

Software Engineering

**Implement COAP module for IoTApi (IOT1.12)**

Submitted by: Hien Pham The (1067411)

First examiner: Prof. Dr. Andreas Pech

Second examiner: Mr. Damir Dobric

Statement

I confirm that I have written this thesis on my own. No other sources were used except those referenced. Content which is taken literally or analogously from published or unpublished sources is identified as such. The drawings or figures of this work have been created by myself or are provided with an appropriate reference. This work has not been submitted in the same or similar form or to any other examination board.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Date, signature of the student

Acknowledgement

I would like to express my deep appreciation Prof. Dr. Andreas Pech of the Frankfurt am Main University of Applied Sciences for being my examiner in this Project.

Besides that, I would like to thank my second examiner Mr. Damir Dobric for organizing the topic and guiding me through all of the tasks.

Moreover, my project would have failed or produced no result if I were not supported dedicatedly about the programming aspects by the module’s tutor: Mr. Mofaggol Hoshen

I finally would like to thank my friends for their suggestions, and assistance during conducting and writing of this report.

Content

[1 Introduction 5](#_Toc487446451)

[2 Theoretical Background 6](#_Toc487446452)

[2.1 Constrained Application Protocol (CoAP) 6](#_Toc487446453)

[2.1.1 Protocol Stack 6](#_Toc487446454)

[2.1.2 Request Methods 8](#_Toc487446455)

[2.1.3 Response Methods 8](#_Toc487446456)

[2.1.4 Messaging Types 9](#_Toc487446457)

[2.1.5 End points 9](#_Toc487446458)

[2.1.6 Messaging Format & Size 10](#_Toc487446459)

[2.1.7 CoAP URIs 11](#_Toc487446460)

[2.1.8 Requests 11](#_Toc487446461)

[2.1.9 Responses 11](#_Toc487446462)

[2.1.10 Request/Response Matching 13](#_Toc487446463)

[3 Realization 14](#_Toc487446464)

[3.1 IotApi Module 14](#_Toc487446465)

[3.2 CoAP Client Connector Implementation 16](#_Toc487446466)

[3.3 Unit Test 17](#_Toc487446467)

[4 Summary and Perspectives 19](#_Toc487446468)

[5 Abbreviations 20](#_Toc487446469)

[6 References 21](#_Toc487446470)

# Introduction

Since the first industrial revolution took place in about 1760, the history has passed more than four centuries. Nowadays, human is preparing to jump into the fourth industrial revolution. There is no more about pure machinery. Particularly, steam power, mass production or automation alone are histories. This is the time for peer to peer network (P2P), machine to machine communication (M2M) and Internet of thing (IoT), where computer system, network participate to connect physical machines together to create the smart home, smart factory or even smart city. In short, they are the future of humanity. Hence, this project was conducted as an application of the IoT Api (Application Programming Interface for Internet of Things).

The primary goal of this project is to implement the IoTApi module to communicate using CoAP (Constrained Application Protocol). Particularly, a middle module is created and developed as a connection between IoTApi module and the CoAP module. Last but not least, unit tests must also be generated so as to validating the possibility and functionality of the middle module.

This report is divided into several main sections as following:

1. Theoretical background: The section contents all of the information regarding background knowledge or previous research about CoAP and the IoTApi.
2. Realization: The realization section gives out every tutorial in details including the process of connecting and handling between modules. The implementation process and results with allegorical figures are also included in this section.
3. Summary and future perspectives: This is the final part of the project. It summarizes all of the result not only in theoretical evaluation but also the implementation results and validation.

# Theoretical Background

In this chapter, theoretical knowledge or previously researched background related to the future evaluation are presented. The reason is to establish the fundamental base for further understanding and accomplishing the goal of this project successfully. Notably, the section consists of CoAP’s basic definition as well as explanation about provided IoTApi module.

## Constrained Application Protocol (CoAP)

The Constrained Application Protocol (CoAP) was designed by the Constrained RESTful Environments (CoRE) workgroup in IEFT. It is a specialized web transfer protocol for use with constrained nodes and constrained networks. In the IoT, constraints on nodes typically emerge in terms of limited power supply, manufacturing costs, RAM, ROM, and generally low processing capabilities. Yet, constrained nodes, i.e., devices, are powerful enough to send and receive network packets and benefit from a connection to the Internet as they can be integrated into a distributed service (Lanter, M., 2013).

Specifically, the nodes often have 8-bit microcontrollers with small amounts of ROM and RAM, while constrained networks such as IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs) often have high packet error rates and a typical throughput of 10s of kbit/s. It uses a compact binary format and runs over UDP (or DTLS when security is enabled), which also enables multicast communication (Kovatsch, M. et al, 2014). The protocol is designed for machine-to-machine (M2M) applications such as smart energy and building automation (IETF RFC 7252, 2014).

CoAP provides a request/response interaction model between application endpoints, supports built-in discovery of services and resources, and includes key concepts of the Web such as URIs and Internet media types. CoAP is designed to easily interface with HTTP for integration with the Web while meeting specialized requirements such as multicast support, very low overhead, and simplicity for constrained environments (IETF RFC 7252, 2014).

### Protocol Stack

CoAP is an application layer protocol, which covers other lower layer protocols. The figure 2.1 is a typical illustration for the CoAP protocol stack, in which includes 5 layers: Physical layer, data/link layer, network layer, transportation layer and application layer. In fact, CoAP could include other protocols in each OSI layer, which will be listed out and explained as following.

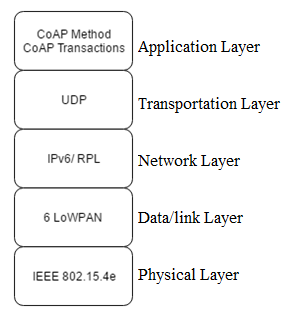


Figure 2.1 Example of a CoAP protocol stack

#### Layer 1: Physical Layer

In CoAP physical layer, PHY/MAC Layers involve all the common wireless communication technology, especially series IEEE 802.15.4 standard specifies MAC/PHY part for low-rate wireless personal area network (LR-WPAN) (Chen X., 2014). The 802.15.4-2006 (successor of the 2003 version) is the physical or layer 1 protocol for low-power and low-rate (data transfer at 250 kbits/s) LLNs. The key technical element of the new proposed 802.15.4e is channel hopping, which significantly increases robustness against noisy and lossy networks and persistent multi-path fading (Sutaria R. and Govindachari R., 2013).

#### Layer 2: Data/Link Layer

Base on the top of layer 1, in data/link layer, 6 LoWPAN is used. 6LoWPAN is an acronym of IPv6 over Low Power Wireless Personal Area Networks. It is an adaption layer for IPv6 over IEEE802.15.4 links. This protocol operates only in the 2.4 GHz frequency range with 250 kbps transfer rate (Postscapes.com). Moreover, a 25-octet header (no security) or a 46-octet header (AES-CCM-128) is added for each 6LoWPAN packet sent (Sehgal A., 2010).

#### Layer 3: Network Layer

According to information provided by (Sutaria R. and Govindachari R., 2013), IP could and should be applied to even the smallest of devices. IPv6, which has succeeded IPv4, has a near infinite address space, allowing for 2128 or approximately 3.4 x 1038 unique addresses. The new 6LoWPAN protocol was defined to enable IPv6 packets to be carried on top of low-powered and lossy personal area networks (LLNs).

#### Layer 4: Transportation Layer

CoAP is network-oriented protocol, using similar features to HTTP but also allows for low overhead, multicast, etc. Nevertheless, Unlike HTTP based protocols, CoAP operates over UDP, whose definition will be mentioned below, instead of using complex congestion control as in TCP. To compensate for the unreliability of UDP protocol, CoAP defines a retransmission mechanism and provides resource discovery mechanism with resource description, which will be discussed on section 2.1.4 (Chen X., 2014).

User Datagram Protocol (UDP) is, on the other hand, provides unreliable transport across the Internet. The reason why it is a best-effort delivery service is that there is no acknowledgment of sent datagrams. UDP adds an 8 octet (8 bytes) header field to datagrams. Even though UDP is not equipped with datagram acknowledgement, it does detect datagrams with errors with a checksum. It is up to higher layer protocols to detect this datagram loss and initiate a retransmission if desired. Sequence numbers, acknowledgments, and window sizes, congestion control or flow control are not included. If any of these functions are needed, they must be built into the application layer protocol (Johnston, 2016: 8).

#### Layer 7: Application Layer

As shown in figure, CoAP employs a two-layer structure: methods and transactions. The transaction sub-layer provides duplicate detection and reliable delivery of messages based on a simple stop-and-wait mechanism for retransmissions. On top, the request/response sub-layer enables RESTful interaction through the well-known methods GET, PUT, POST, and DELETE as well as response codes that are defined in the following sections (Kovatsch, M. et al, 2014).

### Request Methods

CoAP only supports the basic methods of GET, POST, PUT, and DELETE, which are described in detail in table below (IETF RFC 7252, 2014). A request is initiated by setting the Code field in the CoAP header of a Confirmable or a Non-confirmable message to a Method Code and including request information.

Table 2.1: CoAP request method names and description

|  |  |
| --- | --- |
| **Request Method Names** | **Descriptions** |
| GET | Retrieves a representation for the information that currently corresponds to the resource identified by the request URI. |
| POST | Process the representation enclosed in the request. |
| PUT | Update or create the resource identified by the request URI with the enclosed representation. |
| DELETE | Delete the resource identified by the request URI. |

### Response Methods

Besides request methods, response codes should also be noticed according to RFC 7252 in the table 2.2. After receiving and analyzing the request, server end points return CoAP response whose token matches to the token in CoAP request. Similar to other protocol, response messages indicate the result of the attempt to understand and satisfy the request (IETF RFC 7252, 2014).

Table 2.2: CoAP response code

|  |  |  |  |
| --- | --- | --- | --- |
| **Type 2xx: Success: indicates that the clients request was successfully received, understood, and accepted.** | | | |
| 2.01 | Created | 2.02 | Deleted |
| 2.03 | Valid | 2.04 | Changed |
| 2.05 | Content |  |  |
| **Type 4xx: Client error: intended for cases in which the client seems to have errored.** | | | |
| 4.00 | Bad Request | 4.01 | Unauthorized |
| 4.02 | Bad Option | 4.03 | Forbidden |
| 4.04 | Not found | 4.05 | Method not allowed |
| 4.06 | Not acceptable | 4.12 | Precondition Failed |
| 4.13 | Request Entity too large | 4.15 | Unsupported Content-Format |
| **Type 5xx: Server Error: indicates cases in which the server is aware that it has erred or is incapable of performing the request.** | | | |
| 5.00 | Server internal error | 5.01 | Not implemented |
| 5.02 | Bad Gateway | 5.03 | Service unavailable |
| 5.04 | Server Timeout | 5.05 | Proxying not supported |

### Messaging Types

CoAP was designed with options to manipulate the reliability of sending messages. Messages are provided with label or marking as Confirmable (CON), Acknowledgement (ACK), Non-confirmable (NON), Reset (RST). Each particular message type as well as their example will be explained separately.

According to RFC 7252, a confirmable message basically requires receiving peers to send back an acknowledgement or reset messages. A Confirmable message is retransmitted using a default timeout and exponential back-off between retransmissions, until the recipient sends an acknowledgement message with the same Message ID (for example in figure 2.2.a) (IETF RFC 7252, 2014). When a recipient is not at all able to process a Confirmable message, it replies with a Reset message instead of an Acknowledgement.

On the other hand, a non-confirmable message does not require answer messages sent by the receiver. These are not acknowledged, but still have a Message ID for duplicate detection (figure 2.2.b). When a recipient is not able to process a Non-confirmable message, it may reply with a Reset message.

Additionally, in an acknowledgement message, it notifies that messages are received and many carry payload, while reset messages indicate that messages have been received but some context is missing to process.

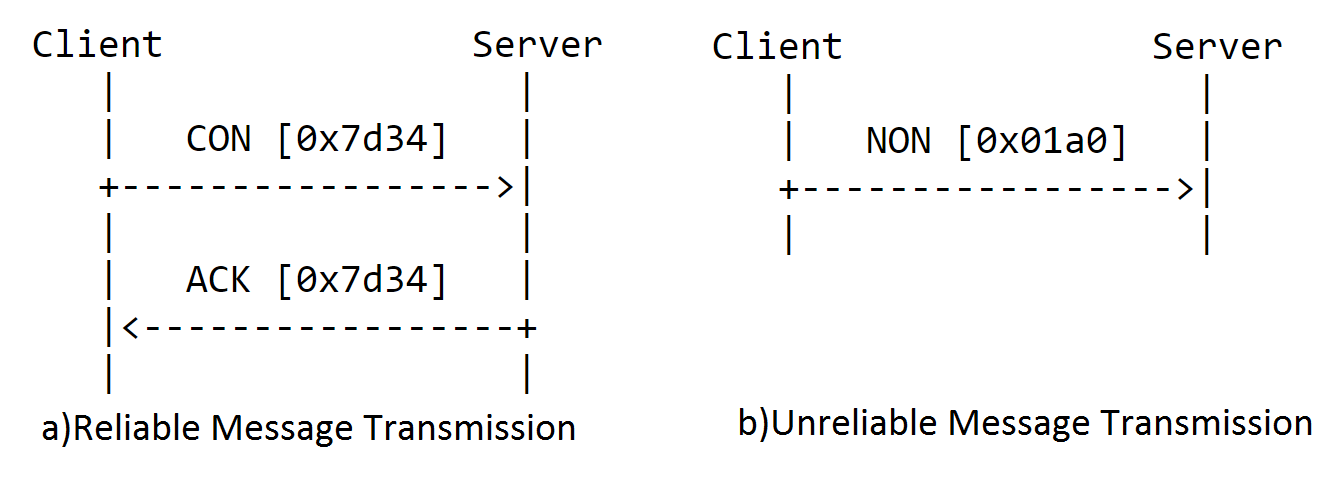


Figure 2.2 CoAP messaging types (IETF RFC 7252, 2014)

### End points

Basically, in CoAp endpoints is considered as end user agent. According to (IETF RFC 7252, 2014: 20), CoAP endpoints are defined as the source and destination where CoAP messages departs and arrives respectively. Depending on transport being used for CoAP, endpoints have specific definition. In the scope of this project, security is out of bound. Thus CoAP endpoints are defined by IP address and UDP port number.

CoAP operates under a similar request/response model. In which CoAP endpoint in the role of a “client” sends one or more CoAP requests to a server, which replies the requests by sending CoAP responses. Requests and responses are exchanged asynchronously over CoAP messages.

Last but not least, at each endpoint, there could exist more than one resource. And for the sake of distinguishing each resource in one endpoint, each resource must be assigned with a name which is also called as resource identifier in implementation.

### Messaging Format & Size

According to the RFC 7252, CoAP was designed to have 4 bytes in header field. As illustrated figure, the header fields contains four fields:

* Ver (Version), which is 2 bits in size, indicates the CoAP version number
* T (Type), which is 2 bits in size, indicates the message types mentioned in section 2.2.4
* TKL (Token length), which is 4 bits in size, indicates number of token bytes after this header
* Code, which is 8 bits in size, indicates the message request method (1-10) or response code (40-255)
* Message ID, which is 8 bits in size, detects message duplication, and matches response

Following header are token and options. The token is used to match a response with a request. The token value is a sequence of 0 to 8 bytes. In which three first bits are used to identify the class of response, while the lower five bits do not have any categorization role; they give additional detail to the overall class. This token is displayed in form of “*c.dd*” with *c* is class in decimal and *dd* is details as two-digit decimal, for instance 2.04 means *change*.

Option is also a part in CoAP message format. It may be used in both request and response. The list of option is as below:

* Content-Format
* Etag
* Location-Path
* Location-Query
* Max-Age
* Proxy-Uri
* Proxy-Scheme
* Uri-Host
* Uri-Path
* Uri-Port
* Uri-Query
* Accept
* If-Match
* If-None-Match
* Size1

The field before payload is the Payload Marker, which is 1 byte (0xFF). This field indicates the end of option and start of the payload. In other to avoid fragmentation, CoAP was designed to fit within a single IP packet. Then CoAP specification itself provides only an upper bound to the message size. For example, assumes an IP MTU is 1280 bytes, if nothing is known about the size of the headers, good upper bounds are 1152 bytes for the message size and 1024 bytes for the payload size.

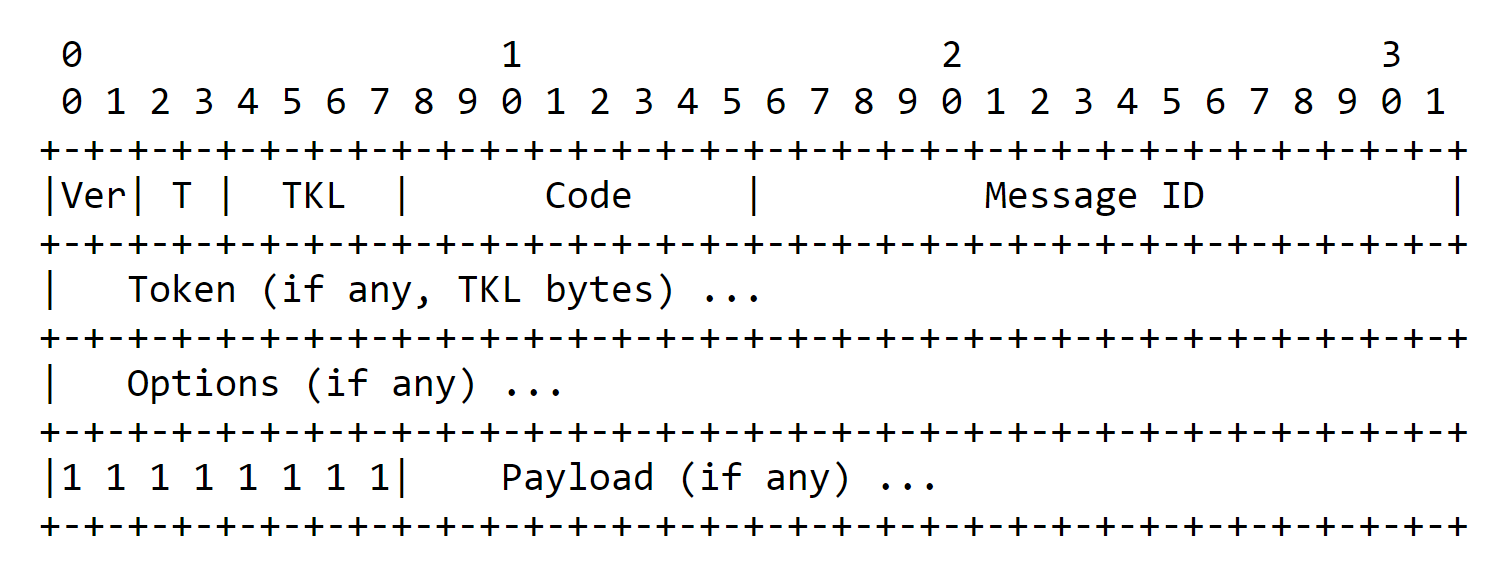


Figure 2.3 Typical CoAP header (IETF RFC 7252, 2014)

### CoAP URIs

There are two schemes of available CoAP URIs defined by the RFC 7252, which are the “coap” and “coaps”. These schemes are used for identifying CoAP resources and providing methods to locate these resources. Resources are organized hierarchically and governed by a potential CoAP origin server listening for CoAP requests ("coap") or DTLS-secured CoAP requests ("coaps") on a given UDP port. Nevertheless, since security will not be discussed in this project, only scheme “coap” is considered.

The coap URI scheme is written as:

coap-URI = "coap:" "//" host [ ":" port ] path-abempty [ "?" query ]

For example, coap://example.se:5683/~sensors./temp1.xml

### Requests

CoAP could support four basic methods namely GET, PUT, POST and DELETE. PUT request is taken as an example for explaination.

The PUT method requests that the resource identifier pointed by the request URI be created or updated with the enclosed representation. The representation here is the message body (a.k.a message payload) included in the request. The representation format is specified by the media type and content coding given in the *Content-format* option.

If the resource exists at the request URI, the enclosed representation is considered as a modified version of that resource. As a result, a *2.04 Changed* response code should be generated. In the case, where no resource exists, server may create a new resource in the address at URI, and server content is the representation. In response, server return to client a *2.01 Created* response code.

### Responses

After receiving and interpreting a request, a server responds with a CoAP response. A response is identified by the *code field* in the CoAP header being set to a Response Code. The CoAP response code indicates the result of the attempt to understand and satisfy the request. Details in response code are illustrated in section 2.2.3. Responses could be sent in three ways, such as piggybacked, separated response and non-confirmable response.

#### Piggybacked Response

This is the most basic case in CoAP response. Immediately right after receiving confirmable requests, responses are carried directly in the acknowledgement message. The responses are independent of being whether success or failure. There is no separate message, which will be discussed next, is required to return the response.

#### Separated Response

Not all of cases where resource could reply for confirmable requests in the piggybacked response. Instead, server might spend time on collecting resource representation so as to send it back in the acknowledgement messages. Consequently, client might reproduce and retransmit requests repeatedly. Therefore, separated response mode should be applied.

Particularly, as illustrated in figure 2.7, when the server chooses to use a separate response, it sends the acknowledgement to the confirmable request as an empty message. Once this message is sent from server, it is not allowed to send any other acknowledgement message to client, even if clients retransmits other identical requests. After some time, whenever server has representation, it sends a confirmable response. And client’s reaction must be an acknowledgement response with empty message. Server stops retransmitting confirmable response until it receives a matching acknowledgement or reset message.

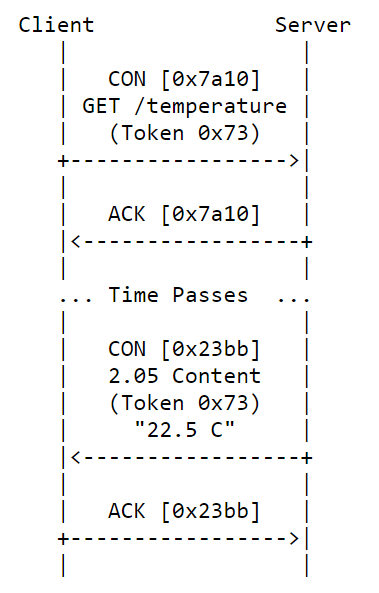


Figure 2.4 A GET request with a seperated response (IETF RFC 7252, 2014)

#### Non-confirmable Response

Non-confirmable response is sent in response to the non-confirmable request message. However, in some minority cases, non-confirmable response could be sent in reply to a confirmable request with a condition that this response must be followed by an empty acknowledgement message.

### Request/Response Matching

As already mentioned in section above, there are two headers namely token and message ID, are used to be as reference for matching request and response in CoAP. Specifically, according to RFC 7252 (IETF RFC 7252, 2014: 34), the exact rules for matching a response to a request are as follows:

* The source endpoint of the response must be the same as the destination endpoint of the original request.
* In a piggybacked response, the Message ID of the Confirmable request and the Acknowledgement must match, and the tokens of the response and original request must match. In a separate response, just the tokens of the response and original request must match.

# Realization

The realization section gives detail information about developing implementation for this project. The section consists of three main sub-sections, namely understanding the provided IoTApi modules, implementing application procedure, implementing unit test classes to test the implemented methods.

* The first subsection introduce and describe the functionality of the provided IoTApi module. Methods to injecting other module will also be mentioned and explained.
* The second subsection mentions about how the implementation was created. The description gathers all of stages in establishing and developing C# application about communication using CoAP.
* The final one explain the purpose of each unit test methods.

## IotApi Module

In the first glance, IoT stands for Internet of Things, and Api stands for Application Programming Interface. The Api is basically a software application that allow devices to talk to each other and exchange information. It also allow developers to have a frictionless but also secure interaction so that they can build apps. In combination, IotApi is the interaction between an IoT device and the Internet or other elements in the network. IoT devices would be useless without the IotApi. APIs are tightly linked with IoT because they allow you to securely expose connected devices to customers, go to market channels and other applications in your IT infrastructure. On the IoT, data is everywhere – flowing from devices to the cloud, from the cloud to your back-end systems, from users back to their devices – all enabled by APIs. API management enables you to govern this flow of data with the security you need to protect sensitive information, and the performance required to support connected cars, connected homes and other Internet-connected things.

In the scope of this project, the IotApi module is an open source .NET Core unified API for IoT. The idea behind this API is to provide various modules, which implement different protocols for different devices and services.

Additionally, modules do not have to provide protocol implementation only. Module can be used for encryption, message retry, message persistence, compression etc. For example, it might be possible to implement an application which uses ProfiNET protocol to provide a communication to some machines. But the same application might use IoTHub, which enables communication to Azure IoTHub cloud service. By using unified API one application, which supports multiple protocols and cloud services can be easier implemented.

IoTApi is designed to provide execution pipeline of different modules. Following picture illustrates sending of the message through pipeline of modules:

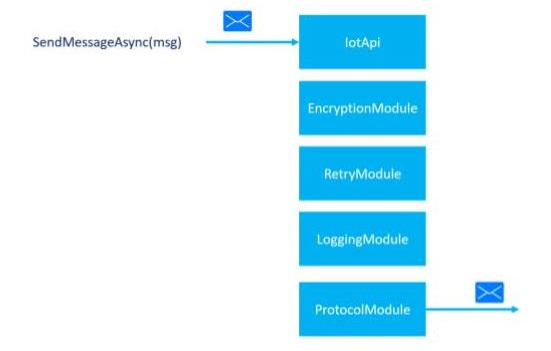


Figure 3.1: IotApi sending procedure

Analog to sending of the message, receiving of the message can use following pipeline:

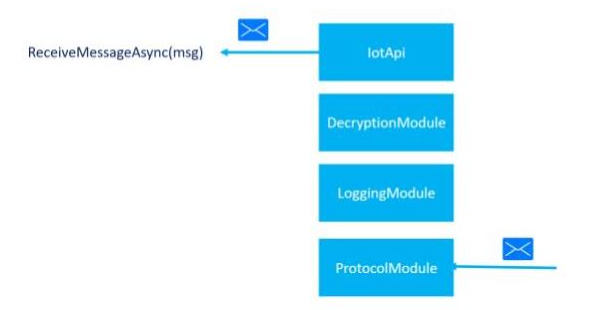


Figure 3.2: IotApi receiving Procedure

## CoAP Client Connector Implementation

As mentioned in the introduction section, the sharp point of this project is to implementing a module in .NET core. The functionality of this module as shown in figure 3.3, is to connecting between a CoAP implementation and the provided IotApi.



Figure 3.3: Middle module

To be more specific, the middle module represent the *CoAPClientConnector* class. This class inherit from the *ISendModule* and *IReceiveModule.* Which gives the possibility to wrap the functionality of the CoAP implementation and register this module to the IotApi. Particularly, this module includes 3 methods: Open, Sendasync, Receiveasync as illustrated in figure 3.4.

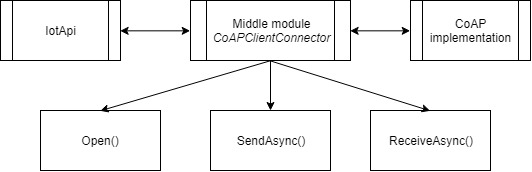


Figure 3.4 Methods in CoAPClientConnector Class

Method Open(): register new module to the IotApi. Besides, in this method, a *client* object is created to start sending to receiving messages later on.

Method SendAsync(): asynchronously send a CoAP message with a message as an input object parameter. At least, this message object must include 2 properties: message ID and message type. These parameters have already explained in the theoretical background section. Another important point in this method, as well as the receiving method is object client must create a listening point before sending or receiving any CoAP message. The process of sending a CoAP message is illustrated in figure 3.5 as following. It could be seen that, 2 compulsory parameters namely message ID and message Type must be included in each CoAP request. After creating the listening point, these two parameters will be checked and then the message could be sent.

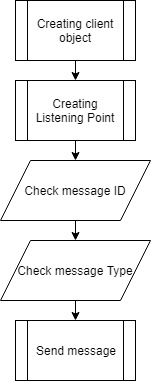


Figure 3.5: Procedure of sending out a message

Method ReceiveAsync(): asynchronously receive a CoAP message. In other to test this functionality, an instance GET request message is sent to an example server. This message request for some type of information, in other for the node to receive. The procedure for testing the receiving asynchronous functionality is presented in figure 3.6.

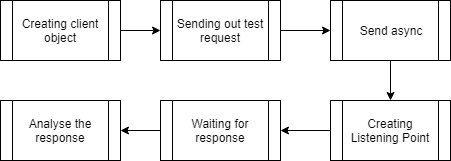


Figure 3.6: Asynchronously receiving CoAP message

## Unit Test

This section will demonstrate all of the implemented unit test in the project.

In the unit testing section, there are two main parts: testing the sending asynchronous functionality, and testing receiving asynchronous functionality.

The first part include the following test:

* Test client request delete
* Test client request get
* Test client request put
* Test client request post

The second one has:

* Test client response get

Besides, some other test cases are implemented. However, these cases are not relevant to the IotApi, instead in test the behavior of the CoAP node.

* Test multicast message is non-confirmable: test the ability of handling multiple endpoint.
* Test multicast message from multicast endpoint:

# Summary and Perspectives

In conclusion, this report is conducted for the course Software Engineering of the Master IT at Frankfurt am Main University of Applied Sciences. The scope of the project is to implement the IotApi with CoAP protocol in .NET Core framework. Based on theoretical background, asynchronous sending and receiving module is generated. The projects also include some unit test to validate the ability of interacting between the given IotApi and the CoAp Protocol.

# Abbreviations

**6**

6LoWPAN Internet Protocol version 6 over Low Power Wireless Personal Area Network

**A**

API Application Program Interface

**C**

CoAP Constrained Application Protocol

**I**

IoT Internet of Things

IP Internet Protocol

IPv4 Internet Protocol version 4

IPv6 Internet Protocol version 6

**T**

TCP Transmission Control Protocol

**U**

UDP User Datagram Protocol

URI Universal Resource Identifier

URL Universal Resource Locator

# References

1. Chen X. (2014). *Constrained Application Protocol for Internet of Things*. [Online] St. Louis: Washington University. Available at: <http://www.cse.wustl.edu/~jain/cse574-14/ftp/coap.pdf> [Accessed 20th Dec. 2016].
2. Johnston, Alan B. (2016). *SIP: Understanding the Session Initiation Protocol*, Fourth Edition, USA: Bartech House.
3. Kovatsch, M., Lanter, M., & Shelby, Z. (2014, October). Californium: Scalable cloud services for the internet of things with coap. In *Internet of Things (IOT), 2014 International Conference on the IEEE*. Switzerland: ETH Zurich, pp. 1-6. Available at: https://www.vs.inf.ethz.ch/publ/papers/mkovatsc-2014-iot-californium.pdf [Accessed 3rd Jan. 2017].
4. Lanter, M., (2013). *Scalability for IoT Cloud Services*. [online] Zurich: ETH Zurich, p5. Available at: https://people.inf.ethz.ch/mkovatsc/resources/californium/cf-scalability.pdf [Accessed 3rd Jan. 2017].
5. Postscapes.com, (n/a). *Internet of Things Protocols & Standards*. [online] Available at: http://postscapes.com/internet-of-things-protocols [Accessed 20th Jan. 2017].
6. Sehgal A. (2010). *A Practical Introduction to 6LoWPAN Programming IPv6 Wireless Sensor Networks with Contiki.* 1st. [pdf] Bremen: Jacobs University Bremen. Available at: <http://cnds.eecs.jacobs-university.de/slides/2010-aims-6lowpan.pdf> [Accessed 9th Jan. 2017].
7. Shelby, Z., Hartke, K. and Bormann, C., (2014). RFC 7252. *The Constrained Application Protocol (CoAP)*, IETF. [Online]. Available at: <https://tools.ietf.org/html/rfc7252> (Accessed 18th Jan. 2017).
8. Sutaria R. and Govindachari R. (2013). *Understanding The Internet Of Things*. [online]. Electronic Design. Available at: http://electronicdesign.com/iot/understanding-internet-things [Accessed 20th Jan. 2016].