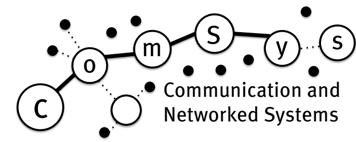
FACULTY OF
COMPUTER SCIENCE



Communication and Networked Systems

Thesis

Design, implementation and testing of a CoAP library on a constrained system for air quality measurements

Lars Hübner

Supervisor: Prof. Dr. rer. nat. Mesut Güneş
Assisting Supervisor: MSc. Kai Kientopf

Institute for Intelligent Cooperating Systems, Otto-von-Guericke-University Magdeburg

July 31, 2019

Abstract

Constrained Application Protocol (CoAP) is an Internet of Things (IoT) protocol for use on constrained devices. We explore if it could be used as a viable alternative to Hypertext Transfer Protocol (HTTP) on an esp8266 based air quality sensor. Therefore, we use the luftdaten.info project that provides assembly instructions and a firmware for measuring particulate matter concentrations using an esp8266 and a SDS011. We extend this firmware with a self written library to support CoAP. Furthermore, we us a cross-protocol proxy to push sensor values to the HTTP based Application Programming Interface (API).

In comparison to the HTTP base version, our implementation requires less flash memory. When using CoAP it takes longer to push values to the API because it takes some time to construct a CoAP message and a full HTTP exchange still takes place. CoAP uses less bandwidth than HTTP. Largely because of the smaller header size in User Datagram Protocol (UDP) but also because of how CoAP encodes information.

We implement and test the CoAP observe extension. Notifications are triggered using a threshold based strategy. The observe extension saves a large number of exchanges especially when monitoring temperature or humidity.

CoAP can be used in combination with a cross-protocol proxy to work on a HTTP API especially when no custom options are required. When bandwidth is of importance, it is also suitable.

Contents

List of Figures vii

List of Tables ix

Listings xi

Acronyms xiii

Glossary xv

1 Introduction 1
- 1.1 IoT . . . 1
- 1.2 Hardware . . . 2
- 1.3 Protocols and OSI Layer Model . . . 2
  - 1.3.1 TCP . . . 2
  - 1.3.2 UDP . . . 3
  - 1.3.3 HTTP . . . 3
- 1.4 CoAP . . . 4
  - 1.4.1 Message Structure . . . 4
  - 1.4.2 Message Types . . . 4
  - 1.4.3 Method Names and REST . . . 5
  - 1.4.4 Messaging Model . . . 5
  - 1.4.5 Options . . . 6
- 1.5 Reliability . . . 6
- 1.6 Observe . . . 7
- 1.7 Proxy operation . . . 7

2 Implementation 9
- 2.1 Airrohr-Firmware . . . 9
- 2.2 Development setup . . . 10
- 2.3 CoAP Library . . . 11
  - 2.3.1 Basic Structure . . . 12
  - 2.3.2 Confirmable . . . 13
  - 2.3.3 Classes . . . 15
  - 2.3.4 Options . . . 15
  - 2.3.5 Observe . . . 15

2.4 Client 16

3 Evaluation **17**

3.1 Custom options 17

3.2 Changes to Californium 19

3.3 Memory Usage 19

3.4 CoAP vs. HTTP: time to send 20

3.5 CoAP vs HTTP: bandwidth usage 21

3.6 CoAP Observe 23

4 Conclusion **25**

Bibliography **27**

List of Figures

1.1 OSI Layers 2
1.2 CoAP message structure 5
1.3 CoAP option structure 6

3.1 Sketch of test environment 18
3.2 Size of memory segments 20
3.3 Time to send comparison 21
3.4 Duration of TCP connections 22
3.5 Push size 22
3.6 Observe exchanges 24

List of Tables

3.1 Comparison of used memory 20

Listings

2.1 Library testbed 11

3.1 Proxy: Proxy.properties 19
3.2 Proxy: OptionNumberRegistry.java 19
3.3 Proxy: ProxyHttpClientResource.java 19

Acronyms

ACK Acknowledgement. 5–7, 15, 23

API Application Programming Interface. iii, 2, 9, 10, 17–22, 25, 26

CoAP Constrained Application Protocol. iii, 1, 2, 4–7, 9, 11–13, 15–21, 23–26

CON Confirmable. 5, 7, 11, 13, 23

CoRE Constrained RESTful Environments. 4

GPS Global Positioning System. 2

GSM Global System for Mobile Communications. 25

HTML Hypertext Markup Language. 5, 9

HTTP Hypertext Transfer Protocol. iii, 1–5, 7, 10, 11, 17–21, 23, 25, 26

ID Identifier. 4, 5, 15

IDE Integrated Development Environment. 10

IO Input/output. 9

IoT Internet of Things. iii, 1, 3, 25, 26

IP Internet Protocol. 1–3, 15

JSON JavaScript Object Notation. 9, 10, 21

mDNS multicast DNS. 10

MQTT Message Queuing Telemetry Transport. 26

NON Non-confirmable. 5, 20, 21, 23

NTP Network Time Protocol. 10

OSI model Open Systems Interconnection model. 2

PC Personal Computer. 11

RAM Random-Access Memory. 20

REST Representational State Transfer. 2, 4, 5, 7, 9, 19, 20, 25, 26

RST Reset. 5, 7, 12, 13, 15

TCP Transfer Control Protocol. 1–3, 10, 21–23

UDP User Datagram Protocol. iii, 2, 3, 5, 6, 11, 12, 25

URI Uniform Resource Identifier. 3, 7, 12–14, 16

WWW World Wide Web. 3

Glossary

Airrohr-firmware Firmare provided by the luftdaten.info project. 1, 2, 9, 10, 16, 19, 20, 25, 26

Arduino An open-source computing platform. 2, 9–11

Arduino core for esp8266 Version of the Arduino environment for esp8266. Includes a WiFi stack. 2, 9, 25

avr-size Utility to list section sizes of a binary file. 19

binding OpenHAB component that provides an interface to interact with physical device. 16

Boost.Asio Asynchronous IO library. We use it for networking. 10, 12

BSS Uninitialized static variables. 20

Californium Eclipse Californium is a Java framework implementation of CoAP.. 16, 17, 19

CBOR Concise Binary Object Representation is an alternative to JSON with a focus on small message and code size. 26

channel A connection between a thing and an item in openHAB. 16

Data Global or static variables with initial values. 20

DHT11 Combined temperature and humidity sensor. 2, 23

Docker Container based virtualization software. 17

DTLS Datagram Transport Layer Security: A transport layer protocol that provides security for datagram based applications. 2

esp8266 The esp8266 is a low cost system on a chip including a WiFi module. iii, 2, 17, 21, 23, 25

flash memory Non volatile memory. iii, 20

openHAB open Home Automation Bus: A java/OSGi based home automation platform. 16, 23

OSGi Open Service Gateway initiative. Description of a modular system and service platform for Java. 16

OTA update Over the Air update. The process of updating the firmware using a WiFi connection. 25

PlatformIO IDE for IoT devices. Used to build Airrohr-firmware. 10

SDS011 Particulate matter sensor for pm10 and pm2.5. iii, 2, 23

Text Code segment. Contains executable instructions. 20

WiFi Wireless fidelity. Collection of technologies used in wireless networks. Also a trademark by the Wi-Fi Alliance. 2, 9, 10, 17, 25

Wireshark Tool to capture and analyze network traffic. 18, 21

CHAPTER 1

Introduction

In recent times it has become clear, that particulate matter pollution has severe adverse health effects[1]. So measuring particulate matter pollution is an important task.

Policymakers and citizens could benefit from improved data quantity. We could measure the effectiveness of policies and improve public health measures. For example, there have already been a number of driving bans due to high particulate matter concentration and cities started to ban certain vehicles in low emission zones[2]. Such measures are controversial and the ability to independently analyze or proof their effectiveness could be very useful.

At the same time small devices have become more widespread. The Internet of Things (IoT) can be applied to a wide range of problems, including measuring air quality data. Low cost of devices makes them accessible to a lot of people.

With the Hypertext Transfer Protocol (HTTP) on the Transfer Control Protocol (TCP)/ Internet Protocol (IP) stack there is already an established infrastructure. A large number of projects depend on this technology however the advent of IoT led to the development of a number of specialized technologies.

The luftdaten.info project is such an effort based on existing technologies. We will explore whether a more IoT specific approach could be beneficial. To that purpose we develop a Constrained Application Protocol (CoAP) library and use it to replace some of the functions on the Airrohr-firmware.

1.1 IoT

Though the concept has been established earlier, IoT is a term that was coined by Kevin Ashton in 1999[3]. The idea is that all possible devices will be connected to the internet and to each other which allows for a wide range of applications, for example, in home automation, production control, environmental monitoring, logistics ,or energy management. Researchers predict a large increase in connected devices during the coming years[4].

These devices need to be cheap and energy efficient because they may need to be powered by battery and communicate over a wireless network. Reducing the amount of transmitted

7-5	Application layer	HTTP, CoAP
4	Transport	TCP, UDP, DTLS
3	Network	IP
2	Data link	Ethernet, IEEE 802.11
1	Physical	

Figure 1.1: OSI Layers

data is an important part of lowering the energy consumption.

1.2 Hardware

The luftdaten.info project is being developed by OK Lab Stuttgart, a non-profit organization that uses open data to further public interest. It provides assembly instructions along with a firmware to provide a low cost solution for measuring air quality data for public and personal use. The collected data is then aggregated and displayed in a map. Readings are also accessible via a Representational State Transfer (REST) Application Programming Interface (API). This firmware is called Airrohr-firmware and is based on the Arduino platform.

A minimal setup consists of an esp8266 that functions as a base and 1 particulate matter sensor. Functionality can be extended by adding other sensors, displays ,or a Global Positioning System (GPS) module. In this project the following components will be used:

- The esp8266, a low cost WiFi chip with a TCP/IP stack.
- A DHT11 for measuring temperature and humidity.
- The SDS011 for measuring particle concentration.

The esp8266 uses a 32-bit RISC CPU running at 160 MHz. It typically has between 512 KiB and 4 MiB flash memory.

The Arduino core for esp8266 project makes it possible to program an esp8266 using the Arduino platform.

1.3 Protocols and OSI Layer Model

The Open Systems Interconnection model (OSI model) is a conceptual model to visualize the interdependencies of networking systems shown in Figure 1.1.

The Internet Protocol is network layer protocol that delivers data packets from source to destination based on IP addresses.

1.3.1 TCP

TCP provides a reliable, ordered, error-checked stream of bytes. Most World Wide Web (WWW) applications rely on TCP and it is usually encapsulated in IP. TCP is connection

based so each time a connection is established a handshake needs to be performed which leads to considerable overhead if a lot of short connections need to be made[5]. If packets are lost or corrupted in transmission, they need to be retransmitted. To correct the order or detect if packets are missing, each TCP header has an acknowledgment and a sequence number.

To detect data corruption a checksum is used.

For supporting all of those features TCP needs relatively large headers. A header without options is 20 bytes long[6].

1.3.2 UDP

User Datagram Protocol (UDP) is a datagram based, connection less protocol. Addressing is based on IP. It is not reliable and non secure. Like TCP it also has a checksum for error detection in each packet, but if other features are needed, they need to be handled in the application layer.

However, it is much lighter than TCP which makes it an appropriate choice for IoT use cases.

1.3.3 HTTP

The HTTP is an application layer protocol primarily intended to transfer documents in the WWW[7]. It follows a request-response paradigm, meaning endpoints will take the roles of client and server. Clients can make requests by sending a string with the method name followed by the Uniform Resource Identifier (URI) and the protocol version. This is optionally followed by a number of header fields and a message body. Line break characters mark the end of a line and there is a blank line between the message body and the initial part. The server then responds with a status code and also with header and payload data. HTTP assumes that the underlying transport layer protocol is reliable. So bytes sent stay the same and their order is intact. It is normally used on top of TCP/IP.

HTTP specifies a number of methods which tell the server what action should be performed on the request's resource. A URI is used to reference the resource.

Among others, clients can use the following methods:

- GET: Retrieve representation of a resource. No other effects.
- POST: Submit enclosed payload to resource.
- PUT: Replace or update the representation with payload.
- DELETE: Remove the indicated resource.

The GET method is safe so it should not change the state of a server. GET, PUT and DELETE are idempotent which means that multiple application of the methods with the same parameters should lead to the same outcome.

REST

HTTP follows the REST architectural style, a term that was defined by Roy Fielding[8]. It states constraints that, when followed, lead to a number of useful properties.

For example the constrained that a client-server architecture should be used leads to a decoupling of components which allows them to be separately developed. Another constraint is statelessness. Each request should contain all information necessary to process it. This improves scalability because a server does not have to maintain client status data anymore.

1.4 CoAP

CoAP is part of the Constrained RESTful Environments (CoRE) project[9]. It is designed to be simple and lightweight but still extensible so that it can be used on systems with very limited resources.

Like HTTP, CoAP uses the REST style. However, opposed to HTTP it uses an asynchronous messaging model.

1.4.1 Message Structure

CoAP aims to reduce message fragmentation by reducing message overhead. Each message has a 4 byte header followed by a number of optional elements. Among others, information about the CoAP version and the messages code is part of the header.

The code field contains either a method or a response code. Those codes are divided into classes. The first 3 bits of the message code represent the general class while the remaining 5 provide more detail.

Messages can be identified by a message Identifier (ID) and a token. The message ID which is part of the header should be used for deduplication.

A token should serve as an exchange identifier for the client in case the request-response pair spans multiple messages. This is used in, for example, CoAP observe or blockwise transfer. Tokens should be non trivial to prevent possible spoofing and they are at most 8 bytes long which allows for one-time use throughout client lifetime[10]. Their length is part of the header.

A number of options are specified each of which has a 1 or 2 byte header depending on the options' length.

A payload marker separates the rest of the package from a possible payload.

Figure 1.2 shows how these elements are arranged in a packet.

1.4.2 Message Types

There are 4 different message types: Confirmable (CON), Non-confirmable (NON), Acknowledgement (ACK), and Reset (RST). NON messages can be used if reliability is not

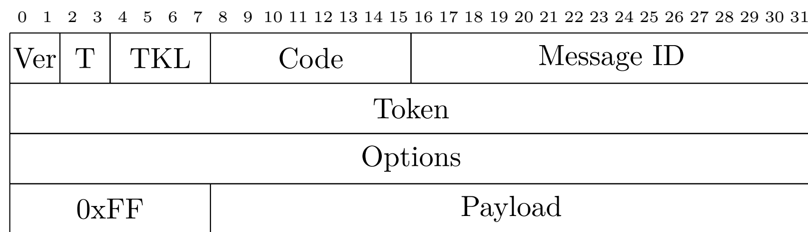


Figure 1.2: CoAP message structure

required. CON messages expect a response to compensate for UDP's lack of reliability. This response needs to be either an ACK or a RST. The former will be sent if a message has been received and processed correctly, while the latter indicates that the server has received a request but can't process it because it lacks context. Clients can also intentionally trigger a RST message to get information about the server's liveliness. This concept is called CoAP-Ping.

1.4.3 Method Names and REST

CoAP specifies a generic web protocol which realizes a subset of REST. As with HTTP, data can be modified with the methods GET, PUT, DELETE and POST. Because of that similarity, web resources can easily be used through intermediaries.

The methods defined are:

- GET: Retrieves a resource representation for the specified resource ID.
- POST: Process the resource representation. That usually means create.
- PUT: Create or update enclosed representation.
- DELETE: Requests that the resource should be deleted.

1.4.4 Messaging Model

The client-server model used by CoAP is similar to the one of Hypertext Markup Language (HTML). Clients send requests to the server which processes the request and responds. Ordering is not guaranteed because UDP is used to transport the CoAP messages. With tokens and message IDs CoAP provides a mechanism to match requests with responses. CoAP can be seen as a two-layer system where a message layer deals with UDP's lossy and asynchronous nature while a request response layer specifies message codes.

A CON will be retransmitted until it can be matched to an ACK or RST response. For communication where reliability is not needed NON messages can be used. A server will still respond with RST if it is not able to process the message.

CoAP's interaction model follows the client-server scheme. A CoAP request sends a method code to the server which returns a response code.

0 1 2 3	4 5 6 7
Option Delta	Option Length
Option Delta extended	
Option Length extended	
Option Value	

Figure 1.3: CoAP option structure. In case of option delta or length larger than 12, the extended fields will be used.

1.4.5 Options

Each CoAP message can have a number of options. Options consist of a number and an elemental data type which can be either opaque data, a string, an integer, or an empty value along with its length.

Some options have default values which can be assumed to be set, so they don't need to be included in the message. Options such as the Uri-Path can be included multiple times.

Each option has a 1 byte header which contains information about both option number and option length. Because we only have 4 bits for each value, the first byte can be followed by additional option number and length information. Figure 1.3 shows how options are structured.

Option classes

It is a feature of CoAP that option numbers provide information about their class. Options can be either elective or critical. Unrecognized elective options will be silently ignored while unrecognized critical options result in a error responses. Options with even number are elective while options with odd numbers are critical.

Delta Encoding

Option numbers won't be used directly. To save bandwidth only an option delta will be used for each option. The option number is the sum of its delta and preceding options number. Therefore, the option number can be calculated by summing up all previous option deltas.

1.5 Reliability

Since CoAP works on top of UDP, a mechanism for dealing with lossy environments is specified. If no ACK is received, requests will be retransmitted with exponential back-off. That means: each time the retransmission timeout is reached, the request will be retransmitted with the retransmission timeout doubled, until either the maximum number of retransmissions is reached or we receive a ACK or RST. Reliable messages need to be marked as CON. A recipient must either respond with an ACK or send a RST if the request

can not be processed. If the sender receives no response, the message will be resent in exponentially increasing intervals until the maximum number of retransmission is reached.

1.6 Observe

CoAP's Observe extension allows one to continuously receive updates to a resource without sending a request each time[11]. Messages that update a resource status are called notification. The observe option controls behavior depending on whether it is part of a request or a notification. If the observe option is set to 0 in a request, the endpoint shows its desire to receive a notification when the observed resource changes. One indicates that the client does not wish to receive further updates.

In a response the presence of the Observe Option shows that the message is a notification while the option value allows the client to reorder messages so that their order is equal to the one in which they were sent.

1.7 Proxy operation

A proxy server acts as an intermediary in the request-response pattern.

Forward proxies relay requests on behalf of local clients to external resources. They are also often used for caching. Reverse proxies relay requests to multiple servers. They are transparent to a client meaning it won't be aware that it is communicating with a proxy. A cross protocol proxy allows translation of one protocol into another.

Since CoAP closely follows the REST paradigm it is possible to map CoAP method names to HTTP method names. Similary we can map response codes, data types and URIs. This allows us to use a CoAP to HTTP cross-protocol proxy to access web-resources and translate the responses back to CoAP. Therefore, a cross proxy needs to understand both CoAP and HTTP[12].

The CoAP specification provides several options to indicate how proxy servers should handle messages. When the Proxy-Uri option is present, a proxy server will forward the request to this URI. The Proxy-Scheme option allows clients to construct the URI by creating a CoAP-URI from Uri-* options and replacing the original URI scheme with the one specified via option.

CHAPTER 2

Implementation

The primary goal of this work is to include CoAP capabilities into the Airrohr-firmware. To realize that, we need to create a library that is able to run in the Arduino environment. That leads to a number of restrictions:

- C++ program code.
- No multithreading: block as short as possible.
- Networking: on Arduino we have to use the WiFi interface provided, so we can not use a multi platform networking socket. In this case we use the WiFiUPD class provided by Arduino core for esp8266.

2.1 Airrohr-Firmware

It reads each configured sensor every 145 seconds and sends the data to a REST web service. The CoAP extension should enable individual queries of sensor data. It returns the last available value. Alternatively it can send a separate response once a value gets updated.

The Airrohr-firmware saves a JavaScript Object Notation (JSON) configuration file which persists between hardware resets. This file contains information about attached sensors, WiFi authentication data, APIs to use as well as the interval to push sensor values. Depending on the sensor, the Airrohr-firmware uses an external library or talks directly to the device.

If there is no existing configuration or no connection can be established to a configured WiFi network, a soft access point will be hosted to provide a configuration interface.

To serve web pages to a user a number of helper functions have been defined. Those functions build a HTML document.

In the Arduino environment a program consists of a setup function, that is being run once, and a loop function, that runs continuously.

Setup phase:

1. Serial Input/output (IO).

2. Read configuration file.
3. Initialize optional external displays.
4. Start web server.
5. Connect to WiFi.
6. Synchronize with Network Time Protocol (NTP) server.
7. Run HTTP auto update.
8. Create Auth strings.
9. MDNS: add service HTTP on TCP port 80.
10. Initialize timekeeping variables.

Loop phase:

1. Update timekeeping variables.
2. Every second: check whether air quality sensors need to be started because they require warm-up time.
3. Handle outstanding HTTP requests.
4. Temperature and humidity sensors deliver a result immediately, so they are only checked when a data string needs to be generated.

For every connected sensor its value is appended to a JSON formatted string. If pushing to the luftdaten.info API is enabled, we additionally send a single HTTP POST request for every sensor. This request can contain multiple readings. In our case we send 1 request that contains temperature and humidity and 1 request that contains particulate matter concentrations. During this time the web server is disabled.

5. Stop web server.
6. Send to luftdaten.info API.
7. Send to optional APIs.
8. Restart web server.
9. Check for auto update.
10. Check if WiFi still connected.
11. Reset for next sampling.

2.2 Development setup

The Airrohr-firmware is written in the Arduino IDE. Because of that, most functionality is in an .ino file which is a combination of a C++ header and source file. To simplify the development process we use PlatformIO development tools.

To increase turnaround time, simplify debugging and to provide a simple testbed for the library, we use a small program that runs on a system supporting Boost.Asio[13] and the standard C++ threads library. We use Boost.Asio for its network abstraction and the threads library to interact with the loop while keeping its structure similar to an Arduino

program. Listing 2.1 shows how to set up a simple CoAP server.

We can check weather we are building for a Linux Personal Computer (PC) or the esp8266 by using `#ifdefs ESP8266` and `__linux__`. This allows us to use the same codebase for offline testing and deployment to the esp8266.

```
1  #include <coap>
2  Coap coap(5683);  // Create coap object that listens on port 5683
3
4  // Example callback function: create a response message based on the incomming request
5  Message get_hello(Message p_request) {
6    std::string s = "World";              // Create response payload
7    std::vector<uint8_t> payload(s.begin(), s.end());
8
9    Message resp;
10   resp.init(p_request);                 // Mirror token, ID & set destination
11   resp.set_type(Message::TYPE::ACKNOWLEDGEMENT);    // Set response message type
12   resp.set_code(69);                    // 2.05 Content
13   resp.set_payload(payload);                // Allways respond with "World"
14   return resp;
15 };
16
17 void setup() {
18   coap.begin();
19   coap.get("/hello", get_hello);
20 }
21
22 void loop() {
23   coap.work();
24 }
25
26 int main(int argc, char** argv) {
27   setup();
28   while(true) loop();
29 return 0;
30 }
```

Listing 2.1: Shows how to use the CoAP library to respond with ”World” if we receive a GET request with the URI ”/hello”.

2.3 CoAP Library

On the Arduino platform there are already a number of librarys following a request response model most notably the HTTP server module. Because of CoAP's close connection to HTTP we try to keep the programmer facing part similar.

The simplest way of getting sensor values from the device with CoAP is requesting them with a CON-GET request with a piggybacked response. This task can be broken down into the following steps:

1. Read incoming datagram from UDP socket.
2. Create a usable data structure from raw UDP payload.
3. Handle the request. In this case: Retrieve sensor value and create a response.
4. Convert the response to raw byte data.
5. Send response back to source of request.

We also want to use some of CoAPs extra features to simplify testing and increase compatibility with generic CoAP client implementations.

- Respond to a CoAP ping.
- Respond to a .well-known/core query with a list of available resources.

Responding to a CoAP ping means answering with RST to an empty GET. To do that, we need a way to add Options to the response.

Based on these requirements we create a basic structure for the library.

2.3.1 Basic Structure

To implement the request-response model we need the ability to send and receive valid CoAP messages. One UDP packet contains exactly 1 CoAP message so whenever there are bytes available to read, the buffer contains at least one CoAP packet. Both WiFiUDP and Boost.Asio allow reading the payload of one such packet even if more packets are buffered.

```
Input: UDP socket
if bytes available then
    buffer ← read packet from socket;
    message ← construct-message(buffer) ;       /* Call Algorithm 2 */
    handle message ;                             /* Call Algorithm 4 */
end
```

Algorithm 1: General structure

```
Input: byte array with message
Result: message object
create message object with information from CoAP header;
if token length > 0 then
    copy token;
end
parse options ;                                  /* Call Algorithm 3 */
```

Algorithm 2: Construct message

Algorithm 1 outlines the main server loop. Algorithm 4 shows how we create a message object. For that we need to parse the messages options which is show in Algorithm 3.

Now that we have the information of a message in a usable format, we can decide how to process it. If the incoming message is a GET request, we need to generate a response depending on the Uri-Path. Because this is a library, users need to be able to supply a programmatic response. In this library we keep a key-value map of function objects with a URI string as key and callback functions as value. A callback function takes a message as parameter and must return a CoAP message which will be sent back to the endpoint of the original request. A function is provided that allows users to add such key-value pairs. To

Input: byte array containing CoAP message
Result: message with options set
start at beginning of option block;
while *not at the end of packet* **do**
  get option delta and option length from option header;
  switch *option delta* **do**
    case *13* **do** option delta = next byte + 13;
    case *14* **do** option delta = next 2 bytes + 269;
    case *15* **do** possible payload marker;
  end
  switch *option length* **do**
    case *13* **do** option length = next byte + 13;
    case *14* **do** option length = next 2 bytes + 269;
  end
  calculate option number;
  convert option payload data to C++ data type;
  add option to message;
end

Algorithm 3: Option parsing. Because of CoAP's delta encoding, we need to get all options in a single pass.

get keys in the correct format, we need to create the URI string from all Uri-Path options of the request.

We can also use this structure to get a list of valid URIs. During the initialization phase we add a function object with the string "/.well-known/core" as key, which creates a list of all keys and sends it back in core-link-format[14].

If the request is malformed or can not be processed, we answer with appropriate error or reset messages. For example: if we receive an empty CON message, we respond with RST. This behavior is also used as a liveliness test (CoAP-Ping). If we don't want to send a response we can just return. The message handling process is shown in Algorithm 4.

Sending works by taking a message object and creating a valid CoAP packet from it. We basically reverse the receiving process. Because options will be stored in a vector and constructed by iterating through it, the programmer needs to add options in the correct order.

2.3.2 Confirmable

Communication in lossy networks is one of CoAPs central features. To implement that, we need to keep track of some additional information. Unanswered requests will be stored in a vector of messages. We also add variables for a retransmission counter, the point in time when another attempt to deliver the message will be made and the destination endpoint. The standard C++ chrono library will be used to keep track of the elapsed time. CoAP gives us default values for transmission parameters. So confirmable sending is initiated by

```
Input: request message class instance
Output: response message
if type of request = ACK then
    clean up outstanding CON sends;
    update timeouts for observers;
else
    switch message code do
        case 0 do
            send RST ;                                          /* CoAP Ping */
        case 1 do
            construct URI string from Uri-Path elements;
            create response by getting and applying callback functino from key-value
              map;
        otherwise do
            send RST;
        end
    end
    if type of resonse = CON then
        send to destination;
        add to send queue;
    else
        send to destination;
    end
end
```

Algorithm 4: Message handling

settings these values and adding the message to the send queue.

Messages will be erased from the queue when we receive an ACK or RST that matches message IDs, if the retransmission counter reaches its maximum allowed value or the whole process exceeds the maximum transmit span parameter.

2.3.3 Classes

We define a number of classes to structure the libraries functionality.

Users create 1 instance of the CoAP class with the desired port number as a parameter.

The Message class represents a CoAP message. It provides functions to query and set header and option information in standard C++ data types. Additionally it contains the retransmission counter, timeout information and its destination IP address and port.

The Options class is a collection of all options set for the current message. It provides functions to get option values and set values and their corresponding option number.

2.3.4 Options

CoAP options use 3 different elemental data types.

- Opaque for which we use a byte vector.
- String for which we use `std::string`.
- Unsigned integer where we use `uint32_t`.

Some options can also be empty.

Repeatable options may appear multiple times. In this case we save the value in a vector of the underlying data type. Set and get functions handle management of that process. Repeatable options are added in the order in which the user sets them.

2.3.5 Observe

To keep track of observers and their status we need to store the token of the original request, the IP address, the port number and timeout information.

The user creates an observable object and 2 support functions. One function returns a Message that contains a notification with the updated resource. Every time the resource status changes, this Message will get sent to all observing endpoints with their respective token filled in. We set the observe option value to the least 3 bytes of the elapsed time as recommended by the specification [11].

The other function handles basic management of the observable object. If a GET indicates an observe request, we add the source of this request to the list of observers along with its token and send the first notification. Likewise, the message source will be removed if it unsubscribes.

If we receive an ACK to a notification, we reset the timeout information. Clients that do not respond to notifications will be removed from the list of observers when their timeout

exceeds a threshold.

2.4 Client

To retrieve readings with a client, we set up the Airrohr-firmware so that each sensor gets a URI.

While generic CoAP clients can be used in this example we use openHAB and the eclipse Californium framework to implement a client.

openHAB is an OSGi based home automation solution. Because Californium is available for the OSGi infrastructure we can directly use it in an openHAB binding. We implement a channel for each provided measurement. Then we establish CoAP observe relationships for each channel and update the channel status with content from the notifications.

CHAPTER 3

Evaluation

We want to analyze whether it could be beneficial to supplant HTTP based communications functionality with CoAP. It is expected that CoAP needs less bandwidth due to the nature of the protocols design as well as the used transport layer protocol. CoAPs close relationship with HTTP means we can use existing web services relatively easily.

To evaluate the implementation we set up a test environment:

- Docker container with luftdaten.info API.
- CoAP-proxy from Californium with modifications to handle 2 specific custom options.
- An esp8266 configured to use the local API.

We connect those components through a WiFi access point so that all the traffic can be captured in the order in which it is generated. Figure 3.1 shows the components and how they are connected.

3.1 Custom options

Since the luftdaten.info API requires HTTP POST request to have the custom header fields X-Pin and X-Sensor, we need a CoAP to HTTP proxy application that can transform outgoing CoAP POST requests to HTTP POST requests with the header fields added. To achieve this we need to add the necessary information to the CoAP request and use a proxy that is able perform the transformation. CoAP allows us to define custom options which we can use to achieve the former. For the latter we need to modify an existing CoAP to HTTP proxy since a generic way to map options to HTTP headers is not part of the specification. Option numbers 6500 to 65535 are reserved for experiments. Our custom options need to marked as Critical and Unsafe so the least two significant bits need to be set to 1. We choose option numbers 65003 and 65007.

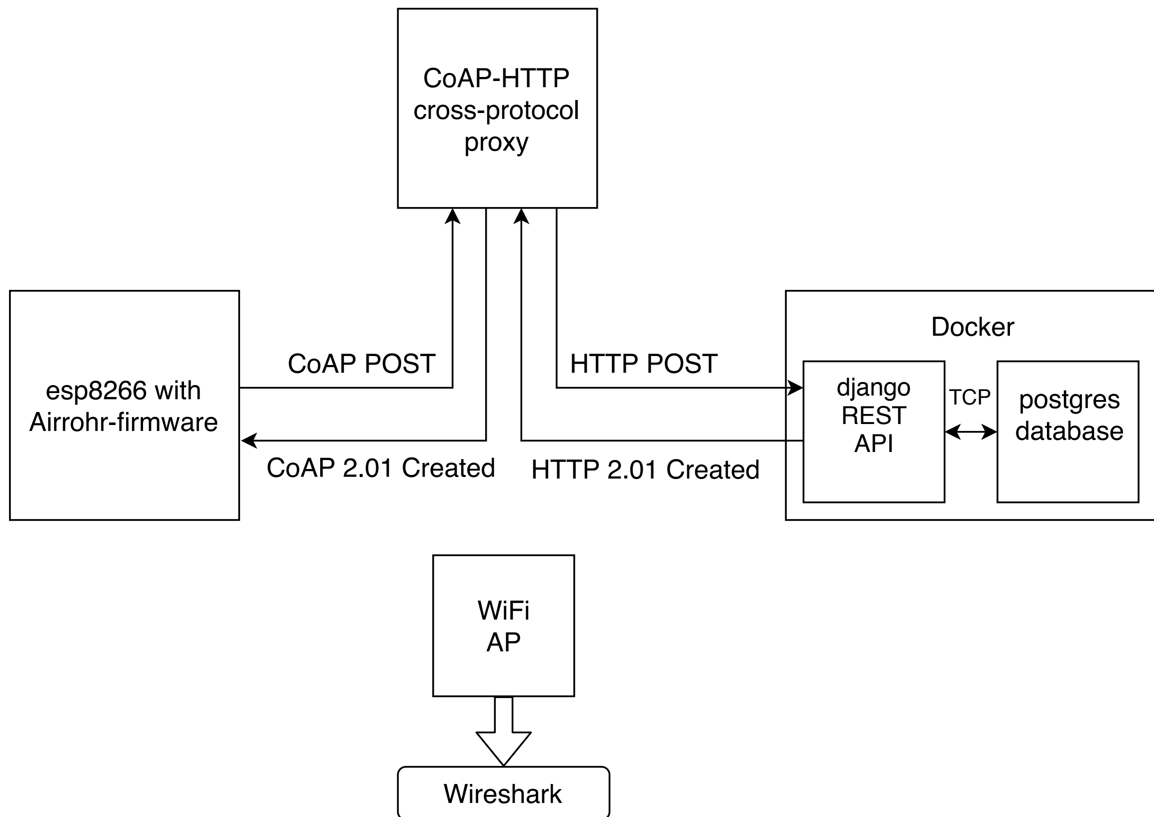


Figure 3.1: Sketch of test environment. The example shows the process of pushing values from 1 sensor to the API. The components are arranged so that all traffic goes through the WiFi access point. We can then observe the communication using Wireshark. The esp8266 sends a CoAP POST request to the cross-protocol proxy which converts it to HTTP and forwards it to the API. The response is then translated back to a CoAP message.

3.2 Changes to Californium

Because of CoAPs REST nature we can use a CoAP to HTTP cross-protocol proxy to deliver CoAP POST requests to an HTTP endpoint. As a result of this, we can move the embedded client's HTTP functionality to the proxy.

Eclipse provides proxy classes as part of Californium along with a demo proxy application. Conversion of CoAP options to HTTP headers happens in `HttpTranslator.java` line 679.

The `Proxy.properties` file contains mappings between CoAP and HTTP codes. We add lines for our custom options as shown in Listing 3.1.

```
2  coap.message.option.65003=X-Pin
3  coap.message.option.65007=X-Sensor
```

Listing 3.1: Added lines in the Proxy.properties file.

Since CoAP option payloads have a data type, Californium-core needs to be aware of those. Changes to `OptionNumberRegistry.java` of Californium-core are shown in Listing 3.2.

```
60    public static final int X_PIN = 65003;
61    public static final int X_SENSOR = 65007;
      ...
127   case X_PIN:
128   return optionFormats.INTEGER;
      ...
136   case X_SENSOR:
137   return optionFormats.STRING;
```

Listing 3.2: These changes make Californium-core aware of the new options and set their basic data type.

In our specific case the sensor sends its requests faster than the proxy can processes them. To prevent it from crashing, multiple connections will be allowed at the same time. We edit the file `ProxyHttpClientResource.java` of Californium-proxy as shown in Listing 3.3.

```
62 private static final AbstractHttpClient HTTP_CLIENT = new DefaultHttpClient(new
       PoolingClientConnectionManager());
```

Listing 3.3: Pass PoolingClientConnectionManager in constructor.

3.3 Memory Usage

Now that we can send updated values to the REST API using only CoAP, we can remove the HTTP functionality. We compare 3 different versions, the unmodified Airrohr-firmware, a version with CoAP included, and a version with CoAP but without HTTP capabilities. The avr-size utility is used to retrieve segment sizes which are shown in Figure 3.2.

The difference in total size between the versions shows us how much memory the CoAP and the HTTP parts need. Our CoAP implementation uses about 21 kB. The HTTP parts of Airrohr-firmware use about 165 kB. These versions are not functionally equivalent so if we

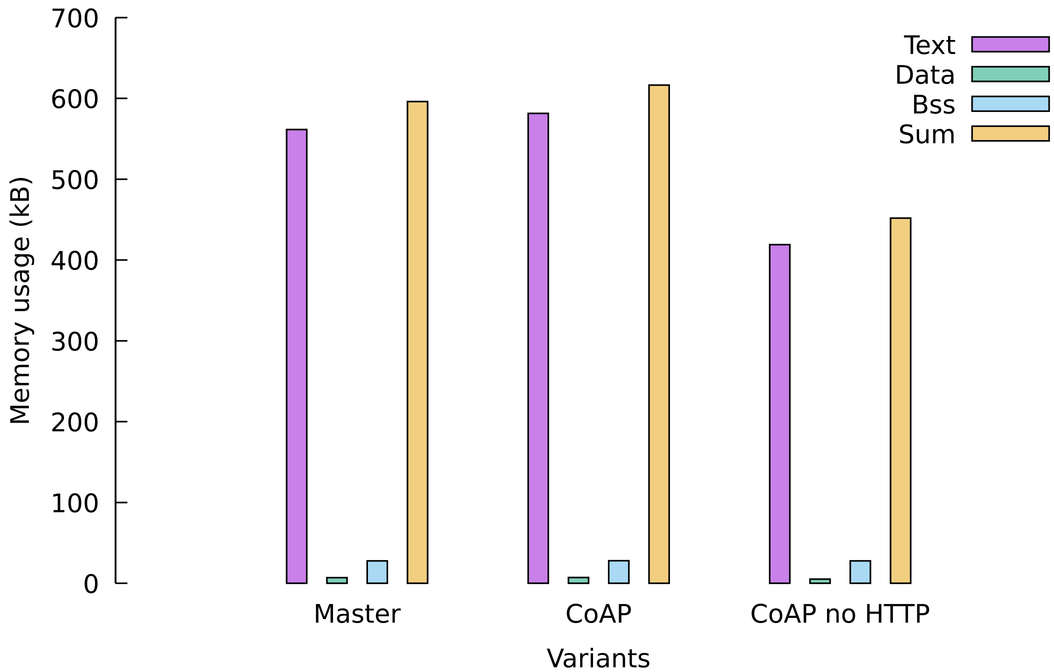


Figure 3.2: Size of memory segments for different Airrohr-firmware variants.

were to completely match all of the original Airrohr-firmwares functions, the CoAP section would increase in size. However, because the difference is relatively large, we can expect it to stay smaller than the HTTP version.

An estimation of flash memory and Random-Access Memory (RAM) requirements is derived by summing up the segments Data and Text, as well as, Data and BSS. This is shown in Table 3.1.

Table 3.1: Comparison of used memory. All values are in bytes.

	Master	CoAP	No HTTP
RAM	568532	588640	424244
Flash memory	623828	644480	479508

3.4 CoAP vs. HTTP: time to send

We look at how long it takes to push values of 1 sensor to the REST API. The Airrohr-firmware opens a single HTTP connection for each sensor, so we can get the duration of a push process by calculating the difference between the point in time when the connection is initiated to the one when the socket is closed. In the case of CoAP-NON requests the duration is obtained by measuring the time elapsed from sending the POST until receiving the NON 2.01 created response.

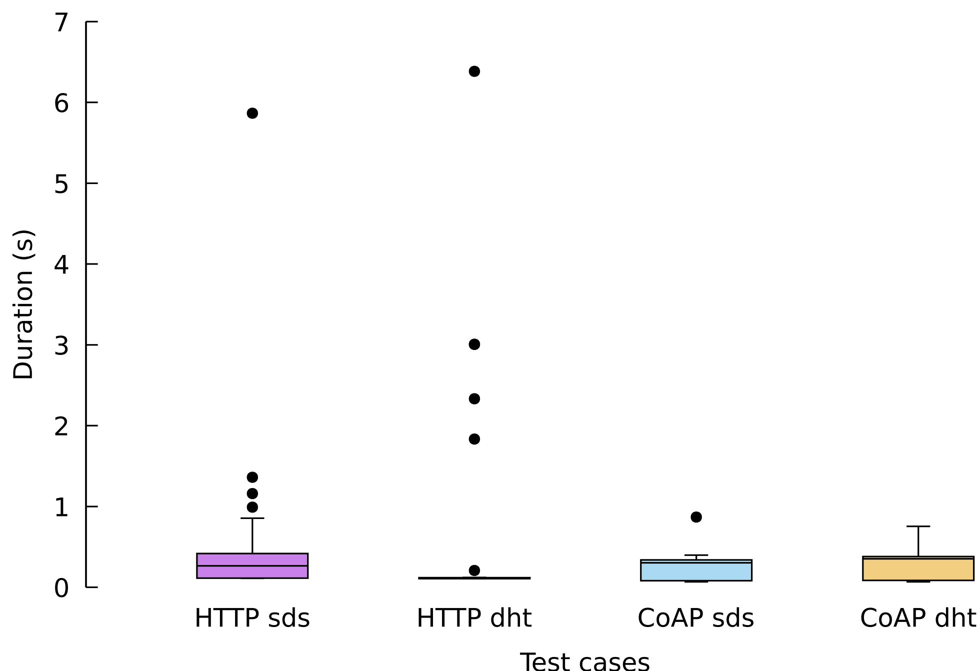


Figure 3.3: The time it takes to complete a POST request. NON messages are used for CoAP.

We measure these values for each sensor 30 times with a polling rate of 145 seconds. The results are shown in Figure 3.3 as boxplots[1].

The median round trip times for HTTP are slightly faster than CoAP round trip times. A contributing factor could be that an HTTP exchanges still needs to take place as part of the CoAP push process.

In the HTTP cases there are several outliers up to 6 seconds. This could be due to delays happening in the esp8266 HTTP stack or due to packet loss. We have recorded the test run and analyze the TCP connection duration as seen by Wireshark in Figure 3.4. The longest connection spans approximately 2.6 seconds so it is more likely that the delays are introduced by the esp8266 HTTP stack.

3.5 CoAP vs HTTP: bandwidth usage

We compare bandwidth usage for a single sensor value push operation in figure 3.5.

Since we are interested in the difference between CoAP and HTTP we look at the traffic between the sensor and the proxy as well as between the sensor and the API.

In all cases the payload contains 1 JSON formatted string in the request and 1 in the

[1] The box is around the region between the first and third quartiles. Whiskers extend to the most distant point whose y value lies within 1.5 times of the interquartile range. These parameters apply to all used boxplots.

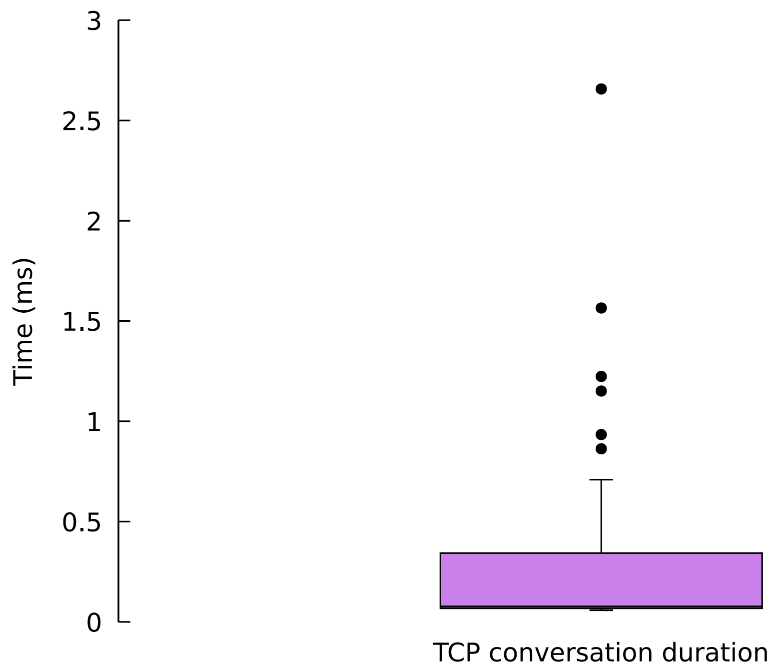


Figure 3.4: Duration of every TCP connection that happened during the response time test.

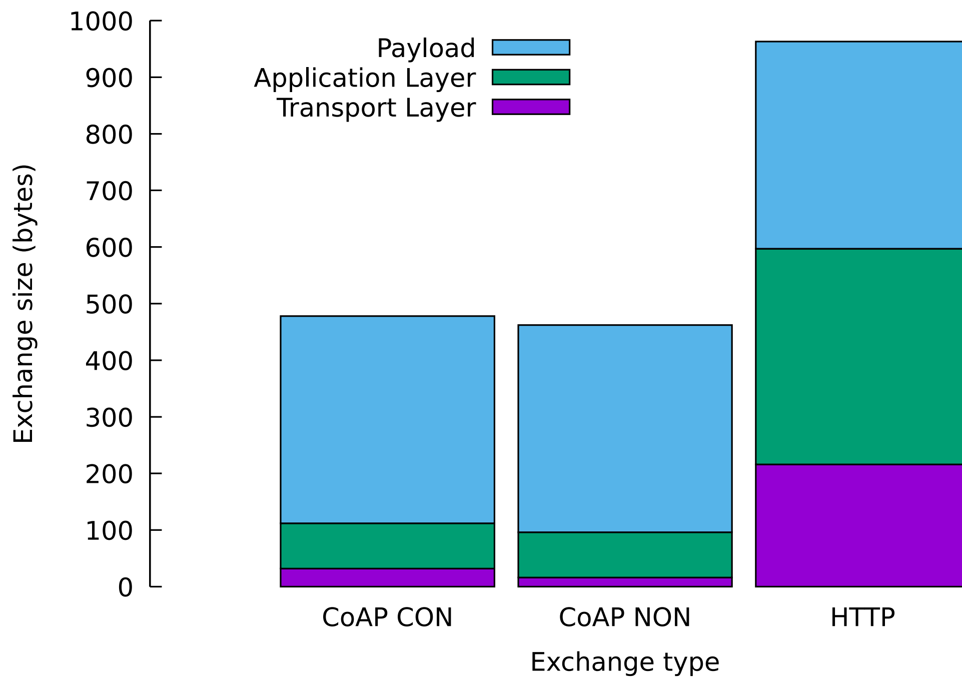


Figure 3.5: Shows how much data is used for pushing the values of 1 sensor to the API.

response. Together they are 366 bytes. The HTTP protocol uses 381 bytes and the TCP protocol accounts for 216 bytes.

A CoAP NON push consists of 2 messages. The POST request and the NON 2.01 created status message. One of the reasons that CoAP is smaller, is that X-Pin and X-Sensor are represented by their option number instead of the string.

In case of CON, the only difference is two additional ACK messages that are 8 bytes each.

By using CoAP we can save about 500 bytes per push operation.

3.6 CoAP Observe

We want to see if the CoAP observe feature could be used to improve upon the polling operational mode. To do that, we use the openHAB based client and set it up to handle observe relationships with the esp8266. We schedule a counter that increments every 145 seconds and calculate the difference to the number of updates we receive per CoAP observe.

Because readings most likely change by minuscule amounts every time we measure, the observers will only be notified when this change is larger than a predefined amount. We use the accuracy values given in the sensors specification sheets as a guideline. For the DHT11 we choose ±5 %RH and ±1 °C[15]. In case of the SDS011 we pick ±15 % and ±10 g/m^3[15]. When the new reading exceeds one of those thresholds, we send a notification.

The client refreshes the observe-relationships every hour. It will also send GET requests based on the Max-Age options value. Since it is 60 seconds by default, the client would request the resource state every minute. To prevent that the server is set to respond with the Max-Age option set to 3600 seconds.

Figure 3.6 shows the result of a 24 hour test run in which we calculated the differences every hour.

In this test configuration we can save a considerable amount of exchanges. However, how much CoAP better observe is depends on the selected thresholds and how frequently the sensor values actually change. Temperature and Humidity are very stable while PM 10 tends to fluctuate. It should also be noted that CoAP observe can never perform worse than polling, provided the Max-Age option is set to a value larger that the polling frequency.

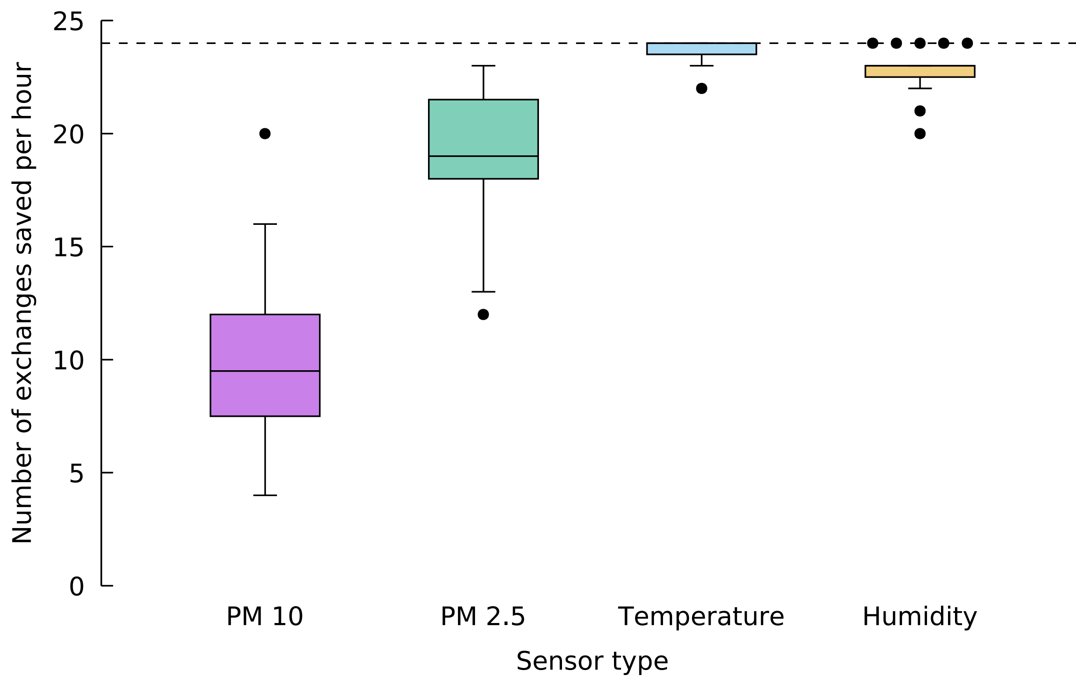


Figure 3.6: Shows how many exchanges were saved by using CoAP observe in comparison to polling. The dashed line represents the maximum amount of exchanges that can be saved in a single hour. If we send no notifications we still refresh the observe-relationships once per hour.

CHAPTER 4

Conclusion

The Airrohr-firmware is an IoT project for measuring and publishing air quality data. Its communication is based on HTTP which is not ideal for embedded devices. We look at CoAP, how it could benefit us in this scenario, and how it is placed in the current infrastructure model. To demonstrate possible benefits of CoAP, we implement a library. We then use this library to add CoAP functions to the Airrohr-firmware.

Because the REST API still depends on HTTP, we implement a cross-protocol proxy for the specific use case of pushing sensor values to the luftdaten.info API. Program size is one of the major constraints for the Airrohr-firmware, so we explore how much memory could be saved by removing and replacing HTTP with CoAP. A significant amount of memory could be saved but not enough to reliably create a firmware version that fits on the smallest esp8266 configuration of 512 KiB, especially with future expansion and the OTA update function in mind. The functionality of the HTTP part would move to the implementations of client and API.

We compare how long it takes to complete one request-response exchange. In this case CoAP is slightly slower than HTTP. This is possibly because in the process of pushing sensor values, we still need to complete an HTTP exchange. It also takes some time to construct a CoAP message out of the data structure.

However, HTTP has a number of high outliers possibly due to delays in the Arduino core for esp8266 HTTP implementation.

Then we compare how much data is transferred when handling 1 sensor. In comparison to HTTP, CoAP messages are much smaller. This is because UDP uses smaller headers and because information is transferred in a more compressed way. For example, when using HTTP we send the name of custom headers while in CoAP this information is encoded in the option number. However, the Airrohr-firmware depends on the availability of a WiFi network with internet connection so in this case saving bandwidth is of limited importance. In a situation where we depend on a battery, or if bandwidth consumption is important, for example if the esp8266 is connected through the GSM network, the smaller data packets could be beneficial.

Finally, we implement and test the CoAP observe extension by comparing it with normal polling operation. The number of transmission could be significantly reduced, especially

in case of temperature and humidity. We used thresholds to trigger notifications and a similar behavior can be realized with HTTP. However, CoAP would provide us with a well specified behavior.

In practice we probably want both CoAP and HTTP functionality. HTTP is still usefull for the initial configuration and pushing data to the public API without running the proxy. Other optional APIs also depend on HTTP.

This work could benefit from a number of future improvements.

The library should be extended so that it is not only a proof of concept. That means it should be able to handle all options and behave in the specified way even if it is used improperly.

There are only very small gains of Flash memory when removing the HTTP POST function however a version without any HTTP might be worth exploring. The Airrohr-firmware in its default configuration has only four other purposes apart from sending to the API: configuring the device, presenting the last read values in a HTML page, downloading new firmware versions from the internet, and support for optional APIs

The first 2 cases can also easily be solved by using CoAP. When using CoAP, functionality would move from the firmware to the client. Updating the firmware would require either a HTTP client or CoAP blockwise transfer and a CoAP capable remote server to host the firmware. Support for optional APIs that depend on HTTP could be replaced by using a cross-protocol proxy.

While removing HTTP, it would also be useful to have a REST-API that is accessible with CoAP.

In the context of such work other IoT technologies could be explored. For example CBOR might allow us to save both bandwidth and memory for the configuration. It would also be worthwhile to see how CoAP compares to other lightweight protocols like Message Queuing Telemetry Transport (MQTT).

Bibliography

[1] Ki-Hyun Kim, Ehsanul Kabir, and Shamin Kabir. “A review on the human health impact of airborne particulate matter”. In: *Environment International* 74 (Jan. 2015), pp. 136–143.

[2] Claire Holman, Roy Harrison, and Xavier Querol. “Review of the efficacy of low emission zones to improve urban air quality in European cities”. In: *Atmospheric Environment* 111 (June 2015), pp. 161–169.

[3] Somayya Madakam, R. Ramaswamy, and Siddharth Tripathi. “Internet of Things (IoT): A Literature Review”. In: *Journal of Computer and Communications* 03.05 (2015), pp. 164–173.

[4] Dave Evans. “The Internet of Things How the Next Evolution of the Internet Is Changing Everything”. In: *Cisco Internet Business Solutions Group (IBSG)* (Apr. 2011).

[5] Tetsuya Yokotani and Yuya Sasaki. “Comparison with HTTP and MQTT on required network resources for IoT”. In: *2016 International Conference on Control, Electronics, Renewable Energy and Communications (ICCEREC)*. IEEE, Sept. 2016.

[6] Jon Postel. *Transmission Control Protocol*. RFC 793. Sept. 1981. URL: `https://rfc-editor.org/rfc/rfc793.txt`.

[7] Henrik Frystyk Nielsen et al. *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616. June 1999. URL: `https://rfc-editor.org/rfc/rfc2616.txt`.

[8] Roy T Fielding and Richard N Taylor. “Architectural styles and the design of network-based software architectures”. PhD thesis. 2000.

[9] Zach Shelby, Klaus Hartke, and Carsten Bormann. *The Constrained Application Protocol (CoAP)*. Tech. rep. 7252. June 2014. 112 pp. URL: `https://rfc-editor.org/rfc/rfc7252.txt`.

[10] M. Kovatsch. *CoAP Implementation Guidance*. July 2018. URL: `https://tools.ietf.org/html/draft-ietf-lwig-coap-06`.

[11] Klaus Hartke. *Observing Resources in the Constrained Application Protocol (CoAP)*. Tech. rep. 7641. Sept. 2015. 30 pp. URL: `https://rfc-editor.org/rfc/rfc7641.txt`.

[12] Angelo Castellani, Thomas Fossati, and Salvatore Loreto. “HTTP-CoAP cross protocol proxy: an implementation viewpoint”. In: *2012 IEEE 9th International Conference on Mobile Ad-Hoc and Sensor Systems (MASS 2012)*. IEEE, Oct. 2012.

[13] Boost Authors. *Boost libraries*. URL: https://www.boost.org/.

[14] Zach Shelby. *Constrained RESTful Environments (CoRE) Link Format*. Tech. rep. 6690. Aug. 2012. 22 pp. URL: https://rfc-editor.org/rfc/rfc6690.txt.

[15] Ltd. Nova Fitness Co. *Laser PM2.5 Sensor specification*. Oct. 2015.