Comparison Between MQTT and WebSocket Protocols for IoT Applications Using ESP8266

Guilherme M. B. Oliveira, Danielly C. M. Costa, Ricardo J. B. V. M. Cavalcanti, Josiel P. P. Oliveira,
Diego R. C. Silva, Marcelo B. Nogueira and Marconi C. Rodrigues
Escola de Ciências e Tecnologia
Universidade Federal do Rio Grande do Norte
Natal, Rio Grande do Norte 59078-275
Emails: guimatheus@ufrn.edu.br, daniellycmcosta@gmail.com, ricavalcanti@bct.ect.ufrn.br, patricio@bct.ect.ufrn.br
diego@ect.ufrn.br, marcelonogueira@ect.ufrn.br, marconicamara@ect.ufrn.br

***Abstract*—As the Internet of Things becomes more popular, applications development using this concept are turning more and more common and accessible. Besides that, high performance and real-time applications require a low latency protocol. Taking these aspects into account, this work aims to compare the application layer network protocols: Message Queue Telemetry Transport and WebSocket, using ESP8266 (SoC with IEEE 802.11) and Node.js servers for data exchange, using the most popular protocol implementation found in Github, considering topics like documentation, round-trip time of packages using local network and memory allocated in a device. Experimental tests were performed to measure latency by calculating the time difference of packets exchanged between a server and an ESP8266 using both protocols. It has been found that the use of WebSocket is more appropriate than MQTT in applications with RTT of at least 1 millisecond according to the data and contexts presented in that article.**

I. Introduction

As the internet presence grows in people's lives, the development of intelligent products, able to make web connection and exchange information between the users or others devices, are becoming even more common and accessible. This may change our living standards as they offer possibilities of measurement, inference, and comprehension of environment indicators [1]. This technology, named Internet of Things (IoT), can be explored in several areas and purposes, that covers base industries, transport, health, and safety departments, reaching the final users [2].

Into IoT projects development, it is necessary:

- A hardware, made up of sensors, actuators, and embedded communication hardware;
- A middleware for data analytics and storage;
- An accessible interface, adapted to different platforms [1].

Referring to the first and second topics, this article used a *Wemos D1 mini* (Figure 1), a prototyping board for IoT applications, that have as positive aspects, besides the low cost: A direct USB connection; WebSocket, MQTT and others protocols compatibility as well as compatibility with most of Arduino libraries because of existing frameworks [3].

As a result of the IoT versatility, the client-server data exchange methods was diversified to serve various purposes. Among these methods, the most traditional is the Hypertext Transfer Protocol (HTTP), that has shown to be efficient when the sending data interval is known and when this interval is not small, as an example in water consumption data transmission, that could be sent hourly.

Figure 1. Wemos D1 mini, source: wiki.wemos.cc

Otherwise, when the application requires a small transmission interval (below 500 milliseconds), using this protocol increases significantly latency rates, network traffic, and data package size, thus making the application unviable [4].

Considering low latency applications, new protocols like Message Queue Telemetry Transport (MQTT) and WebSocket were developed which are going to be compared in this article. These two protocols have differences, but can be used depending on the application with the same purpose. For the comparison, were analyzed topics as: qualitative aspects of each protocol; latency in different situations; and amount of microcontroller programming memory occupied by the library that implements the communication protocol. The last one is a fundamental factor because IoT used microcontrollers may have a low memory yet, on the order of kilobytes, increasing to few megabytes in modern models. For the comparison was used each protocol implementation, found on Github [5], a repositories site.

A. Similar Works

The Comparing Application Layer Protocols for the Internet of Things via Experimentation also suggests a comparison of protocols within an IoT scenario. Using the microcontroller STM32F411RE and ESP8266 only as a wireless module. They compare CoAP, WebSocket and MQTT protocols performance using RTT algorithm and a created coefficient named "effi-

ciency". In order to obtain the RTT, an approach similar to the algorithm used in the present work is used, however, the results do not match due to differences in hardware architecture and network topology. Another difference observed is that the libraries that implement such protocols are not analyzed [6].

II. BIBLIOGRAPHIC REVIEW

A. WebSocket protocol

WebSocket protocol was developed to meet constant data exchange between client and server not supported earlier from HTTP [7]. The protocol consists in a complete bidirectional communication channel that works through a single socket [8], as well as having an asynchronous communication (in contrast with HTTP protocol), in other words, both sides can send data anytime while the connection is established.

The protocol is divided into two parts: handshake and data transfer. In the handshake, basically the client and the server establish initial communication using HTTP and a port, 80 is the default. In this first communication, the client requests a communication type update that once verified, the request is validated and the data exchange can be done using the WebSocket protocol (Fig. 2).

For communication, disregarding IP, TCP, and TLS framing overhead, a single HTTP request could carry an additional 500-800 bytes of metadata plus cookies. In contrast, WebSocket protocol uses a custom binary framing format that divides each message into one or more frames. When these frames reach a destination they are joined and the sender is notified that the entire message has been received. Each frame header can be 2 to 10 bytes in size if sent by the server and 6 to 14 bytes if sent by the client (a client must add a masking key to prevent cache poisoning attacks). WebSocket is also considered one of the most versatile data transport methods available, because of the customization capabilities through of Application Programming Interfaces (API's), extensions and sub-protocols, an example of this is the compression and multiplexing extensions [9].

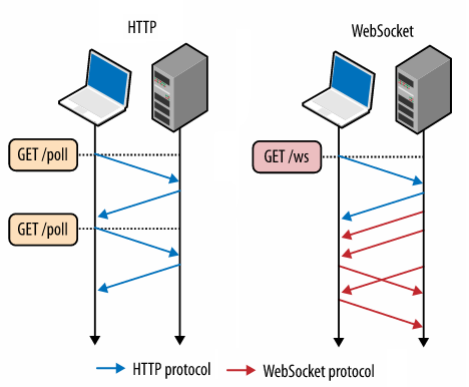

Figure 2. HTTP and WebSocket protocols communication flow, source: hpbn.co (modified)

B. MQTT protocol

MQTT is an exchange messaging protocol that uses the publish-subscribe standard for transporting messages between a server and clients. It runs over TCP/IP and can be run on network protocols that provide ordered, lossless and bidirectional connections. It is standardized by a technical committee of the Organization for the Advancement of Structured Information Standards. MQTT becomes a good candidate to be used for Machine-to-Machine communication and IoT contexts, designed to be lightweight, open and easy to implement, especially in contexts where the internet can be expensive, has low bandwidth, is not secure or when utilizing an embedded device with limited memory resources or processing [10].

In publish-subscribe pattern used at the MQTT, the massages exchange between different clients is through of a server, called the broker. The broker filters the messages and distributes them to the clients according to the topic - an identifier each message has. The client can be an IoT device, Web application, mobile application among others. Those who publish a message to the broker with a topic are called publishers and those who subscribe one or more topics for reading specific messages are called subscribers. The subscribers can receive messages from a number of publishers and can send them to others subscribers, a client can be both publisher and subscriber. All the clients establish the connection with the broker. The publishers do not know the destination of messages sent and the subscribers do not know the origin of messages received. [11] An example of architecture using this pattern is shown in Fig. 3.

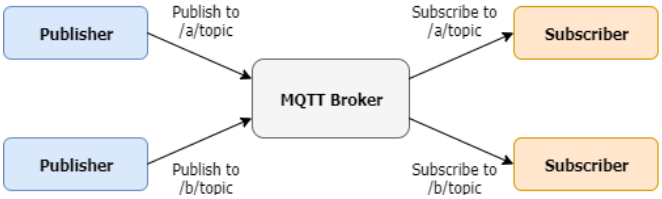

Figure 3. Publish/subscribe architecture example

The format for a control packet of a message using the protocol is divided between fixed header, variable header and payload. The fixed header has the size of two bytes and the variable header and payload size can range from zero to N bytes.

MQTT has three Quality of Services (QoS) levels to message delivery:

- QoS 0 (At most once delivery) Messages are delivered at most once or no, is not stored by either party and there is no acknowledgment of delivery on the network.
- QoS 1 (At least once delivery) Messages must be delivered at least once, the sender stores and tries to send a message until it receives a confirmation, so the receiver can receive and process a message several times.
- QoS 2 (Exactly once delivery) Messages are delivered exactly once, the message is stored in the sender and

receiver until both parties receive confirmation that the message has been delivered exactly once to the receiver.

The levels of QoS can be used according to requirements of applications and have different times for delivery [12].

III. Development

For the comparison, qualitative aspects were considered and quantitative tests were made for WebSocket and MQTT protocols.

The quantitative tests were conducted at the Laboratório de Informática Industrial (LII), located in Núcleo de Pesquisa e Inovação em Tecnologia da Informação (nPITI), with an ESP8266 device and a computer to be the server, the only ones connected to the local network, created by the router TP-Link TL WR 741 ND (Wireless Router N 150Mbps). The computer, which was used as a server, was the model: Samsung NP270E4E, with Intel(R) Celeron(R) CPU 1007U and 4Gb of RAM and operating system Ubuntu 17.10.

The servers were written in Javascript programming language using the Node.js, an asynchronous event-driven JavaScript runtime designed to build scalable network applications. [13] It was necessary to develop different files for each of the protocols, however, maintaining the same algorithm. For the tests using the WebSocket protocol the library called ws [14] was used and for the tests with the MQTT protocol, it was used the broker called Mosca [15].

For the present comparative the one-way delay estimation method chosen was the round-trip time (RTT), also called round-trip delay, that is, the time elapsed between sending data and receive a delivered message acknowledgment [16].

The packets RTT in the local network was measured using each protocol and both following cases were tested: by varying the payload size and keeping it constant. For tests using payload size variation, one side acted passively, only responding with packets without payload, and the other one sent the packets with the payload varying in size. In this case, both the server acting as passive and the device were tested.

The algorithm used for RTT measurement were executed on both the server and the ESP8266 device using the functions ”process.hrtime”, of Node.js, and ”micros”, native to the ESP8266 core for Arduino, respectively. A greater uncertainty was observed on the measurements made at the server, compared to the measurement in the device, considering the standard deviation of the mean of the samples. Therefore, the ESP8266 was chosen to calculate packets round trip time and send the data to the server at the end of the calculations.

On the test algorithm, two parameters were used: the number of samples and payload of packages, both were set before the test starts. For each sample, a package is sent by the sender to the receiver and the sender wait for a response from the receiver through another package; the elapsed time between these is added to a vector in the device. This process is repeated until the number of samples is reached, that triggered the sent of time array to the server for posterior processing (Fig. 4).

Because of the memory limitation of the ESP8266, RTT measurements were divided into five thousand samples, totaling twenty thousand for each test, sufficient to obtain results, without risk of stack overflow.

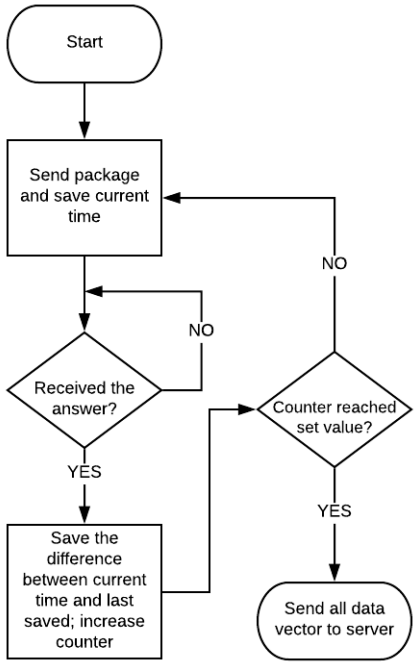

Figure 4. Algorithm used for RTT measurement

The libraries used were searched on the source code hosting platform GitHub, from July 28 to December 20, 2018, considering the compatibility with ESP8266, popularity in the community, and implementation of methods of connecting to the server and sending and receiving data.

The chosen libraries was PubSubClient [17] for MQTT and arduinoWebSockets [18] for WebSocket. Furthermore, the analyzed parameters were divided onto qualitative and quantitative ones. The qualitative are:

- Documentation;
- Usage difficulty and support;
- Source reliability.

The quantitative aspects are:

- Round-trip time;
- Memory consumption in the device;
- Max payload size found.

IV. Results

A. Qualitative Analysis

1) WebSocket: There are a few WebSocket protocol libraries compatible with the ESP8266 which have the parameters evaluated, even though there are more than 100 repositories, between new and projects forks directed to this implementation, few were functional.

Considering the forks and stars quantity, *arduinoWebSockets* library was the most popular for microcontroller projects in the search period and therefore was chosen for testing.

As for documentation chosen as representative of the protocol, the library has documentation that shows some methods, protocol features supported, limitations and supported hardware.

The library can also be considered easy to use and has examples code. It is important to highlight that arduinoWebSockets repository supports not only client side but server side too, allowing the users choose which one will be used in its project.

2) MQTT: There are at least ten MQTT libraries available that are compatible with the ESP8266, some such as esp_mqtt [19] and async-mqtt-client [20] implement the entire protocol specification and features, such as all levels of quality of service.

The PubSubClient library implements some methods, such as connect, subscribe, publish, and setCallback that is called when new messages arrive at the client. It has a documentation with the description of all the methods of the library and the limitations, as well as the hardware compatible and description of versions. Moreover, have at least 15 contributors and is known in the community having at least 600 forks, but it is not updated frequently by the main author. One of the limitations is that to publish a message only QoS 0 is possible and to subscribe to the topic only QoS 0 or 1.

Regarding ease of implementation, the PubSubClient could be used with some ease, respecting the limitations reported in the documentation, and can be configured according to the application.

B. Quantitative tests

Figures 5 and 6 show scatter graphs with values sampled according to the measurement methodology previously described when the server sends messages with payload sizes 5, 25, 50, 100, and 200 bytes, and the ESP8266 acts passively. Tables I and II show measurement statistics of this context.

Using the WebSocket protocol, the values have been concentrated below 1 millisecond and fall into two groups, one above and one below 0.4 milliseconds. Moreover, using MQTT, the values are concentrated below 2 milliseconds, especially for data with a payload size of 200 bytes.

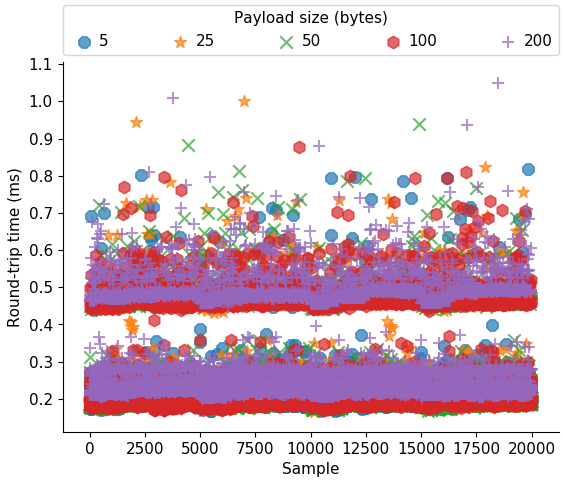

Figure 5. RTT measurements using WebSocket, with microcontroller as passive, as a function of sample size for various payloads

Table I
STATISTICS ABOUT RTT MEASUREMENTS USING WEBSOCKET WITH MICROCONTROLLER AS PASSIVE (FIG. 5)

Payload size (bytes)	RTT(ms)		
	Mean	Std. Deviation	Median
5	0.198	0.032	0.193
25	0.202	0.047	0.194
50	0.206	0.059	0.194
100	0.215	0.075	0.195
200	0.260	0.093	0.224

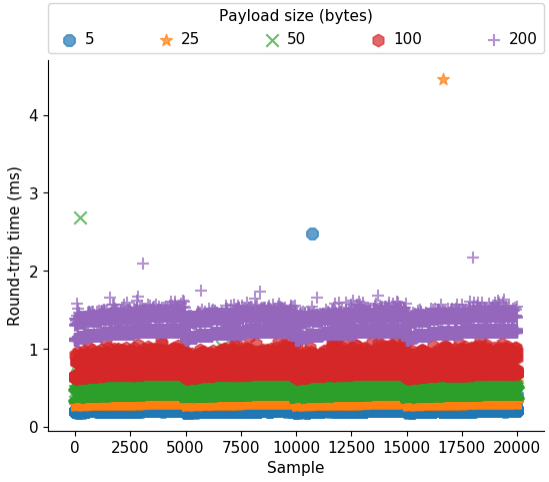

Figure 6. RTT measurements using MQTT, with microcontroller as passive, as a function of sample size for various payloads

Table II
STATISTICS ABOUT RTT MEASUREMENTS USING MQTT WITH MICROCONTROLLER AS PASSIVE (FIG. 6)

Payload size (bytes)	RTT(ms)		
	Mean	Std. Deviation	Median
5	0.212	0.033	0.208
25	0.317	0.048	0.311
50	0.448	0.250	0.439
100	0.705	0.062	0.693
200	1.232	0.107	1.216

Figures 7 and 8 also present scatter plots with values sampled using WebSocket and MQTT protocols respectively, but with the server acting passively and the device sending data with payloads 5, 25, 50, 100, 200, 400, 800, 1200 and 1440 bytes. Tables III and IV show measurement statistics of this context.

After the test, it was observed that using both protocols, the values are concentrated below 1 millisecond, even for data with the larger payload size tested.

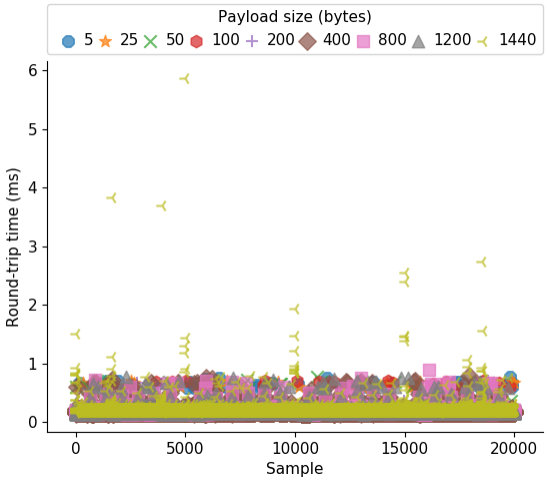

Figure 7. RTT measurements using WebSocket, with microcontroller as ative, as a function of sample size for various payloads

Table III
STATISTICS ABOUT RTT MEASUREMENTS USING WEBSOCKET WITH MICROCONTROLLER AS ACTIVE (FIG. 7)

Payload size (bytes)	RTT(ms)		
	Mean	Std. Deviation	Median
5	0.168	0.025	0.166
25	0.168	0.025	0.166
50	0.167	0.022	0.165
100	0.168	0.022	0.166
200	0.167	0.023	0.165
400	0.170	0.025	0.169
800	0.168	0.023	0.165
1200	0.170	0.027	0.166
1440	0.213	0.124	0.214

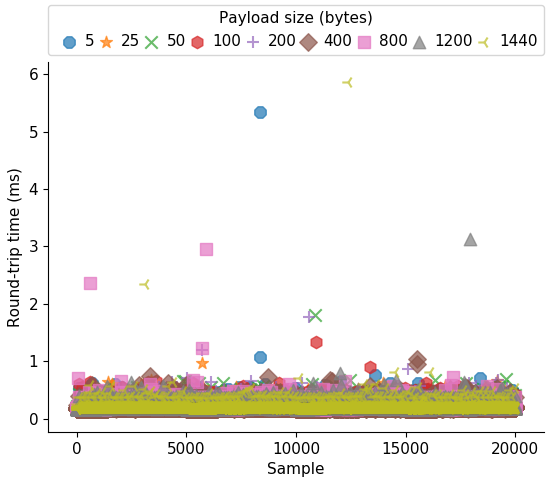

Figure 8. RTT measurements using MQTT with microcontroller as passive as a function of sample size for various payloads MQTT with microcontroller as active

Table IV
STATISTICS ABOUT RTT MEASUREMENTS USING MQTT WITH MICROCONTROLLER AS ACTIVE (FIG. 8)

Payload size (bytes)	RTT(ms)		
	Mean	Std. Deviation	Median
5	0.186	0.046	0.182
25	0.186	0.028	0.182
50	0.187	0.092	0.182
100	0.215	0.029	0.182
200	0.260	0.055	0.182
400	0.187	0.025	0.184
800	0.188	0.036	0.185
1200	0.190	0.090	0.186
1440	0.189	0.049	0.184

For measurements with the ESP8266 acting passively, it was only possible to obtain data up to 200 bytes because there were sudden disconnections during the measurement process from this value, thus affecting to obtain results. For measurements with the server acting as passive, the default value for a message in PubSubClient was changed to different sizes and the maximum experimentally value found to send data to the server was 1440 bytes, consequently limiting data acquisition. The library for WebSocket did not have the same limitation and it was possible to send data with the default maximum size for the payload, 15360 bytes.

Regarding the memory occupied by the libraries used in the tests, it was 6% of the total available for PubSubClient and 7% of the total for arduinoWebSockets.

V. CONCLUSIONS

The present study compared application layer protocols, MQTT and WebSocket, used with the ESP8266 device, analyzing quantitative and qualitative aspects of each protocol most popular implementation available on Github. The analyzed aspects included, for example, documentation and RTT.

Both implementations are promising to be used in IoT projects, even with observed limitations. ArduinoWebSocket

library has support for WebSocket specification features. However, PubSubClient does not implement important features, as all the qualities of service, present on the MQTT protocol. Both have documentation with examples, description of methods and limitations. The WebSocket library takes up more memory of device than the MQTT library, and its use is not feasible depending on the application. In particular, those that require as low memory as possible.

Under the conditions of quantitative tests, WebSocket is more appropriate for applications that require RTT below one millisecond, since the mean of the data measured in both tested situations is less than this time, as can be seen in the tables I and III. When ESP8266 operates in passive mode with MQTT, it is observed that the higher the payload, the higher the RTT. In this context, with a payload size of two hundred bytes, the average is 1.232 milliseconds and the median is 1.216 milliseconds (TABLE II), therefore, it becomes less appropriate for such applications.

Considering the tests when the server performed passively all means and medians values presented in tables IV and III are below 1 millisecond in both protocols, allowing using them in applications in the context that requires this RTT statistical values presented in tables. The datasets obtained from tests with constants payload size have a small standard deviation. This show consistency and reliability of the protocols used since they do not cause abrupt changes on the network RTT.

ACKNOWLEDGMENT

The authors would like to thank Universidade Federal do Rio Grande do Norte (UFRN) and Núcleo de Pesquisa e Inovação em Tecnologia da Informação (nPITI), for the ceded structure, as well as laboratory mates and professors for the support.

REFERENCES

[1] GUBBI, Jayavardhana *et.al.*. Internet of Things (IoT): A vision, architectural elements, and future directions. *Future generation computer systems*, v. 29, n. 7, p. 1645-1660, 2013.

[2] AL-FUQAHA, Ala *et. al*. Internet of things: A survey on enabling technologies, protocols, and applications. *IEEE Communications Surveys & Tutorials*, v. 17, n. 4, p. 2347-2376, 2015.

[3] WEMOS Electronics (2017). D1 mini. [Online] Available: https://wiki.wemos.cc/products/:d1/:d1/_mini.

[4] PIMENTEL, Victoria; NICKERSON, Bradford G. Communicating and Displaying Real-Time Data with WebSocket. *IEEE Internet Computing*, v. 16, n. 4, p. 45-53, 2012.

[5] Github. [Online] Available: https://github.com/

[6] MIJOVIC, Stefan; SHEHU, Erion; BURATTI, Chiara. Comparing application layer protocols for the Internet of things via experimentation. In: Research and Technologies for Society and Industry Leveraging a better tomorrow (RTSI), *2016 IEEE 2nd International Forum on*. IEEE, 2016. p. 1-5.

[7] FETTE, Ian.(2011, Dec.) *The websocket protocol*.[Online]. Available: https://tools.ietf.org/html/rfc6455.

[8] LUBBERS, Peter; GRECO, Frank.*HTML5 WebSocket: A Quantum Leap in Scalability for the Web*.[Online]. Available: http://www.websocket.org/quantum.html.

[9] *Websocket: BROWSER APIS AND PROTOCOLS, CHAPTER 17*.[Online]. Available:https://hpbn.co/websocket/.

[10] MQTT Version 3.1.1. Edited by Andrew Banks and Rahul Gupta. (2014, Oct.). OASIS Standard.[Online]. Available: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html. Latest version: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html.

[11] Hillar, G.C. MQTT Essentials - A Lightweight IoT Protocol. [S.L.]: Packt Publishing, 2017, 280 p.

[12] BOYD, B. et al. Building Real-time Mobile Solutions with MQTT and IBM MessageSight. [S.L.]: IBM Redbooks, 2014. 264 p.

[13] Dahl, R (2009, May.). Node.js. Node.js. [Online]. Available: https://nodejs.org/ [Accessed Oct. 5, 2017].

[14] WebSockets (2011, Nov.). ws. Github. [Online]. Available: https://github.com/websockets/ws [Accessed Oct. 5, 2017].

[15] Collina, M (2013, Feb.). mosca. Github. [Online]. Available: https://github.com/mcollina/mosca [Accessed Oct. 5, 2017].

[16] POSTEL, Jon. Transmission control protocol specification. RFC 793, 1981.[Online]. Available: https://tools.ietf.org/html/rfc793.

[17] O'Leary, N (2009, Feb.). pubsubclient. Github. [Online]. Available: https://github.com/knolleary/pubsubclient/ [Accessed Oct. 10, 2017].

[18] Markus (2015, May.). arduinoWebSockets. Github. [Online]. Available: https://github.com/Links2004/arduinoWebSockets [Accessed Oct. 15, 2017].

[19] PM. T (2016, Dec.). esp_mqtt. Github. [Online]. Available: https://github.com/tuanpmt/esp_mqtt/ [Accessed Oct. 17, 2017].

[20] M. Roger (2016, May.). async-mqtt-client. Github. [Online]. Available: https://github.com/marvinroger/async-mqtt-client/ [Accessed Dec. 15, 2017].

[21] LOCKE, Dave. Mq telemetry transport (MQTT) v3. 1 protocol specification. IBM developerWorks Technical Library, 2010.