Benchmarking Web Communication Protocols in IoT

Daniel Pinho, up201505302@fe.up,pt, João Ferreira, up201303880@fe.up.pt, and Rúben Torres, up201405612@fe.up.pt

***Abstract*—IoT devices are increasing around the world and gaining presence in our lives, and the communication performance of these devices plays a huge role in their dimensions and reliability. Knowing the importance of this topic, we set out to find the most appropriate communication protocol for low processing power devices. Therefore, we created two environments with Vagrant in order to evaluate the performance of the REST, SOAP, MQTT and AMQP protocols and gathered the running time and processing power regarding them.**

***Index Terms*—Benchmark, IoT, protocol, comparison, web communication**

I. Introduction

DISTRIBUTED systems are an ever-increasingly prevalent form of software nowadays. Our everyday lives rely on systems connected to the Internet, where information is transmitted through all our devices, whether we are at home, at our places of work or in the street.

There have been some advancements in technology over the past few years, making it possible to have relatively high processing power in increasingly small hardware systems. This, combined with the increasing presence of the Internet, led to the genesis of the Internet of Things (IoT), where we have devices of all types (such as appliances, lights, and even infrastructure) interacting and communicating with each other.

Despite these advancements in technology, IoT devices often use hardware that, in comparison to personal computers and smartphones, is less powerful. This fact makes them an interesting subject of study, since these devices encounter limitations that are not commonly taken into account when developing software for other, more commonplace systems.

This paper aims to determine which web communication protocols are better for micro-controllers transmitting and receiving data. In order to do this, the performance of four protocols (SOAP, REST, MQTT and AMQP) was evaluated, taking into account measurements such as execution times and RAM and CPU usage. These results are analyzed and discussed in the next sections.

II. Performance metrics and evaluation methodology

In order to take into action our study of the performance of the selected protocols, we prepared a testing platform, which tries to mimic IoT devices, and defined an evaluation methodology.

A. Testing Platform

The testing platform comprised of two virtual machines running Ubuntu 16.04, which where managed using Vagrant. Vagrant empowered us with the flexibility to create different environments with different processing power to test different communication loads. With it, we created two distinct environments.

In figure 1 we find a diagram of the layout of the first environment. The purpose of the first environment was to evaluate the REST and SOAP protocols, as such one machine corresponded to the server and the other the client.

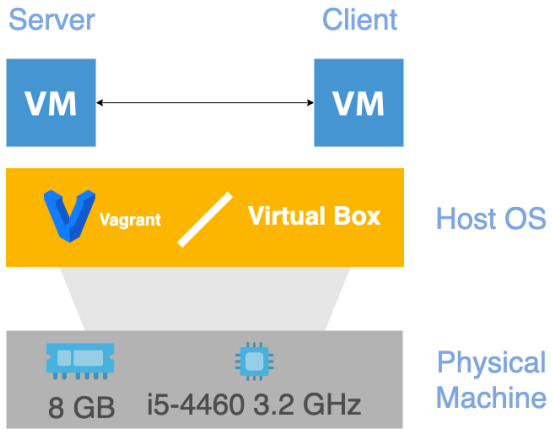

Figure 1. The first environment.

The second environment, as shown in figure 2, was used in order to test the MQTT and AMQP protocols. The publisher and subscriber communicated through the rabbitMQ server running on the host machine.

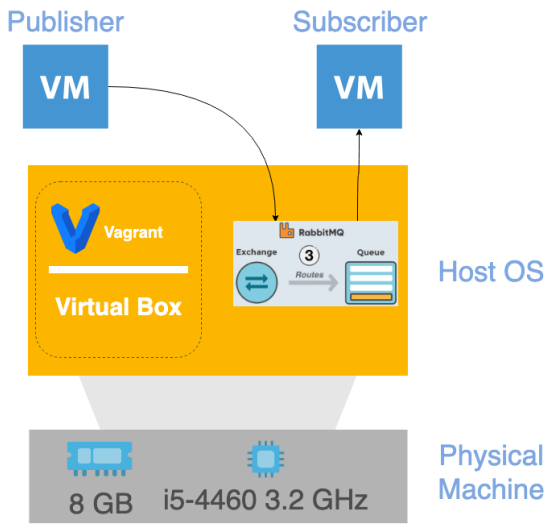

Figure 2. The second environment.

B. Evaluation Methodology

To measure the performance of these web protocols, we took into account the execution time and the run-time usage of CPU and RAM of the process each second for each protocol. Each protocol was tested with several message quantities and sizes. The message sizes varied between 32, 128, 512, and 1024KB, and the number of messages from 128 to 4096 with increments with powers of 2.

In addition to this, we used 3 Vagrant configurations to measure all these tests, which also allowed us to limit the max CPU capacity that each configuration was able to use from the host system. This was vital for us to be able to test the protocols at different capacities.

All configurations had access to 1GB of RAM and 1 CPU core, being the CPU capacity the only varying factor. These configurations had a 15%, 30%, and 50% CPU cap respectively.

CPU cap	Frequency
15%	480MHz
30%	960MHz
50%	1.6GHz

With these tests and performance measures, it's possible to compare and evaluate the web protocols in a fair playing field. However, one can argue that the client-server based protocols have an advantage over the publisher-subscriber ones, since this methodology doesn't explore one of the biggest disadvantages they have in the IoT world; this methodology assumes that there's always information to be retrieved. As a note, the monitorization of CPU and RAM usage each second introduces an associated error, due to the machines having low processing resources and the possibility of the monitorization of the services using some of those resources. However, since each protocol is monitored exactly the same way, the results are able to be reliable and comparable.

All measures of performance were made using a host computer with an (Intel) i5-4460 processor and 8GB of RAM memory. The Vagrant configurations used Ubuntu 16.04 LTS as the operating system and the entire code was executed with the Python 2.7.14 interpreter.

III. Protocols implemented

For this test, four well-known protocols were implemented and evaluated. In this context, we will talk about how the protocols work and their implementation.

A. SOAP

The Simple Object Access Protocol (SOAP) follows a client-server architecture and it is a protocol for information exchange in a decentralized and distributed manner.

It bases its message format in XML and normally uses other protocols for the application layer, most commonly Remote Procedure Call (RPC) and Hypertext Transfer Protocol (HTTP).

This protocol structure consists of three building blocks, an envelope that identifies the XML document as a SOAP message, a header that contains header information, and a body that contains information about the call and the response.

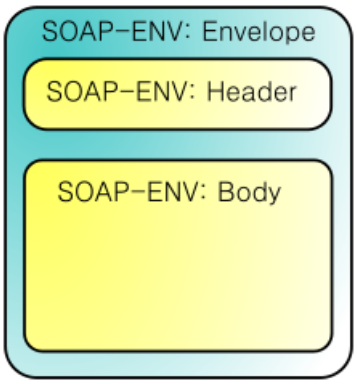

Figure 3. SOAP structure.

This protocol was implemented with the SOAPpy Python Library, using HTTP as the application layer protocol.

B. REST

The Representational State Transfer (REST) follows a client-server architecture and is an abstraction of the web architecture that defines a group of principles/rules/constraints that, when followed, allows the creation of well-defined web services. These provide interoperability between computer systems on the Internet.

The REST web services allow the requesting systems to access and manipulate textual representations of web resources by using a uniform and predefined set of stateless operations, although most Web applications use cookies and sessions to maintain a session state. RESTful systems seek performance, reliability and scaling.

This Protocol was implemented using the Flask python library for the server side and the Requests python library for the client side of the application.

C. MQTT

MQTT stands for Message Queuing Telemetry Transport. It is a simple and lightweight messaging protocol that follows a publisher-subscriber paradigm. It is designed for constrained devices and low-bandwidth, high-latency or unreliable networks.

These design principles exist in order to minimize network bandwidth and device resource requirements whilst also attempting to ensure reliability and some degree of assurance of delivery. IoT communications are in accordance with these principles, making this protocol a good choice for this type of systems.

An MQTT system consists of clients communicating with a server, often called a "broker". A client may be either a publisher of information or a subscriber. This protocol bases its way of working around notifications all around the system, and different elements may not have information on the organization of the network.

This protocol was implemented using the paho-mqtt Python library and rabbitMQ as a broker. This implementation uses a quality of service equal to 2 and a non-persisting connection.

D. AMQP

The Advanced Message Queuing Protocol (AMQP) is a standard used for asynchronous messaging. Its defining features are message orientation, queuing, routing (including point-to-point and publish-and-subscribe), reliability and security.

AMQP mandates the behavior of the messaging provider and client to the extent that implementations from different vendors are interoperable.

This protocol was implemented using the pika Python library and rabbitMQ as a broker.

IV. Results and Analysis

A. Execution time

During the tests that were executed, the running time was measured, and the information that was gathered can be found in the graphs present in figures 4 to 14.

1) 15% cap: In this section, execution times of the 4 different protocols were analyzed while performing the machines capped at 15%. The results where compiled into the graphs present in figures 4, 5 and 6.

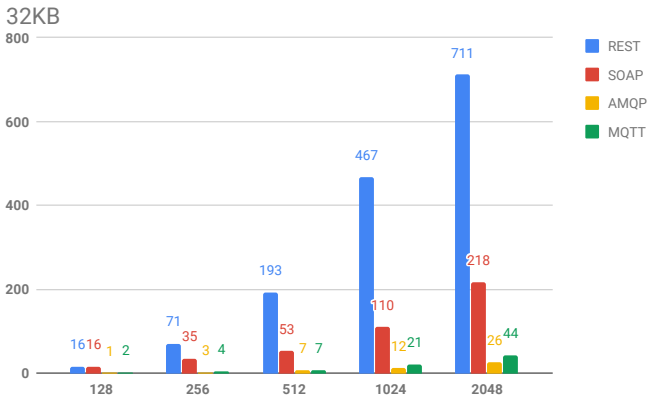

Figure 4. Execution times sending 32KB messages at 15%cap.

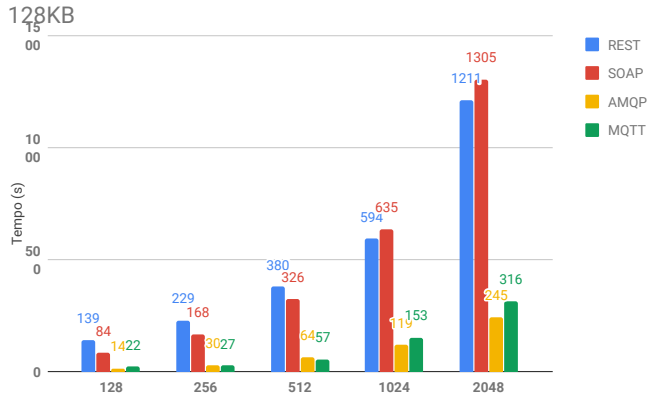

Figure 5. Execution times sending 128KB messages at 15%cap.

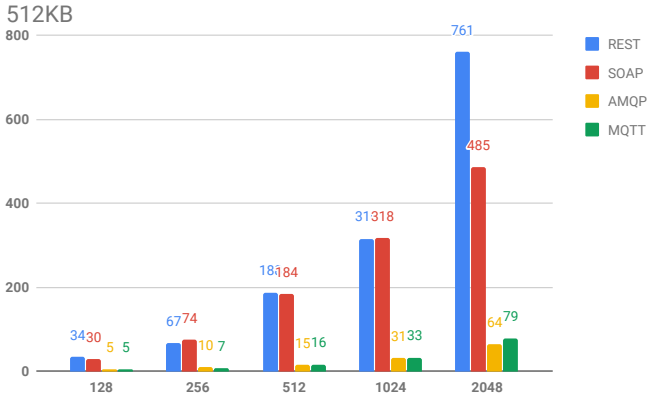

Figure 6. Execution times sending 512KB messages at 15%cap.

Analyzing the graphs, we can see that the execution time is significantly higher in the REST protocol compared to the rest. The performance is also not so great when sending messages using SOAP.

The AMQP protocol is the best one of the four, although its execution times are similar to the ones obtained using MQTT when sending a lower amount of messages.

2) 30% cap: Similarly to the previous section, the graphs for the experiments can be found in figures 7, 8, 9 and 10. It is important to note that, starting with the CPU cap at 30%, there were also experiments made sending messages of 1024KB.

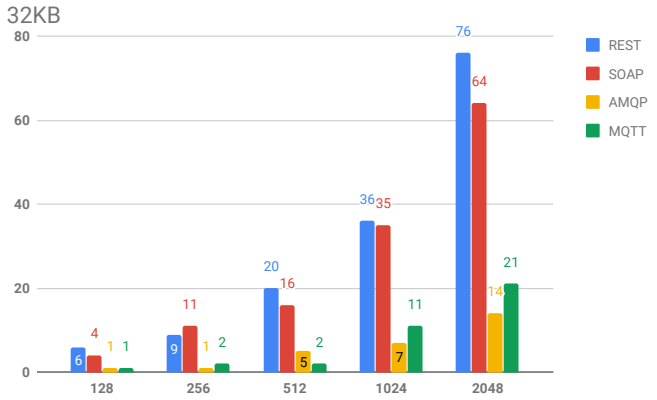

Figure 7. Execution times sending 32KB messages at 30%cap.

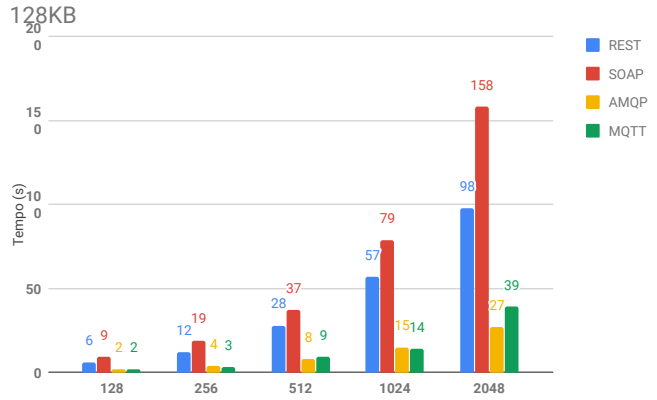

Figure 8. Execution times sending 128KB messages at 30%cap.

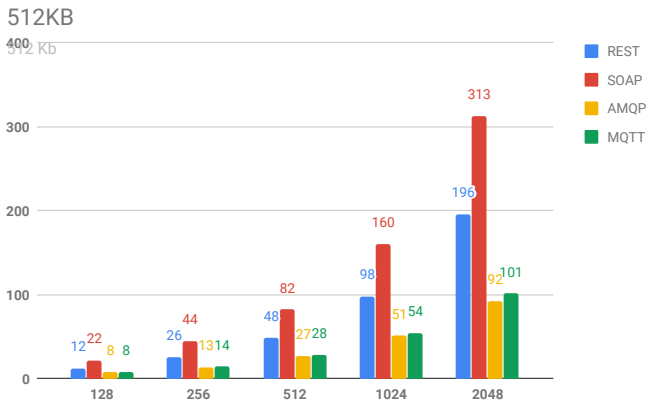

Figure 9. Execution times sending 512KB messages at 30%cap.

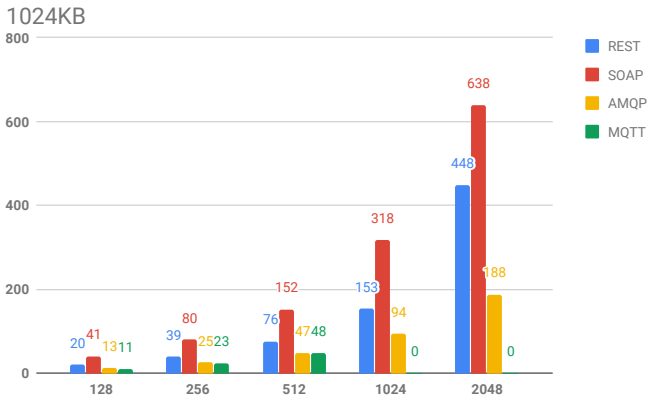

Figure 10. Execution times sending 1024KB messages at 30%cap.

Analyzing the graphs shows us that the scenario does not change much when compared to the previous one, but there are two visible changes: the SOAP protocol ends up being the one that takes the most time (followed by REST), and the MQTT protocol's publisher stops working when sending a large amount of large messages, such as sending 1024 messages of 1024 KB.

3) 50% cap: Akin to previous sections, the graphs for the experiment made capping resources at 50% can be found in images 11, 12, 13 and 14. In addition to the scenarios found with the 30% cap, we also started sending 4096 messages in a test.

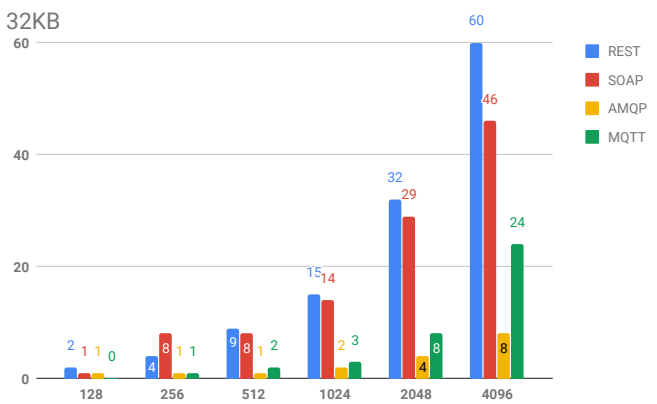

Figure 11. Execution times sending 32KB messages at 50%cap.

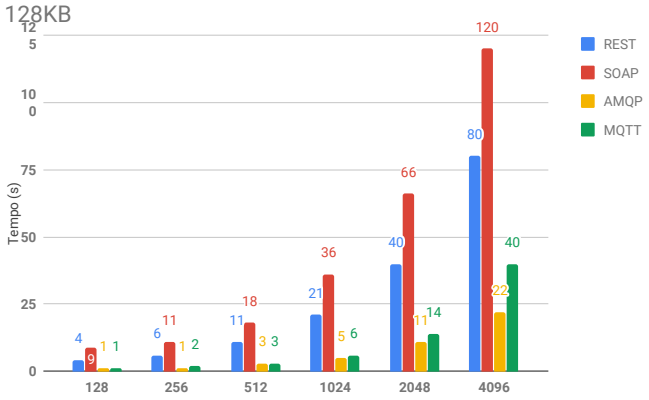

Figure 12. Execution times sending 128KB messages at 50%cap.

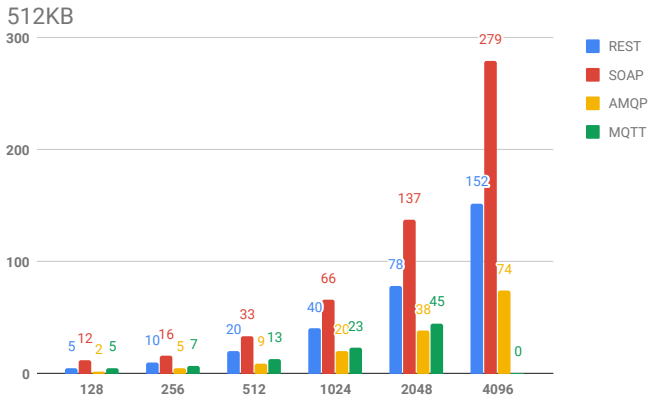

Figure 13. Execution times sending 512KB messages at 50%cap.

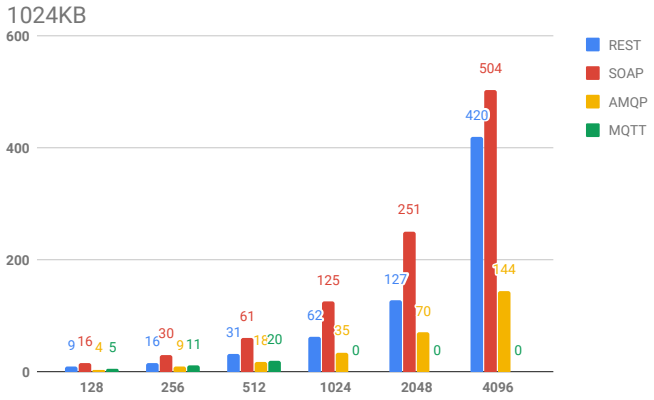

Figure 14. Execution times sending 1024KB messages at 50%cap.

Looking at the graphs, we can conclude that nothing has changed from the previous scenario, with SOAP having the largest running time and the MQTT publisher stopping its work when sending large amounts of data (e.g. 1024 messages of 1MB and 4096 messages of 512 KB).

B. Resource Usage

During the execution of the protocols, the percentage of the CPU and RAM usage was also measured, which can be found in tables I, II, III, IV, V and VI. Data was gathered for both the subscribers and the publisher, which ended up providing with differing results.

Considering the server/publisher, using the REST or the

Table I
SUBSCRIBER RESOURCE USAGE AT 15% CAP

Msg size	Qty.	REST		SOAP		MQTT		AMQP	
		CPU%	RAM%	CPU%	RAM%	CPU%	RAM%	CPU%	RAM%
32	128	53,5	2,0	64,6	1,9	93,5	1,3	94,5	1,6
	256	58,3	2,0	60,4	1,9	77,1	1,3	66,4	1,6
	512	50,7	2,0	59,0	1,9	83,7	1,3	42,0	1,6
	1024	47,3	2,0	49,2	1,9	43,4	1,3	45,1	1,6
	2048	45,4	2,0	53,5	1,9	36,7	1,3	45,4	1,6
128	128	75,0	2,0	62,7	1,8	72,8	1,3	56,5	1,6
	256	58,9	2,0	58,7	1,9	92,9	1,3	61,0	1,6
	512	56,4	2,0	53,9	1,8	71,0	1,3	59,8	1,6
	1024	56,9	2,0	52,0	1,8	71,6	1,3	69,9	1,6
	2048	51,7	2,0	55,8	1,9	69,0	1,3	74,6	1,6
512	128	62,0	2,2	59,5	2,1	39,0	1,5	81,2	1,6
	256	56,3	2,2	74,2	2,1	74,5	1,7	77,6	1,6
	512	57,7	2,2	58,1	2,0	71,4	1,6	72,5	1,6
	1024	59,4	2,3	58,7	2,0	62,4	1,7	75,3	1,7
	2048	61,2	2,3	58,6	2,0	58,5	1,7	75,4	1,6

Table II
PUBLISHER RESOURCE USAGE AT 15% CAP

Msg size	Qty.	REST		SOAP		MQTT		AMQP	
		CPU%	RAM%	CPU%	RAM%	CPU%	RAM%	CPU%	RAM%
32	128	33,9	2,1	49,0	1,8	98,0	1,3	93,6	1,6
	256	20,3	2,1	33,9	1,8	112,6	1,3	103,1	1,6
	512	62,0	2,0	29,9	1,8	87,4	1,4	110,0	1,6
	1024	18,4	2,1	35,1	1,8	91,6	1,5	100,7	1,6
	2048	19,1	2,1	33,9	1,8	87,4	1,7	87,8	1,6
128	128	21,0	2,0	29,5	1,8	84,1	1,5	65,5	1,6
	256	28,8	2,1	30,1	1,8	82,5	1,4	72,7	1,6
	512	24,3	2,1	31,0	1,8	75,9	1,5	79,0	1,6
	1024	23,6	2,1	27,8	1,8	93,5	1,6	69,5	1,6
	2048	24,2	2,1	30,7	1,9	87,5	1,8	70,6	1,6
512	128	77,6	2,0	27,7	2,1	62,1	3,3	65,6	1,7
	256	16,8	2,1	28,3	2,1	63,3	6,9	58,2	1,7
	512	15,7	2,1	26,8	2,1	66,5	11,4	60,0	1,7
	1024	16,3	2,1	26,4	2,1	70,5	6,4	56,7	1,7
	2048	15,9	2,1	26,8	2,1	78,8	6,1	52,0	1,7

Table III
SUBSCRIBER RESOURCE USAGE AT 30% CAP

Msg size	Qty.	REST		SOAP		MQTT		AMQP	
		CPU%	RAM%	CPU%	RAM%	CPU%	RAM%	CPU%	RAM%
32	128	33,9	2,1	49,0	1,8	98,0	1,3	93,6	1,6
	256	20,3	2,1	33,9	1,8	112,6	1,3	103,1	1,6
	512	62,0	2,0	29,9	1,8	87,4	1,4	110,0	1,6
	1024	18,4	2,1	35,1	1,8	91,6	1,5	100,7	1,6
	2048	19,1	2,1	33,9	1,8	87,4	1,7	87,8	1,6
128	128	21,0	2,0	29,5	1,8	84,1	1,5	65,5	1,6
	256	28,8	2,1	30,1	1,8	82,5	1,4	72,7	1,6
	512	24,3	2,1	31,0	1,8	75,9	1,5	79,0	1,6
	1024	23,6	2,1	27,8	1,8	93,5	1,6	69,5	1,6
	2048	24,2	2,1	30,7	1,9	87,5	1,8	70,6	1,6
512	128	77,6	2,0	27,7	2,1	62,1	3,3	65,6	1,7
	256	16,8	2,1	28,3	2,1	63,3	6,9	58,2	1,7
	512	15,7	2,1	26,8	2,1	66,5	11,4	60,0	1,7
	1024	16,3	2,1	26,4	2,1	70,5	6,4	56,7	1,7
	2048	15,9	2,1	26,8	2,1	78,8	6,1	52,0	1,7
1024	128	84,9	2,6	73,7	2,2	75,2	2,2	44,9	1,7
	256	80,2	2,6	71,9	2,2	71,0	2,0	46,0	1,7
	512	84,9	2,5	70,3	2,7	76,3	1,9	41,1	1,6
	1024	80,9	2,6	73,3	2,3	**72,4**	**2,1**	46,8	1,6
	2048	72,7	2,6	71,9	2,2	-	-	40,3	1,7

Table IV
PUBLISHER RESOURCE USAGE AT 30% CAP

Msg size	Qty.	REST		SOAP		MQTT		AMQP	
		CPU%	RAM%	CPU%	RAM%	CPU%	RAM%	CPU%	RAM%
32	128	36,4	2,0	50,2	1,8	137,0	1,3	85,4	1,6
	256	22,0	2,1	21,3	1,8	149,0	1,3	82,5	1,6
	512	13,9	2,1	25,5	1,8	67,3	1,4	88,4	1,6
	1024	20,4	2,1	28,7	1,8	95,2	1,5	88,3	1,6
	2048	18,5	2,1	33,3	1,8	100,5	1,7	89,1	1,6
128	128	14,6	2,1	24,6	1,8	110,0	1,4	69,5	1,6
	256	21,7	2,1	23,1	1,8	78,0	1,5	72,8	1,6
	512	22,1	2,1	25,3	1,8	104,9	1,5	102,5	1,6
	1024	16,5	2,1	27,0	1,9	99,2	1,6	73,5	1,6
	2048	18,2	2,1	30,1	1,9	101,6	1,8	74,3	1,6
512	128	17,4	2,1	19,2	2,0	81,5	2,3	57,2	1,6
	256	28,3	2,0	17,5	2,0	70,8	2,3	78,1	1,7
	512	13,5	2,1	17,6	2,0	69,5	13,1	60,6	1,7
	1024	12,6	2,1	16,7	2,0	78,8	15,2	60,7	1,7
	2048	12,6	2,1	18,6	2,0	91,5	10,4	53,2	1,7
1024	128	13,0	2,1	22,6	2,2	59,9	3,4	61,3	1,8
	256	13,2	2,2	21,7	2,2	63,3	11,0	55,0	1,8
	512	8,6	2,2	22,3	2,2	72,1	21,1	53,1	1,8
	1024	8,6	2,2	20,8	2,3	**74,9**	**24,2**	55,4	1,8
	2048	7,3	2,2	21,2	2,3	-	-	56,3	1,8

Table V
SUBSCRIBER RESOURCE USAGE AT 50% CAP

Msg size	Qty.	REST		SOAP		MQTT		AMQP	
		CPU%	RAM%	CPU%	RAM%	CPU%	RAM%	CPU%	RAM%
32	128	73,5	1,9	111,0	1,8	56,5	1,3	33,4	1,6
	256	52,5	2,0	37,2	1,8	73,0	1,3	41,1	1,6
	512	46,7	2,0	39,8	1,9	35,8	1,3	39,7	1,6
	1024	60,4	2,0	43,1	1,8	46,6	1,3	43,5	1,6
	2048	63,7	2,0	48,4	1,9	29,3	1,3	53,2	1,6
	4096	69,2	2,0	59,0	1,8	19,7	1,3	51,2	1,6
128	128	38,8	2,0	44,9	1,8	90,8	1,3	91,0	1,6
	256	51,9	2,0	50,7	1,8	44,5	1,3	93,0	1,6
	512	61,6	2,0	50,7	1,8	95,8	1,4	101,8	1,6
	1024	61,5	2,0	56,4	1,8	91,8	1,3	96,3	1,6
	2048	68,8	2,0	55,8	1,8	48,2	1,3	91,4	1,6
	4096	69,0	2,0	60,0	1,9	25,6	1,4	76,7	1,6
512	128	76,4	2,1	61,6	2,0	13,1	1,5	98,8	1,7
	256	58,5	2,2	67,6	2,0	31,8	1,5	92,3	1,7
	512	75,3	2,2	75,6	2,0	37,8	1,6	90,0	1,7
	1024	76,1	2,3	78,3	2,0	53,4	1,5	76,6	1,7
	2048	78,4	2,3	73,5	2,0	69,3	1,7	86,9	1,6
	4096	76,7	2,3	62,0	2,0	-	-	81,2	1,7
1024	128	61,8	2,4	72,4	2,3	21,0	2,1	94,0	1,7
	256	66,2	2,5	72,0	2,1	77,7	2,1	88,2	1,7
	512	69,4	2,5	72,8	2,3	68,0	2,1	88,7	1,7
	1024	75,3	2,6	70,7	2,2	**42,7**	**1,7**	85,5	1,7
	2048	76,4	2,6	70,9	2,2	-	-	80,5	1,7
	4096	70,2	2,6	70,9	2,3	-	-	81,5	1,7

SOAP protocols, we can see that the server doesn't really change the amount of resources it uses; they would use up between approximately 15-20% of the CPU using REST and a little more using SOAP. Both protocols would expend around 2% of the available RAM.

The resources are used more intensively in the other two protocols. While RAM usage hovers around 1.6% using the AMQP protocol, this value grows heavily using the MQTT protocol, particularly when big amounts of large messages are sent.

We can see, in tables IV and VI, that the publisher crashes while sending large amounts of data (1024 messages of 1MB or 4096 messages of 512KB; these cases are identified in bold in the tables, since the resource usage value was sometimes

Table VI
PUBLISHER RESOURCE USAGE AT 50% CAP

Msg size	Qty.	REST		SOAP		MQTT		AMQP	
		CPU%	RAM%	CPU%	RAM%	CPU%	RAM%	CPU%	RAM%
32	128	10,8	2,1	13,0	1,8	109,6	1,4	95,7	1,6
	256	15,6	2,1	21,8	1,8	104,3	1,3	93,3	1,6
	512	16,2	2,1	28,1	1,8	111,0	1,4	74,3	1,6
	1024	16,6	2,1	30,9	1,8	108,3	1,5	56,0	1,6
	2048	18,4	2,1	30,5	1,8	101,7	1,7	106,7	1,6
	4096	15,5	2,1	36,1	1,8	94,4	2,1	103,6	1,6
128	128	14,8	2,1	20,1	1,8	99,1	1,3	89,7	1,6
	256	16,8	2,1	29,2	1,8	106,0	1,4	91,0	1,6
	512	17,1	2,1	36,2	1,8	106,0	1,4	92,0	1,6
	1024	17,9	2,1	31,7	1,8	105,7	1,6	55,2	1,6
	2048	18,4	2,1	35,4	1,8	94,6	1,8	82,8	1,6
	4096	17,8	2,1	36,7	1,9	99,4	2,1	80,0	1,6
512	128	23,8	2,1	30,9	2,1	93,0	2,3	72,0	1,7
	256	16,5	2,1	22,4	2,0	73,5	4,4	71,9	1,7
	512	15,9	2,1	22,7	2,1	70,5	2,4	75,7	1,7
	1024	15,1	2,1	22,3	2,0	92,7	2,5	63,4	1,7
	2048	15,1	2,1	21,8	2,1	89,2	3,1	62,6	1,7
	4096	13,7	2,1	23,3	2,1	-	-	60,3	1,7
1024	128	14,1	2,2	20,7	1,9	66,1	5,0	63,0	1,8
	256	12,1	2,2	24,3	2,2	82,2	3,5	48,8	1,8
	512	13,5	2,2	23,6	2,3	77,2	3,6	60,4	1,8
	1024	11,5	2,2	24,1	2,2	**63,9**	**19,6**	56,3	1,8
	2048	10,6	2,2	22,9	2,3	-	-	51,0	1,8
	4096	8,7	2,2	23,4	2,3	-	-	51,9	1,8

recorded). The absence of values after the crash is due the lack of enough RAM memory for the process, consequently being killed by the OS itself.

The results, while having different values, ended up being similar in the subscribers. RAM usage stays about the same when looking at the REST, SOAP and AMQP protocols, and stays low using MQTT, in the order of 1-2%. CPU usage is a lot higher and variable, however.

We can take from here that resource caps don't really change the protocols' resource usage, which is low in the SOAP, REST and AMQP protocols. However, things start grinding to a halt when the publisher has to send large amounts of large messages using MQTT.

RAM availability seems to be the bottleneck present here, and its usage increases exponentially using this latter protocol, resulting in the abrupt termination of the execution of its publisher.

V. DISCUSSION

The results presented in the previous section allow us to have an acceptable overview of the way the four web communication protocols behave.

Taking into account the junction of the running time and the resources each protocol uses, we can group the protocols into two groups. The first one includes SOAP and REST, client-server-based protocols, in which we find longer running times and low efficiency in resource usage. This last remark can be seen in the virtually null variation in resources used even when the resource cap is different. This shows us that there is the possibility of an existing bottleneck somewhere in the architecture or in the implementation.

The second group includes the other two protocols, AMQP and MQTT. They follow a publisher-subscriber architecture, and are characterized by using more of the available resources

and lower running time compared to the other group. Taking this into account, we can assert that these running times are a result of using more resources.

We can ascertain that out of these two groups, the protocols in the second one, since they follow the aforementioned publisher-subscriber architecture, function in a way that is more in line with the reality of IoT systems. Additionally, we can conclude that the AMQP protocol is the most adequate one out of these four to use, given that it does not fail when encountering high loads of data and traffic, unlike MQTT, whose publisher performs acceptably in low-to-medium loads but fails otherwise.

VI. Conclusion and Future Work

From the analysis performed, we can conclude that the AMQP protocol is the most reliable and versatile of the four, taking into account all situations, however, MQTT is a strong contender for more lightweight scenarios. REST and SOAP, even though they had an advantage in the situations tested, they didn't compare to the publisher-subscriber protocols in an IoT or low-processing power environment with high number of messages being exchanged.

Further work could be done in order to measure other performance metrics, including, but not limited to, Data Cache Misses(DCM) and millions of instructions per second (MOP/s). Additionally, all tests executed could be performed in actual IoT devices. For example, Arduino boards and Raspberry Pis.

Appendix

Implementation source code and result

The source code developed in the context of this work and the results obtained can be found in the following GitHub repository:

`https://github.com/vordep/ASSO19_Proj`

References

[1] SOAPpy documentation, URL:https://pypi.org/project/SOAPpy/ (last visited on 2019/06/17)

[2] Requests documentation, URL:https://2.python-requests.org/en/master/ (last visited on 2019/06/17)

[3] Flask documentation, URL:http://flask.pocoo.org/ (last visited on 2019/06/17)

[4] Paho-mqtt documentation, URL:https://pypi.org/project/paho-mqtt/ (last visited on 2019/06/17)

[5] Pika documentation, URL:https://pika.readthedocs.io/en/stable/ (last visited on 2019/06/17)

References

[1] SOAPpy documentation, URL:https://pypi.org/project/SOAPpy/ (last visited on 2019/06/17)

[2] Experiment: How Fast Your Brain Reacts To Stimuli, URL:https://backyardbrains.com/experiments/reactiontime (last visited on 2019/06/16)

[3] Course slides, ETRS-Sched-3-5-1.pdf

[4] Course slides, ETRS-Sched-6-8.pdf