# Analysis of CoAP Implementations for Industrial Internet of Things: A Survey

Markel Iglesias-Urkia[1], Adrián Orive[2], Aitor Urbieta[3]

IK4-Ikerlan Technology Research Centre
Information and Communication Technologies Area

Pº J.M.Arizmendiarrieta, 2, 20500
Arrasate-Mondragón, Spain

Tel.: +34 943 712 400
Fax.: +34 943 796 944

1: miglesias@ikerlan.es
orcid.org/0000-0001-7708-3252

2: aorive@ikerlan.es
orcid.org/0000-0003-2919-5799

3: aurbieta@ikerlan.es
orcid.org/0000-0001-5836-4198

Diego Casado-Mansilla

Deusto Institute of Technology
DeustoTech-INTERNET

Av. Universidades, 24, 48007
Bilbao, Spain

Tel.: +34 944 139 003 (Ext: 2977)

dcasado@deusto.es
orcid.org/0000-0002-1070-7494

***Abstract*** **-** Over the last few years, the Internet of Things (IoT) has grown in protocols, implementations and use cases. In terms of communication protocols, the Constrained Application Protocol (CoAP) prevails among the rest, such as MQ Telemetry Transport (MQTT) or Advanced Message Queuing Protocol (AMQP). This protocol is lightweight and capable of running in resource constrained devices and networks and can be securized using Datagram Transport Layer Security (DTLS). Having a secure channel of communication is important in IoT environments, since IoT devices affect the physical world and exchange personal private data. There exist many implementations of CoAP, each of these with its own particular features and requirements. Therefore, it is important to choose the CoAP implementation that suits better to the specific requirements of each application. This paper presents a feature and empirical comparison of several open source CoAP implementations and also analyzes the security libraries they use. First of all, it surveys current CoAP implementations, and compares them in terms of built-in core, extensions, target platform, programming language and interoperability. Then, a theoretical analysis of the security libraries is provided. Finally, it analyzes CoAP libraries' performance in terms of latency, memory and CPU consumption in a real testbed deployed in an industrial scenario, in order to help in adopting a decision criteria for similar deployments.

***Keywords*** - CoAP; Benchmarking; DTLS; Survey; Internet of Things; Industrial Internet of Things

***Acknowledgements*** - This work has been partially supported by the Basque Government through the Elkartek program under the LANA II project (Grant agreement no. KK-2016/00052), as well as the Spanish Government and the H2020 research framework of the European Commission.

# 1. Introduction

"The Internet of Things is a system of interrelated computing devices, mechanical and digital machines, objects, animals or people that are provided with unique identifiers and the ability to transfer data over a network without requiring human-to-human or human-to-computer interaction" (Rouse, 2016). According to Evans (Evans, 2011) and Chase (Chase, 2013), by 2020, 50 billion devices will be connected to the IoT while Jeon (Jeon, 2011) further claims that they will reach 75 billion. Even though these predictions are probably too optimistic (as by 2015 we are far behind the expected growth predicted by (Evans, 2011)), the amount of interconnected objects grows daily at a fast pace.

Some common Internet protocols, such as HTTP (Fieldings, et al., 1999), have been originally used to connect legacy IoT devices. However, new, lighter ones have emerged, to fit the constrained requirements imposed by IoT environments, including CoAP (Shelby, et al., 2014), MQTT (Banks & Gupta, 2014), AMQP (AMQP, 2014) and DDS (Object Management Group, 2016) among others. Whereas CoAP is one of the newest protocols, it is getting great popularity as it follows the REST paradigm, making the adaptation process from HTTP easy for developers. Besides, it is very lightweight both in terms of power, resources and network requirements.

One of the final goals of the IoT industry is to have the possibility to deploy as many different kind of devices as possible. As suggested in the previous paragraph, CoAP could be a good fit to do so, but it is very important to correctly choose among the wide pool of implementations that currently exist on the literature (Bormann, 2016). The selected one has to support the requirements of the system and it has to correctly balance the required CPU, memory resources and energy consumption.

On industrial applications, securing IoT devices is of pivotal importance due to the huge number of connected devices. Hence, security features can not be left out of an implementation analysis. In this regards, two concepts arise: authentication and privacy. On the one hand, as demonstrated on the 2016 Dyn attack (Hilton, 2016), a Distributed Denial of Service (DDoS) attack can be very effective using small, connected devices. This attack was carried out with the Mirai botnet, a malware that targets IoT devices (The New Jersey Cybersecurity & Communications Integration Cell , 2016). To achieve this attack, the firmware of the devices has to be changed, which should be avoidable with proper authentication mechanisms. On the other hand, the importance of privacy is more straightforward to understand: from video streams from home cameras to bank account information for self-refilling fridges, private data is constantly being send.

Having provided a general review of IoT, the main objective of the work presented in this paper is to ease the selection of the CoAP implementation that better fits each project by analyzing different open source implementations' features and experimental behavior. The selection of the proper implementation for each project must be based both on offered features and performance. The implementations selected for the analysis target several platforms and are written in different languages. All the tests have been conducted on an industrial deployment over the Raspberry Pi (https://www.raspberrypi.org/) platform, analyzing the response time or latency, as well as the memory and CPU needed to deploy the implementations. A preliminary theoretical analysis of the ongoing security development of these libraries has also been carried out, as there is still no library with security features fully implemented.

The rest of this paper is organized as follows. First, the related work is presented. Next, CoAP is described along with some extensions and implementations and an explanation on how to securize CoAP communications. Section 4 features a comparison of different implementations, followed by a discussion about the security features of the analyzed CoAP libraries. Section 6 describes the experiment set-up and the measured parameters and their results are presented in Section 7. Finally, conclusions and guidance for future work are provided in the last two sections.

## 2. Related Work

In the earliest state of the implementation, when CoAP's standardization was not complete, there were some pieces of research analyzing different implementations that are summed up in Table 1. In (Villaverde, et al., 2012), the authors analyze CoAP theoretically short after the first draft was presented and analyze some of the implementations at that time. The paper presents a list of available implementations and some conclusions based on analysis from surveyed researches. Lerche et al. (Lerche, et al., 2012) summarize the results of the first ETSI CoAP Plugtest (ETSI, 2012). They present the participant implementations and their interoperability but they do not offer a performance analysis of any sort. Both these articles present a list of available implementations, but since they were written, CoAP's standardization went on and new libraries have been implemented, so they are not up to date. From the best of our knowledge, there are no updated surveys of the current implementations since Table's authors. Therefore, it is necessary to widen the search parameters to other protocols or focus on a single implementation on different hardware, such as (Kruger & Hancke, 2014) where a benchmarking of a single CoAP implementation on different hardware (Raspberry Pi, BeagleBone and BeagleBone Black) is presented. They measure latency, bandwidth and CPU and memory usage through the loopback interface over class 4 and 10 SD cards with active, deactive and uninstalled GUI.

| Authors | Platform / Use case | Metrics | Downsides |
|---|---|---|---|
| **Lerche et al.** | 1st Interoperability plugtest & performance analysis | Performance based on other papers | Old implementations, no own performance analysis |
| **Villaverde et al.** | 1st CoAP draft implementations | Interoperability | No performance analysis |
| **Kruger et al.** | Raspberry Pi, BeagleBone & BeagleBone Black | Latency, bandwidth, CPU & memory | Same implementation on different hardware |

**Table 1: Previous CoAP benchmarks**

Expanding the analyzed protocols scope, CoAP and HTTP have similar structure even though they target different environments. In (Ludovici, et al., 2013), the authors present their own implementation for TinyOS called TinyCoAP. They compare its performance against HTTP/TCP, HTTP/UDP and the original TinyOS CoAP implementation, CoAPBlip, on TelosB motes and they measure latencies and memory and energy consumption. Colitti et al. (Colitti, et al., 2011) work on Tmote Sky and Zolertia Z1 motes to measure response time and energy consumption. Kuladinithi et al. (Kuladinithi, et al., 2011) present their own implementation called libcoap and port it to Contiki and TinyOS to compare the performance against HTTP. Elmangoush et al. (Elmangoush, et al., 2015) present CoAP, HTTP, MQTT and AMQP, but they only work on the two former protocols. They use the OpenMTC platform to measure the bandwidth per request interval time, the response time per request interval time and the response time for different payload sizes.

Another popular IoT protocol is MQTT and De Caro et al. (De Caro, et al., 2013) measure several parameters such as the latency, the bandwidth usage and the package loss ratio over different QoS and network loss configurations on Smartphone implementations of CoAP and MQTT. Thangavel et al. (Thangavel, et al., 2014) present a common middleware based for both MQTT and CoAP and measure the latency and bandwidth consumption.

There are more popular IoT protocols besides CoAP and MQTT. Talaminos-Barroso et al. (Talaminos-Barroso, et al., 2016) implement a tool for an eHealth application with the possibility of using DDS, MQTT, CoAP, JMS, AMQP and XMPP. They benchmark the CPU and memory usage, the bandwidth consumption and the messages' latency and jitter. Mun et al. (Mun, et al., 2016) selected CoAP, MQTT, MQTT-SN, WebSockets and TCP. Similar as we do in this paper, the authors aim to ease programmers make a good choice selecting the protocol that better fits for their applications, and to do so, they measure the performance, the energy efficiency and the memory and CPU usage. Chen et al. (Chen & Kunz, 2016)

analyze MQTT, CoAP, DDS and a custom protocol over UDP. They use a network emulator to configure different parameters and measure consumed bandwidth, latency and package loss.

As it has been presented in previous paragraphs and grouped in Table 2, CoAP's performance has been compared against other protocols, such as HTTP and MQTT. Some CoAP implementations' performance has also been studied on different hardware. However, the only work done analyzing different implementations of CoAP is based on early drafts of the CoAP specification; therefore it is updated. Besides, it analyzes their interoperability and main features; but it does not conduct a performance analysis. Thus, this paper's goal is to fill that gap offering an up to date comparison, both empirical and feature-based, to help system designers choose the implementation that better fits their requirements for industrial environments.

| **Authors** | **Protocols** | | | | | | | | | | **Platform / Use case / Operative System** | **Metrics** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | HTTP | HTTP/UDP | MQTT | DDS | JMS | AMQP | XMPP | MQTT-SN | C. UDP | TCP | | |
| **Ludovici et al.** | ✔ | ✔ | | | | | | | | | Telos B | Latency, memory, energy |
| **Colitti et al.** | ✔ | | | | | | | | | | Tmote Sky & Zolertia Z1 | Latency, energy |
| **Kuladinithi et al.** | ✔ | | | | | | | | | | Contiki & TinyOS | Performance |
| **Elmangoush et al.** | ✔ | | | | | | | | | | OpenMTC | Bandwidth, latency, latency/ payload |
| **De Caro et al.** | | | ✔ | | | | | | | | Smartphone | Latency, bandwidth and package loss ratio |
| **Thangavel et al.** | | | ✔ | | | | | | | | Middleware | Latency, bandwidth |
| **Talaminos-Barroso et al.** | | | ✔ | ✔ | ✔ | ✔ | ✔ | | | | eHealth | CPU, memory, bandwidth, latency, jitter |
| **Mun et al.** | | | ✔ | | | | | ✔ | | ✔ | | Performance, energy, memory, CPU |
| **Chen et al.** | | | ✔ | ✔ | | | | | ✔ | | | Bandwidth, latency, package loss |

**Table 2: CoAP and other protocols comparisons**

Regarding security for IoT, the adoption is in its early phases. Discussion is still been carried out to analyze the feasibility of adding security to CoAP based communications, using the Datagram Transport Layer Security (DTLS) (Rescorla & Modadugu, 2012) in resource constrained devices, due to the costs of the handshake and cryptographic calculations.

Kothmayr et al. presented a preliminary work in (Kothmayr, et al., 2012) to later expand it in (Kothmayr, et al., 2013). In this article, the authors point out the need to add security to communications to avoid attackers to capture the traffic between IoT devices. In this case, they propose to use DTLS and evaluate its performance in an OPAL sensor node. They conclude that it is possible to add two way authenticated asymmetric encryption through X.509 certificate exchange with less than 20kB RAM usage and small power consumption.

Kwon et al. (Kwon, et al., 2015) analyze the challenge of using DTLS with the different available modes. To do that, the authors compare the RawPublicKey and PreSharedKey modes of DTLS both in a simulation and in a real test-bed to analyze the comparison in terms of code size, energy consumption, and processing and receiving time. Finally, they conclude that Class 1 devices can use PreSharedKey

mode but even though RawPublicKey is mandatory to implement CoAP over DTLS, Class 1 and Class 2 devices can not implement it well due to resource limitations.

Based on the literature presented in previous paragraphs, it can be concluded that DTLS support for resource constrained devices is still a work in progress. Very resource constrained devices need to be secured in order to avoid attacks, but nowadays there is not yet a complete solution.

# 3. CoAP

Published as RFC 7252 by the IETF CoRE Working Group (IETF, 2017) on June 2014, the Constrained Application Protocol (Shelby, et al., 2014) is a web transfer protocol designed for resource constrained devices and networks. It is built on top of UDP and it follows the REST paradigm, so it works similar to HTTP. The different technologies involved in IoT applications compared to the general use case stack are shown in Table 3. CoAP can not use TLS, as it requires ordered and guaranteed message delivery and the underlying UDP does not provide those guarantees, to securize the connections. Instead, DTLS is UDP's alternative to TLS, but it pays the cost of losing some of the benefits derived from using UDP such as not requiring an open connection as the endpoints need to save session information to be able to code and decode messages.

| Layer | IoT | General Applications |
|---|---|---|
| **Application** | CoAP(s) | HTTP(s) |
| **Security** | DTLS | TLS |
| **Transport** | UDP | TCP |
| **Network** | IPv6<br>6LowPAN | IPv4, IPv6 |
| **Link** | IEEE 802.15.4 | IEEE 802.3, 802.11 |

**Table 3: IoT and general application stacks**

CoAP uses a subset of HTTP's request verbs: GET, POST, PUT and DELETE; also response codes and content-formats, even though in this last case there is an additional one. The default port for CoAP is 5683 and the one for secure CoAP 5684. A CoAP URI consists on the following format: *coap://host[:port]/[path][?query]*. The user needs to know the path in order to access the available resources, and it will also be able to send queries to the server in the URI.

CoAP messages follow the structure shown in Figure 1 with a 4 byte fixed-length header and a variable-sized part. The first 2 header bits define the version of the CoAP specification and the following 2 bits represent the message's type. As mentioned, the underlying UDP protocol does not guarantee message delivery so CoAP implements this guarantee in a higher stack level though two of its four message types: confirmable messages and non-confirmable ones. The other two type of messages are acknowledgements that may contain piggybacked data and resets. The rest of the first byte defines the length of the token, present in the variable-sized part and limited to 8 bytes. Next header byte represents the message code and the remaining 2 bytes contain the message ID. The variable-sized part has different sections: 0-8 bytes long token as specified in the header, and a variable number of options preceding a full set byte followed by the optional payload.

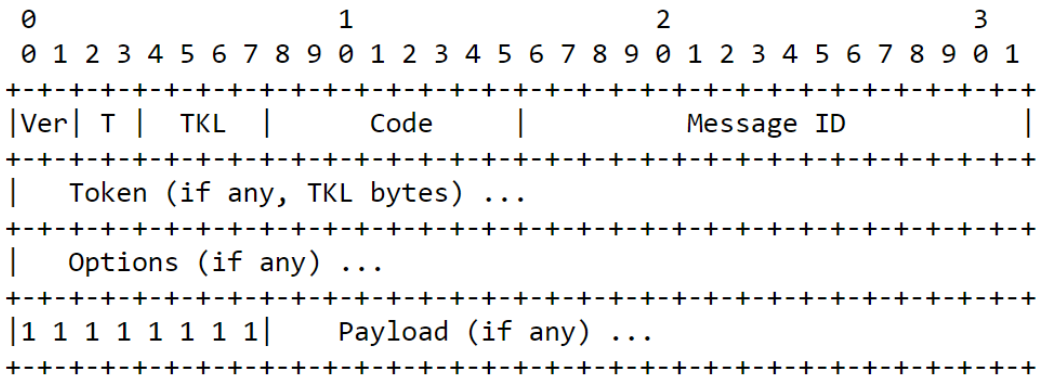



**Figure 1: CoAP message**

The IETF proposes some extensions to broaden the capabilities of the CoAP specifications:

- *Constrained RESTful Environments (CoRE) Link Format* (Shelby, 2012): The IETF defines the format for the links that constrained servers use to describe their resources, attributes and relationships between links. This format is used for the payload and it uses a well-known URI as a default point of entry to get the resources of the server. It is based on the HTTP Link Header field (RFC 5988) (Nottingham, 2010) and it defines a new Internet media type, "application/link-format".
- *Block-Wise Transfers in the Constrained Application Protocol (CoAP)* (Bormann & Shelby, 2016): CoAP tries to avoid IP fragmentation and to do so, it uses small payloads. That is not a problem with most use cases, as sensors will not send large data. But in other cases large payloads will be needed, for example for firmware updates. To solve this, the Block-Wise Transfer extension adds two new options, one for the request and another for the response. It uses the stop and wait paradigm to send the data sliced in pieces with a flag indicating whereas it is the last piece or not.
- *CoRE Resource Directory* (Shelby, et al., 2016): The IETF proposes an entity named Resource Directory (RD), which has the information about resources of other CoAP servers. It is useful to save battery in those other servers and to ease the server discovery, without needing to scan the entire network or use multicast addresses. There are three ways for the RDs to get the information about the servers. The servers can send their information themselves, the RD can ask the servers to send the information and in some cases the information can be achieved in boot time from a third device, called the commissioning tool. The commissioning tool gets that information from a database or similar.
- *Observing Resources in the Constrained Application Protocol (CoAP)* (Hartke, 2015): With this extension, a CoAP server is able to send push notifications to registered clients, like other publish/subscribe based protocols such as MQTT or XMPP. It enables clients to subscribe to a resource and receive changes of the resource's data. The server is the device that decides what a change is, i.e., it can decide to send a notification when a value changes, when a value changes out of a range or just periodically. A petition to subscribe is a GET request with the observe option enabled. To stop the subscription, the client needs to send another GET request to the same resource but with the observe option disabled or a reset message.
- *Group Communication for the Constrained Application Protocol (CoAP)* (Rahman & Dijk, 2014): CoAP is designed to use IPv6 in the internet layer and IPv6 supports multicast by default. This extension explains how CoAP should be used in a multicast environment and it also defines some new features, such as new CoAP processing functionalities (e.g., new rules for reuse of Token values, request suppression and proxy operation) and a new management interface.
- *CoAP Simple Congestion Control/Advanced CoCoA* (Bormann, et al., 2015): The CoAP specification proposes basic behaviors to avoid network congestion. To add more sophisticated methods, the IETF is working on the CoCoA draft.

There are many available CoAP implementations with different features and targets. The aim of this work is to use and compare open source libraries that target several platforms and environments. We tried to cover different programming languages and runtime environments and to do so the following implementations have been selected:

- *libcoap* (Bergmann, 2016) is a library written in C and it is designed to fit in a wide range of devices, from embedded devices to big POSIX ones. It supports the official RFC 7252 for the client and server side and it also provides support for several extensions, i.e. Observe mode, Block-Wise transfer and Resource Directory. The source code comes with very complete examples.
- *smcp*[1] (Quattlebaum, 2016) is another C library. It aims to be implemented in devices from bare-metal sensors to Linux-based devices, including embedded ones. It supports client and server sides following the RFC 7252 and it is possible to use it with BSD sockets or µIP. As for CoAP extensions, it supports the Observe mode and Multicast groups. It provides a command line client called smcpctl.
- *microcoap* (1248, 2016) is a limited CoAP library in C that targets small microcontrollers. The source code provides examples for POSIX and Arduino. It follows the RFC 7252 but it does not support it entirely. It supports only the server side and has limited features. It does not support DELETE requests, only GET, PUT and POST, the ACKs can only be piggybacked and it does not support retries.
- *FreeCoAP* (Cullen, 2016) is the last C library analyzed in this paper and it targets GNU/Linux devices. It follows the stable specification RFC 7252.
- *Californium* (The Eclipse Foundation, 2014) is a very complete Java library for not so constrained devices. It targets backends with JVM and it offers both client and server sides. In addition to the RFC 7252, it also supports some extensions, i.e. Observe mode, Block-Wise transfer and Resource Directory.
- *h5.coap* (Walukiewicz, 2014) is a JavaScript library that targets the Node.js platform. It provides only the client side and it follows the stable definition of the RFC 7252. In addition, it supports the Observe mode and the Block-Wise transfer.
- *node-coap* (Collina, 2016) is another JavaScript library for Node.js. It provides both client and server sides and follows the stable version of the RFC. It supports not only the core but also the Observe mode and Block-Wise transfer extension.
- *CoAPthon* (Tanganelli, 2016) is a RFC 7252 compliant Python library that supports both client and server sides. In addition to the core features, it also supports Observe mode, Core-Link format, Multicast and Block-Wise transfer extensions.
- *CoAPy* (CoAPy, 2010) is another Python library. It follows an old draft of CoAP (draft-ietf-core-coap-02) making it theoretically not compatible with the others. In addition to CoAP's specification, it also supports Block-Wise transfer.

Despite the list above that covers different targets and languages, there are other implementations worth mentioning that have been discarded because the working environments for this paper is going to be a Raspberry Pi, hence, they do not fit. *Copper* (Kovatsch, 2014) is a visual client implemented as a Firefox plug-in. *Erbium* (ETH Zurich, 2014) is a widely used C implementation, targeted towards ContikiOS and *TinyCoAP* (Ludovici, 2013) targets tinyOS. As in this paper the working environment is going to be Raspberry Pi, they do not fit.

## 3.1. Security in CoAP: DTLS

As UDP does not guarantee message delivery nor ordered arrival, TLS (T. Dierks, 2008) can not be used to add security to CoAP. Instead, the IETF proposes DTLS (Rescorla & Modadugu, 2012), which is hugely based on TLS but adds some features to overcome the issues related to delivery failure or

[1] Since this work has been carried out, smcp has evolved to go beyond CoAP. A spin off was created on March 2017 to focus just on the CoAP part of the implementation, named libnyoci, https://github.com/darconeous/libnyoci.

unordered message arrival. DTLS securizes the messages, but not the routing information. For example, in a mesh network, the nodes need to know the intended receiver in order to route the datagram through the network. Denial of Service attacks, resource consumption and similar security issues are out of the scope of this section as they are related to other levels of the TCP/IP stack.

The DTLS specification defines 4 working modes, depending on the message encryption:

- NoSec: Criptography not enabled.
- PreSharedKey: Symmetric cryptography.
- RawPublicKey: Asymmetric cryptography.
- Certificate: Asymmetric cryptography with keys provided through certificates.

Symmetric and asymmetric cryptography differ in the keys used to cipher the messages. The former uses the same key both for ciphering and deciphering, meaning that a secret common key needs to be held in both the communication endpoints. The later uses a pair of keys, a public and a private one, being each of them only able to decode a message ciphered with the opposite. The main disadvantage of the symmetric approach is key management as it requires to be distributed to both devices, while the coding speed is its main advantage. On the other hand, the asymmetric approach is easy to distribute as the secret private key should never be shared and the public key can be freely distributed without compromising security. It is important to point out that as one of the keys is of public domain, messages ciphered with the private key do not prevent other listeners to decode it, but instead offers a way of authenticating the sender of a message as it is only owned by one endpoint. This means that messages may be ciphered twice, once with the desired receiver's public key to ensure its privacy and another with the sender's private key for authentication purposes.

To overcome the high ciphering resource consumption that asymmetric cryptography requires, usually it is only used for the initial process named handshake, where a symmetric temporal key is negotiated between both endpoints. This approach benefits from the lighter symmetric key encryption while its main drawback is overcome as the generated key is temporal and different for each session.

In the handshake process, there are three authentication options: no authentication; one way authentication, where only one party of the communication is authenticated; and mutual or two way authentication, where both client and server are authenticated. Each of them follows a different handshake message exchange. Figure 2 shows the three handshake processes.

From the modes specified by DTLS, CoAP requires NoSec to be implemented as messages do not always need to be ciphered. RFC 7252 also states that when using DTLS in *PreSharedKey* mode, the default and hence mandatory suite to support, is *TLS_PSK_WITH_AES_128_CCM_8*, while the others are optional. When asymmetric modes are supported, *TLS_ECDHE_ECDSA_WITH_AES_128_CCM_8* is the mandatory cipher suite. Both *RawPublicKey* and *Certificate* modes are set up using mutual authentication. All cipher suites are registered in the IANA registry[2].

There are several mathematical options to generate asymmetric key pairs. For resource constrained devices, elliptic curve algorithms are more optimal as they use shorter keys for the same level of security. The calculations are more complex, implying higher resource consumption when coding and decoding, but it is made up for as shorter keys are transmitted, which is much less consuming.

[2] http://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml

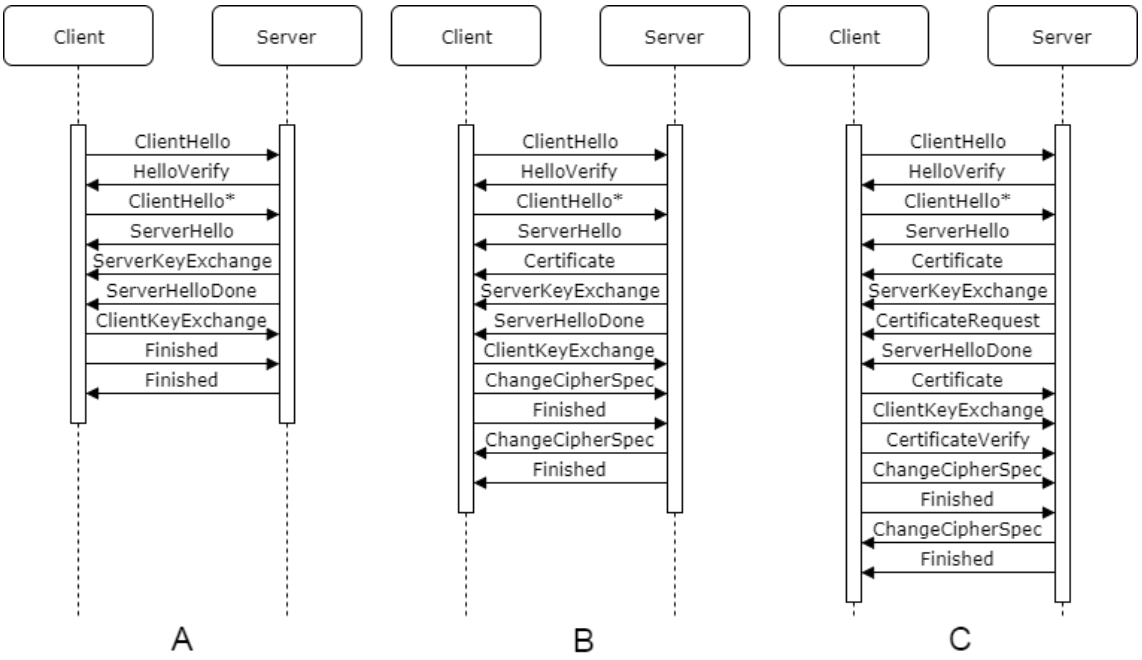



**Figure 2: DTLS handshake with no authentication (a), server-side only (b) and mutual authentication (c)**

# 4. Feature Comparison

Once the different implementations overview have been presented, the next step is to compare them in order to offer a guideline for the library selection based on their features. TThe first step is to compare the libraries based on their characteristics (language, target, extensions, etc.) and then to describe how the library integration process is for new implementations.

In Table 4 specifies the used version of each one of the analyzed libraries, followed by the programming language in which they are written and the target platforms. Four of them are written in C and one in Java, while the four left are evenly distributed between Python and Javascript. The target platforms vary for different libraries, but a Raspberry Pi running GNU/Linux can run all of them.

| Library | Version | Language | Target platform |
|---|---|---|---|
| **libcoap** | Develop (Sept. 24, 2016) | C | POSIX, Contiki, lwIP, TinyOS |
| **smcp** | Master (Sept.24, 2016) | C | Embedded devices, bare-metal sensors, Linux-based devices |
| **microcoap** | Master (Sept.24, 2016) | C | Arduino, POSIX |
| **FreeCoAP** | Master (Sept.24, 2016) | C | GNU/Linux |
| **Californium** | 1.1.0-SNAPSHOT (Sept.24, 2016) | Java | JVM supporting devices |
| **h5.coap** | 0.0.0 (Sept.24, 2016) | JavaScript | Node.JS supporting devices |
| **node-coap** | 0.18.0 (Sept.24, 2016) | JavaScript | Node.JS supporting devices |
| **CoAPthon** | Master (Sept.24, 2016) | Python | Python supporting devices |
| **CoAPy** | 0.0.3-DEV (Sept.24, 2016) | Python | Python supporting devices |

**Table 4: Features – Part 1**

Table 5 further summarizes the main features by defining the specification of each library, whether if it implements client and server sides, the supported extensions and the security libraries for upcoming

DTLS support. CoAPy is based on an early draft (draft-02) and has not been active recently. The rest claim to comply with RFC 7252, although microcoap does that just partially. Regarding client and server implementations, they all support both, except for microcoap (server side only) and h5.coap (client side only). Observe mode and Block-Wise transfer are implemented in most of the libraries, and the most mature ones also support some of the other extensions. CoRE link format, despite not explicitly being mentioned in the description of the libraries, is supported by most of them.

| Library | Specification | Client | Server | Extensions | | | | Security |
|---|---|---|---|---|---|---|---|---|
| | | | | Observe mode | Block-Wise transfer | Resource Directory | Multicast | |
| **libcoap** | RFC 7252 | ✓ | ✓ | ✓ | ✓ | ✓ | | tinydtls, GnuTLS, OpenSSL |
| **smcp** | RFC 7252 | ✓ | ✓ | ✓ | | | ✓ | OpenSSL |
| **microcoap** | RFC 7252 | - | ✓ | | | | | - |
| **FreeCoAP** | RFC 7252 | ✓ | ✓ | | | | | tinydtls, GnuTLS |
| **Californium** | RFC 7252 | ✓ | ✓ | ✓ | ✓ | ✓ | | Scandium |
| **h5.coap** | RFC 7252 | ✓ | - | ✓ | ✓ | | | - |
| **node-coap** | RFC 7252 | ✓ | ✓ | ✓ | ✓ | | | node-mbed-dtls |
| **CoAPthon** | RFC 7252 | ✓ | ✓ | ✓ | ✓ | | ✓ | PyDTLS |
| **CoAPy** | Draft-2 | ✓ | ✓ | | ✓ | | | - |

**Table 5: Features – Part 2**

When considering interoperability, CoAPy uses some options such as deprecated Path-Uri, where code 9 is used. The next draft (draft-03) changed this option to code 11, thus making implementations from the previous drafts non-interoperable. The rest of the libraries claim to use the latest specification and are therefore expected to be interoperable.

Regarding performance, the first four libraries are expected to be faster and more lightweight. This is due to the fact that they are developed in native C instead of a non-native language that requires an additional layer: Java Virtual Machine (JVM) for Californium, Node.js and the V8 JavaScript engine for JavaScript libraries and a Python interpreter for Python ones.

Some of the implementations are more mature than others, this has made that few of them have evolved not only to offer basic functionalities but also to provide advanced mechanisms to deal with resources, available request types and response codes. Due to this, the creation and management of resources varies between implementations and this affects considerably the development process of applications, as it is described in the following lines:

- *libcoap*: it has an interface which makes adding new resources very easy. The library itself manages the response codes, so the developers only need to add the name of the resource, which request types it supports and link each kind of request to a handler.
- *smcp*: similar to libcoap, the developers only need to create the handlers and resources and add them to the system with the help of an interface. The library handles the response codes.
- *microcoap*: to add new resources to the server, they have to be defined and added to a resource array along with the handlers. The library manages everything else by itself.
- *FreeCoAP*: the response codes have to be defined by the developers and the resources' path too, but this implementation does not include an interface to ease the handling of resources. Adding new resources and handling the response codes or actions is a bit tricky.
- *Californium*: this is a very mature implementation and it makes it easy to add and manage resources. To add a new resource, a Java class needs to be created, with the handlers for the different types of supported requests. Then, through an interface, the resources are easily added to the server and the library itself handles the rest.

- *node-coap*: the handling of resources and response codes is on the developers' hands. This implementation does not provide an interface to ease the creation of resources and management of the response codes, so the handling of resources and request has to be made by the application itself, not the library.
- *CoAPthon*: a Python class has to be created for each resource, with the methods that the application supports. The library manages by itself the response codes and everything else.
- *CoAPy*: a Python class has to be created for each resource, but the application needs to handle the response codes, the library does not provide this feature.

To summarize, libcoap, smcp, microcoap, Californium and CoAPthon are the easiest libraries to build a server, because they all handle the response codes within the library itself. The developers just need to define the resources and the handlers and link then to the server, the library handles everything else. The rest of libraries require the developer to handle this explicitly adding unneeded complexity to the application.

# 5. Security Comparison

Previously analyzed libraries use different DTLS implementations as shown in Table 5 from Section 4, for a total of six different DTLS libraries: tinydtls, GnuTLS, OpenSSL, Scandium, PyDTLS and node-mbed-dtls. The latter two are Python and JavaScript wrappers around OpenSSL and mbed TLS respectively. The features of the five remaining implementations, as one of the wrapped libraries overlaps with the previous four, are compared in Table 6. First, the evaluated version is specified, followed by the language the library is implemented on and the intended target device type. Finally, the last column divides the libraries in general security libraries that implement both TLS and DTLS and those specifically developed to support just DTLS. The supported DTLS version has not been included as all of them implement the latest, DTLS 1.2.

| Library | Version | Language | Target | DTLS/TLS+DTLS |
|---|---|---|---|---|
| **OpenSSL** | 1.1.0f | C, assembly | UNIX and Windows | TLS+DTLS |
| **GnuTLS** | 3.6.0 | C | UNIX and Windows | TLS+DTLS |
| **mbed TLS** | 2.6.0 | C | Embedded devices; ports for ARM, PowerPC, MIPS, Motorola 68000, x86, x64 | TLS+DTLS |
| **tinydtls** | 0.9 | C | Embedded devices | DTLS |
| **Scandium** | Part of Californium Framework 1.0.0 | Java | JVM supporting devices | DTLS |

**Table 6: DTLS libraries' features**

OpenSSL (OpenSSL Software Fundation, 2016) is one of the most mature and complete cryptographic libraries including all versions of TLS and DTLS and certificate handling among other features. Some of the codebase is implemented directly in assembly what makes it, in theory, one of the fastest libraries. Conceived to be installed in desktop computers, the devices' requirements are not optimized for more resource constrained devices.

The license used by OpenSSL was not compatible with GPL projects, what was solved by GnuTLS (GnuTLS, 2017), a library that shares most of OpenSSL features. GnuTLS is not as optimized as OpenSSL so it is expected to be slightly slower, but it also supports all TLS and DTLS versions, certificates and is also targeted towards desktop devices.

mbed TLS (ARM, 2016) is the name that ARM gave to PolarSSL when they bought the company. As expected from a library property of ARM, its footprint is smaller than the previous two, despite offering similar features including all TLS and DTLS versions and certificate handling. They also focus on developers by offering an easy to use API but it is not as optimized in terms of speed.

Under the Eclipse Foundation's umbrella, two completely different DTLS implementations have been developed: tinydtls (The Eclipse Foundation, 2017) and Scandium (The Eclipse Foundation, 2017). The former was specifically developed to target resource constrained devices only implementing DTLS basic features in order to minimize the footprint. The latter is a pure Java library that implements DTLS 1.2 developed to be part of Californium framework (The Eclipse Foundation, 2014), so it supports all CoAP requirements.

There are also other DTLS libraries targeting resource constrained devices that have not been included in this survey, as they are not being used by the CoAP libraries. These are WolfSSL (formerly CyaSSL) (WolfSSL, 2017), Lithe (Raza, et al., 2013) or tinyECC (Cyber Defense Laboratory, 2011).

# 6. Experiment Setup

Current industry solutions rely on wired networks such as Profibus, Modbus, CAN, Profinet, to cite some. These are not very flexible and modifying their set-ups requires wiring changes; so as wireless networks are becoming more reliable, their acceptance as a valid alternative is growing. The present experiment has been deployed on an industrial prototype over Raspberry Pi, which is a widely used platform for gateways and Industry 4.0 scenarios. Selecting the Raspberry Pi enables the opportunity of testing a wider pool of implementations than those available for constrained devices. In this case, as both client and server were needed, two Raspberry Pi 3 model B were connected via WiFi through a local 56 Mbps router as can be seen in Figure 3.

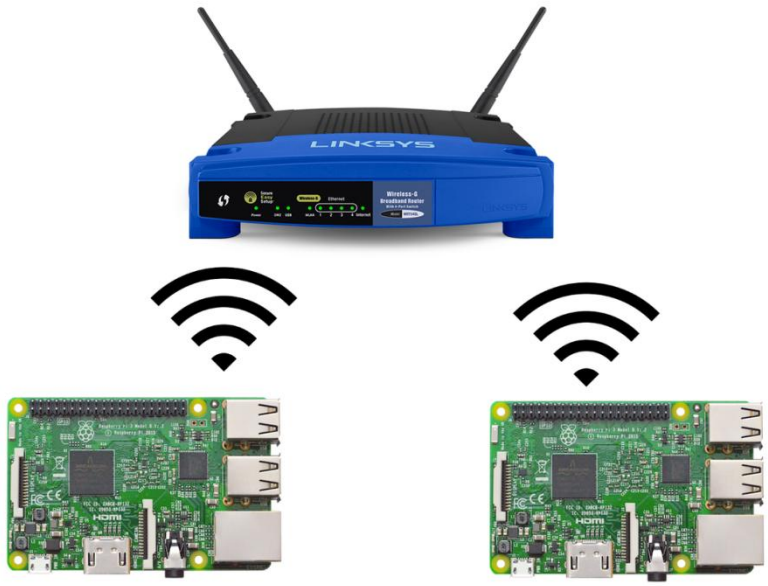



**Figure 3: Experiment setup**

The listed libraries in Table 5 were tested unmodified (except for FreeCoAP, which was ported to IPv4) both in terms of interoperability and performance. All implementations were tested against each other except h5.coap server and microcoap client as they are not implemented. The requests have a single byte payload, while the responses' are alternatively 7 and 8 bytes long. However, it is important to note that at the time that the tests were conducted it was discovered that CoAPthon did not allow adding any payload to 2.04 Changed responses (Github, 2016), thus it sent less bytes in the response, saving time and resources. Regarding the metrics, memory usage, CPU consumption and latency was measured. For ROM usage, both the executable and library files' sizes were taken into account. RAM and CPU consumption was analyzed with the GNU/Linux time tool, which shows statistics of an execution. Round Trip Time (RTT) was selected to measure latency with *tcpdump* network sniffer on the client. 50 requests were sent for each combination of client and server, with a single second waiting interval between send requests.

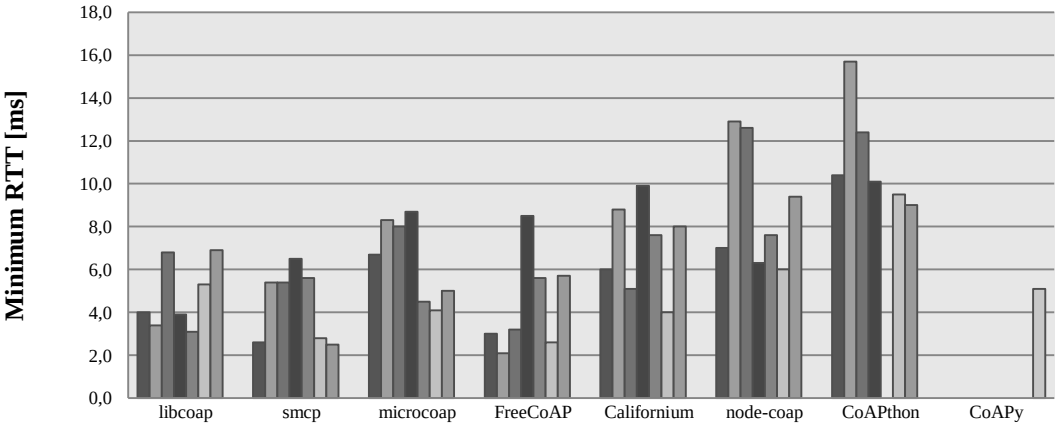
# 7. Results

After describing the experiment scenario, this section presents the results. Table 7 shows the outcome of the interoperability test between different implementations, where PUT requests were sent to the server in order to observe its response. As expected from Section 4, all but CoAPy are interoperable. This test was carried out without a network packet analyzer, but when adding one for the performance test, an issue between h5.coap client and CoAPthon server was found that was not obvious on the interoperability test. CoAP messages use two kind of identifiers, message identifiers that allow to pair a message with its acknowledge and tokens for a more generic purpose, that may be empty. CoAPthon server was generating a token when the client sent an empty one, contrary to RFC 7252, and h5.coap was checking the tokens to match in order to accept the acknowledgment, so they were being rejected and the message was sent again until the maximum allowed number of retries. The issue was reported and later corrected in the CoAPthon's repository (Github, 2017).

| Client \ Server | libcoap | smcp | microcoap | FreeCoAP | Californium | Node-coap | CoAPthon | CoAPy |
|---|---|---|---|---|---|---|---|---|
| **libcoap** | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | |
| **smcp** | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | |
| **FreeCoAP** | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | |
| **Californium** | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | |
| **h5.coap** | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | |
| **node-coap** | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | |
| **CoAPthon** | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | |
| **CoAPy** | | | | | | | | ✓ |

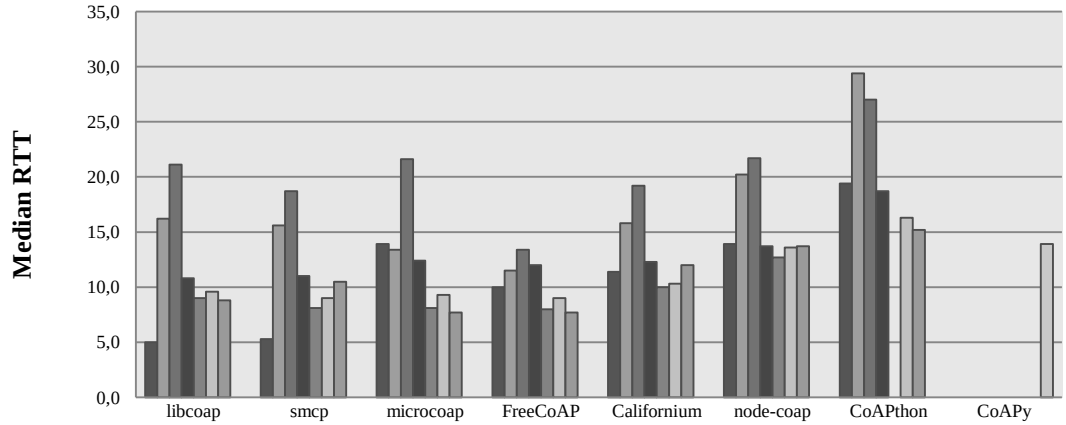
**Table 7: Interoperability results**

Regarding the latency, Figure 4, Figure 5 and Figure 6 show the minimum, median and the maximum values respectively, to represent best, normal and worst case scenarios for every server-client combination. The horizontal axis contains the different server implementations and the vertical bars represent the round-trip time for each of the clients in milliseconds, which have also been tabulated below the graph rounded to the first decimal. As expected, libcoap, smcp and microcoap are faster servers, as C is a lower level language not requiring additional abstraction layers and thus being more optimized. Java (Californium), Node.js (h5.coap and node-coap) and Python (CoAPthon) implementations have given surprisingly good results as clients even in comparison to most of the C ones, being libcoap (C) the fastest library for most cases.



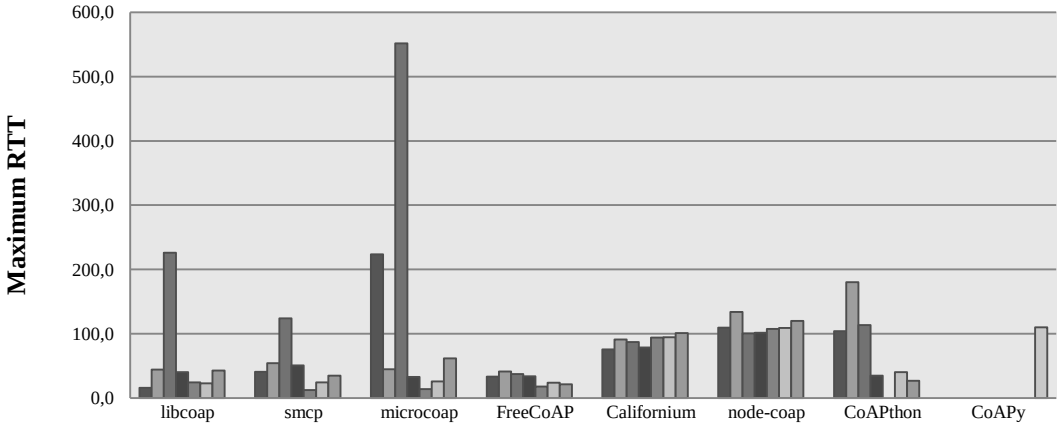

| | libcoap | smcp | microcoap | FreeCoAP | Californium | node-coap | CoAPthon | CoAPy |
|---|---|---|---|---|---|---|---|---|
| libcoap | 4,0 | 2,6 | 6,7 | 3,0 | 6,0 | 7,0 | 10,4 | |
| smcp | 3,4 | 5,4 | 8,3 | 2,1 | 8,8 | 12,9 | 15,7 | |
| FreeCoAP | 6,8 | 5,4 | 8,0 | 3,2 | 5,1 | 12,6 | 12,4 | |
| Californium | 3,9 | 6,5 | 8,7 | 8,5 | 9,9 | 6,3 | 10,1 | |
| h5.coap | 3,1 | 5,6 | 4,5 | 5,6 | 7,6 | 7,6 | | |
| node-coap | 5,3 | 2,8 | 4,1 | 2,6 | 4,0 | 6,0 | 9,5 | |
| CoAPthon | 6,9 | 2,5 | 5,0 | 5,7 | 8,0 | 9,4 | 9,0 | |
| CoAPy | | | | | | | | 5,1 |

**Figure 4: Minimum RTT in ms**




| | libcoap | smcp | microcoap | FreeCoAP | Californium | node-coap | CoAPthon | CoAPy |
|---|---|---|---|---|---|---|---|---|
| libcoap | 5,0 | 5,3 | 13,9 | 10,0 | 11,4 | 13,9 | 19,4 | |
| smcp | 16,2 | 15,6 | 13,4 | 11,5 | 15,8 | 20,2 | 29,4 | |
| FreeCoAP | 21,1 | 18,7 | 21,6 | 13,4 | 19,2 | 21,7 | 27,0 | |
| Californium | 10,8 | 11,0 | 12,4 | 12,0 | 12,3 | 13,7 | 18,7 | |
| h5.coap | 9,0 | 8,1 | 8,1 | 8,0 | 10,0 | 12,7 | | |
| node-coap | 9,6 | 9,0 | 9,3 | 9,0 | 10,3 | 13,6 | 16,3 | |
| CoAPthon | 8,8 | 10,5 | 7,7 | 7,7 | 12,0 | 13,7 | 15,2 | |
| CoAPy | | | | | | | | 13,9 |

**Figure 5: Median RTT in ms**




| | libcoap | smcp | microcoap | FreeCoAP | Californium | node-coap | CoAPthon | CoAPy |
|---|---|---|---|---|---|---|---|---|
| libcoap | 16,1 | 40,8 | 223,4 | 33,5 | 75,6 | 109,6 | 104,3 | |
| smcp | 44,5 | 54,5 | 45,1 | 41,3 | 91,2 | 134,1 | 180,3 | |
| FreeCoAP | 226,3 | 124,1 | 551,6 | 37,6 | 87,1 | 100,8 | 113,8 | |
| Californium | 40,2 | 51,0 | 32,8 | 33,8 | 79,0 | 101,8 | 35,0 | |
| h5.coap | 24,3 | 12,7 | 14,0 | 18,1 | 94,4 | 107,5 | | |
| node-coap | 22,9 | 24,4 | 25,8 | 24,1 | 94,6 | 109,0 | 40,4 | |
| CoAPthon | 42,9 | 35,0 | 61,6 | 21,7 | 101,0 | 120,0 | 26,9 | |
| CoAPy | | | | | | | | 110,3 |

**Figure 6: Maximum RTT in ms**

Considering the system resource consumption, Table 8 shows the ROM usage in bytes. Some of the studied libraries (libcoap, smcp, h5.coap, node-coap, CoAPthon and CoAPy) need to be installed, showing either the *.a* files' or *lib* folder sizes in the library row; while the rest (microcoap, FreeCoAP and Californium) include all the dependencies in the executable. In the Server and Client rows, the sizes of the respective executables are shown, corresponding to the examples that have been used for the previous analysis. C implementations have bigger executable and library sources than Python or JavaScript, while Java's executables are considerably heavier. The size of I/O libraries has not been taken into account except for Californium and using non native languages require adding a runtime environment (Node.js), an interpreter (Python) or a Virtual Machine (Java), which is heavier than the size difference.

| | libcoap | smcp | microcoap | FreeCoAP | Californium |
|---|---|---|---|---|---|
| **Library** | 296150 | 383366 | - | - | - |
| **Server** | 21812 | 18356 | 18700 | 39772 | 4257012 |
| **Client** | 33328 | 22684 | - | 31616 | 4257329 |

| | h5.coap | node-coap | CoAPthon | CoAPy |
|---|---|---|---|---|
| **Library** | 106097 | 47341 | 277095 | 76074 |
| **Server** | - | 1329 | 2508 | 4563 |
| **Client** | 3621 | 883 | 1672 | 1794 |

**Table 8: ROM usage in bytes**

Table 9 shows the CPU and RAM usage of a server (left column) and a client (right) execution for 1000 requests. The data has been collected using the Gnu/Linux tool *time*, executing */usr/bin/time -v "server/client execution command"*. This command shows the peak usage of the RAM (KB) and the total CPU time (seconds) the application has used, both in user and system space. The results show that in execution time C implementations are the fastest and most lightweight. All four tested C implementations are similar in terms of RAM consumption with about 3.3 MB, while Californium and both Node.js implementations are around 10 times heavier. Python implementations are more lightweight but are still far behind C ones. In terms of CPU usage, especially the User Time, it is also in clear favor of C implementations, which are much faster due to compilation and execution of native code.

| | | libcoap | smcp | microcoap | FreeCoAP | Californium | h5.coap | node-coap | CoAPthon | CoAPy |
|---|---|---|---|---|---|---|---|---|---|---|
| **Server** | User time | 0.09 | 0.04 | 0.13 | 0.06 | 2.30 | - | 2.35 | 6.60 | 0.99 |
| | System Time | 0.13 | 0.08 | 0.09 | 0.13 | 0.27 | - | 0.24 | 1.23 | 0.16 |
| | Peak RAM | 3252 | 3232 | 3236 | 3240 | 24492 | - | 31008 | 14348 | 8332 |
| **Client** | User Time | 0.06 | 0.10 | - | 0.10 | 4.66 | 4.22 | 2.73 | 5.77 | 4.96 |
| | System Time | 0.08 | 0.31 | - | 0.21 | 0.41 | 0.62 | 0.21 | 1.25 | 0.13 |
| | Peak RAM | 3240 | 3312 | - | 3256 | 29676 | 37732 | 34484 | 12924 | 10644 |

**Table 9: CPU and RAM usage in second and Kbytes**

# 8. Discussion

After comparing several CoAP libraries, we confirm that libcoap, smcp, microcoap, FreeCoAP, Californium, node-coap and CoAPthon are interoperable, while CoAPy is not, because it is based on an outdated draft of CoAP's specification.

Regarding server performance, C-based implementations prevail over the rest. Among them, libcoap and smcp are the fastest libraries, while microcoap's and FreeCoAP's memory requirements are one order of magnitude lower. However, microcoap does not include all the specifications of the RFC 7252 and FreeCoAP does not handle response code generation tasks and URIs in a transparent way. At the client side, where disk space requirements are not critical, the Java, Node.js and Python implementations are surprisingly close to libcoap and smcp in terms of speed. Moreover, thanks to the higher abstraction level of their languages, Californium (Java), CoAPthon (Python), h5.coap and node-coap (Node.js) are also recommended for CoAP clients.

From the security perspective, most of the active libraries are under current development to add cryptographic features using different DTLS libraries. OpenSSL, GnuTLS and mbed TLS were not developed for CoAP so they are more mature than the other two and also include TLS support. Additionally, mbed TLS and tinydtls target more resource constrained devices.

From this work, different lines for future work arise. On one hand, the evaluation of the proposed libraries in highly constrained platforms, such as devices based on the ARM Cortex-M architecture, would be valuable. Using some Real Time Operative System or baremetal, this would probably mean to discard Python, Java and Node.js implementations, but if the industry needs to point to smaller devices, it might be very useful.

On the other hand, backbone systems may need to serve thousands of devices. A comparison of CoAP libraries in terms of scalability may be relevant, since the libraries based on Java, Python or Node.js may overcome the performance of C-based libraries in larger scenarios. The selected implementation for a backbone system needs to be able to satisfy the demands of large systems with a huge number of connected devices.

Being security an important issue in industrial scenarios, testing the libraries with the cryptographic features enabled and configured with different cipher suites may help to extend the analysis of the libraries further. When versions including these features are released, comparing the results of securized transactions among them and against non-securized tests should be worthy.

Finally, a comparison between implementations of CoAP and other IoT lightweight protocols, such as MQTT, HTTP, XMPP, AMQP and DDS, may also provide useful insights for their use in resource constrained platforms. Comparing not only the protocols' performance but also concrete

implementations' behavior might give system designers the tools they need to choose both the protocol and the implementation for their applications.

## 9. Conclusion

In this paper we have compared several CoAP implementations' features and behavior. From this comparison we conclude that all the analyzed libraries but CoAPy have their use case. More heavyweight implementations can be used as clients in backbone systems as they are fast and those systems do not have a lack of resources, while more lightweight libraries can be used in more resource constrained devices. In this work we point out the strengths and weaknesses of each of them, but it is up to the system designers to choose the more adequate library for their systems.

## 10. References


1248, 2016. *microcoap.* [Online]
Available at: https://github.com/1248/microcoap

AMQP, 2014. *AMQP.* [Online]
Available at: https://www.amqp.org/

ARM, 2016. *mbed TLS.* [Online]
Available at: https://tls.mbed.org/

Banks, A. & Gupta, R., 2014. *MQTT Version 3.1.1,* : OASIS standard.

Bergmann, O., 2016. *libcoap: C-Implementation of CoAP.* [Online]
Available at: https://libcoap.net/

Bormann, C., 2016. *CoAP Implementations.* [Online]
Available at: http://coap.technology/impls.html

Bormann, C., Betzler, A., Gomez, C. & Demirkol, I., 2015. *CoAP Simple Congestion Control/Advanced.* [Online]
Available at: https://tools.ietf.org/html/draft-ietf-core-cocoa-01

Bormann, C. & Shelby, Z., 2016. *Block-wise transfers in the Constrained Application Protocol.* [Online]
Available at: https://tools.ietf.org/html/rfc7959

Chase, J., 2013. *The Evolution of the Internet of Things.* , Texas Instruments.

Chen, Y. & Kunz, T., 2016. Performance evaluation of IoT protocols under a constrained wireless access network. *2016 International Conference on Selected Topics in Mobile and Wireless Networking, MoWNeT 2016.*

CoAPy, 2010. *CoAPy.* [Online]
Available at: http://coapy.sourceforge.net/

Colitti, W. et al., 2011. Evaluation of constrained application protocol for wireless sensor networks. *18th IEEE Workshop on18th IEEE Workshop on Local & Metropolitan Area Networks (LANMAN), 2011.*

Collina, M., 2016. *node-coap.* [Online]
Available at: https://github.com/mcollina/node-coap

Cullen, K., 2016. *FreeCoAP.* [Online]
Available at: https://github.com/keith-cullen/FreeCoAP

Cyber Defense Laboratory, 2011. *TinyECC: A Configurable Library for Elliptic Curve Cryptography in Wireless Sensor Networks.* [Online] Available at: http://discovery.csc.ncsu.edu/software/TinyECC/

De Caro, N. et al., 2013. Comparison of two lightweight protocols for smartphone-based sensing. *IEEE SCVT 2013 - Proceedings of 20th IEEE Symposium on Communications and Vehicular Technology in the BeNeLux.*

Elmangoush, A. et al., 2015. *Application-derived communication protocol selection in M2M platforms for smart cities.* Paris, 18th International Conference on Intelligence in Next Generation Networks.

ETH Zurich, 2014. *Erbium.* [Online] Available at: http://people.inf.ethz.ch/mkovatsc/erbium.php

ETSI, 2012. *IoT CoAP Plugtests.* [Online] Available at: http://www.etsi.org/plugtests/coap/coap.htm

Evans, D., 2011. *The Internet of things. How the Next Evolution of the Internet Is Changing Everything.* , Cisco.

Fieldings, R. et al., 1999. *HTTP.* [Online] Available at: https://www.ietf.org/rfc/rfc2616.txt

Github, 2016. *PUT response with payload.* [Online] Available at: https://github.com/Tanganelli/CoAPthon/issues/45

Github, 2017. *Server auto-token generation breaks RFC 7252.* [Online] Available at: https://github.com/Tanganelli/CoAPthon/issues/48

GnuTLS, 2017. *The GnuTLS Transport Layer Security Library.* [Online] Available at: http://www.gnutls.org/

Hartke, K., 2015. *Observing Resources in the Constrained Application Protocol (CoAP).* [Online] Available at: https://tools.ietf.org/html/rfc7641

Hilton, S., 2016. *Dyn Analysis Summary Of Friday October 21 Attack.* [Online] Available at: https://dyn.com/blog/dyn-analysis-summary-of-friday-october-21-attack/

IETF, 2017. *IETF CoRE Working Group.* [Online] Available at: https://datatracker.ietf.org/wg/core/charter/

Jeon, J., 2011. *Web Browser as Universal client for IoT.* [Online] Available at: https://es.slideshare.net/hollobit/web-browser-as-universal-client-for-iot

Kothmayr, T. et al., 2012. *A DTLS based end-to-end security architecture for the Internet of Things with two-way authentication.* Clearwater, Proceedings - Conference on Local Computer Networks.

Kothmayr, T. et al., 2013. DTLS based security and two-way authentication for the Internet of Things. *Ad Hoc Networks,* 11(8), pp. 2710-2723.

Kovatsch, M., 2014. *Copper (Cu) CoAP user-agent for Firefox.* [Online] Available at: http://people.inf.ethz.ch/mkovatsc/copper.php

Kruger, C. P. & Hancke, G. P., 2014. Benchmarking Internet of things devices. *Proceedings - 2014 12th IEEE International Conference on Industrial Informatics, INDIN 2014.*

Kuladinithi, K. et al., 2011. Implementation of coap and its application in transport logistics. *Proc. IP+ SN*.

Kwon, H., Park, J. & Kang, N., 2015. Challenges in Deploying CoAP Over DTLS in Resource Constrained Environments. *WISA 2015 Revised Selected Papers of the 16th International Workshop on Information Security Applications.*

Lerche, C., Hartke, K. & Kovatsch, M., 2012. Industry adoption of the Internet of Things: A constrained application protocol survey. *IEEE International Conference on Emerging Technologies and Factory Automation, ETFA.*

Ludovici, A., 2013. *TinyCoAP.* [Online]
Available at: https://github.com/AleLudovici/TinyCoAP

Ludovici, A., Moreno, P. & Calveras, A., 2013. TinyCoAP: A Novel Constrained Application Protocol (CoAP) Implementation for Embedding RESTful Web Services in Wireless Sensor Networks Based on TinyOS. *Journal of Sensor and Actuator Networks.*

Mun, D.-h., Dinh, M. L. & Kwon, Y.-w., 2016. *An Assessment of Internet of Things Protocols for Resource-Constrained Applications.* Atlanta, IEEE 40th Annual Computer Software and Applications Conference (COMPSAC).

Nottingham, M., 2010. *Web Linking.* [Online]
Available at: https://tools.ietf.org/html/rfc5988

Object Management Group, 2016. *DDS.* [Online]
Available at: http://portals.omg.org/dds/

OpenSSL Software Fundation, 2016. *OpenSSL Cryptography and SSL/TLS Toolkit.* [Online]
Available at: https://www.openssl.org/

Quattlebaum, R., 2016. *SMCP - A Full-Featured Emedded CoAP Stack.* [Online]
Available at: https://github.com/darconeous/smcp

Rahman, A. & Dijk, E., 2014. *Group Communication for the Constrained Application Protocol (CoAP).* [Online]
Available at: https://tools.ietf.org/html/rfc7390

Raza, S. et al., 2013. Lithe: Lightweight Secure CoAP for the Internet of Things. *IEEE Sensors Journal*, 13(10), pp. 3711-3720.

Rescorla, E. & Modadugu, N., 2012. *Datagram Transport Layer Security Version 1.2.* [Online]
Available at: https://tools.ietf.org/html/rfc6347

Rouse, M., 2016. *Internet of Things (IoT).* [Online]
Available at: http://internetofthingsagenda.techtarget.com/definition/Internet-of-Things-IoT

Shelby, Z., 2012. *Constrained RESTful Environments (CoRE) Link Format.* [Online]
Available at: https://tools.ietf.org/html/rfc6690

Shelby, Z., Hartke, K. & Bormann, C., 2014. *The Constrained Application Protocol (CoAP).* [Online]
Available at: https://tools.ietf.org/html/rfc7252

Shelby, Z., Koster, M., Bormann, C. & van der Stok, P., 2016. *CoRE Resource Directory.* [Online]
Available at: https://tools.ietf.org/html/draft-ietf-core-resource-directory-11

T. Dierks, E. R., 2008. *The Transport Layer Security (TLS) Protocol Version 1.2.* [Online] Available at: https://tools.ietf.org/html/rfc5246

Talaminos-Barroso, A. et al., 2016. A Machine-to-Machine protocol benchmark for eHealth applications - Use case: Respiratory rehabilitation. *Computer Methods and Programs in Biomedicine.*

Tanganelli, G., 2016. *CoAPthon.* [Online] Available at: https://github.com/Tanganelli/CoAPthon

Thangavel, D. et al., 2014. Performance evaluation of MQTT and CoAP via a common middleware. *IEEE ISSNIP 2014 - 2014 IEEE 9th International Conference on Intelligent Sensors, Sensor Networks and Information Processing, Conference Proceedings.*

The Eclipse Foundation, 2014. *Californium.* [Online] Available at: https://eclipse.org/californium/

The Eclipse Foundation, 2017. *Eclipse tinydtls.* [Online] Available at: https://projects.eclipse.org/projects/iot.tinydtls

The Eclipse Foundation, 2017. *Scandium (Sc) - Security for Californium.* [Online] Available at: https://github.com/eclipse/californium/tree/master/scandium-core

The New Jersey Cybersecurity & Communications Integration Cell , 2016. *Mirai Botnet.* [Online] Available at: https://www.cyber.nj.gov/threat-profiles/botnet-variants/mirai-botnet

Villaverde, B. C. et al., 2012. *Constrained application protocol for low power embedded networks: A survey.* Palermo, Sixth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing.

Walukiewicz, Ł., 2014. *h5.coap.* [Online] Available at: https://github.com/morkai/h5.coap

WolfSSL, 2017. *WolfSSL.* [Online] Available at: https://www.wolfssl.com/wolfSSL/Home.html