in table 3.2 to send new data for temperature, humidity and air pressure. The only other paths are the auto discovery under /.well-known/core and the observable resource /led.

If data is sent to one of the data accepting routes, the incoming message payload is parsed from CBOR back to a number value. Depending on the route which was requested, the value is stored to the database using the timestamp of the current time at the server and the type of the measurement based on the chosen resource route. The data is saved using the database helper class, which accepts the mentioned options and stores the data.

For the observable resource each observation requester is written to a list of observers and the current state of the desired color is returned as an integer encoded in CBOR. The global count of observers on the resource is updated as well and can be used by the HTTP component of the server to check if the Thingy:91 has already connected. If the led resource is updated, all observers in the list—in this case the Thingy:91—are notified the same way the original response was sent.

Method	URI
GET	`/.well-known/core`
GET (OBSERVE)	`/led`
PUT	`/temp`
PUT	`/humid`
PUT	`/air_press`

Tab. 3.2.: Routes for the CoAP server component

3.3.3. MySQL Database

The database runs in its own container and only exposes the standard port 3306. Upon initial application startup, no schema is present. The schema is created by the server upon startup. It detects if the schema with the correct tables already exists on the database. If not, the schema and the corresponding tables are created.

The schema includes one table for each measurement type, consisting of the types temperature, humidity and air pressure. Other types can quickly be added. Each table entry has an id, a value of type float and a time stamp of type datetime.

Data is read and written by the python helper class included in the server components. It uses an asynchronous MySQL adapter to keep the advantage of the asynchronicity.

3.4. Web Client

The web client shown in Figure 3.4 is loaded upon the first site visit. Once loaded, the page does not need to be reloaded for updates. It automatically polls all needed data for the charts. Every 15 seconds all the data for the charts as well as the online status of the Thingy:91 is requested from the server. If the Thingy:91 is online and connected to the server, a green button with the text "CONNECTED" will indicate the online status. If

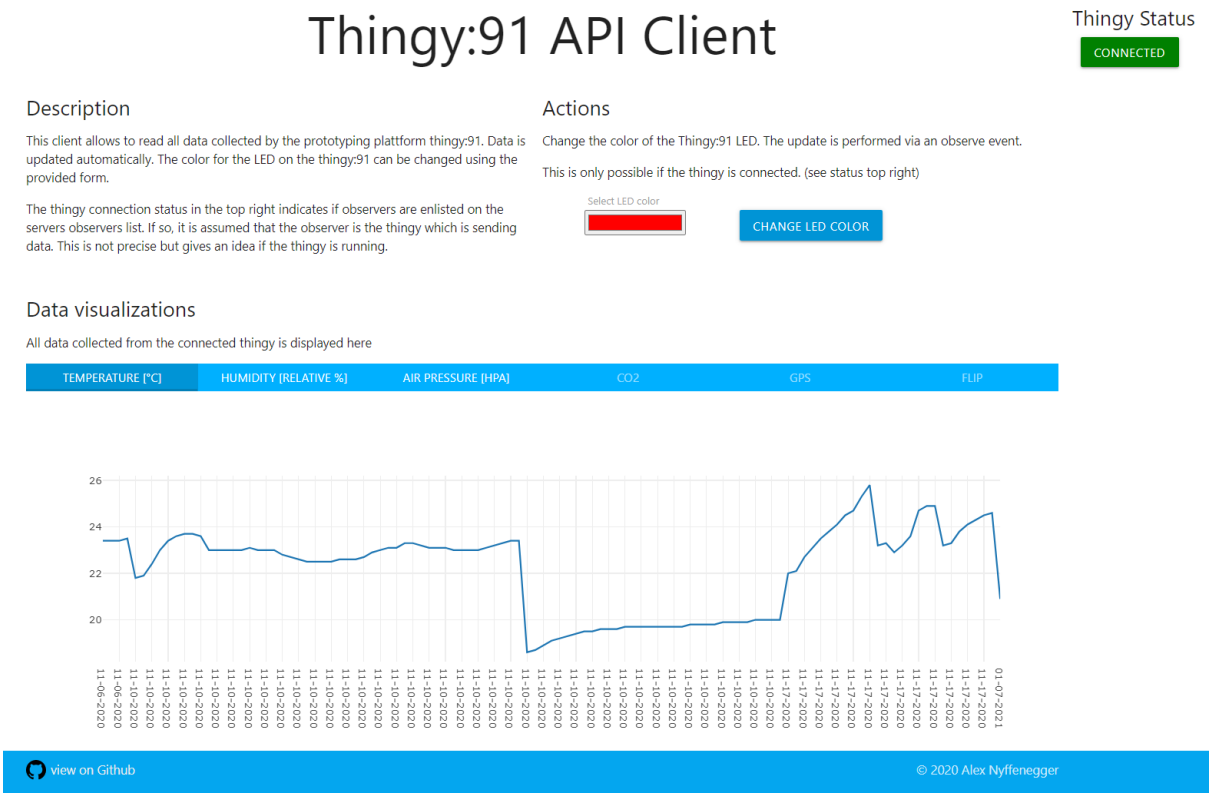

Fig. 3.4.: Web client interface with connected Thingy:91

the Thingy:91 is disconnected the status indication will display a "DISCONNECTED" info in a grey tone.

In the "Actions" section of the web client users can alter the color of the LED of the Thingy:91. This action is only possible if the status of the Thingy:91 is shown as connected. Otherwise the color update is not possible and the corresponding action button is grayed out.

The data visualizations at the bottom are always accessible. The different measurements are separated into several tabs, each showing a different measurement timeline. The tabs for CO_2, GPS and the orientation of the Thingy:91 (FLIP) are already present, but currently not available as the server does not provide this data. The shown time line can be zoomed, panned and downloaded as a chart.

3.5. Thingy:91 Firmware

3.5.1. Overview

As Nordic Semiconductors—the manufacturer of the Thingy:91—suggests, the application was built on top of an already existing application. The Asset Tracker is an application provided by Nordic Semiconductors. It collects sensor and GPS data from the sensors on the Thingy:91 and sends it to a cloud application by connection to either the NB-IoT or LTE-M network. To implement the project in this work, the cloud backend had to be

swapped to use CoAP and CBOR, while changes to the main application explained in the section 3.5.2 were minimal.

3.5.2. Base application: Asset Tracker

The asset tracker—which formed the basis of this work—consists of a cloud backend and a main application. The main application handles all necessary tasks. At startup it initialises all sensors. Then the modem of the Thingy:91 is connected to the next cell tower. A work thread is initialized at application startup which executes all tasks pushed to the work queue by different parts of the main application. This allows the main application to handle and coordinate several tasks without getting stuck on a single one when a resource might block or some error occurs. Once the network is connected and the applications initial setup is done, the cloud backend is started. In a separate thread, the specified cloud backend is executed and the exposed functionalities are called. In the case of the asset tracker, the default backend is implemented to use MQTT and allows for FOTA updates. It connects to the specified Nordic cloud server. Once the connection to the cloud suceeds, the main application starts to read data from the sensors every 15 seconds and sends it to the Nordic cloud by calling the according cloud backend methods. The same happens for other data such as the GPS, buttons and accelerometers. On the Nordic cloud website, users can activate the buzzer. The cloud backend receives the MQTT messages and forwards them to the main application. The main application handles the messages and executes the corresponding actions, in this case starting the buzzer on the Thingy:91. To ensure that all messages can be parsed, a cloud message standard is defined in the asset tracker. It defines format, types and other options allowed for the messages. The format specified is JSON and contains properties such as the command type and exact value for incoming command messages, and the sensors type, measurement type and other parameters for outgoing messages.

3.5.3. Cloud backend

The cloud backend in the asset tracker works as an encapsulated component. It communicates with the main application by exposing actions and sending messages of a special cloud event type which is received by the main application. Therefore, the implementation of a cloud backend using CoAP consists of building a backend which offers the same actions to the main application and is able to generate events and send them to the main application at the required moments.

The actions minimally required are *init, connect, disconnect* and *send*. They are handled by the main cloud backend thread.

Init starts the cloud backend and returns the configured backend. For the implemented backend this is just a struct containing the most important configurations for the backend and a pointer to the backend itself.

Connect tells the cloud backend to connect to the configured cloud server. For this, the IP address is resolved and two sockets are initialized with the resolved IP address and UDP as protocol. The first socket is used for the main thread which will send data to the cloud server. The second thread will wait for updates from the remote server and parse those. There are two reasons why two separate threads each with a separate

connection exist: first of all, it is easier to implement two different flows for the two conceptually different actions. The second reason is that to send data to the cloud it is not very important to wait for acknowledgements of messages. If data does not arrive at the server because of some small outages or similar problems, the application is not affected. It can keep sending data and the server will receive it once it is back up. If the sending thread would block until an acknowledgement for each message was received it might cause the thread to lag behind with its messages when some ACKs return with a delay. The listening thread on the other hand has to block and wait for incoming data. Without being busy sending data and receiving ACKs it can quickly react on incoming commands from the cloud server.

The listening thread is started upon application start, but does not execute any code. It listens on a semaphore which is activated in the main thread of the cloud backend in the connect function. Once the semaphore is set at the end of the connect function, the thread will start receiving commands on the socket provided by the connect function of the main cloud backend thread.

Send takes a struct of type message, containing data which should be sent to the remote server. The message is parsed and compiled into CBOR format. Then the CoAP request is prepared and the CBOR payload appended. Once the message is sent, the socket is read for new incoming messages. If a confirmation is received, an ACK is sent, otherwise the function is skipped. If the cloud backend would block on confirmations, other messages might get slowed down. Confirmations can be acknowledged on the next pass as well.

Disconnect releases the connection to the server and changes the semaphore which allows the message receiving thread to listen for messages to terminate its connection as well. The sockets are closed and the cloud has to be connected again to do any further messaging.

3.5.4. Observation thread

As explained in the previous section, the cloud backend consists of two threads, of which one sends sensor data to the cloud server while the other waits for incoming commands from the cloud server. This thread is the second one of those two and started at the startup of the application. It does however not take any action until the main thread of the cloud backend has changed a semaphore instructing this thread to use the given socket to listen for incoming commands. To receive any instructions, the first action this thread does is to send an OBSERVE request to the cloud server. It then waits for the confirmation of this request and sends an ACK upon arrival. From then on it enters a loop listening for incoming messages on the socket. If a message is received, the contents are parsed from CBOR to a JSON format and an event (see 3.5.5) containing the message contents is sent to the main application. Once the event is sent, the thread goes back to waiting for incoming messages on the socket.

3.5.5. Messaging

Application internal messaging

For the internal communication between the main application and the cloud backend on the Thingy:91 itself messages of the type event and an API on the cloud backend are used. The API on the cloud backend exposes the methods explained in section 3.5.3, which accept arguments to send messages. Each message contains the *appId*, which represents the entity the message is for. This can be a sensor, a button or the LED. In the *data* parameter are the corresponding values, either a single value in case of a DATA message or otherwise an attribute hash. The last attribute *messageType* contains as the name implies the type of the message. CFG_SET is used to define commands changing settings for the Thingy:91, such as the LED. *DATA* is used for any messages containing only data. There are other types available, but they are not used in the current implementation.

The messages received on the cloud backend from the cloud server get encapsulated in events mentioned in 3.5.4. As shown in the snippet below the event contains the message received, its size and the type of the event so that the main application can interpret it correctly.

```
// trigger cloud event with received data
    struct cloud_event cloud_evt = {
        .type = CLOUD_EVT_DATA_RECEIVED,
        .data.msg.buf = message,
        .data.msg.len = sizeof(message)
    };
    cloud_notify_event(coap_cloud_backend, &cloud_evt, message);
```

Network messaging

The communication uses the Constrained Application Protocol (CoAP) and serializes messages in CBOR format. The data which is sent to the cloud backend from the main application. To reduce size, the data is not sent to a single server resource containing all information for the data type sent, the cloud backend parses the data from the main application and reads the sensor data type. Depending on the type, the message will be sent to a different resource on the cloud server. The sensor data for temperature for example will be sent to the server resource at */temperature*. To send the value, the cloud backend reads the measurement data from the JSON string and encodes it in CBOR format of type float. This will then be appended to the CoAP packet and sent to the cloud server. The response does not need to be parsed, as it is just a confirmation or acknowledgement message with no payload.

The messaging for the observation thread works very similar, just the other way around. If a message is received, the payload is parsed. As the only supported command in the implemented cloud backend is the LED color change, the incoming data is a color in hexadecimal representation. To make payload size smaller and reduce parsing effort, the hexadecimal value is converted to an integer on the cloud server before being encoded in CBOR and sent. The observation thread parses the CBOR payload and converts the integer back to a hexadecimal string. This string is then used to build an event message for the main application as explained in section 3.5.5.

3.5.6. Application Flow

The execution flow of the application is similar to the initial asset tracker explained in section 3.5.2. The Figure 3.5 shows the build up and running structure of the application. On the left is the main part of the application and on the right the cloud backend. Startups and terminations of threads are indicated with black dots. The execution is split into three parts annotated on the left of the illustration.

Once the device is started, the initialization phase starts: the main thread starts a worker thread handling all the heavy work tasks the application requires. As soon as the worker thread is started, tasks to start the LTE connection and sensor initialization are delegated to the work thread. After this, the cloud backend main thread is started. If this is successful, the cloud backend's functions to initialize and connect to the cloud server are called. With the process elaborated in section 3.5.3 the already running thread for the observation of cloud server events sends its requests and starts listening for messages. With this step the initialization phase is finished and the repeated loop comes into play.

Every 15 seconds the main application thread delegates a task to read sensor data to the work thread. The returned data is passed to the cloud backend main thread using the send function discussed in section 3.5.3. Data is sent to the cloud server. While this process is done periodically, the arrival of cloud server messages is irregular. If a message comes in from the web server, the observe thread forwards the message as a cloud event presented in section 3.5.5. The message is passed to the worker and evaluated there, where necessary actions are taken.

3.6. Capabilities and Limitations

The current state of the project serves as a prototype to show the capabilities of CoAP and CBOR in this context and evaluating the used tools and technologies. It is therefore quite limited in its capabilities.

The application only checks the server connection at the connecting stage, but not further on. If the server fails in operation, data will be sent anyway by the Thingy:91 while commands will no longer be received as the server does not have any count on its observers after a restart or crash. Fail-safe mechanisms are not in place to handle such errors.

As of now, the Thingy:91 can only receive LED updates. While allowing for a very small packet size and a simple implementation, this holds off the possibility to send other commands for now. It would however not take much effort to expand it. The same goes for the sending of other sensor data. The three environmental parameters sent in the current implementation can all be represented in the format of a float. Communication of GPS information and other data would have to be implemented in separate functions.

Another limitation is that the web client can show the online status of the Thingy:91. As this only references the number of connected observers on the server, it can happen that the Thingy:91 runs into an error, loses connection or gets shut down without proper disconnection and still be on the observers list of the server.

While CoAP supports security through DTLS it is not implemented in this prototype. This is not a serious limitations though. What does become a serious limitation is the message complexity and size. In its current implementation, the web client receives a full

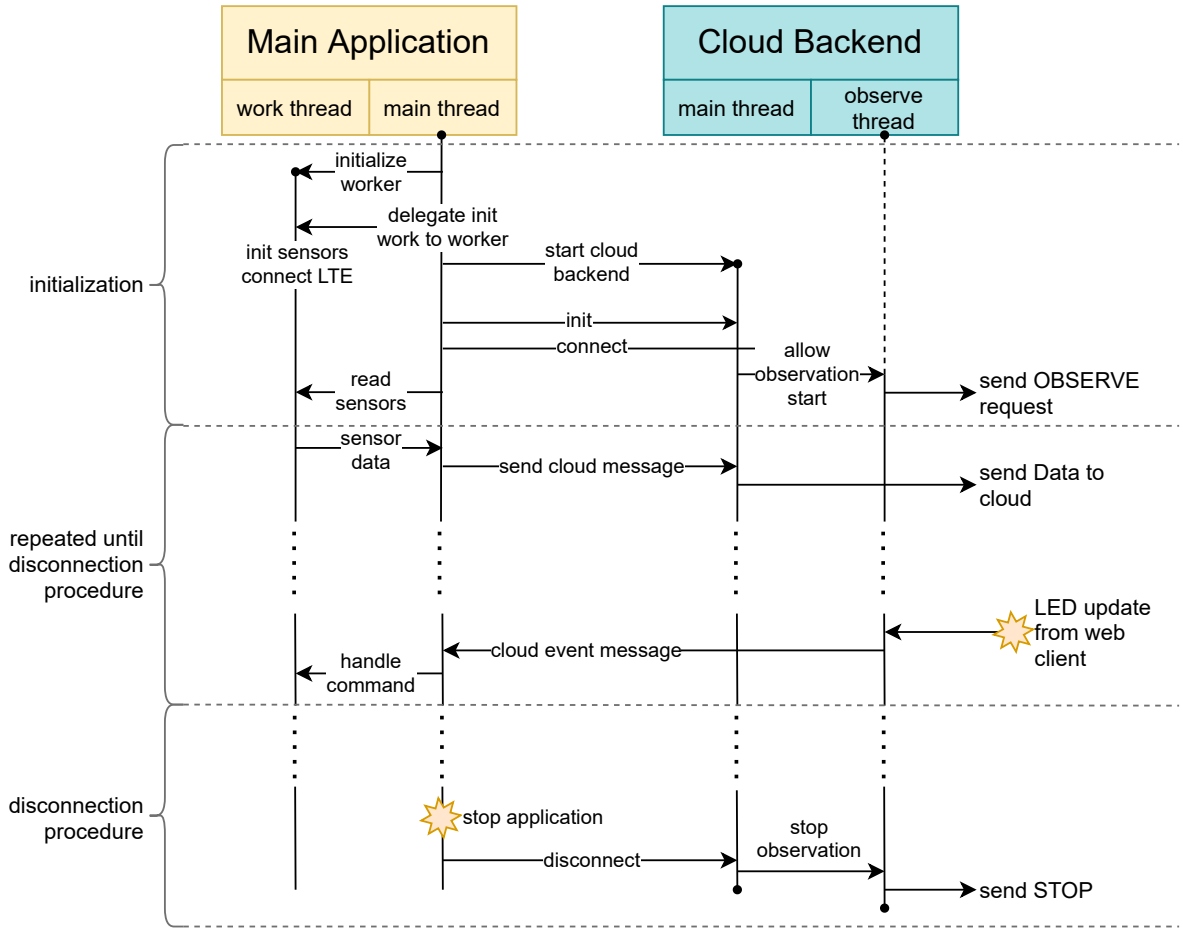

Fig. 3.5.: Internal flow and threading

JSON of all available data. After several hours of operation this can grow to a very large packet which makes the HTTP server lag behind and the web client side consumes most of the cloud backends resources, leaving little for the responsiveness on the CoAP end.

4 Evaluation

4.1. Prototype

The prototype works fairly well on first sight. The Thingy:91 connects quickly to the cloud server and properly sends data and observes LED updates. The cloud server starts without any issues and generally does its job well. They interact using CoAP and CBOR without any severe issues.

The web client is responsive and allows for good user interaction. Data is shown accordingly and the connection status as well as the LED update work in a basic form. The goals of this thesis have been reached.

There are however a few problems still open. Most of them are limited by the time and knowledge available to the developer. Cleaner code and better error handling would lead to a more stable application.

4.1.1. Communication

Though the communication works, several drawbacks hold the success of the prototype somewhat back. As already mentioned in section 3.6, the Thingy:91 does not always properly handle ACK messages and sometimes does not send ACK messages itself. Timing issues do not help either. This means that retransmissions of messages are no rarity and occur often. For an application that focuses on a lightweight application with a small

need for bandwidth and power this certainly is a problem. Another problem is the number of messages. As the base application used for this work sends a message for each data point, the same behavior now causes a lot of messages with very little content. If some of those get retransmitted the problem is even more severe. The message complexity could be reduced, but was kept at a high level to stay compliant with the base asset tracker application.

Similar problems occur on the web side of the application. A request is sent for each data type. This leads to many more requests. Each one of these requests is much larger than needed, as the time format is unnecessary large and the format of the JSON used quickly get very large. This also comes from the fact that there is no restriction on the data retrieved, the server just returns all available data, even if the messages reaches megabytes in size. A better approach would restrict the data sent to size or even better to certain date ranges. Another good solution would be paging.

Overall, the communication works but is riddled with small errors which quickly lead the application to its limits.

4.1.2. Quality

In regards to overall quality the prototype receives a similar verdict as its communication. On the surface everything is fine, but at lower levels small issues are at place. On the Thingy:91 software the capabilities of the cloud backend are quite restricted. Error handling is implemented but not always executes good recovery actions. Of course some issues are difficult to resolve on a constrained device, but connection errors, bad message transmissions or missing responses could be handled with clean fallbacks instead of error logs. The same goes for messaging on the web part. There are no fallbacks on the server and wrong formats might already cause the loss of a message. This is not a severe issue as the server only ever communicates with a device where the format of messages is controlled. The only part properly handling problems is the web client, which shows proper error dialogues to the user.

Regarding code quality, parts of the project lack refactoring. Functions often fulfill very specific purposes and cannot be reused. Many of them should be split into smaller entities. It can be argued that for a prototype this is not a severe problem, but it does prevent quick adjustments in the development process, which is a desirable trait for a prototype under development.

Both the limited implementation and the code quality can be attributed to the limited knowledge and time given for this project. For a first project in the field of IoT without prior experience the quality is sufficient.

4.1.3. Usability

From a users perspective the prototype is a success. It is easy to setup and use and provides users with easy access to measured data. Furthermore, the LED can be changed without issues. Simply turning the Thingy:91 on and taking it wherever it should take measurements is enough. If it ever loses connection it will eventually regain it by itself and start sending data again. If the LED update suddenly does not work anymore or the connection is not recovered quickly enough, the user can just restart the Thingy:91.

For a final product more stable runtime and scalability would definitely be indispensable, but for the given requirements the prototype functions well from a user perspective.

4.2. Advantages and Disadvantages

4.2.1. Technology

TCP vs. UDP layer

For the purposes of this project the UDP protocol used for CoAP is the better option. Constrained devices do not need the high reliability that TCP offers. To communicate reliably, the tokens in the CoAP protocol are definitely enough to ensure good communication. The two main advantages UDP has over TCP in this context are its size and capability to be sent without negotiating a connection first. Publishing new data on the Thingy:91 and sending it to the server does not need to be reliable. For the Thingy:91 it does not matter why exactly the connection broke; if it doesn't arrive there is no big issue. The server might have an incomplete time series, but given the type of series recorded they can easily be recovered.

The size aspect of the packets is another point. It is not the packet size itself that matters, packets can vary in size. The difference comes in the way the data is sent. It takes a lot more resources to negotiate a connection first like TCP does. At least three handshaking messages for every new connection are required. On a constrained device, this uses much more power than necessary. And as the TCP handshake cannot be skipped, there is no way to reduce this complexity. In UDP there is no connection negotiation and every message can be sent as a single package. If the user decides he needs the additional reliability he can use the specifications built into CoAP skipping handshaking and needing one less message for a reliable connection.

The drawback that UDP has in this context is its limit in size of 65'507 bytes, but as constrained devices barely ever surpass that, this is not an issue.

CoAP, MQTT and HTTP comparison

As mentioned previously, CoAP has a very small header compared to HTTP. Secondly, it runs on UDP which is—as discussed in the previous section—a better solution for this project. The other main difference between the two protocols is the OBSERVE option built into CoAP which eliminates the need for polling. With smaller packet sizes, less messages and less resource intensive creation of packets, CoAP is as expected the better protocol for an IoT application such as this one.

The comparison of CoAP with another very common IoT protocol MQTT is more interesting. Both are built to perform well on constrained devices in the IoT [8]. There are however quite some differences which disqualify MQTT for the better solution in this work. As HTTP, MQTT uses TCP, which forces the devices to build up a connection and send more messages. This makes it more resource hungry [25]. Furthermore, the header of MQTT is 2 bytes larger than the header of CoAP [10].

The most important reason why MQTT is commonly used is its capability to let publishers publish messages and reroute them to several subscribers using topics. CoAP is meant for

criteria	**HTTP**	**MQTT**	**CoAP**
transport layer protocol	TCP	TCP	UDP
security	TLS	TLS	DTLS
headersize	undefined	2 bytes	4 bytes
communication model	request/response	publish/subscribe	publish/subscribe or request/response
messaging	sync	async	sync or async
number of message types	-	14	4
RESTful	yes	no	yes
message identification	provided by TCP	provided by TCP	message tokens
multicast	no	no	yes

Tab. 4.1.: comparison of key features of HTTP, MQTT and CoAP

direct communication rather than distribution. In this work the distribution to several clients is not necessary as there are only two devices. If there would be several devices the Thingy:91 could send its data to several other clients with CoAP as well, as the UDP layer of the the CoAP protocol allows for multicasts.

Another factor rendering MQTT the less desirable option is its need for a broker. The broker is resource intensive and not meant to run on constrained devices. The idea of the broker is to forward messages from several clients to others, but the scenario in this work only has a single client. There is no need for a broker which can forward messages to subscribers.

MQTT supports direct communication with the broker, but in the chosen approach not only does the server receive data from the Thingy:91, but the Thingy:91 itself is observing the LED resource on the server. A broker does not publish messages itself and the server would therefore not be able to inform the Thingy:91 about updates on the LED resource. To have a communication going both ways, the publisher-subscriber model is not a good solution. If it is used anyway, CoAP can implement the same architecture as MQTT using less and smaller messages. Instead of complicating the system with a publisher-subscriber model, a simple REST communication with CoAP is a better solution. CoAP performs well with REST architectures [5].

To conclude it can be said that HTTP would be a bad choice, as it has too many disadvantages over the other two candidates. The race between CoAP and MQTT is closer. Both could be used for IoT projects. For the exact architecture of this project however, CoAP outperforms MQTT. Its smaller package sizes, lower message complexity and simple architecture for only two clients fulfills the requirements very well.

JSON vs. CBOR

CBOR should outperform JSON in this project. As described in section 2.5 CBOR is not only smaller, it also is created faster than JSON. In the case of this project however, the difference was not very notable. Messages only contained single integers and floats, for

python object	**CBOR**	**JSON**
data: [1,'foobar',('ad','ff'),'cool':4353,'kk':45.342,'fff':'aa']		
88 bytes	77 bytes	119 bytes
data: 22 (integer)		
28 bytes	34 bytes	51 bytes
data: "foobar" (string)		
55 bytes	40 bytes	57 bytes
data: 22.55 (float)		
24 bytes	42 bytes	54 bytes
data: ["hello", "foo", "bar"] (string array)		
80 bytes	48 bytes	72 bytes
data: [1,2,3,4,5] (integer array)		
96 bytes	39 bytes	64 bytes

Tab. 4.2.: Comparison CBOR and JSON serialized data sizes in Python (D.1)

which the representation and precision matters more than the encoding. Table 4.2 shows a comparison between encoded sizes of the two formats for different datasets. It can be argued that a JSON string requires a few more bytes than CBOR, but on such a small scale the difference is marginal. As packet sizes are around 70 bytes the few bytes for the message did not matter too much.

Testing with the python library cbor2 showed that the difference for the used data is not important for projects on this small scale. It could become an advantage if projects get larger with more devices and every byte counts. In this case better optimisations could help reduce the size of CBOR encoded data packets. For the development of the prototype the CBOR encoding was rather a disadvantage. Debugging is much more complicated as messages are no longer human readable and debugging with networking tools such as Wireshark become tedious. The only advantage CBOR brought was that handling of CBOR encoded data was to some extent simpler than parsing and handling JSON strings, as low levels languages are easier to use on binary data than handling complex string parsing.

Regarding the usage of the tinyCbor library used for the Thingy:91, the usage was very tedious. Several buffer handlers, readers and encoders have to be initialized and if an error occurs it is often very cryptic and the library offers no help resolving the issue. The incorrect and imprecise documentation don't help either. A more abstract interface would be desirable. The Python library cbor2 offers two methods, *dumps()* and *loads()* which are used to encode and decode a whole CBOR data packet. Given that CBOR is built on the idea of JSON with keys and values in a serialized format one would expect that a library has the capability to nest it. It could be that the tinyCbor library has better options to further optimize the process, but for simple applications this is most likely not necessary.

4.2.2. Zephyr Platform

The Zephyr OS is very new and built with many of the important features for its usage on constrained devices in mind. Like most open source projects, it has good documentation and supports a tremendous amount of devices. It can run on most modern chips and even runs in emulators. Not only does it run on a lot of hardware, but also uses very few resources—allowing for powerful and large applications being run on very small and constrained devices. Its incredibly high modularity allows users to configure the operating system to their needs to keep resource requirements and power consumption low, while getting the most performance out of it. All the configuration is documented in all details and a plethora of small examples help developers to get started with new projects. There are however not just good things. The operating system being fairly new on the market means that it has not yet as many users and support is sometimes hard to get as only few people might have run into the same issue and sometimes none of them was able to resolve the issue properly. The configuration for the examples can be copied and single variables are documented, but there is a lack of configuration overviews which would help new users a great deal to configure their projects themselves without reading a large documentation of very fine grained instructions.

Not only the initial configuration for projects is hard for new users, but the whole ecosystem has a high entry level skill requirement. A good understanding of the Linux build system with its configuration and Makefiles is required to achieve more than simply running the hello world examples. Even the small examples are often not explained and following the poorly commented code might already pose problems when getting started with Zephyr.

The learning curve stays quite steep even after the initial steps have been made. There is more often than not no standard way to do something, as standards just not have had the time to be developed yet. This results in developers spending a lot of time on searching the documentation for good ways to achieve their task while not knowing if there was a capability in the OS which would have done exactly what they needed.

4.2.3. Nordic nRF Connect

nRF Connect Toolkit

Expanding the Zephyr OS, the nRF-SDK does a great job adding custom functionalities to help developers prototype their application. A lot of base applications are provided which can directly be used and expanded for custom purposes. A drawback of the many example applications is that the usage is well documented, but not necessarily the inner working. This can be a problem for newbies as they have to browse trough code and try to understand the functionality while reading the documentation for all the functions. Nevertheless the samples work very well and can quickly be deployed, tested and altered using the many provided tools.

A nice feature—while a drawback at the same time—is the quick development and advancements on the SDK. Every few months a new version is released. This means older versions might quickly not work anymore. This would not be a problem when users use the newer tags, but an issue is that the documentation sometimes cannot keep up with the rapid development. The installation assistant might advise to use the stable tagged

version v1.0.0 to get started, while tag 1.3 was already available. This caused some dependency issues at times. A good example of this might be that on the version v10.0 in the requirements a Python library is required which does no longer exist[1]. In the very long list of dependencies it is very hard for an individual to find out what exactly the library did and how to replace it or find it anyway. In the tag v1.0.0 for example the dependency hub=2.0 failed because the package was no longer available. For first time users getting into nRF SDK it would be very helpful that Nordic ensures their recommended SDK version works out of the box using their instructions.

On top of the Zephyr environment the nRF Connect toolkit provides several helpful tools as explained in section 2.3.1. They allow developers to get started much more quickly. It became clear during this work, that the nRF Connect toolkit is marketed as platform independent, but it quickly showed that the version for linux systems had problems sorting out all dependencies and the build chain failed more often than not. At times, flashing of software was a problem as well. Even compilation failed from time to time, for example because Segger Studio passed incorrect command line arguements to the compiler. A common error was the option -m32 instead of using the -m32b option. There were no instructions found on how such options could be configured and the only possible solutions were to remove the added functionalities again or to use the west build tools instead of the Segger building capabilities.

The Windows version presented in most tutorials on the other hand works nearly flawless. The installation process and dependency setup is very easy and quick, the toolkit components and Segger Studio work right out of the box and compile and flash applications with ease. Drivers for flashing are directly installed during the nRF Connect installation assistant while on the linux version drivers and toolchain have to be installed manually.

There are however still small flaws in the tools which are from a users perspective very annoying. The LTE Link Monitor to supervise application output for example would not show more than a few hundred lines. After the maximum was reached, it would not show any errors or be unresponsive, but the new output will just not be displayed. Users would think their application is not running or the connection has gone bad, while the actual fault lies with the monitoring tool. Similar issues could happen when the debugging mode of the Segger Studio software was used. If the Thingy:91 ever lost connection to the cell tower, no reconnecting was possible and the debug session had to be closed and everything restarted. Another interesting thing with Segger Studio on windows was, that if the flashing of an application to a device ever failed, it would never recover again. Every try would stagnate at exactly 37% percent progress and would only every work again if the Thingy:91 was restarted and the connection to the computer was disconnected and connected again.

Such issues had no clear explanation and no direct solutions were found on the help forums as pinpointing the issue was very hard given there were no error messages. Even with error messages present, the resolution could often not directly be found as error messages are often cramped into a small non-scrollable window meaning the message could not be read nor copied. A tedious error search was often the resolution for such problems.

The issues with the provided tools are time consuming and require some perseverance from time to time, but once the most common ones are found and users can quickly

[1] https://devzone.nordicsemi.com/f/nordic-q-a/51051/nrf-connect-sdk-install-issue-hub-2-0-not-found-leading-to-build-errors

mitigate them with a short application restart the development is fast and easy. During this project there were already many improvements which leads to the conclusion that tools will soon be much more stable and a great toolkit for development.

nRF SDK

While the provided extensions and functionalities of the SDK are very helpful there is sometimes confusion on how those functionalities interact with the ones of the Zephyr OS. There are cases where both implement the same function, for example the well known linux function "recv" used to read messages from a socket. It is implemented in Zephyr as "recv" and in the nRF SDK as "nrf_recv". While they are supposed to do the same and the recv should be just a wrapper for the nrf_recv according to newest information on the development help board[2], they seem to behave somewhat different and mix very poorly together. A good advise would be to just use either the Zephyr functions or the nRF ones. The problem with this is that there might be different behaviors that one might want to access. An example of this is that the Zephyr recv function allows for the argument flag "MSG_WAITALL" which should be available in the nrf_recv as well, but fails upon compilation. Such inconsistencies are not necessarily a big deal, but cause a lot of anger and frustration when developing.

Similarly, issues with nRF libraries caused troubles as well. Specifically the CoAP library provided by the SDK for example would cause a stack overflow when initialized, even though the stack size should have been large enough according to documentation and other examples. Those are not terrible problems but can get newbies into struggling more than necessary.

Another library causing issues is the tinyCBOR library. It is included in the SDK, but not developed by Nordic. The Nordic team changed some of the specification to still work with the initial source which has changed. The documentation is not available on the nRF SDK docs and developers are redirected to the documentation of the source, which however was not updated. This lead to severe inconsistencies, where header files are named differently, and functions are either not named the way the documentation says or do not what they are supposed to. In the case of the tinyCBOR library even the small example at the very beginning of the tutorials failed, resulting in an in depth analysis of the source code of the library. This kind of problems should not occur, especially if they are already known by the Nordic team[3].

As a last point regarding libraries, the usage could be more clear. Zephyr shows the exact configurations and header files which have to be set and included for each additional library. For the nRF SDK, this is not always the case, and it can be a hassle to find the headers and options necessary to use them.

It seems that overall many functions in the nRF SDK are very capable when used for their exact purpose, which is nearly always well displayed in example applications. The problem with the extensions is that they are seemingly wrappers for different Zephyr capabilities. In the case of the CoAP client the nRF SDK extension is abstracted so much, that only very basic actions can be achieved. As mixing Zephyr and nRF works

[2] https://devzone.nordicsemi.com/f/nordic-q-a/65848/nrf_recv-vs-recv

[3] https://devzone.nordicsemi.com/f/nordic-q-a/66409/missing-documentation-on-tinycbor-for-nrfconnect-sdk-1-3-0

poorly, it is better to directly use the Zephyr functions when developing something more specific. For the very basic prototypes which just need the most basic capabilities, the nRF SDK functions are a good choice as long as they are kept up to date with the sources.

When functions of the nRF SDK do not work the same as the documentation states it is hard to find the problem, as the SDK is proprietary and the source is not always open and can therefore not be investigated. Given it is closed source there are less people looking at it and support can only be found at the devzone forum of Nordic [19]. It is nearly always just the Nordic employees who can answer questions.

In conclusion it can be stated that the additional functionalities of the SDK are very helpful at times, especially in the many examples. Using them lets developers start very quickly with their custom development and prototyping simple applications is very quick. Many things are however already present in Zephyr on a lower level, and the additions in the nRF SDK are often just wrappers with abstractions only good for the rapid and simple development of prototypes. This is not an issue, as Nordic markets their SDK as a prototyping platform, which it does well.

Thingy:91

The Thingy:91 is built as a prototyping device and therefore worked well for this project. It has all necessary tools to build any kind of simple application. The sensors work well and network connectivity behaves correctly. The firmware is stable and can easily be updated with the nRF Connect toolkit. Setup is simple as well, the Thingy:91 works right out of the box.

A drawback is its high cost. Some applications might not require GPS or high precision air quality sensors, which are expensive. The Thingy:91 is not necessarily suited for specific applications which do not require all the features. It could even be argued that for this work another device with fewer capabilities would have been enough. The advantage of the simple setup and tools the Thingy:91 and its surrounding tools of the nRF Connect environment though are most likely worth the price.

Looking at the simplicity when using it, the Thingy:91 is mostly convenient. It is chargeable over USB and has an ON/OFF switch, allowing for easy resets. One thing that could be easier is the flashing of the firmware. To install a new application a user has to either buy the development board from Nordic or get a J-Link debugger and programmer from Segger. For a device which should serve as a rapid prototyping platform quickly accessible for users, one could think it would have a separate controller on the board to allow flashing over USB. Even more so as a USB-mini port is installed and can be used to read the output of the Thingy:91.

5 Conclusion

5.1. Review

This work gave an introduction to IoT and elaborated on some of the current technologies in the field. The RTOS Zephyr, the nRF Connect development environment by Nordic Semiconductor and the CoAP protocol as well as several other technologies were explained in more detail.

With the goal to reduce bandwidth for constrained devices, a prototype was implemented. It also served as a learning experience and allowed to evaluate the used technologies. To communicate it uses the CoAP protocol and serializes messages in CBOR format. It is capable of tracking environmental data and collecting it on a central server. The server is able to deserialize the data and save it to a MySQL database. It provides the saved data to users on a web client, which also allows to change the color of an LED on the measuring device using the observe pattern incorporated into CoAP.

The final prototype was evaluated for its approach, architecture and overall design. The technologies used were examined and compared to alternatives. The quality of the prototype is not very high, as it has several smaller issues. However, as a prototype is performs well and allows for a good evaluation of the chosen approach and the used technologies.

It was found that the used technologies are well suited for the goal of connecting a constrained device to the cloud and publishing its data. The CoAP protocol outperforms HTTP and MQTT for the purposes in this project due to its superior architecture in this context, smaller message sizes and lower message complexity.

The nRF SDK proved to be a helpful tool to develop IoT applications, but it became clear that the initial knowledge required to build an application is high. With similar knowledge required, the RTOS Zephyr is not easy to setup, but runs stable and is a good solution for IoT applications.

5.2. Outlook

The prototype in its current implementation fulfills all requirements, but lacks quality in several sections. Further development would be required to make the prototype into a valuable product. Benchmarking and testing would be required to refine the used techniques and further increase performance. The next steps would be to overthink the

message complexity and introduce more reliable error handling to ensure a communication without redundant or lost messages.

5.3. Final statement

This thesis grew slowly as many problems arose from the new technologies throughout the project, hindering progress but bringing new insights. Especially the start turned out to be difficult, as unexpected errors and a defect SIM-card stalled any advancements. With perseverance and intensive debugging the project reached its success at last.

A
Common Acronyms

CSS	Cascading Style Sheet
HTTP	Hypertext Transfer Protocol
IoT	Internet of Things
JSON	JavaScript Object Notation
REST	Representational State Transfer
TCP	Transmission Control Protocol
OS	Operating System
RTOS	Real Time Operating System
CoAP	Constrained Application Protocol
CBOR	Concise Binary Object Representation
MQTT	Message Queuing Telemetry Transport
MTU	Maximum Transfer Unit
UDP	User Datagram Protocol
URI	Uniform Resource Identifier
M2M	machine-to-machine
IETF	Internet Engineering Task Force
DTLS	Datagram Transport Layer Security
JS	Javascript
DOM	Document Object Model
SDK	Software Development Kit

B

License of the Documentation

C
Website of the Project

The web client if online runs on `http://212.47.236.188:4100/` and can be accessed without credentials.

The code of the project is hosted on Github.

The firmware for the Thingy:91 is hosted at `https://github.com/Skatinger/sdk-nrf/tree/new_approach`. The repository is a fork of the nRF-SDK.

The cloud server and web client are in the repository at `https://github.com/Skatinger/coap_server`.

D

Python benchmarking scripts

D.1. Script to benchmark CBOR and JSON encoding parameters

```
import cbor2
import json
import time
import sys

def testing(data):
    # cbor
    start = time.time()
    cbor_obj = cbor2.dumps(data)
    cbor_time = time.time() - start

    # json
    start = time.time()
    json_obj = json.dumps(data)
    json_time = time.time() - start

    #print
    print("DATA:")
    print(dataset)
    print("python object: " + str(sys.getsizeof(data)) + " bytes")
    print("cbor: " + str(sys.getsizeof(cbor_obj)) + " bytes in " + str(cbor_time) + "
        ms")
    print("json: " + str(sys.getsizeof(json_obj)) + " bytes in " + str(json_time) + "
        ms")

datasets = [
    [1,'foobar',('ad','ff'),{'cool':4353,'kk':45.342,'fff':'aa'}],
    22,
    "foobar",
    22.55,
    ["hello", "foo", "bar"],
    [1,2,3,4,5]
]

for dataset in datasets:
    testing(dataset)
```

References

[1] Amin Azari, Čedomir Stefanović, Petar Popovski, and Cicek Cavdar. On the latency-energy performance of nb-iot systems in providing wide-area iot connectivity. *IEEE Transactions on Green Communications and Networking*, PP:1–1, 10 2019.

[2] Bormann C. and Hoffman P. Concise binary object representation (cbor). RFC 8949, RFC Editor, 12 2020.

[3] A. P. Castellani, M. Gheda, N. Bui, M. Rossi, and M. Zorzi. Web services for the internet of things through coap and exi. In *2011 IEEE International Conference on Communications Workshops (ICC)*, pages 1–6, 2011.

[4] Muhammad Farooq, Muhammad Waseem, Sadia Mazhar, Anjum Khairi, and Talha Kamal. A review on internet of things (iot). *International Journal of Computer Applications*, 113:1–7, 03 2015. 3

[5] M. Iglesias-Urkia, D. C. Mansilla, S. Mayer, and A. Urbieta. Validation of a coap to iec 61850 mapping and benchmarking vs http-rest and ws-soap. In *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, volume 1, pages 1015–1022, 2018.

[6] M. Kovatsch, S. Duquennoy, and A. Dunkels. A low-power coap for contiki. In *2011 IEEE Eighth International Conference on Mobile Ad-Hoc and Sensor Systems*, pages 855–860, 2011.

[7] S. Madakam, R. Ramaswamy, and S. Tripathi. Internet of things (iot): A literature review. *Journal of Computer and Communications*, 3:164–173, 2015. 3

[8] N. Naik. Choice of effective messaging protocols for iot systems: Mqtt, coap, amqp and http. In *2017 IEEE International Systems Engineering Symposium (ISSE)*, pages 1–7, 2017.

[9] S. Raza, H. Shafagh, K. Hewage, R. Hummen, and T. Voigt. Lithe: Lightweight secure coap for the internet of things. *IEEE Sensors Journal*, 13(10):3711–3720, 2013.

[10] Vasil Sarafov. Comparison of iot data protocol overhead. *Network Architectures and Services*, pages 7–14, 2018.

[11] Shelby Z., Hartke H., and Bormann C. The constrained application protocol (coap). RFC 7252, RFC Editor, 06 2014.

Referenced Web Resources

[12] aiocoap project page. https://aiocoap.readthedocs.io/en/latest/ (accessed January 02, 2021).

[13] aiohttp project page. https://docs.aiohttp.org/en/stable/ (accessed January 02, 2021).

[14] asyncio library documentation page. https://docs.python.org/3/library/asyncio.html (accessed November 08, 2020).

[15] Narrow-band IoT. https://www.bearingpoint.com/de-ch/unser-erfolg/insights/narrow-band-iot/ (accessed January 09, 2021).

[16] cbor2 library documentation page. https://cbor2.readthedocs.io/en/latest/usage.html (accessed November 12, 2020).

[17] cJSON Github repository. https://github.com/DaveGamble/cJSON (accessed November 15, 2020).

[18] Offical CoAP page. https://coap.technology/ (accessed January 05, 2021).

[19] Nordic Semiconductors Devzone. https://devzone.nordicsemi.com/ (accessed January 06, 2021).

[20] Differences between LTE-M and NB-IoT. https://www.digi.com/videos/what-are-the-differences-between-lte-m-and-nb-iot (accessed January 09, 2021).

[21] Docker product page. https://www.docker.com/ (accessed January 13, 2021).

[22] Free Documentation Licence (GNU FDL). http://www.gnu.org/licenses/fdl.txt (accessed July 30, 2005).

[23] jQuery project page. https://jquery.com/ (accessed November 11, 2020).

[24] Materialize project page. https://materializecss.com/ (accessed November 11, 2020).

[25] Offical MQTT page. https://mqtt.org/ (accessed January 02, 2021).

[26] MySQL product page. https://www.mysql.com/ (accessed November 11, 2020).

[27] Plotly project page. https://plotly.com/javascript/ (accessed December 23, 2020).

[28] QEMU project page. https://www.qemu.org/ (accessed November 11, 2020).

[29] IoT Insights. https://www.telenorconnexion.com/iot-insights/lte-m-vs-nb-iot-guide-differences/ (accessed January 09, 2021).

[30] west Github repository. https://github.com/zephyrproject-rtos/west (accessed November 12, 2020).

[31] Zephyr Project. https://zephyrproject.org/ (accessed January 08, 2021).

[32] Zephyr Project Documentation. https://docs.zephyrproject.org/latest/ (accessed January 12, 2021).