An Experimental Evaluation of Application Layer Protocols for the Internet of Things

Lavinia NĂSTASE[1], Ionuț Eugen SANDU[2], Nirvana POPESCU[1]*

[1] University Politehnica of Bucharest, Computer Science Department, Bucharest, Romania
contact@lavinianastase.com; nirvana.popescu@cs.pub.ro (**Corresponding author*)

[2] National Institute for Research & Development in Informatics, Bucharest, Romania
ionut@rotld.ro

Abstract: The Internet of Things is envisaged to interconnect billions of devices, from sensors and actuators to smart objects, computers and vehicles. The main obstacle that arises is that technology should allow physical objects, usually constrained devices, to interact with applications. Therefore, already existing technologies and patterns should be adapted to the new requirements, or novel ones must be created. In this context, the application layer protocols play an important part in orchestrating an IoT network. This paper focuses on the comparison of different protocols by carrying out experiments and prototyping a real-world interaction between IoT network actors.

Keywords: Internet of Things (IoT), Application layer protocols, CoAP, MQTT, XMPP, AMQP, HTTP, WebSockets.

1. Introduction

As society is rapidly leading to an innovative concept, named the Internet of Things, technology must quickly adapt to new business and user requirements and needs. The IoT comprises smart devices, sensors, networks, cloud solutions, all of them being connected through common standards [1, 6].

A commonly agreed architecture of the IoT comprises three layers: application layer, network layer and perception layer. Since the complexity of such systems is constantly increasing, the necessity of adapting existing protocols, for choosing the appropriate ones and minimizing vulnerabilities grows exponentially.

In this paper, our focus is on the application layer protocols used in the Internet of Things. In [5], we described three of the most used protocols, MQTT, CoAP and XMPP, with emphasis on their characteristics, their suitability for certain applications and the security they provide.

The second section of this paper provides a short overview of related work in researching IoT protocols. The third section illustrates six application layer protocols and aims to emphasize their characteristics in an IoT environment. The fourth section focuses on the experimental setup, while the fifth shows the obtained results and their interpretations. The last section concludes the work in this paper and presents further research possibilities.

2. Related work

The communication protocols for IoT have been a popular research subject in the last years, given the demand of innovation and advance in this field. Several surveys were conducted in terms of application layer protocols, which were also considered in [5], where we extended the presentation by taking into account the security MQTT, XMPP and CoAP can provide. A comprehensive study on IoT protocols, including lower layer protocols, is conducted in [2]; in this survey, the authors document the current status in this research area and summarize security mechanisms for each layer. Other surveys that discuss application layer protocols by comparison are [3] and [7]. In these papers, authors present MQTT, XMPP, CoAP, AMQP, HTTP and WebSockets in the context of IoT. In [4], authors conduct experiments to compare MQTT, WebSocket and CoAP performances on LAN, ISP and cellular network. In a scenario similar to a real network, they compared average RTT and they concluded that changing from LAN to an IoT network does not trigger a fall of a protocol's efficiency.

3. Application layer protocols

In this section, we extend the protocols discussion from [5] and, in addition, we introduce three more protocols, widely used in IoT: AMQP, HTTP RESTful services and WebSockets.

3.1 MQTT

MQTT is a lightweight publish-subscribe messaging protocol, based on brokers. It uses TCP/IP protocol and it is suitable in constrained environments, for devices with low memory resources or a limited processor. Furthermore, the message payload is limited to a maximum of 256

MB of information, which makes this protocol suitable in expensive and unreliable networks.

MQTT, through its publish-subscribe architecture, "decouples a client, who is sending a particular message (called publisher) from another client (or more clients), who is receiving the message (called subscriber)"[1]. The broker publishes one or more topics, to which clients subscribe to get messages. Each topic can have multiple subtopics used for filtering messages from connected clients. For example, in case of a smart home, a topic would be:

"home/firstfloor/bedroom/temperature".

According to the MQTT protocol specification[2], there are three quality of service (QoS) modes provided for message delivery:

- "Fire and forget" mode, also known as "at most once": in this case, a message arrives once or not at all. It is suitable in environments where an individual measurement is not vital, since a next one would be published afterwards.
- "Acknowledged delivery" or "at least once": a message arrives at least once and therefore duplicates can occur.
- "Assured delivery" or "exactly once": this mode is the highest QoS; all messages are ensured to arrive exactly once. It is useful in applications where missing or duplicate messages may lead to unwanted results, for example a payment service[3].

The last two levels are the most reliable ones, nonetheless they impose an overhead and bandwidth requirements that are not feasible in most of IoT environments. However, clients may choose the desired level of QoS.

In order to further adapt MQTT to constrained devices requirements, a new protocol emerged, named MQTT-SN (where SN stands for "sensor networks"). According to [10], published by IBM in 2013, the differences between those two protocols are the following:

1. The topic name in PUBLISH messages are replaced by a two-byte long id, instead of a string.
2. Short topic names are introduced and they do not require registration.
3. Multiple gateways may exist in the networks and they advertise their presence periodically by broadcasting massages to connected devices. Clients which do not have a pre-configured server or gateway address use a discovery procedure to find the network address of a server or gateway.
4. Devices can go in a sleeping mode during the time when their messages are buffered at the gateway/server; they are woken up when messages are delivered. This improvement is particularly useful for battery-operated devices.

To sum up, MQTT-SN's purpose is to be used in a local network. The specification states that it can use a datagram protocol (such as UDP, instead of TCP). Since MQTT usually connects directly to the cloud, it would be inefficient to use UDP in such an error prone scenario. However, for local networks with a small number of clients and constrained devices, it is more suitable to reduce the number and the size of the packets.

As we stated in [5], the MQTT specification does not impose any security mechanism, because it is designed to be used in secure networks. Therefore, it is not feasible to create a globally MQTT network.

Since the IoT requires a standard for authentication, MQTT relies on SSL/TLS encryption. The drawback is that SSL/TLS is quite an expensive protocol to use in memory and power constrained devices. Nowadays, certain brokers accept anonymous clients and therefore the username and password are no longer required, but it is not desirable in most of the cases. The client validates the server certificate during the handshake phase, when it verifies its identity to authenticate it. Client certificates can also be used; the broker can authenticate the client that requests the connection[4]. In an extended network, a poor MQTT application design may be easily prone to harmful messages injection into the network.

For these reasons, security must be implemented on top of MQTT, depending on the specificity of the network. For example, in a secured and isolated network, this is not the case.

3.2 XMPP

XMPP (Extensible Messaging and Presence Protocol) is a client-server protocol mostly used for chat, instant messaging, video and voice calls and is standardized by the IETF[5]. In this protocol, data is exchanged between two or more network entities using "XML stanzas", which are small pieces of XML structured data. Although XML is an appropriate choice in terms of interoperability, it usually creates an overhead in terms of processing. XML tag parsing increases power consumption and computational power and might be unnecessary.

An XMPP client is required to connect to a server to gain access to the network. Therefore, it is the entity that establishes the XML stream, after authenticating through a SASL negotiation. A XMPP server manages open streams with the connected clients.

The process of connection is usually the following:

1. Determine the IP address and the port to which the client will connect.
2. Open a TCP connection and then an XML stream over that connection.
3. Negotiate TLS channel encryption (not mandatory, but preferable).
4. Authenticate via SASL.
5. Bind a resource to the stream.
6. Begin to exchange XML stanzas with other entities in the network.
7. Close the XML stream and the TCP connection.

Server to server connection is also allowed, after negotiating and allowing inter-domain communication.

There are two possible paths, since XMPP based solutions are usually deployed in decentralized client-server architectures: client-to-server stream and server-to-server stream. If both entities are clients, they need an intermediate entity (a server) with a certain level of trust; it is not possible for them to open a communication channel directly between them.

As fas as security is concerned, the XMPP community did not propose so far an end-to-end encryption technology suitable for widespread deployment. The IETF recommends, according to the RFC, support to authentication via SASL and transport security with TLS. SASL provides a number of authentication methods from which the client can choose; the disadvantage is that a weak mechanism can be chosen. SASL uses Base64 encoding, that hides easily recognized information; however, it doesn't provide computational confidentiality. As discussed in [5], IETF recommends secure mechanisms for peer authentication, such as SCRAM-SHA-1 or SCRAM-SHA-1-PLUS, to offer protection against man-in-the-middle-attacks, spoofing and unauthorized access.

An acceptable strategy for security would be to employ a combination of TLS encryption and SASL authentication, to provide both mutual authentication and integrity. Channel encryption is usually based on a PKIX certificate presented by the receiving entity or both the receiving and the initiating entity, for mutual authentication. The signature algorithm should be SHA-256 at minimum.

The vulnerabilities to which unprotected XMPP systems are exposed are various: sniffing and breaking passwords, eavesdropping, replaying, inserting, deleting, modifying stanzas, discovering usernames through directory harvesting attacks, spoofing, gaining unauthorized entry, man-in-the-middle attacks and more. One of the issues in XMPP emerges from the fact that an XML stanza can transit multiple streams and some of them might not be protected with TLS.

To sum up, although XMPP has built-in security feature, it might not be practical for M2M communication, because it does not provide QoS modes [3]. By comparison with MQTT, it has the advantage of being a mature and well-established protocol and its pub/sub architecture leverages it for the IoT.

3.3 CoAP

The third protocol we discussed in [5] is CoAP (Constrained Application Protocol).

CoAP is designed to be interoperable with HTTP, since it uses a subset of HTTP methods (GET, PUT, POST, DELETE) [3]. It is specialized for use with constrained nodes (in terms of memory and processing power) and networks (lossy and low power), by obeying a request/response model between application endpoints, similar to the client/server HTTP model.

However, unlike HTTP, uses a datagram-oriented transport, such as UDP, most suitable in constrained environments. Moreover, requests and responses are not sent over a previously established connection, but are exchanged asynchronously over CoAP messages. CoAP supports four types of messages: (i) Confirmable (CON), (ii) Non-confirmable (NON), (iii) Acknowledgement (ACK) and (iv) Reset (RST) [5].

As CoAP is bound to UDP, an unreliable transport method, messages may arrive unordered, be duplicated or missing. Therefore, CoAP implements a reliability mechanism similar to TCP, but more lightweight. Message reliability is provided by marking it as CON, eventually retransmitting it on a default timeout basis until a corresponding ACK is received from the corresponding endpoint. When a message does not require reliable transmission, however, it can be sent as NON. This could be the case of each single measurement out of a stream of sensor data (temperature, pressure etc).

The CoAP security mechanisms are largely discussed in [2]. We also described DTLS in [5], as the substitute of TLS for UDP transport layer.

Many architectures do not require to employ any security at the transport layer level, due to the physically limited access. Instead, there are mechanisms to provide lower-layer security, such as IPSec at network layer, when connecting to the outside network. In this case, the packets are sent over usual UDP over IP. The security is provided by routing techniques and by keeping attackers from gaining access to packets to or from the CoAP nodes.

Three DTLS modes are available and by implementing them, the new architecture is called CoAPs (secured), similar to HTTP secured with SSL/TLS which became HTTPS [8]. In this way, the security association can be used to authenticate and authorize the communication peer. CoAP does not provide in the specification any imposed mechanisms for authentication or authorization.

The four CoAP modes that can be employed in addition to DTLS are described below:

The *NoSec* mode does not provide any security and messages are sent without being encrypted.

The *PreSharedKey* mode is usually implemented for devices that cannot support the public key cryptography, due to memory limitations. It is based on a list of pre-shared keys, each one including a list of nodes it can be used to communicate with, installed on a device at manufacturing time [2]. There may be one or more keys for each node. When negotiating a connection to a new node, the system selects a key based on the nodes it is trying to contact and then begins a DTLS session using Pre-Shared Key (PSK) mode of DTLS.

In *RawPublicKey* mode, the device has an asymmetric key pair without a certificate (a raw public key), an identity calculated from the public key and a list of identities of the nodes it can communicate with; most commonly, the asymmetric key pair is generated and installed during the manufacturing process. A device may be set up with multiple raw public keys.

The RawPublicKey mode is “appropriate for devices requiring authentication based on public keys, but which are unable to participate in public-key infrastructures” [2].

The *Certificate* mode is employed for applications that support PKI and public-keys based authentication. In this mode, the device has an asymmetric key pair with an X.509 certificate signed by a common trust root and bound to its subject. It also has a list of root trust anchors that can be used for validating a certificate.

ECC (elliptic curve cryptography) was adopted in CoAP for RawPublicKey and Certificate modes. The advantage of ECC over other algorithms such as RSA is that the key length is smaller and therefore the computational time is smaller, while the security is the same. ECC uses ECDSA (Elliptic Curve Digital Signature Algorithm) and ECDHE (Elliptic Curve Diffie-Hellman Algorithm with Ephemeral keys), supporting device authentication [2].

3.4 AMQP

AMQP (Advanced Message Queuing Protocol) is sometimes regarded as an IoT protocol, as authors in [8] state, since it is about server inter-communication using queues. Historically, AMQP emerged from the financial services, where the focus was on interoperability and on not losing messages [7].

AMQP is a binary protocol built on top of TCP and provides a pub/sub architecture. It ensures reliability in the case of network disruptions by storing the messages in queues (“store-and-forward” feature) [3]. It also provides three QoS:

1. *At most once*: the message is sent once and it can be delivered or not.

2. *At least once*: the message is ensured to be delivered one time or more.

3. *Exactly once*: the message is delivered once and only once.

Security is provided by SSL/TLS and/or SASL protocols.

3.5 HTTP RESTful services

REST is more of an architectural style rather than a protocol, but it is usually considered in terms of HTTP. It relies on exposing resources that can be consumed by clients via request/response commands. It supports both XML and JSON, which is vital for interoperability. Even though it is a mature and widely spread protocol, it was not optimized for IoT use; however, it is adopted by many cloud platforms due to useful features such as content-type negotiation, authentication mechanisms, caching [3].

Nonetheless, HTTP has a myriad of drawbacks for the Internet of Things. Firstly, the overhead of the request/response model, as well as the possibly long polling are not suitable not only on constrained devices, but also on smartphones that rely on battery usage. Secondly, HTTP headers are very large and contain a lot of information that might not be useful in many IoT applications. It is likely to have a bigger header than the body of the request, which might overuse the network. Finally, it is envisaged that the JSON will be replaced by binary formats for the IoT environments. The advantage of binary encodings is that the schemas are automatically enforced and therefore they provide less overhead at reading data.

As we discussed in section 3.3, CoAP is a REST-like, lightweight alternative to HTTP. Even though the optimizations brought by replacing TCP with UDP are important, it still has the disadvantage of the request/response architecture.

RESTful services use SSL/TLS for securing communication. However, there are M2M platforms that do not have support for HTTPS [3] and in this case authentication keys are carried in the header of every request, which diminishes the overall system security.

3.6 WebSocket

WebSocket is a protocol developed by a HTML5 initiative and it is built on top of TCP. The specification states that it is a “full-duplex communication channel that operates through a single socket over the Web”[6].

In terms of architecture, it can be considered neither a publish/subscribe nor a request/response protocol. The connection process is the following: the client initializes a conversation by starting a handshake action with a server; after the connection has been established, messages can be exchanged. The process can be perceived as similar to HTTP, but the difference is that headers are removed and messages are delivered asynchronously, in a full-duplex connection [3]. Therefore, WebSockets are improved in terms of packet payload, but it is still not well suited for constrained devices.

The RFC[7] states that WebSocket is a standalone protocol and its only relationship with HTTP is the handshake, which is “interpreted by HTTP servers as an Upgrade request”. It uses port 80 for regular connections and 443 for secured communication with TLS. Client authentication is not imposed, however available mechanisms can be used, such as HTTP or TLS authentication, cookies etc.

4. Experimental setup

The purpose of the experiment is to test the six protocols we described in the previous chapter and to evaluate them from different perspectives. In relation to the IoT context, we are interested in comparing them from the following points of view:

- Data bytes and total bytes for each protocol.
- Ratio between the useful bytes (i.e. actual information sent over the network) and the total number of bytes exchanged, also called protocol efficiency in [4].
- Data packets and total packets.
- Ratio between the useful packets and the total number of packets.
- Average of packet size in bytes.
- Round-trip time (RTT), defined as the time required by a packet to travel between the source and the destination.

To simulate realistically and efficiently an IoT network, we used a PC as a server and a Raspberry Pi 3 as a client, which has the following technical specifications:

- Broadcom BCM2837, quad core ARM Cortex-A53, 1.2GHz;
- 1GB RAM, LPDDR2 (900 MHz);
- 10/100 Ethernet, 2.4GHz 802.11n wireless;
- Micro SD port – we used a 1GB card for loading Raspbian OS and storing data.

In an IoT network, Raspberry Pi usually acts as a gateway between the wireless sensors, which measure and send data, and the cloud server, whose purpose is to collect the data.

Figure 1 illustrates the setup of our network:

1. the sensors and the actuators are replaced by a simulator which generates environment data: temperature, motion, light, image.
2. the Raspberry Pi, having the characteristics we defined above, acts as a gateway. It also acts as a simulator and as a client, since it interacts with the server.
3. the server, which simulates the cloud server of a real IoT network, is a PC with Intel Xeon Quad Core processor and 16GB of RAM (3GHz), running Ubuntu 16.04.
4. the Raspberry Pi and the PC are connected to a 100 Mbps LAN switch.

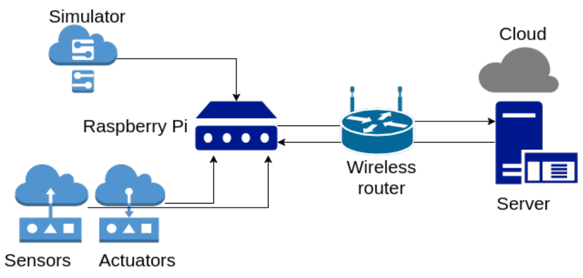

Figure 1. General network setup

In our setup, we simulate the measurements by generating ten messages that contain a JSON with five fields, populated with random data:

```
{
 "time": "2017-07-08T10:00:00.000Z",
 "temperature": 25,
 "motion": 1,
 "light": 50,
 "image": "AybdYAbfmfiwbs[...]"
}
```

The "time" field is the UTC timestamp when the message is sent; the "temperature" is a float between 20 and 50; the "motion" field is a boolean which tells is the sensors perceives movements around; the "light" is a percent; the "image" field is a generated string of 100 characters.

Then, the data is sent to the cloud server via various application layer protocols.

As far as software is concerned, we implemented applications for each of the six protocols using open-source libraries in Python. The servers are hosted on the PC, acting as a cloud server and the clients are on the Raspberry Pi. The experiment consists in sending the same data from all the clients to their corresponding servers, via various application layer protocols. The traffic is then captured and analyzed using Wireshark.

Implementation details for each protocol are further explained in the next sections.

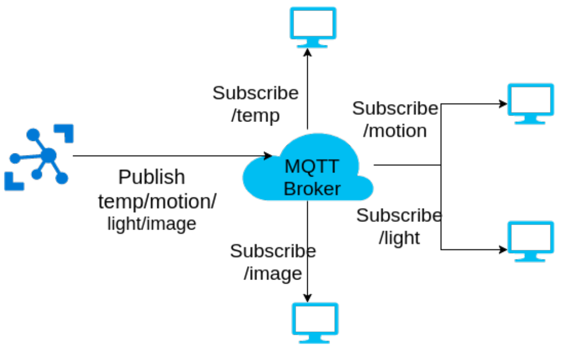

Figure 2. MQTT network

4.1 MQTT

To establish a MQTT network, there are three actors needed: a broker, a publisher and a subscriber.

Figure 2 shows the network we implemented: there is a publisher which sends messages on four topics and four subscribers, each subscribing to one topic.

The broker has the role of a server; we use Mosquitto, hosted on the server. The publisher and the subscribers are clients, written in Python using Paho Client library[8]. The usage flow for the clients is the following:

- Creation of a client instance.
- Connection to the broker.
- Maintain traffic with the broker by using a loop() function.
- Subscribe to a topic to receive messages – subscribe() function.
- Publish messages to the broker via the publish() function.
- Disconnect from the broker.

4.2 XMPP

Since XMPP's architecture is client-server, we can either create a client and make request to an existing server or create our own server. In order to implement XMPP, we used XMPPPy[9] library.

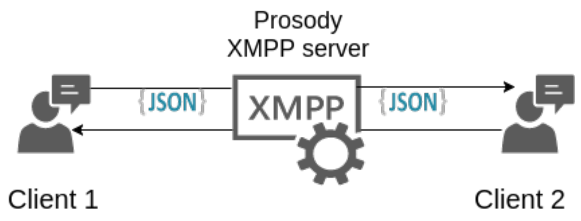

Figure 3. XMPP network

The server is Prosody and it is hosted on the PC, while the two clients are run from the Raspberry Pi. To simulate a chat-like conversation, we implemented two clients connected to the Prosody server; each client sends five messages to his peer and the other receives them. Therefore, we have a total of ten messages exchanged, as we have in all the implementations.

4.3 CoAP

There is a myriad of CoAP open-source implementations available, however we chose CoAPthon [11], a Python library built on top of an event-driven networking engine named Twisted.

The CoAP server defines a resource ('measurements' in our case), and the client must query that resource in order to get data. The network setup is quite simple and it is depicted in Figure 4. The client is hosted on the Raspberry Pi and the server on the PC.

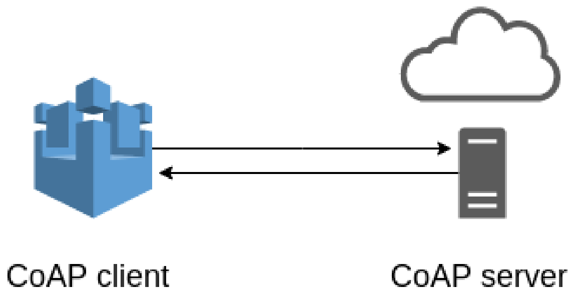

Figure 4. CoAP network

4.4 AMQP

To run AMQP, it is necessary to install RabbitMQ on the server and declare a queue. After setting the environment, clients and servers can send messages in the queue and/or consume messages. As the documentation states, "AMQP is a two-way RPC protocol where the client can send requests to the server and the server can send requests to a client"[10]. The Python library used for AMQP is Pika.

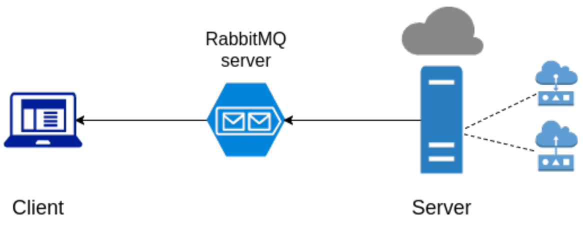

Figure 5. AMQP network

The network setup, as displayed in Figure 5, has the RabbitMQ server as a broker between the two actors: the server simulates the sensor measurement data and sends them to the queue, while the client consumes the messages from RabbitMQ. Therefore, the server is a publisher and the client is a subscriber.

4.5 HTTP RESTful Service

Since HTTP RESTful services require a client-server architecture, we build a server using Python Eve Framework[11]. The client can be a browser, a HTTP client such as Postman or SOAP UI or the API can also be consumed with "curl" commands, for CLI.

An example of such a command is:

```
$ curl -i http://test.com/example
HTTP/1.1 200 OK
```

The framework requires three features:

- a MongoDB database,
- a launch script and
- a configuration file, where the necessary settings are defined, such as: database host, port and name, schema, allowed methods bound to certain resources and others. For example, for certain resources, the use of HTTP's DELETE command can be prohibited.

We define the resource named 'measurements', which contains information about temperature, motion, light and image. The only methods allowed are GET – for retrieving data and POST – for saving data in the MongoDB database. Furthermore, as a custom setting, search can also be done on an 'id' field.

After setting up the server environment, installing MongoDB, installing eve library and running the main script, the server can accept requests. An example of a POST request, with curl, is the following:

curl -d '[{"id": 1, "temperature": 23, "motion": 0, "light": 50, "image": "[..]" }]' -H 'Content-Type: application/json' http://*host_ip*:5000/measurements.

The HTTP Status 200 means that the data was successfully saved and it can now be queried using a GET:

curl -i http://*host_ip*:5000/measurements

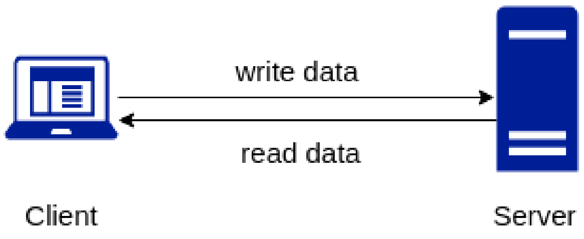

Figure 6. HTTP network

The HTTP setup, as depicted in Figure 6, is composed of a server application, run on the PC, and two clients on Raspberry Pi. The first client writes data and the second queries.

4.6 WebSocket

The WebSocket client-server architecture is more like a consumer-producer pattern; therefore, there are two actors which must be implemented:

- a client, which sends messages and
- a server, whose responsibility is to receive and store messages from clients.

The Python implementation[12] provides supports for implementations of both client and server and it is built according to the WebSocket protocol:

- An HTTP upgrade request as an opening handshake,
- data transfer, ended with a closing handshake.

Furthermore, the API supports asynchronous operations. Once the server has started, it loops and waits for clients to connect. A 'hello' method can also be configured to be sent to a newly connected client. When a new client wants to connect, the server accepts the connection, starts the opening hadshake and delegates to the WebSocket handler. The handler is executed and the client sends messages; the closing handshake and the end of the connection is also performed by the server.

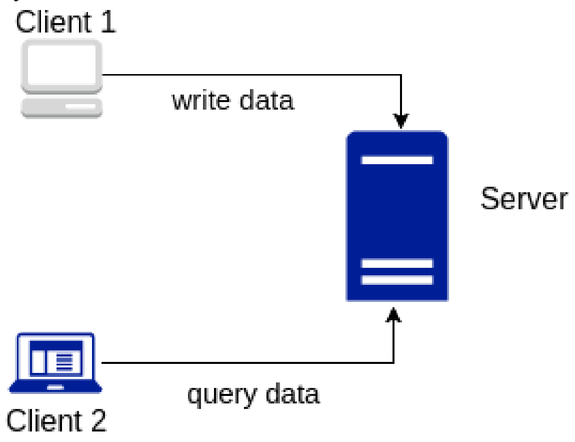

Figure 7. WebSocket network

The WebSocket architecture is simple, as depicted in Figure 7. The client, hosted on the Raspberry Pi, sends simulated data to the server and the server responds with a message of confirmation.

5. Experimental results

The tests were conducted using the same set of data, on six different protocols. A raw message calculation, outside any protocol encapsulation, reveals that one message has a length of 212 bytes. The experiment consisted, as stated before, in sending ten messages in a JSON format.

The first aspect we are interested in is the relation between useful data bytes and total bytes. Figure 8 shows a big discrepancy between AMQP and the other protocols. AMQP sends few packets, but with a large amount of data. The most efficient protocol from this point of view is WebSocket, which sends only 3514 bytes of useful information and 9800 bytes in total.

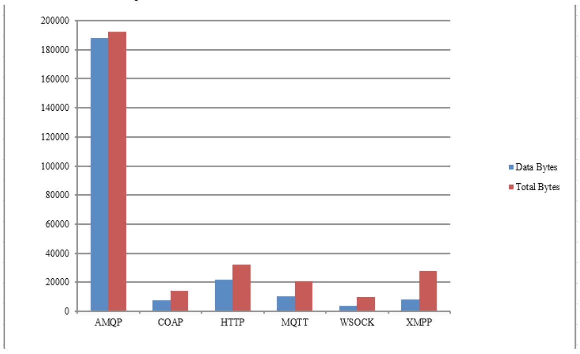

Figure 8. Data bytes and total bytes

Figure 9 shows the ratio between data bytes and total bytes per protocol. Therefore, as far as protocol efficiency is concerned, XMPP has the smallest percent value, of 28,11%. While the useful traffic for XMPP was of only 7864 bytes, the total was almost four times bigger. The difference is explained by ACK, RST and FIN packets.

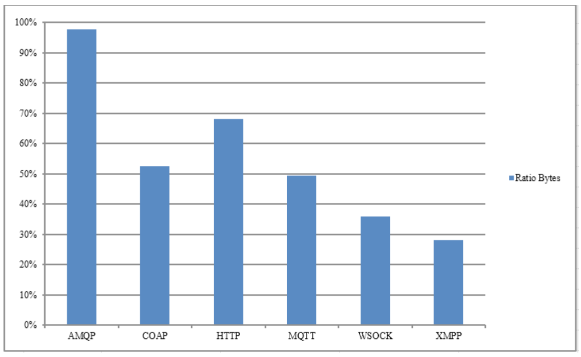

Figure 9. Ratio between databytes and total traffic bytes

After analyzing data bytes, it is also practical to consider data packets. Figure 10 reveals a histogram of data packets and total packets and Figure 11 presents the ratio between useful packets and total of packets.

While in the previous measurements AMQP was emerging as the protocol with the highest traffic, in this case CoAP and MQTT have the largest number of packets exchanged. However, this is not necessarily relevant in the context of IoT, since more packets with few bytes might be preferable to few packets with a huge amount of data.

WebSocket and XMPP exchange the smallest amount of packets, from two reasons:

- The data is not separated on topics.
- The information sent in a packet appart from the useful payload is negligible.

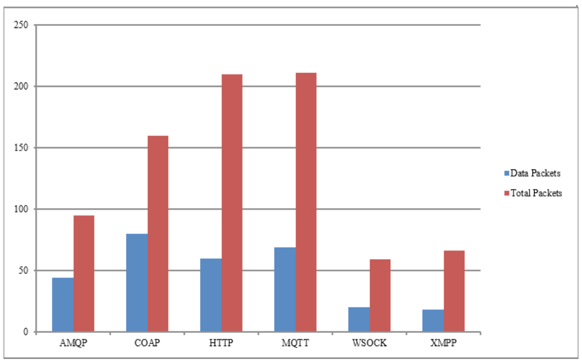

Figure 10. Data packets and total packets

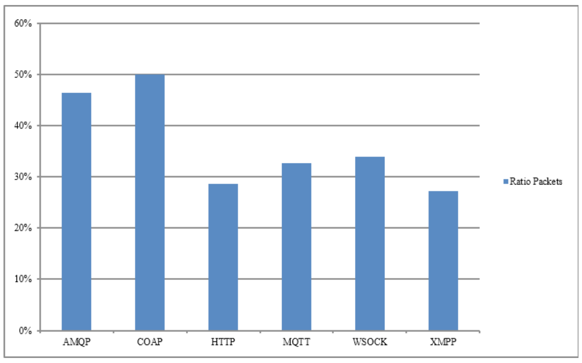

Figure 11. Ratio between data packets and total packets

Since we discussed about packets and number of exchanged bytes, another interesting topic is the average number of bytes for packets. Figure 12 shows the results for each protocol. As described in the previous section, CoAP and MQTT implemented four topics – temperature, motion, light and image and the data queries were made on each one of them. On the other hand, the other protocols write all topics in one message and also read also in one message. Therefore, this protocol implementation explains the small number of bytes per packet and the relatively big amount of packets in comparison with the other protocols. However, although WebSocket does not implement topics, it still has a good performance in terms of average number of bytes per packet (approximately 175 bytes), but also on data bytes and total bytes, where it has the lowest values from all discussed protocols.

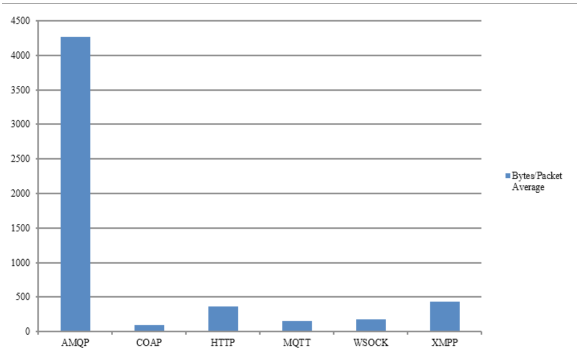

Figure 12. Average bytes per packet

Finally, the most relevant measurement, the average round-trip time, is presented in Figure 13. MQTT and XMPP performed the best, with 0.448 ms and 0.373 ms, respectively. On contrary, AMQP has an average of 90.75 ms, a value that can by explained by the big amount of data that travels in one packet. Therefore, as expected, AMQP is a heavy-weight protocol, not suitable for constrained devices; it is nonetheless a reliable and secure protocol, suitable for handling sensitive data, such as e-payments.

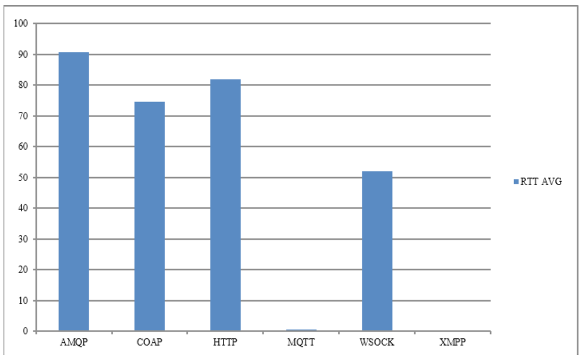

Figure 13. Average RTT

6. Conclusions and future research

This paper presented an experimental approach towards six of the most used application layer protocols in IoT: AMQP, CoAP, HTTP RESTful services, MQTT, WebSocket, XMPP. The experiment was conducted by simulating a simple IoT network and by generating and exchanging environment data.

The results show by comparison the performances of the protocols in a common scenario of a wireless network. We can conclude that AMQP is not suitable for constrained devices, given the large amount of traffic it triggers, while XMPP, MQTT and WebSocket perform very well in terms of RTT, number of relevant bytes exchanged and number of packets. CoAP and HTTP also have satisfactory results and they also have the advantage of interoperability. Furthermore, the specific of the actual IoT environment should also be taken into consideration when comparing protocols. For example, in a smart home context, where measurements are sent frequently, XMPP may be suitable because of its chat-like architecture or MQTT with QoS=0, for a lightweight communication. If payments are processed, AMQP would be suitable. Finally, for an eHealth system, CoAP or HTTP are suitable, due to their interoperability and proxy-ing capabilities.

In conclusion, considering the benefits of scalability and interoperability at the application layer, further research must be conducted in the area of cross-protocol proxies for the application layer IoT solutions.

Endnotes

[1]MQTT essentials, <http://www.hivemq.com/blog/mqtt-essentials-part2-publish-subscribe>.

[2]MQTT V3.1 Protocol Specification, <http://public.dhe.ibm.com/software/dw/webservices/ws-mqtt/mqtt-v3r1.html>.

[3]<http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html>.

[4]TechTarget, IoT Agenda – "MQTT (MQ Telemetry Transport)", <http://internetofthingsagenda.techtarget.com/definition/MQTT-MQ-Telemetry-Transport>.

[5]XMPP RFC, <https://tools.ietf.org/html/rfc6120>.

[6]HTML5 Websocket, <https://www.websocket.org/aboutwebsocket.html>.

[7]The WebSocket Protocol, <https://tools.ietf.org/html/rfc6455>.

[8]Eclipse Paho, Python Client, <http://www.eclipse.org/paho/clients/python/docs/>.

[9]XMPPPy, <http://xmpppy.sourceforge.net/>.

[10]Pika, <http://pika.readthedocs.io/en/0.10.0/intro.html>.

[11]Python Eve, <http://python-eve.org>.

[12]Python WebSockets, <http://websockets.readthedocs.io>.

REFERENCES

1. Dumitrache, I., Sacala, I. S., Moisescu, M. A. & Caramihai, S. I. (2017). A conceptual framework for modeling and design of Cyber-Physical Systems, *Studies in Informatics and Control*, *26*(3), 325–334.

2. Granjal, J., Monteiro, E. & Sa Silva, J. (2015). Security for the Internet of Things: A Survey of Existing Protocols and Open Research Issues, *IEEE Commun. Surv. Tutorials*, *17*, 1294–1312.

3. Karagiannis, V., Chatzimisios, P., Vazquez-Gallego, F. & Alonso-Zarate, J. (2015). A Survey on Application Layer Protocols for the Internet of Things, *Trans. IoT Cloud Comput.*, *3*, 11–17.

4. Mijovic, S., Shehu, E. & Buratti, C. (2016). Comparing application layer protocols for the Internet of Things via experimentation. In *2016 IEEE 2nd Int. Forum Res. Technol. Soc. Ind. Leveraging a Better Tomorrow. RTSI.*

5. Nastase, L. (2017). Security in the Internet of Things: A Survey on Application Layer Protocols. In *IEEE International Conference on Control Systems and Computer Science*.

6. Park, J., Joe, I., & Kim, W. T. (2014). An efficient discovery protocol of large-scale CPS middleware for real-time control system, *Studies in Informatics and Control*, *23*(1), 23–30.

7. Rajandekar, A. & Sikdar, B. (2016). *A Survey of MAC Layer Issues and Protocols for Machine-to-Machine Communications*, 1–12.

8. Raza, S., Shafagh, H., Hewage, K., Hummen, R. & Voigt, T. (2013). Lithe: Lightweight secure CoAP for the internet of things, *IEEE Sens. J.*, *13*, 3711–3720.

9. Schneider, S. et al. (2013). Understanding The Protocols Behind The Internet Of Things, *Electron. Des.*, 1–9.

10. Stanford-Clark, A. & Truong, H. L. (2013). "MQTT for Sensor Networks (MQTT-SN) Protocol Specification", *Mqtt.Org* [Online]. Available *<http://mqtt.org/new/wp-content/uploads/2009/06/MQTT-SN_spec_v1.2.pdf>*.

11. Tanganelli, G., Vallati C. & Mingozzi, E. (2015). CoAPthon: Easy Development of CoAP-based IoT Applications with Python. In *IEEE World Forum on Internet of Things (WF-IoT)*.