Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 109C (2017) 188–195

Procedia Computer Science

www.elsevier.com/locate/procedia

The 8th International Conference on Ambient Systems, Networks and Technologies (ANT 2017)

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

Markel Iglesias-Urkia, Adrián Orive, Aitor Urbieta

IK4-Ikerlan Technology Research Centre, Information and Communication Technologies Area.
P^{o} J.M.Arizmendiarrieta, 2. 20500 Arrasate-Mondragón, Spain

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) stands out among the rest. This is extremely lightweight and capable of running in resource constrained devices and networks. 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. 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, it analyzes their performance in terms of latency, memory and CPU consumption in a real testbed deployed in an industrial scenario.

Peer-review under responsibility of the Conference Program Chairs.

Keywords: Benchmarking, CoAP, IIoT, Internet of Things, IoT, Industrial Internet of Things, Lightweight, Protocols, Survey

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"[1]. According to Evans[2] and Chase[3], by 2020, 50 billion devices will be connected to the IoT while Jeon[4] 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[2]), the amount of interconnected objects grows daily at a fast pace thanks to the communication protocols.

* Corresponding author. Tel.: +34 943 712 400 ; fax: +34 943 796 944.
** ORCID: http://orcid.org/{0000-0001-7708-3252, 0000-0003-2919-5799, 0000-0001-5836-4198}
E-mail address: {miglesias,aorive,aurbieta}@ikerlan.es

Peer-review under responsibility of the Conference Program Chairs.
10.1016/j.procs.2017.05.323

Some common Internet protocols, such as HTTP[5], have been originally used to connect the first IoT devices but new, lighter ones have emerged, to fit the constrained requirements imposed by IoT environments, including CoAP[6], MQTT[7], AMQP[8] and DDS[9] among others. Even though it is one of the newest protocols, CoAP is getting some momentum as it follows the REST paradigm, making the adaptation process from HTTP easy for developers. Besides, it is very light both in terms of device and network requirements.

One of the final goals of the IoT industry is to get to as many different kind of devices as possible. As said 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 existing implementations. The selected one has to support the requirements of the system and it has to correctly balance the required CPU and memory resources and energy consumption.

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 platform, analyzing the response time or latency, as well as the memory and CPU needed to deploy the implementations.

The rest of this paper is organized as follows. First, related work is presented. Next, CoAP is described along with some extensions and implementations. After that, in Section 4, a feature comparison of different implementations is carried out. Following section describes the experiment set-up and the measured parameters. The results are presented in Section 6 and finally conclusions and guidance for future work are offered in Section 7.

2. Related Work

There is some previous work analyzing CoAP's performance in different platforms and its comparison against other IoT protocols. Early on, when CoAP's standardization was not complete, there was some work made analyzing different implementations, like Lerche et al.[10] and Villaverde et al.[11] In the latter, the authors analyze CoAP theoretically short after the first draft of CoAP was presented and analyze some of the implementations from that time. The paper presents a list of available implementations and some conclusions based on other papers' analysis. Lerche et al.[10] summarize the results of the first ETSI CoAP Plugtest[12]. They present the participant implementations and their interoperability but they do not offer a performance analysis of any sort. Both Lerche et al.[10] and Villaverde et al.[11] present a list of available implementations, but since they were written, CoAP's standardization went on and new implementations have shown up, so they are not up to date. To research more current work on CoAP's performance, it is necessary to widen the search parameters to other protocols or focus on a sigle implementation on different hardware.

Focusing on CoAP, Ludovici et al.[13] 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. Kruger et al.[14] present a benchmarking of the same CoAP implementation on different hardware, i.e. Raspberry Pi, BeagleBone and BeagleBone Black. They compare the performance on class 4 and class 10 SD cards and with running, off and uninstalled GUI. The parameters they measure are latency, bandwidth and CPU and memory usage, all in the loopback interface. They also compare the latency on a TelosB mote server with a BeagleBone gateway and a laptop client.

Widening the analyzed protocols scope, CoAP and HTTP have similar structure even though they target different environments. Colitti et al.[15] work on Tmote Sky and Zolertia Z1 motes to measure response time and energy consumption. Kuladinithi et al.[16] present their own implementation called libcoap and port it to Contiki and TinyOS to compare the performance against HTTP. Elmangoush et al.[17] present CoAP, HTTP, MQTT and AMQP, but they only work on the two formers. 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.[18] 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.[19] 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.[20] 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.[21] selected CoAP, MQTT, MQTT-SN, WebSockets and TCP. 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.[22] 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, CoAP's performance has been compared against other protocols, such as HTTP, MQTT and others. 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 not up to date. Besides, it analyzes their interoperability and main features, 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.

3. CoAP

Published as RFC 7252[23] by the IETF CoRE Working Group[24] on June 2014, the Constrained Application Protocol[6] 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. It 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 in 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. As it is implemented over UDP, CoAP does not guarantee the arrival of the packets, and therefore implements two main types of messages that differ in whether they require acknowledgment: Confirmable and Non-confirmable. The other two types are Reset and ACK, which can piggyback data. To securize the connections, CoAP can not use TLS due to the underlying UDP protocol as it requires ordered and guaranteed message delivery. DTLS is UDP's alternative to TLS, but it also pays the cost of losing some of the benefits of using UDP derived from not requiring an open connection.

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

- **Constrained RESTful Environments (CoRE) Link Format**[25]: this extension defines the format for the links that constrained servers use to describe their resources, attributes and relationships between links.
- **Block-Wise Transfers in the Constrained Application Protocol (CoAP)**[26]: this extension allows large payloads to be sent when needed (e.g. firmware updates), avoiding IP fragmentation.
- **CoRE Resource Directory**[27]: the IETF proposes an entity called Resource Directory, which has the information about resources of other CoAP servers, allowing battery saving and making the server discovery easy.
- **Observing Resources in the Constrained Application Protocol (CoAP)**[28]: this extension enables CoAP servers to send push notifications to clients, like other publish/subscribe protocols such as MQTT or XMPP.
- **Group Communication for the Constrained Application Protocol (CoAP)**[29]: this extension explains how CoAP should be used in a multicast environment.
- **CoAP Simple Congestion Control/Advanced CoCoA**[30]: the CoAP specification proposes basic behaviors to avoid network congestion but to add more sophisticated methods, the IETF is working on this 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**[31] 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 and it is designed to easily add DTLS with OpenSSL or tinydtls.
- **smcp**[32] 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 and has an, at this time, experimental DTLS branch.

- **microcoap**[33] 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**[34] is the last C library analyzed in this paper. It targets GNU/Linux devices as it uses GnuTLS for security even if it has a new tinydtls branch. It follows the stable specification RFC 7252.
- **Californium**[35] 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. It has optional DTLS support with the Scandium project.
- **h5.coap**[36] 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**[37] 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**[38] 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**[39] 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 for different reasons. Copper[40] is a visual client implemented as a Firefox plugin. Erbium[41] is a widely used C implementation, targeted towards ContikiOS and TinyCoAP[42] targets tinyOS. As in this paper the working environment is going to be Raspberry Pi, they do not fit.

4. Feature Comparison

Once the different implementations overview has been presented, the next step is to start comparing them in order to offer a guideline for the library selection based on their features. To get this, the 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.

Table 1 summarizes the features of the analyzed libraries. In the first place the used version of each library is listed. The first differentiating characteristic is the programming language in which they are written and the targeted platform, being four of them written in C and one in Java, while the four left are evenly distributed between Python and Node.js (Javascript). All of them are supposed to comply with RFC 7252 except for CoAPy, that is based on a previous draft (draft-02). Regarding client and server implementations, they all support both sides except for microcoap (server side only) and h5.coap (client side only). Most of them also implement the most important extensions: the Observe mode and the Block-Wise transfer. In addition, the most mature ones (i.e. libcoap and Californium) also support the Resource Discovery extension.

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 the Python interpreter for Python ones.

Table 1. Characteristics of the implementations

Library	Version	Language	Target platform	Specification	Client/Server	Extensions*	Notes
libcoap	Develop Sept. 24, 2016	C	POSIX, Contiki, lwIP, TinyOS	RFC 7252	Client & Server	Observe, Block-Wise, Resource directory	–
smcp	Master Sept. 24, 2016	C	Embedded devices, bare-metal sensors, Linux-based devices	RFC 7252	Client & Server	Observe, Multicast	–
microcoap	Master Sept. 24, 2016	C	Arduino, POSIX	RFC 7252	Server	–	Partial Support
FreeCoAP	Master Sept. 24, 2016	C	GNU/Linux	RFC 7252	Client & Server	–	–
Californium	1.1.0-SNAPSHOT	Java	JVM supporting devices	RFC 7252	Client & Server	Observe, Blockwise Resource Directory	–
h5.coap	0.0.0	JavaScript	Node.js supporting devices	RFC 7252	Client	Observe, Block-Wise	–
node-coap	0.18.0	JavaScript	Node,js supporting devices	RFC 7252	Client & Server	Observe, Block-Wise	–
CoAPthon	Master Sept. 24, 2016	Python	Python supporting devices	RFC 7252	Client & Server	Observe, Core-Link Multicast, Block-Wise	–
CoAPy	0.0.3-DEV	Python	Python supporting device	Draft-2	Client & Server	Blockwise	Inactive

* Even if most of them do not explicitly say it, they support Core Link-Format extension.

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 support. 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 with, 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. Experiment Setup

Current industry solutions mainly use wired networks such as Profibus, Modbus, CAN, Profinet, etc. 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-s, 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.

The listed libraries have been tested unmodified (except for FreeCoAP, which has been ported to IPv4) both in terms of interoperability and performance. All implementations have been tested against each other except h5.coap server microcoap client as they are not implemented. The requests have a single byte payload, while the responses' ones are alternatively 7 and 8 bytes long. However, it is important to note that at the time that the tests have been conducted it has been discovered that CoAPthon does not allow to add any payload to 2.04 Changed responses[43], thus it sends less bytes in the response, saving time and resources. Regarding the metrics, memory usage, CPU consumption and latency have been measured. For ROM usage, both the executable and library files' sizes have been taken into account. RAM and CPU consumption has been analyzed with the GNU/Linux time tool, which shows statistics of an execution. Round Trip Time (RTT) has been selected to measure latency with *Tcpdump* network sniffer on the client. 50 requests have been sent for each combination of client and server, with a single second waiting interval between send requests.

6. Results

After describing the experiment scenario, this section presents the results. Table 2 shows the outcome of the interoperability test between different implementations, where PUT requests have been sent to the server in order to observe its response. As expected from Section 4, all but CoAPy are interoperable. This test has been 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 is generating a token when the client sends an empty one, contrary to RFC 7252, and h5.coap is checking the tokens to match in order to accept the acknowledgment, so they are being rejected and the message is sent again until the maximum allowed number of retries.

Table 2. Implementation interoperability

Client \ Server	libcoap	smcp	microcoap	FreeCoAP	Californium	node-coap	CoAPthon	CoAPy
libcoap	✔	✔	✔	✔	✔	✔	✔	–
smcp	✔	✔	✔	✔	✔	✔	✔	–
FreeCoAP	✔	✔	✔	✔	✔	✔	✔	–
Californium	✔	✔	✔	✔	✔	✔	✔	–
h5.coap	✔	✔	✔	✔	✔	✔	✔	–
node-coap	✔	✔	✔	✔	✔	✔	✔	–
CoAPthon	✔	✔	✔	✔	✔	✔	✔	–
CoAPy	–	–	–	–	–	–	–	✔

Regarding the latency, Table 3 shows the median (left column) and maximum (right) values in milliseconds (rounded to the first decimal), to represent normal and worst case scenarios for every server-client combination. 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.

Table 3. Median and Max of RTT in ms

Client \ Server	libcoap		smcp		microcoap		FreeCoAP		Californium		node-coap		CoAPthon		CoAPy	
libcoap	5,0	16,1	5,3	40,8	13,9	223,4	10.0	33,5	11,4	75,6	13,9	109,6	19,4	104,3	–	–
smcp	16,2	44,5	15,6	54,5	13,4	45,1	11,5	41,3	15,8	91,2	20,2	134,1	29,4	180,3	–	–
FreeCoAP	21,1	226,3	18,7	124,1	21,6	551,6	13,4	37,6	19,2	87,1	21,7	100,8	27,0	113,8	–	–
Californium	10,8	40,2	11,0	51,0	12,4	32,8	12,0	33,8	12,3	79.0	13,7	101,8	18,7	35,0	–	–
h5.coap	9.0	24,3	8,1	12,7	8,1	14,0	8.0	18,1	10.0	94,4	12,7	107,5	–	–	–	–
node-coap	9,6	22,9	9,0	24,4	9,3	25,8	9.0	24,1	10,3	94,6	13,6	109,0	16,3	40,4	–	–
CoAPthon	8,8	42,9	10,5	35,0	7,7	61,6	7,7	21,7	12,0	101.0	13,7	120.0	15,2	26,9	–	–
CoAPy	–	–	–	–	–	–	–	–	–	–	–	–	–	–	13,9	110,3

Considering the system resource consumption, Table 4 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.

Table 4. ROM usage

	libcoap	smcp	microcoap	FreeCoAP	Californium	h5.coap	node-coap	CoAPthon	CoAPy
Library	296150	383366	-	-	-	106097	47341	277095	76074
Server	21812	18356	18700	39772	4257012	-	1329	2508	4563
Client	33328	22684	-	31616	4257329	3621	883	1672	1794

Table 5 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 favour of C implementations, which are much faster due to compilation and execution of native code.

Table 5. CPU and RAM usage

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

7. Conclusion

In this paper we compare several CoAP implementations' features and behavior. From this comparison, 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. Regarding server performance, C-based implementations stand out. 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 to 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 recommended for CoAP clients.

From this work, different lines of development 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. On the other hand, a comparison of CoAP libraries in terms of scalability may also be relevant, since the libraries based on Java, Python or Node.js may overcome the performance of C-based libraries in larger scenarios. Finally, a comparison between implementations of CoAP and other IoT lightweight protocols, such as MQTT, AMQP and DDS, may also provide useful insights for their use in resource constrained platforms.

Acknowledgment

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.

References

1. Internet of Things (IoT). http://internetofthingsagenda.techtarget.com/definition/Internet-of-Things-IoT
2. D. Evans. The Internet of Things, How the Next Evolution of the Internet Is Changing Everything. *Cisco White Paper*, 2011.
3. J. Chase. The Evolution of the Internet of Things. *Texas Instruments White Paper*, 2013.
4. J. Jeon. Web Browser as Universal client for IoT. http://www.slideshare.net/hollobit/web-browser-as-universal-client-for-iot, 2011.
5. HTTP. https://www.ietf.org/rfc/rfc2616.txt
6. CoAP. http://coap.technology/
7. MQTT. http://mqtt.org/
8. AMQP. https://www.amqp.org/
9. DDS. http://portals.omg.org/dds/
10. C. Lerche and K. Hartke and M. Kovatsch. Industry adoption of the Internet of Things: A constrained application protocol survey. *IEEE International Conference on Emerging Technologies and Factory Automation, ETFA*, 2012.
11. B.C. Villaverde and D. Pesch and R. De Paz Alberola and S. Fedor and M. Boubekeur. Constrained Application Protocol for Low Power Embedded Networks: A Survey. *Sixth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, 2012
12. IoT CoAP Plugtests http://www.etsi.org/plugtests/coap/coap.htm
13. A. Ludovici and P. Moreno and A. Calveras. 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*, 2013.
14. C.P. Kruger and G.P. Hancke. Benchmarking Internet of things devices. Proceedings. *12th IEEE International Conference on Industrial Informatics, INDIN 2014*, 611-616, 2014.
15. W. Colitti and K. Steenhaut and N. De Caro and V. Buta and V. Dobrota. Evaluation of constrained application protocol for wireless sensor networks. *18th IEEE Workshop on18th IEEE Workshop on Local & Metropolitan Area Networks (LANMAN)*, 1-6, 2011.
16. K. Kuladinithi and O. Bergmann and T. Pötsch and M. Becker and C. Görg. Implementation of coap and its application in transport logistics. *Proc. IP+SN*, 2011.
17. A. Elmangoush and R. Steinke and T. Magedanz and A.A. Corici and A. Bourreau and A. Al-Hezmi. Application-derived communication protocol selection in M2M platforms for smart cities. *International Conference on Intelligence in Next Generation Networks, ICIN*, 2015.
18. N. De Caro and W. Colitti and K. Steenhaut and G. Mangino and G. Reali. 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*, 0-5, 2013.
19. D. Thangavel and X. Ma and A. Valera and H.X. Tan and C.K. Tan. 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*, 21-24, 2014.
20. A. Talaminos-Barroso and M.A. Estudillo-Valderrama and L.M. Roa and J. Reina-Tosina and F. Ortega-Ruiz. A Machine-to-Machine protocol benchmark for eHealth applications - Use case: Respiratory rehabilitation. *Computer Methods and Programs in Biomedicine*, 1-11, 2016.
21. D.-h .Mun and M.L. Dinh and Y.-w. Kwon. An Assessment of Internet of Things Protocols for Resource-Constrained Applications. *IEEE 40th Annual Computer Software and Applications Conference*, 2016.
22. Y. Chen and T. Kunz. Performance evaluation of IoT Protocols under a Constrained Wireless Access Network. *International Conference on Selected Topics in Mobile and Wireless Networking, MoWNeT*, 2016.
23. RFC 7252. https://tools.ietf.org/html/rfc7252
24. IETF CoRE Working Group. https://datatracker.ietf.org/wg/core/charter/
25. Constrained RESTful Environments (CoRE) Link Format. https://tools.ietf.org/html/rfc6690
26. Block-Wise Transfers in the Constrained Application Protocol (CoAP). https://tools.ietf.org/html/rfc7959
27. CoRE Resource Directory draft-ietf-core-resource-directory-08. https://tools.ietf.org/html/draft-ietf-core-resource-directory-08
28. Observing Resources in the Constrained Application Protocol (CoAP). https://tools.ietf.org/html/rfc7641
29. Group Communication for the Constrained Application Protocol (CoAP). https://tools.ietf.org/html/rfc7390
30. CoAP Simple Congestion Control/Advanced draft-bormann-core-cocoa-04. https://tools.ietf.org/html/draft-bormann-core-cocoa-04
31. libcoap. https://libcoap.net/
32. smcp. https://github.com/darconeous/smcp
33. microcoap. https://github.com/1248/microcoap
34. FreeCoAP. https://github.com/keith-cullen/FreeCoAP
35. Californium. https://eclipse.org/californium/
36. h5.coap. https://github.com/morkai/h5.coap
37. node-coap. https://github.com/mcollina/node-coap
38. CoAPthon. https://github.com/Tanganelli/CoAPthon
39. CoAPy. http://coapy.sourceforge.net/
40. Copper. https://github.com/mkovatsc/Copper
41. Erbium. http://people.inf.ethz.ch/mkovatsc/erbium.php
42. TinyCoAP. https://github.com/AleLudovici/TinyCoAP
43. CoAPthon PUT response. https://github.com/Tanganelli/CoAPthon/issues/45