Constrained Application Protocol

MOD250 AUTUMN 2016

LARS-JO RØSBERG, ROY NESBØ

Table of Contents

1 Introduction

1.1 Goal and motivation

The main goal of this project was to evaluate the "Constrained Application Protocol" (CoAP) software technology. The team wanted to investigate how easy it is to learn and use the technology, and how much work is required to enable communication between two or more devices using this technology.

The second goal was to determine whether CoAP is less expensive, in terms of reducing traffic overhead compared to HTTP, and in which scenario it is more appropriate to use HTTP implementations.

The project posed an excellent opportunity to learn more about the Internet of Things devices, and get some hands-on experience with the combination of electronics and programming. Furthermore, the team was excited to be able to choose frameworks freely to achieve the main goals.

1.2 Limitations

While the CoAP specification is not too complex, implementing the protocol itself would be too time consuming for this project. For this reason, the decision was made to limit the scope by using existing implementations of the protocol, and focus on how the technology is applicable in "real world" applications instead.

1.3 History of the technology

As the "Internet of Things" (IoT) started to gain popularity, the need for better communication protocols became apparent. A recent report published by company Ericsson, which specializes in networking and telecommunications technology, states that an estimated number of IoT devices reaches close to 16 billion by the year 2021 [1]. By developing technology that allows these devices to use less powerful hardware, the cost of production and maintenance of devices could be greatly reduced. The potential toll on network infrastructure, both locally and in the aspect of the internet at large, is enormous. Thus, figuring out new ways to reduce the amount of data being sent by such a huge number of devices is another important perspective to consider, when developing new communication protocols.

CoAP, or the "Constrained Application Protocol", is one possible solution to these issues. It was developed as an *Internet Standards Document*, and was designed for constrained devices and networks. [2] The protocol is very modern, and was developed with both established and state of the art communication technologies in mind, such as "IPv6 over Low-Power Wireless Personal Networks" (6LoWPANs). It was published as RFC 7252 in June 2014. [3]

1.4 Organization of the report

The main content of this report is organized into the following chapters:

- Chapter 2 presents the software technology.
- Chapter 3 introduces the prototype system, and elaborates on its design.
- Chapter 4 explains in details how the prototype system was implemented.
- Chapter 5 contains the execution of the project's second goal, and a discussion of CoAP.
- Chapter 6 draws conclusions on the team's findings and experiences during this project

2 The software technology

2.1 Key concepts

2.1.1 Constrained devices

One of the most important aspects of CoAP is that it is designed to be used on constrained devices. More specifically, it supports Class 1 constrained devices, described in section 3 of RFC 7228 - Terminology for Constrained-Node Networks [4]. The class consists of devices with 10 KiB of data (e.g. RAM) and 100 KiB code size (e.g. flash storage), which lacks the resources to utilize more complex protocols, such as HTTP.

2.1.2 Constrained networks

The term *constrained network* is defined in [4] as "A network where some of the characteristics pretty much taken for granted with link layers in common use in the Internet at the time of writing are not attainable.". Examples of such characteristics are:

- Low achievable bitrate and/or throughput.
- Packet loss, including high variability of packet loss and penalties for using larger packets.
- Limits on reachability over time/power disruptions, such as battery-powered devices which inevitably will run out of power.

CoAP addresses these issues in several ways. The countermeasures are described in more detail in the next section.

2.2 Features

2.2.1 Resources

CoAP uses the coap and coaps (secure) URI scheme as means to locate the available resources. CoAP URI's are composed of the same basic elements as the HTTP URI scheme. The below snippet is an example of CoAP resource URI's.

```
// Standard CoAP resource URI
coap://hostname:port/path/to/resource
// Resource with query parameters
coap://hostname:port/path/to/queriedResource?query
```

CoAP resources are typically sensory data, such as temperatures, humidity, distance etc. A request might look like this:

```
// Distance CoAP resource URI
coap://hostname:port/measurements/distance
// Temperature CoAP resource URI
coap://hostname:port/ovaloffice/temperature
```

Like HTTP, CoAP is based on the Representational State Transfer (REST) model, in which resources are made available under a URL. CoAP supports the same standard methods such as GET, PUT, POST and DELETE, and resembles HTTP in this regard. CoAP also supports different media types, such as XML and JSON and any other data format of your choice.

This makes the transition from REST to CoAP easy for developers already familiar to the REST model.

2.2.2 Reliability

The CoAP specification uses the User Datagram Protocol (UDP), which enables constrained devices, such as small microcontrollers, to use full IP networking. Other protocols like TCP and SMS may be used, but only UDP is a part of the specification. This is a message based connectionless protocol where messages are sent in one direction, from source to destination, without any verification or acknowledgement of the transmissions.

The CoAP transmission headers therefore include a type field (T) which indicated whether the message is confirmable (CON) not. If the message is marked confirmable, the transmission will use a default timeout, and an exponential back-off between retransmission [2]. The transmission will then continue until the "max retransmit" limit is reached, or until the client receives an ACK message with the corresponding ID from the recipient. CoAP also include duplicate detection for both Confirmable and Non-Confirmable messages, to ensure data integrity.

To avoid packet loss, caused by large packages, CoAP implementations *may* include block-wise transfers, which can split the payload into several message blocks.

The CoAP headers are small, and consist of only 4-bytes. This decreases the communication overhead and facilitates effective transmissions. The contents of the header are depicted in the table below.

<table>
<tr><td colspan="8">0</td><td colspan="8">1</td><td colspan="8">2</td><td colspan="8">3</td></tr>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>0</td><td>1</td></tr>
<tr><td colspan="2">Ver</td><td colspan="2">T</td><td colspan="4">TKL</td><td colspan="8">Code</td><td colspan="16">Message ID</td></tr>
<tr><td colspan="32">Token (if any, TKL bytes) ...</td></tr>
<tr><td colspan="32">Options (if any) ...</td></tr>
<tr><td colspan="8">1 1 1 1 1 1 1 1</td><td colspan="24">Payload (if any) ...</td></tr>
</table>

Ver – Version (1)
T – Message Type (Confirmable, Non-Confirmable, Acknowledgement, Reset)
Code – Request method (1-10) or Response Code (40-255)
Message ID – 16-bit identifier for matching responses
Token – Optional response matching token
Token Length (TKL): 4-bit unsigned integer. Indicates the length of the variable-length Token field (0-8 bytes). Lengths 9-15 are reserved, MUST NOT be sent, and MUST be processed as a message format error.

2.2.3 Protocol compatibility

A limited subset of the functionality from HTTP is available in CoAP, and thus cross-protocol proxying to HTTP is straightforward. CoAP also might be proxied to other protocols such as Extensible Messaging Presence Protocol (XMPP) or Session Initiation Protocol (SIP) [3].

2.2.4 Service discovery

Resource discovery in CoAP can be performed using either unicast or multicast. The RFC 7252 specification emphasizes the importance of supporting the Constrained Restful Environments (CoRE) Link Format of discoverable resources [2]. This scheme entails that the discoverable resources can be explored using a standardized URI for all constrained nodes. The discovery sequence is triggered by sending a GET request to "/.well-known/core" on the server, which returns a payload in the CoRE Link

Format. A client would then match the appropriate resource type, interface description, and possible media type for its application [5].

The CoRE link format can also be used to register new resources in a resource directory, or to allow a resource directory to poll for resources. This can be achieved by having each server sending a POST request to "/.well-known/core" on the resource directory. This will add a discoverable link to the resource directory under an appropriate resource.

2.2.5 Service subscriptions

The proposed standard RFC 7641, *Observing Resources in the Constrained Application Protocol (CoAP)*, adds the option of making CoAP-resources observable [6]. Essentially, it enables clients to observe a *subject*, a resource or a service, located on a CoAP server. The subject owner (the server) is responsible of sending notifications to the observer when the subject changes. If the client wishes to stop the observation, it may simply remove the observation. The next time a notification is sent by the server, the client is not able to recognize is and responds with a reset message, removing itself from the observer list. Figure 1: The Observer Design Pattern illustrates how the client and server converse using this pattern.

An example use case for this functionality, is a temperature sensor resource, where clients may want to be notified when the temperature changes.

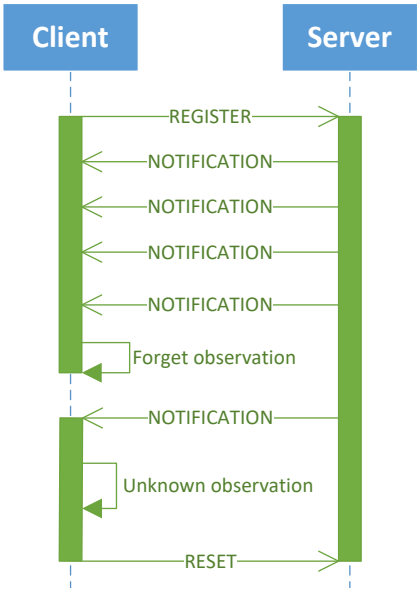

Figure 1: The Observer Design Pattern

2.2.6 Security

One of the most important features of CoAP is its security feature.
Class 1 constrained devices are unable to use the transport layer security protocols we're used to, such as TLS and its predecessor SSL. However, devices in this class can support transport layer security by using protocols which are specifically designed for constrained devices.

The security in CoAP uses DTLS (Datagram Transport Layer Security) binding to enable encrypted communication. The RFC document for CoAP defines 4 different modes which are mandatory for implementations of the current specification:

- NoSec: No protocol-level security.
- PreSharedKey: DTLS is enabled, uses a list of pre-shared keys and allowed nodes for all communication.
- RawPublicKey: DTLS is enabled, uses an asymmetric key pair without a certificate.
- Certificate: DTLS is enabled, uses an asymmetric key with a certificate signed by a trusted root signer.

2.3 Constrained environments

2.3.1 Interoperability

The interaction model between devices using CoAP shares some similarity to how it's done in HTTP, as both protocols relies heavily on the client/server model. However, due to the nature of *M2M,* many of the devices using CoAP will take on the roles of both client and server. This is necessary to enable two-way communication between the devices, and differs from HTTP where the roles of clients and

servers are clearly defined. Implementing proxies enables bridging between CoAP and other web transfer protocols, such as HTTP.

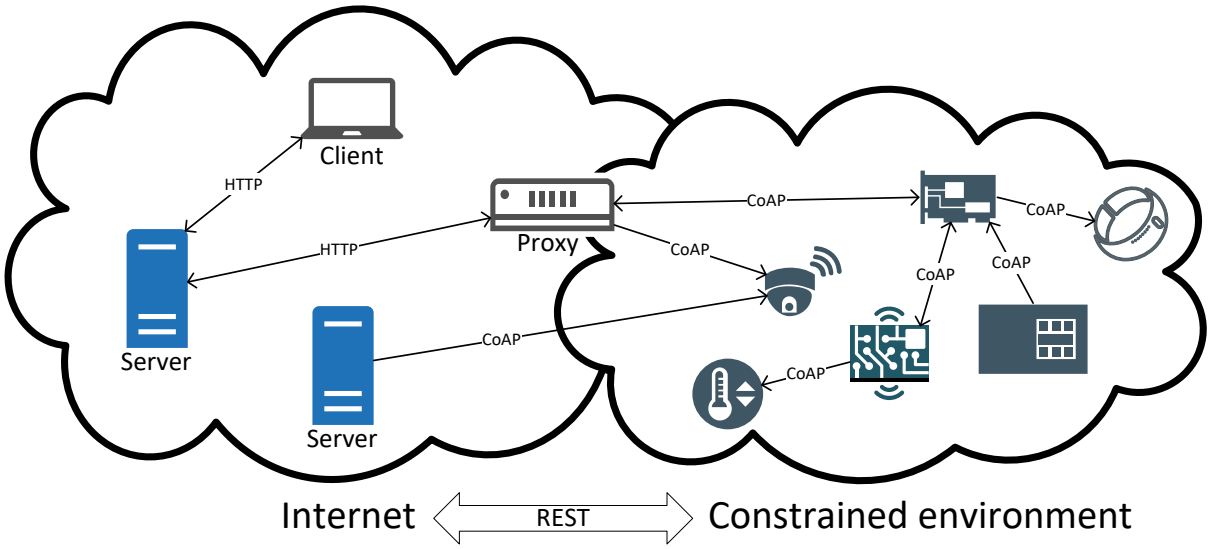

Figure 2: The illustration demonstrates a network, in which multiple devices using both CoAP and HTTP can communicate. Inspired by the illustrations presented in [7].

2.3.2 Proxies

Multiple types of proxies are described in the specification of CoAP, and facilitates several useful extensions to the network architecture. By using a proxy, which acts as an intermediary between HTTP and CoAP devices, enables high interoperability between the two protocols, as both are built on the client/server model and supports the REST architecture. The CoAP specification includes descriptions of proxies for both "HTTP client to CoAP server" and "CoAP client to HTTP server".

Additionally, some use cases for CoAP-CoAP proxies are also described. These proxies facilitate several useful actions, such as request forwarding and buffering of resources hosted on less powerful hardware.

2.4 Related technologies

There are several related data protocols, such as Advanced Message Queuing Protocol (AMQP), Websocket and Node. But the most relevant with regards to constrained applications, is the Message Queuing Telemetry Transport (MQTT) protocol.

MQTT is a message oriented publish/subscribe message protocol, designed for machine-to-machine (M2M) communications. IBM originally developed it, but it has now become an open standard. The key components of an MQTT setup are clients, and a central server, known as a broker. The clients can subscribe to string based topics, and the broker will relay messages from a source, to the clients subscribing to the topics.

Although MQTT is designed to be lightweight, it has a couple of obvious drawbacks. It utilizes TCP, and the connections are persistent throughout the communication period. Secondly, all topics are string based. This becomes a disadvantage in a constrained network scenario, where package loss might be common, and the computing resources are minimal [8].

3 Prototype design

3.1 CoAP implementation

3.1.1 Analysis of available implementations

There are several CoAP implementations available in several languages online. To limit the scope of this project, this report will only discuss three implementations, two C# .NET implementations and one Java implementation.

3.1.1.1 Requirements

The baseline requirements for selecting a CoAP implementation for this project was based on the team's previous experience, interests, and ease of use. The implementation had to support the object-oriented paradigm. It should be relatively easy to get started with, to reduce the time on initial setup of the development environment, and leave more time to explore the main features of the CoAP specification. Furthermore, it should be based on a familiar programming language, such as C# or Java, which the team already has some experience with.

3.1.1.2 Californium

Californium (Cf) is a complete Java implementation of the CoAP specification. The project is divided into five sub-projects; Californium (Cf) Core, Scandium (Sc), Actinium (Ac), CoAP Tools and Connector. The Californium Core is the central framework with the protocol implementation [9].

The Scandium project is the DTLS 1.2 security implementation, which can be used to secure a CoAP application with pre-shared keys, certificates, or raw public keys.

The Actinium project allows you to import and expose JavaScript resources as web-services and run them in a Java environment.

The CoAP tools provides a command-line interface to communicate with the devices directly, and means to benchmark the application.

The connector project was originally a separate component, but has now been included into the core module of Californium. By abstracting the connector from the rest of the implementation, it enables the developer to create new connectors, permitting additional transport layer protocols and means of encryption to be used.

3.1.1.3 CoAP.NET

This implementation is .NET framework written in C#, ported from the Californium Core, by SmeshLink Technology. The latest release is version 1.1.0. Being a port of Californium, and written in a very similar programming language, using it is essentially identical in term of how CoAP clients and servers is written in Californium. Some documentation is available, but some parts are unfinished or incomplete. Some code examples are also available for simple use case scenarios. Security/DTLS is not available in this port, as this is not a part of Californium Core [10].

3.1.1.4 CoAP Sharp

The first .NET-implementation of CoAP, written in C# by EXILANT Technologies. The current release is version 0.3.1. It is designed to run on constrained devices with support for the .NET Micro Framework. An experimental build with support for Windows 10 IoT Core is also available. The framework lacks proper documentations, with simple tutorials being the only available help for developers. This implementation does not include security/ DTLS.

A deep understanding of CoAP is required to use this implementation. Where most of the other object-oriented implementations tries to abstract the more complex parts of the protocol, CoAP Sharp requires the developer to handle all possible outcomes and errors manually. To implement a server which supports both reliable and unreliable communication, the developer must use separate logic for each method of communication. Sending and receiving multi-block payloads must be done manually on both the server and the client [11].

3.1.1.5 Discussion of implementations

The initial goal was to use the CoAP.NET implementation with a Universal Windows Platform (UWP) application on both devices. It turned out that CoAP.NET does not currently support UWP applications, and the decision therefore had to be reconsidered. The second .NET alternative was CoAP Sharp, which does include an experimental UWP build. However, after having implemented a basic server instance, it was deemed unnecessarily complex and cumbersome, leaving the team with doubts as to if the project could be completed in time with it. The third viable option was the Java implementation, Californium. This is the only complete implementation, and it is quite easy to use. The only drawback with this implementation is that it must run on another operation system than Windows 10 IoT Core, which only supports UWP applications. The Windows 10 IoT Enterprise version does support Windows Presentation Foundation (WPF) applications, which is compatible with the CoAP.NET implementation. However, this is not freely available.

3.2 Tools and frameworks

3.2.1 Available hardware

There are several do-it-yourself (DIY) devices on the market. The most versatile, and perhaps most relevant devices for this project are the Arduino UNO and the Raspberry PI.

The Arduino UNO microcontroller is a small 7V board with 14 digital and 6 analog pins, and only 32 KB Flash memory. It has a clock speed of 16 MHz and weigh only 25 grams. The Arduino program language is a set of C and C++ functions, and is supported across all platforms [12].

The Raspberry Pi 3 is not a very constrained device. With 1.2GHz quad-core processor, 1GB Ram, 802.11 Wireless adapter, Bluetooth, USB, and HDMI ports, it is a small computer [13]. However, it supports several operation systems, which makes it flexible, and allows for testing and prototyping of new ideas rapidly. That is the main reason behind the purchase of Raspberry Pi's for this project.

3.2.2 Choosing platform(s)

The open source community has developed a Debian-based operation system, optimized for the Raspberry PI, called Raspbian [14]. This is a good option for testing the Java implementation, as both the Java Runtime Environment and Java Development Kit is included in the current distribution. Even though the CoAP.NET implementation is lacking support for security, it would be interesting to create a hybrid scenario with devices running Java on Raspbian, and a central dashboard running a WPF application with CoAP.NET. In this way, it becomes possible to both test the most promising CoAP implementation, as well as building a quick visual presentation of the intercommunication between the devices.

3.2.3 Software

The following software was used in this project:

- Visual Studio 2015 for developing the C# display implementation
- IntelliJ for developing the Java implementations
- Raspbian Jessie With Pixel operation system on the Raspberry Pi's

3.3 IoT environment

3.3.1 High level design

To make a prototype which displays CoAP in a realistic environment, the team decided to design and implement a system consisting of 3 devices:

- A device with a temperature sensor.
- A device with a motion detection sensor.
- A device connected to a display, which shows the temperature for a while after motion has been detected.

The illustration below depicts the environment. Since we have access to 2 Raspberry Pi's, which have general purpose input/output capabilities well suited for sensors. These will fulfill the role of devices with sensors in this prototype. The display device will be implemented as a windows desktop application. All devices will be connected to the same Wi-Fi access point, and all communication will be done using CoAP.

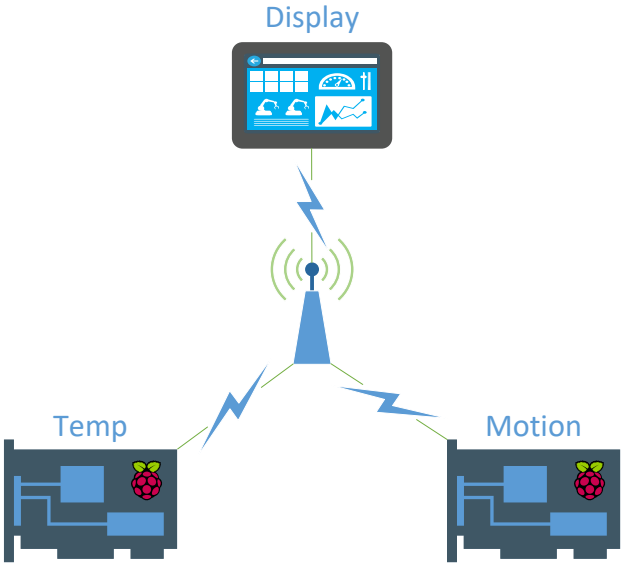

Figure 3: The prototype environment

3.3.2 Temperature device

The device is a Raspberry Pi with the operating system Raspbian, with a temperature sensor connected. To enable other devices to access this sensor, a CoAP server with the following available resource will be implemented:

- GET /sensors/temperature

The server implementation will use the Californium implementation of CoAP, on the Java platform. Resources/services should be implemented in a such way, that it is observable by CoAP clients.

3.3.3 Motion device

Like the temperature device, this is a Raspberry Pi with the operating system Raspbian. However, this device will only have a motion detector sensor connected. To enable other devices to access the sensor resource, a CoAP server with the following service will be implemented:

- GET /sensors/motion

This server will also use Californium on the Java platform, and clients should be able to observe the services/resources.

3.3.4 Display device

This device will consist of a laptop running the Windows 10 operating system. Considering the team's prior knowledge and experience with development on this platform, developing a WPF application and displaying the sensor data of the other devices by using the CoAP.NET framework is the safest approach.

The following illustration demonstrates how this device will use the services available on the other devices:

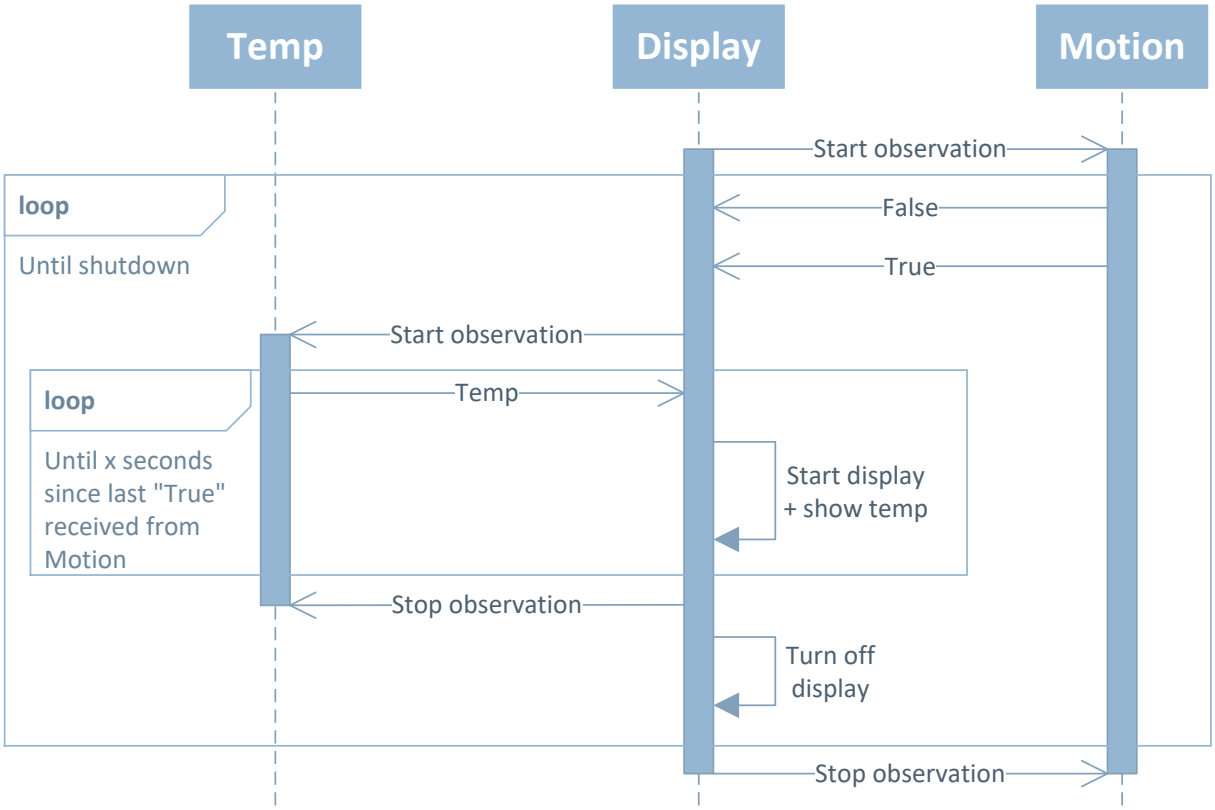

Figure 4: Sequence diagram showing the behavior of the display device.

When this device starts, it begins an observation of the motion detector resource. It remains dormant until it receives a message that motion has been detected, and then starts observing the temperature resource. When it receives the first temperature data message, it turns on the display and displays the temperature. It continues to receive updated temperature readings, until a predetermined timespan has elapsed from the last time it received a "motion detected" message. It then stops observing the temperature resource, since it's no longer necessary to do so, and then turns off the display and waits for the next motion to be detected.

4 Implementation

4.1 Overview

The final solution is composed of three separate implementations: Two Java applications (on both devices), and one C# application on the main dashboard. Due to time constraints, the decision was made to settle for simulated raw data, as opposed to spending valuable time on wiring the pieces together.

4.2 Implementation of temperature and motion devices

4.2.1 Preparing hardware and operating system

The first step was to follow the instructions on how to install Raspbian, located under the download section on the Raspberry Pi web site [13]. Then, the SD-cards and copying the NOOBS (New Out Of the Box Software) operating system installer files to it, the operating system were installed with ease. Only two button presses were necessary to install the operating system after powering on the Pi's for the first time. A simple "Hello world" was run in the preinstalled Java Runtime Environment to verify that the device was ready to run the server.

4.2.2 Adding the sensors

The original plan included using real sensors in this project. In the pre-planning stages, the team made several successful attempts at connecting and reading data from several types of sensors, including the types mentioned in this report. However, this was accomplished on another platform, using a different set of frameworks and tools. The group members decided that, rather than using the remaining time to get these sensors to work with the new environment, simulated sensors should be used instead to complete this project in time. The reasoning for this is that CoAP itself should take precedence, however interesting working with sensors may be.

The simulated temperature sensor was implemented by setting a value to a random decimal number between 0 and 20, and then periodically increasing or decreasing the value. The motion detector was implemented by setting the default "motion detected" value to false. Then, every second the value has a 1 in 15 chance to switch to true. When this happens, a timer is started to automatically return the value to false after 5 seconds. Both sensor implementations include a list of "listeners", enabling a simple form of the observer pattern. The listeners are notified of any changes in the sensor values. Java interfaces were made to simplify the process of enabling the observe function in the CoAP sensor resources.

4.2.3 Implementing the server applications

The CoAP servers and resources for both devices were implemented similarly. The first step consisted of creating a Java class which extends the *CoapResource* class, and overriding the default constructor to mark the resource as observable. The default constructor also instantiates the sensor object and adds the resource as a listener. Next, the "sensor listener" interfaces described earlier was implemented. This is what enables the CoAP resource to notify observers of the resource when it changes.

```
/**
 * Handles GET-requests to this service
 * @param exchange
 */
@Override
public void handleGET(CoapExchange exchange) {
    exchange.respond(String.valueOf(motionDetector.getMotionDetected())); //Responds with the sensor value
}
```

Figure 5: The resource handler which responds to requests sent with the GET method.

```java
public class MotionDetectorResource extends CoapResource implements MotionDetectorSensorListener{
    private DummyMotionDetectorSensor motionDetector;

    /**
     * Constructor for the class
     * @param name The name of the URI the resources will be reachable at
     */
    public MotionDetectorResource(String name) {
        super(name);
        setObservable(true); //Enables this resource to be observable

        motionDetector = new DummyMotionDetectorSensor(); //Instantiates the motion detector sensor
        motionDetector.addListener(this); //Adds this object to the sensors list of listeners
    }
```

Figure 6: Motion detector resource class, including the constructor.

The server classes extend the *CoapServer* class, and the overridden default constructors for each implementation adds the device-specific sensor resource. A main-method were added to each implementation. Overall, implementing the server application were very simple and straight forward. However, if any of the more advances features turned out to be required, such as binding the server to specific network interfaces, it would be very hard to implement since there's no documentation available, other than simple code examples for a small set of scenarios.

```java
11  public class MotionServer extends CoapServer {
12      /**
13       * Default constructor. Adds the motion detector sensor resource on the "/motion" URI.
14       */
15      public MotionServer() {
16          add(new MotionDetectorResource("motion"));
17      }
18
19      /**
20       * The main method that creates and runs the server.
21       * @param args Main arguments
22       */
23      public static void main(String[] args) {
24          MotionServer server = new MotionServer();
25          server.start();
26      }
27  }
```

Figure 7: The motion server class.

4.3 Implementation of the display device

The display device is implemented as a Windows Presentation Foundation application, with a Model View View-Model design pattern. This framework is optimal for creating graphical user interfaces, and facilitates good separation of concern between the components. The CoAP.NET library was used to communicate with the devices. The components of the application are; the main window, two clients and two View-Models. The View-Models represent the state of the model, being the returned response from the sensors. When the response arrives, an event is fired and the View-Model property is updated. The properties of the View-Model are bound to the View, resulting in an instant update of the value.

In the initial application state, the display device starts a subscription on the motion detector sensor service. The motion sensor will reply with true / false to indicate whether motion has been detected or not. One the value "true" is received, a new subscription is submitted to the temperature service. This subscription is maintained until the value "false" is received, and a timeout value has been exceeded. The idea behind having a time-constrained subscription, is to alleviate the potential constrained network for unnecessary traffic.

5 Evaluation and discussion

5.1 Transmission size comparison experiment

5.1.1 About this experiment

One of the goals of this project, was to investigate the difference in network load when using CoAP, compared to HTTP. This is one of the most interesting features of CoAP, as it enables constrained devices to communicate over the internet, even if there's only a very limited connection available. This experiment attempts to determine how much smaller the network packets are, when using CoAP as opposed to HTTP.

This was done by implementing two servers with some simple REST-services, one using CoAP and the other using HTTP. Both servers, and all related code, was written in Java. Californium was used for the CoAP implementation, and includes a main method which starts the server. The HTTP server was built using the JAX-RS framework featured in this course, and is deployed to a glassfish server. All packets are sent reliably, as this is the default behavior of both the Californium CoAP implementation and the TCP protocol used with the HTTP. Services used in this experiments were designed to test 3 scenarios, all using the GET-method of REST:

1. No payload returned. Payload length = 0.
2. Returns a «Hello world!» text string. Payload length = 12 bytes, if each character takes 1 byte to transmit.
3. Return a list of 100 randomly generated «person»-objects, formatted using the media type JSON. Payload length = 13190 bytes, if each character takes 1 byte to transmit.

Figures 8 and 9 shows the pair of services which returns a "Hello world!" text string.

```
public class HelloWorldResource extends CoapResource {
    public HelloWorldResource(String name) {
        super(name);
    }

    @Override
    public void handleGET(CoapExchange exchange) {
        exchange.respond("Hello world!");
    }
}
```

Figure 8: The CoAP "hello world" service.

```
@Path("/helloworld")
public class HelloWorldService {
    @GET
    @Produces("text/plain")
    public String get() {
        return "Hello world!";
    }
}
```

Figure 9: The HTTP "hello world" service.

5.1.2 Testbed environment

To determine the size of the network packets being sent through the network when the services are used, the packets were captured in a software by the name of Wireshark. This software has good support for all protocols used in the experiment, both in the transport layer (UDP, TCP) and the application layer (CoAP, HTTP). Using the filter feature it was easy to display only the relevant packets.

5.1.3 Results

Collecting the results was done by using each service on both servers, while capturing the packets both to and from the server in Wireshark. Then the sizes of the transmitted packets for each capture session were added together, as shown in the table below.

Protocol	Service	Packets sent	Total bytes transmitted	Difference from empty
CoAP	Empty	2	106	-
HTTP	Empty	6	911	-
CoAP	Hello world	2	122	16
HTTP	Hello world	6	1009	98
CoAP	JSON data	52	16086	15980
HTTP	JSON data	7	14223	13315

5.1.4 Discussion

The results indicate that both protocols have both pro's and con's. As one would expect, CoAP shows its strengths when the payload size of the transmission is small. This is a typical scenario for constrained devices, due to the restrictions caused by limited processing power and memory available. Comparing the captured packet sets from the "Hello world" service on both protocols, depicted below in figure 10 and 11, we see that the total transmission size of the CoAP service is less than the first two packets in the 3-way handshake (SYN, SYN+ACK, ACK) of TCP. This result really shows the potential of CoAP as a preferred protocol for constrained devices.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	10.0.0.238	10.0.0.4	CoAP	60	CON, MID:8957, GET, End of Block #0, /helloworld
2	0.010576	10.0.0.4	10.0.0.238	CoAP	62	ACK, MID:8957, 2.05 Content, End of Block #0 (text/plain)

Figure 10: Transmitted packets when the "Hello world" CoAP service is accessed.

No.	Time	Source	Destination	Protocol	Length	Info
8990	6.003282	10.0.0.238	10.0.0.4	TCP	66	10947→8080 [SYN] Seq=0 Win=
8991	6.003416	10.0.0.4	10.0.0.238	TCP	66	8080→10947 [SYN, ACK] Seq=0
8992	6.004693	10.0.0.238	10.0.0.4	TCP	60	10947→8080 [ACK] Seq=1 Ack=
8993	6.004913	10.0.0.238	10.0.0.4	HTTP	382	GET /helloworld HTTP/1.1
9005	6.056139	10.0.0.4	10.0.0.238	TCP	54	8080→10947 [ACK] Seq=1 Ack=
9006	6.056761	10.0.0.4	10.0.0.238	HTTP	332	HTTP/1.1 200 OK (text/plain
9101	6.118846	10.0.0.238	10.0.0.4	TCP	60	10947→8080 [ACK] Seq=329 Ack

Figure 11: Transmitted packets when the "Hello world" HTTP service is accessed.

However, HTTP overtakes CoAP when larger payloads are transmitted. In the JSON data example, the total amount of data transmitted is lower for the HTTP service, than the CoAP service. Looking at the captured packets, shown in figure 12 and 13, the reason is obvious. Where HTTP and TCP allows the whole payload to be sent at once, CoAP, limited by the UDP protocol, splits the payload into multiple blocks. Each block contains about 500KB of the payload, requiring 26 request-response interaction before finishing the transmission, "wasting" about 25 times 60 bytes compared to HTTP.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	10.0.0.238	10.0.0.4	CoAP	60	CON, MID:7834, GET, End of Block #0, /jsondata
2	0.001057	10.0.0.4	10.0.0.238	CoAP	563	ACK, MID:7834, 2.05 Content, Block #0 (application/json)
3	0.044680	10.0.0.238	10.0.0.4	CoAP	60	CON, MID:7835, GET, End of Block #1, /jsondata
4	0.045600	10.0.0.4	10.0.0.238	CoAP	563	ACK, MID:7835, 2.05 Content, Block #1 (application/json)
5	0.095294	10.0.0.238	10.0.0.4	CoAP	60	CON, MID:7836, GET, End of Block #2, /jsondata
6	0.096402	10.0.0.4	10.0.0.238	CoAP	563	ACK, MID:7836, 2.05 Content, Block #2 (application/json)
7	0.146819	10.0.0.238	10.0.0.4	CoAP	60	CON, MID:7837, GET, End of Block #3, /jsondata
8	0.147979	10.0.0.4	10.0.0.238	CoAP	563	ACK, MID:7837, 2.05 Content, Block #3 (application/json)

Figure 12: The first 8 packets, out of 52, transmitted when the "JSON data" CoAP service is accessed.

10906	8.256453	10.0.0.238	10.0.0.4	HTTP	380	GET /jsondata HTTP/1.1
10909	8.262867	10.0.0.4	10.0.0.238	TCP	13532	[TCP segment of a reassembled PDU]
10910	8.263344	10.0.0.4	10.0.0.238	HTTP	59	HTTP/1.1 200 OK (application/json)

Figure 13: The request and response packets captured when the "JSON data" HTTP service is accessed.

5.2 Discussion of the prototype system

The team was predetermined to explore the .NET implementations of CoAP. Due to limitations in the implementation, this became difficult to achieve, and had to be reconsidered, as mentioned in section 3.1.2. This was not discovered until the implementation stage of the project, which meant potential trouble in terms of time left to finish the project. Luckily, the Raspberry Pi is versatile, and it was possible to install a new operating system quickly.

The Java Californium library was easy to use, and the server were implemented without any problems. The intercommunication between the devices and the CoAP.Net-based dashboard was established without any unforeseen issues.

However, due to time constraints it was not feasible to test all the planned features. Although a machine-to-machine communication was never established between the Pi's, this is a trivial part which could easily be implemented in the future. The prototype setup represents an autonomous environment with no human interference. Once the system is started, the components interact automatically with each other and the data is transmitted only when necessary. One could easily connect several other devices to the environment and expand the services available, to create an intelligent home-automation system with many nodes with different purposes.

The technology feels very familiar to programmers with experience within HTTP and RESTful web-services. With the advantage of limited amount of connections and low overhead, such a system would certainly be far more feasible in a constrained network, than any other HTTP implementation.

6 Conclusion

The main goal in this project, was to evaluate the CoAP technology, and how easy it is to learn, and use as the communication protocol to connect two or more devices. To learn the protocol thoroughly, the team went through the specifications and collected important features and concepts in this document. After having explored some of the CoAP implementations in both Java and C# through the prototype development, we can claim to have gained some insight to the technology. We have had some previous experience with building REST API's, and this has made the transition to CoAP rather smooth.

The common denominator for all implementations available, is the lack of documentation they provide. The implementations have been partially documented through examples on GitHub, but we have not been able to find any comprehensive or complete documentation online. This lead us to draw the conclusion that the technology still is immature. However, the specification seems promising, and we claim that this is a very exciting technology with lots of promising features.

To evaluate the possible improvements this technology has in regards to constrained networks, an experiment comparing the amount of data transmitted when using CoAP and HTTP services was performed. The results showed vast improvements when using CoAP to transmit small amounts of data, but due to the limits of UDP, larger payloads were more efficiently transferred with HTTP over TCP. Considering the limits of constrained devices, which may not be able to handle larger amounts of data, the conclusion is that CoAP is an excellent choice as a communication protocol for these devices, just like it was designed to be.

7 References

[1] Ericsson AB, "Ericsson Mobility Report," 2016. [Online]. Available: https://www.ericsson.com/res/docs/2016/ericsson-mobility-report-2016.pdf. [Accessed 19 11 2016].

[2] CoAP, "RFC 7252 Constrained Application Protocol," [Online]. Available: http://coap.technology/. [Accessed 14 11 2016].

[3] The Internet Engineering Task Force (IETF), "RFC 7252 - The Constrained Application Protocol (CoAP)," 06 2014. [Online]. Available: https://tools.ietf.org/html/rfc7252. [Accessed 19 11 2016].

[4] The Internet Engineering Task Force (IETF), "RFC 7228 - Terminology for Constrained-Node Networks," 05 2014. [Online]. Available: https://tools.ietf.org/html/rfc7228. [Accessed 17 11 2016].

[5] The Internet Engineering Task Force (IETF), "RFC 6690 - Constrained RESTful Environments (CoRE) Link Format," [Online]. Available: https://tools.ietf.org/html/rfc6690. [Accessed 20 11 2016].

[6] The Internet Engineering Task Force (IETF), "RFC 7641 - Observing Resources in the Constrained Application Protocol (CoAP)," 09 2015. [Online]. Available: https://tools.ietf.org/html/rfc7641. [Accessed 20 11 2016].

[7] Z. Shelby and ARM, "ARM CoAP Tutorial," 05 2013. [Online]. Available: http://www.slideshare.net/zdshelby/coap-tutorial. [Accessed 10 11 2016].

[8] The Eclipse Foundation, "MQTT and CoAP, IoT Protocols," [Online]. Available: https://eclipse.org/community/eclipse_newsletter/2014/february/article2.php. [Accessed 19 11 2016].

[9] T. E. Foundation, "Californium (Cu) CoAP Framework," [Online]. Available: http://www.eclipse.org/californium/. [Accessed 22 11 2016].

[10] SmeshLink Technology Co, "CoAp.NET," [Online]. Available: http://open.smeshlink.com/CoAP.NET/. [Accessed 22 11 2016].

[11] EXILANT Technologies Private Limited, "CoAPSharp," [Online]. Available: http://www.coapsharp.com/documentation/. [Accessed 22 11 2016].

[12] Arduino, "Arduino - ArduinoBoardUno," [Online]. Available: https://www.arduino.cc/en/Main/ArduinoBoardUno. [Accessed 22 11 2016].

[13] RASPBERRY PI FOUNDATION, "Raspberry Pi," [Online]. Available: https://www.raspberrypi.org/downloads/noobs/. [Accessed 22 11 2016].

[14] Open Source Community, "Raspbian," [Online]. Available: https://www.raspbian.org/. [Accessed 22 11 2016].