PERFORMANCE EVALUATION OF DIFFERENT RASPBERRY PI MODELS AS MQTT SERVERS AND CLIENTS

Trent N. Ford, Eric Gamess and Christopher Ogden

MCIS Department, Jacksonville State University, Jacksonville, AL, USA

ABSTRACT

Performance analysis for devices in Internet of Things (IoT) environments is an important consideration, especially with their increasing integration in technological solutions, worldwide. The Single Board Computers (SBCs) of the Raspberry Pi Foundation have been widely accepted by the community, and hence, they have been incorporated in numerous IoT projects. To ease their integration, it is essential to assess their network performance. In this paper, we made an empirical performance evaluation of one of the most popular network protocols for IoT environments, named the Message Queuing Telemetry Transport (MQTT) protocol, on Raspberry Pi. To do so, we set up two different testbeds scenarios and assessed the performance with benchmarks. At the software level, we focused on Mosquitto, a popular open-source MQTT broker implementation and client library. Our principal metric is the transmission time, but we also investigated the throughput. In our experiments, we varied several parameters, such as the size of the payload of the published messages, the WiFi bandwidth, the QoS level, the security level (MQTT vs. MQTT with TLS), and the hardware for the clients and broker. We focus mainly on packet sizes ranging from 100 to 25,000 bytes. We also investigate how these low-cost devices handle a TCP SYN flood attack. In the research work presented within this paper, we aim to guide developers, researchers, network administrators, and hobbyists who plan to use these low-cost devices in an MQTT or IoT network by showing the performance that they should expect according to different Raspberry Pi options.

KEYWORDS

MQTT, Mosquitto, Benchmark, Performance Evaluation, Raspberry Pi, DoS Attack.

1. INTRODUCTION

Internet of Things (IoT) devices are devices that have some data processing and transmission capabilities. Since their cost is going down, they have been increasingly integrated into all kinds of technological solutions. Hence, the performance analysis of IoT devices is becoming an important consideration. To foster their smoother integration, it is fundamental to expand researcher, industrial, and consumer knowledge concerning their network performance. Common messaging protocols that have been used in IoT environments include Message Queuing Telemetry Transport (MQTT) [1][2], Constrained Application Protocol (CoAP) [3][4], Advanced Message Queuing Protocol (AMQP) [5][6], eXtensible Messaging and Presence Protocol (XMPP) [7][8], and DDS (Data Distribution Service) [9]. MQTT and CoAP appear to be the most frequently used.

In this work, we evaluate the performance of several Raspberry Pi Single Board Computers (SBCs) as MQTT servers (brokers) and clients (publishers and subscribers). Our experiments were done in testbed environments, using a transmission time benchmark that we previously developed [10] and a throughput benchmark proposed in [11]. We varied several parameters such

DOI: 10.5121/ijcnc.2022.14201

as the size of the payload of the published messages, the WiFi bandwidth, the QoS level, the security level (MQTT vs. MQTT with TLS), and the hardware for the clients and broker. With this work, we aim to guide developers, researchers, network administrators, and hobbyists who are planning to use low-cost Raspberry Pi SBCs in an MQTT or IoT network, and empirically examine and analyze what the expected performance is between these options.

The rest of the paper is structured as follows. In Section 2, we discuss a number of peer-reviewed literature work conducted within this research area. Section 3 provides a high-level overview of the technologies used in this research: MQTT, TLS, and Raspberry Pi SBCs. We describe the benchmarking tools used for our performance assessment in Section 4. In Section 5, we present the testbeds utilized for our experiments. We show, examine, and analyze the results of our experiments in Section 6. Finally, Section 7 concludes and summarizes the results of this paper as well as discusses future avenues for further research work within the area of study.

2. RELATED WORK

Some related works are dedicated to comparing several messaging protocols for IoT environments. Various research groups focused on contrasting functionalities, while others are more dedicated to performance evaluation using real testbeds or simulation tools. For example, Imane, Tomader, and Nabil [12] presented the relevance of IoT for the implementation of smart healthcare systems. They started with a summary of some existing healthcare systems by enumerating their strengths and weaknesses, before discussing in which context CoAP or MQTT should be chosen. They did not conduct any performance comparison evaluation but offered advice on selecting the most adequate network protocol for several different situations. The authors of [13] did a practical assessment of CoAP and MQTT in a testbed environment. They reported results related to the communication delay and the network traffic required for a variety of simple tasks. As clients, an ESP8266 [14] and an ODROID [15] were used. Çorak, Okay, Güzel, Murt, and Ozdemir [16] quantitatively compared the performance of CoAP, MQTT, and XMPP in a real-world testbed, and reported results such as the packet creation time and packet transmission time. The testbed consisted of one Intel Galileo Gen 2 board, one laptop (Intel Core i7-6700HQ CPU at 2.60 GHz and 4 GB of RAM), and some sensors. The sensors were connected to the Intel Galileo Gen 2 board, where the publisher was running. In the laptop, the authors executed the broker and the subscriber. Banno, Ohsawa, Kitagawa, Takada, and Yoshizawa [17] compared the performance of several MQTT brokers (Mosquitto [18], HiveMQ [19], and EMQ X [19]) and reported metrics such as latency and throughput. The authors developed their own MQTT performance testing tool. The testbed consisted of one Ethernet switch and four PCs that were running a publisher, a broker, a subscriber, and an NTP server, respectively. The publisher and subscriber synchronized their time using the NTP server. Laaroussi and Novo [20] presented a performance analysis of the security protocols employed with MQTT and CoAP, and reported parameters such as latency and throughput. The MQTT clients (both the publisher and the subscriber) were installed in an Intel Core i5 CPU at 2.60 GHz with 16 GB of RAM, while the broker was running in a Dell Latitude 7480 laptop with an Intel Core i7 CPU at 2.80 GHz and 16 GB of RAM. While the previous research groups used empirical experimentation for their evaluations, a number of other research groups have opted for conducting their assessments using simulators. As an example, Larmo, Carpio, Arvidson, and Chirikov [21] studied the performance of sensors reporting their captured samples over CoAP and MQTT, running on top of two different radio interfaces, namely Bluetooth Low Energy (BLE) and IEEE 802.11ah. The evaluations were performed using an Ericsson internal event-based radio network system simulator with detailed models of application, transport, network, and datalink layers for both BLE and WiFi.

Other performance evaluation works are focused on MQTT. Pipatsakulroj, Visoottiviseth, and Takano [22] developed muMQ, a high-performance MQTT broker that efficiently utilizes multi-core CPUs. To assess their new broker, the research team compared its performance with the one of Mosquitto, for parameters that include latency, throughput, and CPU usage. They conducted their experiments by running the broker in a machine equipped with a 20-core CPU (Intel Xeon E5-2650 @ 2.30 GHz) and 252 GB of RAM. As for client machines, they used virtual machines created on a computer with an 8-core CPU (Intel Xeon E620 @ 2.4 GHz) and 20 GB of RAM. In [23], the authors evaluated the usage of a Raspberry Pi Zero W as an MQTT gateway between sensors and a broker. They installed the sensors locally, while the broker was in the cloud (public broker). The gateway worked as the data-processing device closer to the sensors, and as a bridge to the Internet. The authors reported performance metrics such as the processor temperature and CPU utilization. In a home environment, MQTT performed better than CoAP on a Raspberry Pi 3 [24]. In [25], Baranauskas, Toldinas, and Lozinskis assessed a Raspberry Pi 2 as a broker in a testbed environment. The clients (both the publisher and the subscriber) were running in an ESP32 [14]. When using a Raspberry Pi 2 as a broker, the authors found that QoS 0 consumed 6.7% more energy than QoS 1 for MQTT with TLS. In [10], Gamess, Ford, and Trifas evaluated the MQTT protocol considering large payload sizes (ranging from 512 to 1,048,576 bytes) for the published messages, in testbed environments primarily based on conventional PCs. In one of the scenarios, the broker was changed to different models of Raspberry Pi.

As can be seen from this literature survey, little research work has been published that evaluates the performance of MQTT through the use of Raspberry Pi devices in an IoT network. In contrast to all the previous literature work done within this research area that we surveyed, in this effort, we conducted a number of empirical assessments that varied the role of low-cost devices, Raspberry Pi SBCs, and determined how they compare to each other and against a PC with average specifications, using benchmarks. We focused on two parameters, namely the transmission time and throughput. We also investigated how these low-cost devices handle a DoS attack, using Hping3 [26] to flood them with TCP SYN segments.

3. TECHNOLOGIES

3.1. Message Queuing Telemetry Transport

Message Queuing Telemetry Transport (MQTT) [1][2] is a machine-to-machine data exchange protocol for IoT devices. MQTT uses a publish/subscribe model with a broker. There are a few versions of the standard, with versions 3.1.1 [27][28] and 5.0 [29] as the most popular. TCP is the transport protocol used by MQTT, with two reserved ports: 1883 (MQTT without security) and 8883 (MQTT over TLS). MQTT topics are a form of addressing that allows MQTT clients to share information, through a broker. An MQTT broker, also known as an MQTT server, is the message hub. It keeps track of which devices subscribe to which topics, receives all published messages, and forwards them according to subscriptions. A client can be a publisher, a subscriber, or both as necessary.

3.2. Transport Layer Security

Transport Layer Security (TLS) [30][31][32] is one of the most frequently used security standards for network communication. Its most common usage is for securing sessions between a web server and a web browser [33]. It is a continuation of earlier SSL protocols. The main distinction between the TLS and SSL protocols is a question of who authored and who controlled them. SSL was initially proprietary from Netscape; TLS has always been an open standard created by the Internet Engineering Task Force since version 1.0 [30][33]. Both protocols use

X.509 [34] certificate standards, issued by an entity known as a Certificate Authority (CA). The certificates allow TLS to use asymmetric encryption to exchange a symmetrical key that is unique to a session. This is known as the handshake [33]. The asymmetrical aspect of the handshake makes it challenging to get access to the session key, while the session key allows for faster communication. The initial handshake is slightly resource-intensive for lower-powered and less capable devices, but the session key encryption has been noted to only have a small effect on performance [35]. The MQTT protocol does not have its own security layer, and it relies on the security provided by TLS [35]. Different ciphers are being explored for use with MQTT to lessen the overhead [35]. Shapsough, Aloul, and Zualkernan [35] concluded that low-cost, low-power MQTT devices increasingly offer a challenge for developers and security researchers to address the vulnerabilities while minimizing the impact on performance.

3.3. Raspberry Pi

The Single Board Computers (SBCs) from the Raspberry Pi Foundation are cheap, effective, and highly versatile. They run a flavor of Linux, so technically, they can do anything that a desktop computer with Linux can do. For example, a Raspberry Pi can be used as a desktop computer, an email or web server, a network storage device, a gaming console, a media center, or a controller for sensors and actuators [36]. They are also used to promote STEM education and for custom DIY projects [37]. The Raspberry Pi Foundation wants to democratize technology and provide access to tools to do so [38]. They created cheap general-purpose computers (e.g., US$15 for a Raspberry Pi Zero 2 W) and claim that their SBCs are actively being used in many places such as interactive museum exhibits, schools, national postal sorting offices, and government call centers [38]. Their goal is to eliminate the barriers that priced people out of technology for education, entertainment, and creativity. There is also a large open-source community and forums for Raspberry Pi OS [39] and troubleshooting projects using Raspberry Pis [40].

For this paper, we focused our experiments on the Raspberry Pi Zero W [41] (RPi Zero W), the Raspberry Pi Zero 2 W [42] (RPi Zero 2 W), and the Raspberry Pi 3 Model B [43] (RPi 3B). We chose these models for their popularity and their low-cost (US$10 for RPi Zero W, US$15 for RPi Zero 2 W, and $US35 for RPi 3B).

4. BENCHMARKING TOOLS

In a previous paper [10], we developed a novel benchmark for determining the transmission time (time required for a PUBLISH message to go from the publisher to the subscriber, through the broker) in an MQTT environment. Briefly, this benchmark tool is written in the C programming language and uses the Mosquitto client library [18]. It requires two clients (clt1 and clt2) where one client (clt1) publishes a message that is sent to the broker and forwarded to the other client (clt2), triggering that client (clt2) to do the same thing. This gives a round-trip transmission time, so the one-way transmission time (or transmission time) can be calculated as half of it. The benchmark repeats the message's exchange multiple times, and the overall time is divided by the number of round-trip exchanges to minimize errors. The researcher can set several parameters, including the ability to specify the payload size of the PUBLISH messages and the QoS level. In this paper, we will refer to this benchmark as the transmission time benchmark.

Another benchmarking tool used in this paper is from [11]. It is written in the Go programming language and uses the Eclipse Paho library [44] (the Eclipse Paho project provides open-source, mainly client-side, implementations of MQTT and MQTT-SN in a variety of programming languages). It reports the throughput, defined as the number of PUBLISH messages sent/received per second. The benchmark needs two clients (a publisher and a subscriber) and a common topic.

The subscriber establishes a connection with the broker, subscribes to the common topic, and waits for the messages published by the publisher. The publisher establishes a connection with the broker and sends a specific number of messages to the common topic (the inter-departure time of the PUBLISH messages is one of the benchmark parameters). Once having sent all the PUBLISH messages, the publisher ends its connection with the broker and shows the throughput at its level (sending throughput). As soon as the subscriber has received all the PUBLISH messages, it ends its connection with the broker and reports the throughput at its level (receiving throughput). This benchmark has a number of other parameters, such as the payload size of the PUBLISH messages and the support for TLS. In this paper, we will refer to this benchmark as the throughput benchmark.

5. EXPERIMENTAL SETUP

For our experiments, we used three Raspberry Pi Zero W, three Raspberry Pi Zero 2 W, two Raspberry Pi 3 Model B, and four standard PCs of identical characteristics. We removed the original microSD cards of the SBCs and replaced them with 64 GB SanDisk Extreme microSDXC UHS-I Memory Cards (SDSQXA2-064G-GN6MA). They are considered as one of the fastest microSD cards of the market, with up to 160 MB/s and 60 MB/s for the reading and writing speeds, respectively. The specifications of the PCs were: Dell OptiPlex 3030 AIO, with a 64-bit Intel quad-core i3-4130 CPU at 3.4 GHz, 16 GB of RAM, a 512 GB SSD, a 1 Gbps Ethernet NIC, and an Intel Wireless 7260 Network Adapter (dual-band WiFi adapter with support to IEEE 802.11 a/b/n/g/ac). Debian amd64 11.1 (codename “Bullseye”) was installed as the operating system.

The Raspberry Pi Zero W (RPi Zero W) [41] is based on a 32-bit Broadcom BCM2835 single-core ARM1176JZF-S SoC @ 1.0 GHz, 512 MB of RAM, one 2.4 GHz IEEE 802.11b/g/n WiFi interface, one micro USB On-The-Go port, and one mini HDMI connector. The Raspberry Pi Zero 2 W (RPi Zero 2 W) [42] was released in October 2021. It is the last SBC of the Raspberry Pi Foundation. It is based on an RP3A0, which consists of the integration of a 64-bit Broadcom BCM2710A1 quad-core Cortex-A53 @ 1.0 GHz and 512 MB of RAM, in a single chip. It also has one 2.4 GHz IEEE 802.11b/g/n WiFi interface, one micro USB On-The-Go port, and one mini HDMI connector. It can be easily overclocked to 1.3 GHz, with an adequate heat sink. The Raspberry Pi 3 Model B (RPi 3B) [43] is based on a 64-bit Broadcom BCM2837 quad-core Cortex-A53 SoC @ 1.2 GHz, 1 GB of RAM, one 10/100 Mbps Ethernet interface, one 2.4 GHz IEEE 802.11b/g/n WiFi interface, four USB 2.0 ports, and one full-size HDMI connector.

Many operating systems are available for Raspberry Pi, but we opted for the 32-bit version of the Raspberry Pi OS (armhf), released in October 2021. This operating system is based on Debian Bullseye. Of the three distributions (“Lite”, “Desktop”, and “Desktop and Recommended Software”) supported by the Raspberry Pi Foundation, we chose the “Lite” distribution, which consists of 493 packages, that do not include a desktop environment. It is worth mentioning that the 64-bit version of Raspberry Pi OS (arm64) is still in the beta stage, and cannot be run on an RPi Zero W.

The WiFi router had the following characteristics: NETGEAR AC1200 Smart WiFi Router R6220, with an 880 MHz MediaTek processor that has two radio bands (IEEE 802.11b/g/n in the 2.4 GHz band and IEEE 802.11a/n/ac in the 5 GHz band), 128 MB of flash, and 128 MB of RAM. In the 2.4 GHz band, the bandwidth can be set up to a maximum of 54, 145, or 300 Mbps. We did not use the 5 GHz band, since the SBCs that we tested do not support it.

We opted for Mosquitto [18] version 2.0.11 as the broker in all the experiments and configured it for MQTT version 3.1.1. It is an open-source message broker that implements MQTT versions

3.1, 3.1.1, and 5.0. We selected Mosquitto as the broker of all our experiments due to its open-source license, maturity, wide availability, simple installation, and overall ease of use.

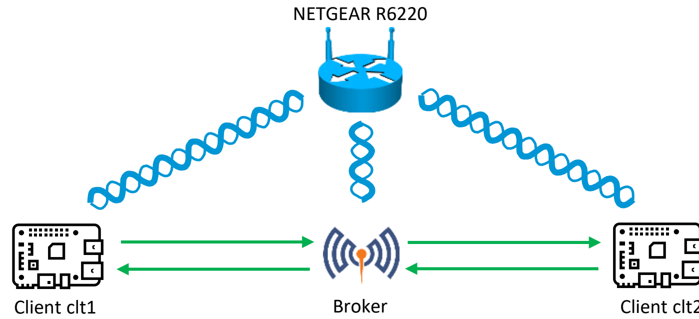

Figure 1. Testbed without a DoS Attack

We used two testbeds for our experiments. Figure 1 was the testbed for all the experiments without a DoS attack and consisted of two clients and one broker. We varied the hardware for the clients and the broker. However, the hardware for both clients was always identical. Each subsequent experiment will specify which hardware was used.

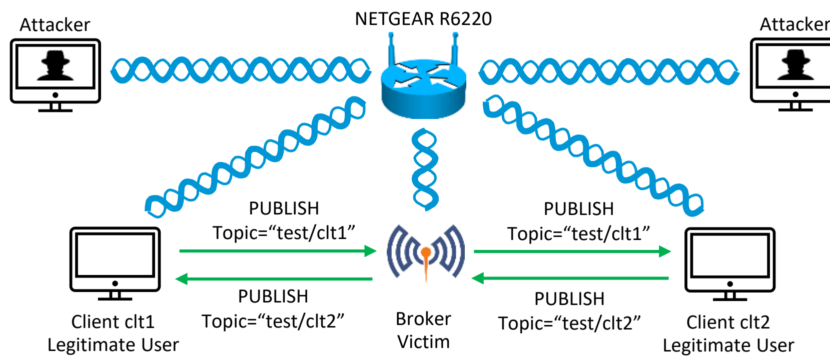

Figure 2. Testbed for the DoS Attack

As shown in Figure 2, the testbed for the DoS attack is very similar to the setup in Figure 1. The two clients and the two attackers were PCs, with the specifications mentioned previously. The broker device was a Raspberry Pi and was varied for different experiments.

For both Figures 1 and 2, the communication between devices was controlled by the WiFi router (NETGEAR AC1200 Smart WiFi Router R6220), using WiFi Protected Access 2 (WPA2). The PCs and SBCs were placed 4 meters from the wireless router, with no obstacles between them. Also, IPv4 was employed as the network protocol for all the experiments discussed within this paper.

6. PERFORMANCE RESULTS AND ANALYSIS

In this section, we give and analyze the results of our experiments. In order to ensure the consistency and validity of our empirical results, each experiment was conducted several times (minimum 10 times), and the results depicted in the figures correspond to the performance results averaged from the repeated experimental runs.

6.1. WiFi Bandwidth Variation

The objective of this experiment is to determine the impact of the WiFi bandwidth over the transmission time of a PUBLISH message. To do so, we used the testbed depicted in Figure 1. Clients clt1 and clt2 were running the transmission time benchmark described in Section 4, with QoS 0. In this experiment, we varied the WiFi bandwidth on the 2.4 GHz band using the router's built-in settings (maximum 54, 145, and 300 Mbps). Raspberry Pi devices acted as the clients and published the messages, while a PC was the broker.

Figures 3, 4, and 5 show the experiment's results using the RPi Zero W, the RPi Zero 2 W, and the RPi 3B, respectively. The MQTT payload of the PUBLISH messages was varied from 100 to 25,000 bytes. The figures have three bars for each payload size: 54 Mbps in blue, 145 Mbps in orange, and 300 Mbps in grey. It is noted that when the wireless router was configured at 54, 145, and 300 Mbps, respectively, all the Raspberry Pi SBCs had a bitrate that capped out at 54, 72.2, and 72.2 Mbps, respectively, while the PC's bitrate capped at 54, 144.4, and 144.4 Mbps, respectively.

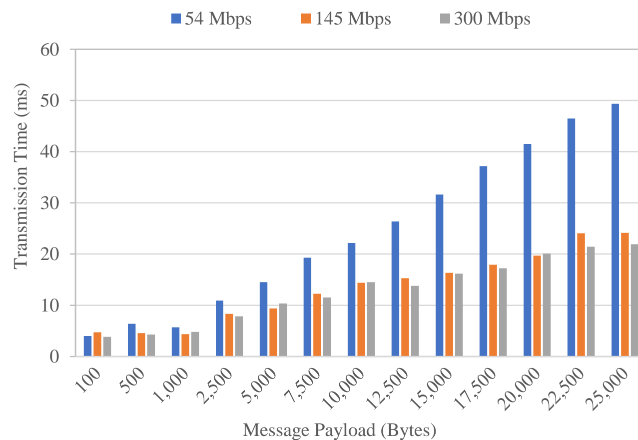

Figure 3. Transmission Time when Using RPi Zero W as Clients with WiFi Bandwidth Variation

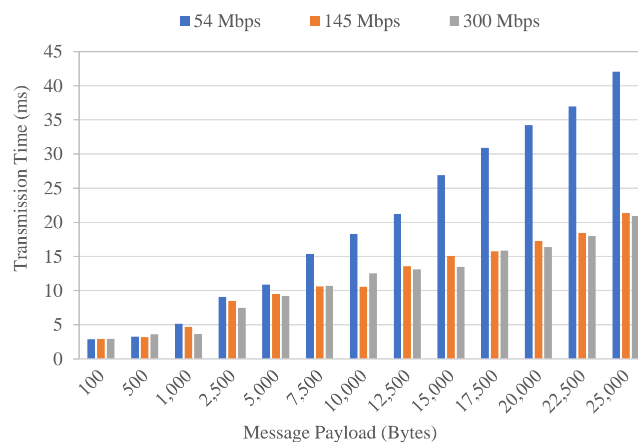

Figure 4. Transmission Time when Using RPi Zero 2 W as Clients with WiFi Bandwidth Variation

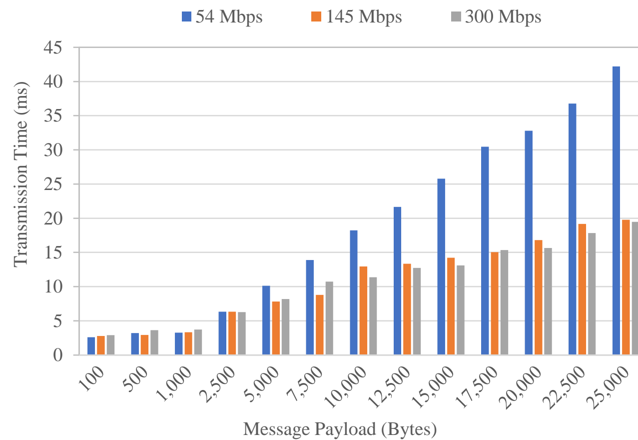

Figure 5. Transmission Time when Using RPi 3B as Clients with WiFi Bandwidth Variation

Across all devices, as seen in Figures 3, 4, and 5, 54 Mbps had the slowest transmission times. Likely due to the bitrate caps innate to the device hardware, 145 Mbps and 300 Mbps yielded very similar results. As a consequence of this bitrate cap, all the subsequent experiments were carried out by setting up the wireless router at a maximum bandwidth of 145 Mbps.

6.2. Client Hardware Variation

This experiment aims to determine the impact of the client hardware over the transmission time of a PUBLISH message. To do so, we used the testbed depicted in Figure 1. Clients clt1 and clt2 were running the transmission time benchmark described in Section 4, with QoS 0. In this experiment, we varied the clients' hardware; clients clt1 and clt2 were both changed out so that both clients in the experiment were running the same hardware and software. Raspberry Pi devices acted as the clients and published the messages, while a PC was the broker.

The MQTT payload of the PUBLISH messages was varied from 100 to 25,000 bytes. The WiFi bandwidth was set to a maximum of 145 Mbps at the router using the 2.4 GHz band. Figure 6 has four bars for each payload size: RPi Zero W in blue, RPi Zero 2 W in orange, RPi 3B in grey, and RPi Zero 2 W Overclocked in yellow. It is worth remembering that all the Raspberry Pi SBCs had a bitrate that capped out at 72.2 Mbps, while the PC's bitrate capped out at 144.4 Mbps.

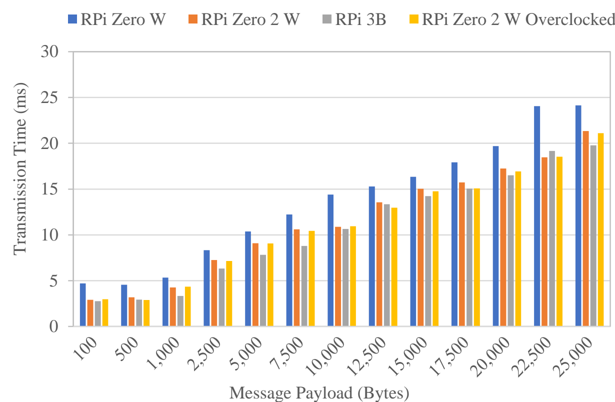

Figure 6. Transmission Time when Making Client Variation

As shown in Figure 6, the RPi 3B performed slightly better overall compared to the RPi Zero 2 W (when overclocked and when not), but their performance is nearly the same. Overclocking the RPi Zero 2 W did not have a significant impact but was slightly faster. We think that most of the time is spent in transmission, and not in computing, resulting in this slight difference. The poorest performance was for the RPi Zero W, with the difference becoming greater with the increasing payload size.

6.3. Large Payload Variation

The goal of this experiment is to determine the impact of a large payload over the transmission time of a PUBLISH message. To do so, we used the testbed depicted in Figure 1. Clients clt1 and clt2 were running the transmission time benchmark described in Section 4, with QoS 0. Raspberry Pi devices acted as the clients and published the messages, while a PC was the broker.

The MQTT payload of the PUBLISH messages was varied from 100,000 to 1,000,000 bytes. The WiFi bandwidth was set to a maximum of 145 Mbps at the router using the 2.4 GHz band. Figure 7 has three bars for each payload size: RPi Zero W in blue, RPi Zero 2 W in orange, and RPi 3B in grey.

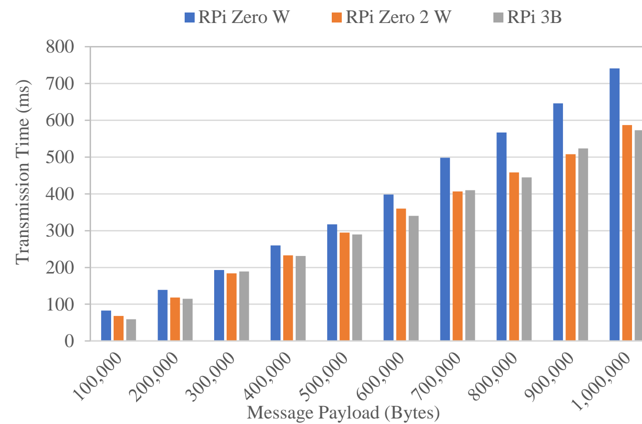

Figure 7. Transmission Time of Large Payloads with Client Variation

As shown in Figure 7, the performance of the RPi 3B was slightly better overall compared to the RPi Zero 2 W. Both of these Raspberry Pi devices significantly outperformed the RPi Zero W in this experiment, with the performance differences becoming greater with increasing payload sizes.

6.4. Client and Broker Hardware Variation

Choosing the most suitable hardware for the broker often comes down to a compromise between cost and capability. This experiment aimed to determine the impact of the client and broker hardware over the transmission time of a PUBLISH message. To do so, we used the testbed depicted in Figure 1. Both clients clt1 and clt2 were running the transmission time benchmark described in Section 4, with QoS 0. In this experiment, we varied the hardware of the clients and the broker so that all devices in the experiment were running the same hardware and software. Raspberry Pi devices acted as the clients and published the messages, while an identical Raspberry Pi device performed as the broker.

The MQTT payload of the PUBLISH messages was varied from 100 to 25,000 bytes. The WiFi bandwidth was set to a maximum of 145 Mbps at the router using the 2.4 GHz band. Figure 8 has three bars for each payload size: RPi Zero W in blue, RPi Zero 2 W in orange, and RPi Zero 2 W Overclocked in grey.

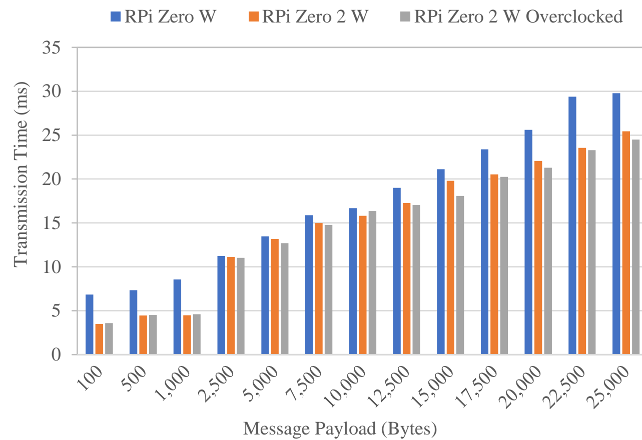

Figure 8. Transmission Time when Making Client and Broker Variation

By analyzing Figure 6 (the broker is a PC) and Figure 8 (the broker is a Raspberry Pi), we can observe that the overall transmission time increases when using a Raspberry Pi as a broker. When finding the average percentage difference across the tested points, the RPi Zero W, the RPi Zero 2 W, and the RPi Zero 2 W Overclocked had 37.47%, 32.90%, and 29.26% slower transmission times than when using the PC as the broker. Unfortunately, we did not have a third RPi 3B available for our experiments, but it is likely to follow a trend similar to the one observed for the RPi Zero 2 W, based on the results shown so far.

6.5. QoS Level Variation

It is worth reminding that MQTT has native support for Quality of Service (QoS). It defines three levels of QoS where QoS 0 is at most one delivery (fire and forget), QoS 1 is at least one delivery (acknowledged delivery), and QoS 2 is exactly one delivery (assured delivery). With QoS 0, a publication is done through a single message between the MQTT client and server (PUBLISH). With QoS 1 and when there are no communication problems, a publication requires two messages (PUBLISH and PUBACK). With QoS 2 and without issues in the communication, a publication requires four messages (PUBLISH, PUBREC, PUBREL, and PUBCOMP). Hence, there is an increase in network traffic with an upper QoS level, but it is usually acceptable for important messages.

The goal of this experiment is to determine the impact of the QoS level over the transmission time of a PUBLISH message. To do so, we used the testbed depicted in Figure 1. Clients clt1 and clt2 were running the transmission time benchmark described in Section 4, where we varied the QoS level (0, 1, and 2). Raspberry Pi devices acted as the clients and published the messages, while a PC performed as the broker.

Figures 9, 10, and 11 depict the results of the experiments using RPi Zero W, RPi Zero 2 W, and RPi 3B as clients, respectively. The MQTT payload of the PUBLISH messages was varied from 100 to 25,000 bytes. The WiFi bandwidth was set to a maximum of 145 Mbps at the router using

the 2.4 GHz band. The figures have three bars for each payload size: QoS 0 in blue, QoS 1 in orange, and QoS 2 in grey.

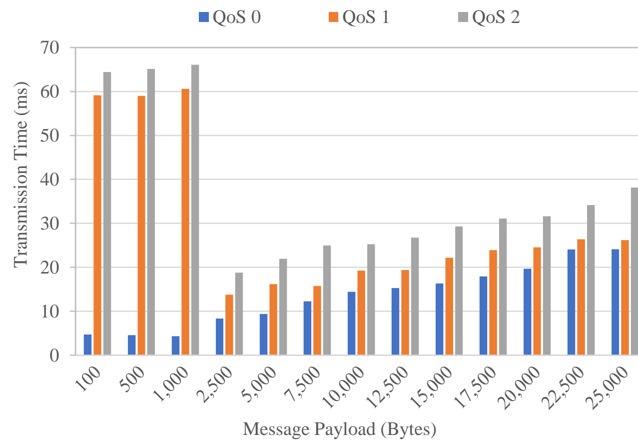

Figure 9. Transmission Time when Using RPi Zero W as Clients with QoS Variation

As can be seen in Figures 9, 10, and 11, QoS 0 (i.e., at most once) has the shortest transmission time, followed by QoS 1 (i.e., at least once), then QoS 2 (i.e., exactly once). The RPi Zero W is the slowest of the devices, while the RPi Zero 2 W and the RPi 3B perform similarly. Interestingly, there are significant spikes in transmission time for a payload of a PUBLISH message that can be sent within one TCP segment, for both QoS 1 and QoS 2. Hence, in these cases, it might be better for programmers to extend the payload with random bytes and force the usage of a second TCP segment since it should significantly reduce the transmission time.

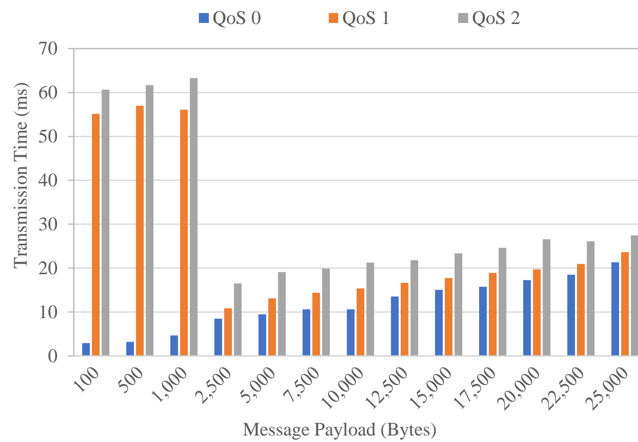

Figure 10. Transmission Time when Using RPi Zero 2 W as Clients with QoS Variation

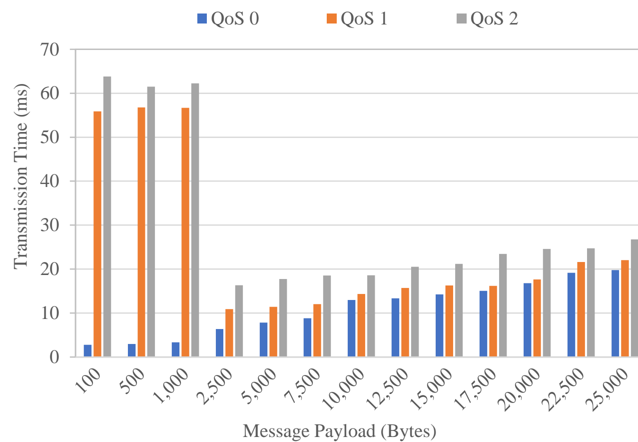

Figure 11. Transmission Time when Using RPi 3B as Clients with QoS Variation

6.6. Throughput of Messages with Security Variation

This experiment aims to determine the impact of security on the transmission of PUBLISH messages. To do so, we used the testbed depicted in Figure 1. Clients clt1 and clt2 were running the throughput benchmark described in Section 4, with QoS 0. We varied whether the communication was secured (with TLS) or not (without TLS) in this experiment. We used TLS version 1.2. Raspberry Pi devices acted as the clients and published the messages, while a PC performed as the broker.

The publisher (clt1) was set to publish 500 messages per second (inter-departure time of two milliseconds). Figures 12 and 13 depict the throughput at the level of the subscriber (clt2) without and with TLS, respectively, that is, the number of PUBLISH messages that could make it to the subscriber, per second. The MQTT payload of the PUBLISH messages was varied from 100 to 25,000 bytes. The WiFi bandwidth was set to a maximum of 145 Mbps at the router using the 2.4 GHz band. Figures 12 and 13 have three bars for each payload size: RPi Zero W in blue, RPi Zero 2 W in orange, and RPi 3B in grey. The experiments were run for thirty seconds for each payload size.

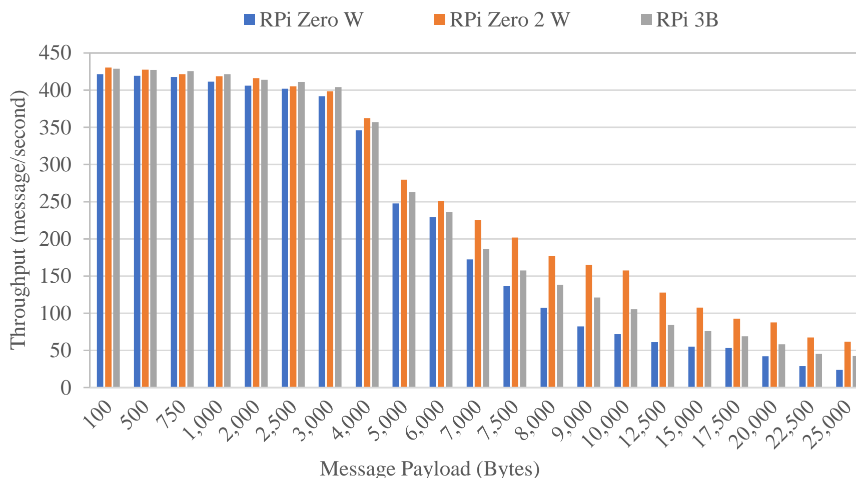

Figure 12. Throughput at the Level of the Subscriber without Security (without TLS)

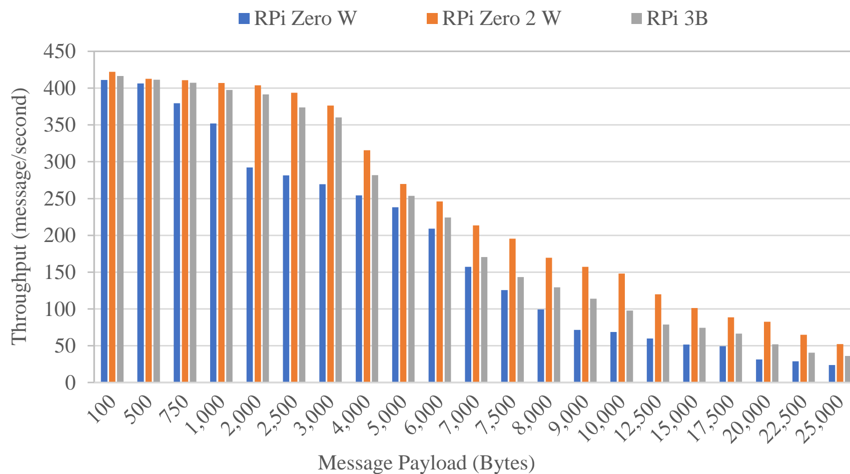

Figure 13. Throughput at the Level of the Subscriber with Security (with TLS version 1.2)

According to Figures 12 and 13, adding TLS security slowed the performance. The majority of the differences are minor, except for the RPi Zero W, where the impact of TLS is noticeable for payload sizes between 1,000 and 5,000 bytes.

The worst performance recorded in this experiment was for the RPi Zero W. Meanwhile, the RPi Zero 2 W and the RPi 3B yielded similar results to each other for small payloads. However, the RPi Zero 2 W outperformed the RPi 3B for larger payloads in this experiment.

6.7. Performance under a DoS Attack

A Denial-of-Service (DoS) attack is an attack meant to slowdown or shutdown a network service or resource, making it inaccessible to the intended users. In general, DoS attacks are accomplished by flooding the target with traffic, or sending it packets that trigger a crash due to bugs. Constrained devices are an easy target for DoS attacks, resulting in a great deal of time and money for the victim to handle. Hence, the importance is assessing DoS attacks on MQTT for Raspberry Pi.

The goal of this experiment is to determine the impact of a DoS attack on the transmission time of a PUBLISH message. To do so, we used the testbed depicted in Figure 2. Clients clt1 and clt2 were running the transmission time benchmark described in Section 4, with QoS 0. In this experiment, we varied the hardware of the broker. Clients clt1 and clt2 were both PCs, and another two PCs were used as the attackers. All four PCs have the specifications described in Section 5.

To carry out the DoS attack, we flooded the broker with TCP SYN segments generated by Hping3 [26], an open-source packet generator and analyzer for the TCP/IP protocol suite. We used the following command, where options –V, –S, –i, and –p enable verbose output, send TCP SYN segments to the target, indicate the interval of time between the sending of two consecutive segments (inter-departure time), and specify the destination TCP port (1883 for MQTT), respectively. The argument at the end of the command is the victim’s IP address (i.e., MQTT broker).

```
hping3  -V -S -i <interval> -p 1883 <brokerIPAddress>
```

Figures 14, 15, and 16 illustrate the effects of the TCP SYN flood attack over the transmission time with an inter-departure time of 1 ms for an RPi Zero W, RPi Zero 2 W, and RPi 3B as a broker, respectively. The MQTT payload of the PUBLISH messages was varied from 100 to 25,000 bytes. The WiFi bandwidth was set to a maximum of 145 Mbps at the router using the 2.4 GHz band. For each payload size, the figures have three bars: no attacker in blue, 1 attacker in orange, and 2 attackers in grey.

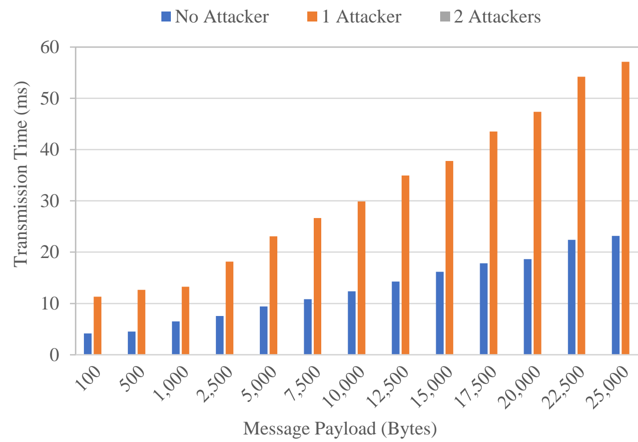

Figure 14. Transmission Time During a DoS Attack when Using an RPi Zero W as the Broker

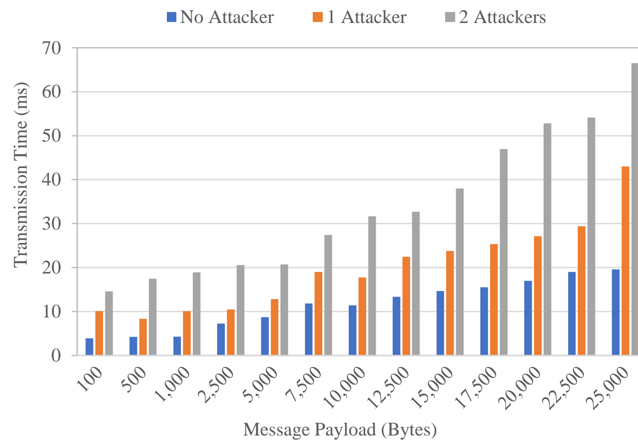

Figure 15. Transmission Time During a DoS Attack when Using an RPi Zero 2 W as the Broker

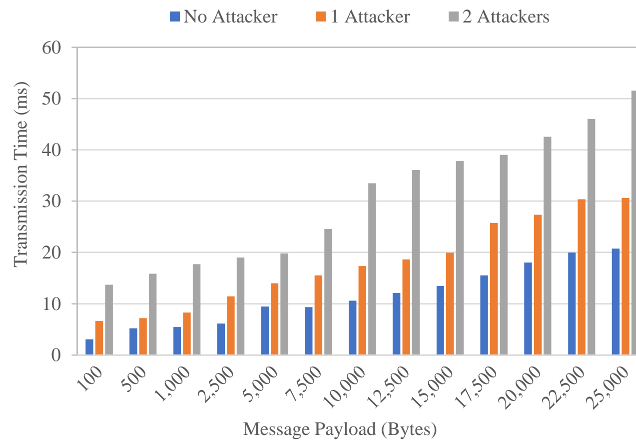

Figure 16: Transmission Time During a DoS Attack when Using an RPi 3B as the Broker

In Figure 14, the two attackers successfully stalled the RPi Zero W broker, so we were unable to get any MQTT transmissions through the network. This is the reason why there are only results for no attacker (in blue) and one attacker (in orange) in this case. As shown in Figure 16, the RPi 3B handled the attack the best, followed by the RPi Zero 2 W shown in Figure 15. In all figures, it is noticeable that each attacker creates a significant increase over the transmission time observed by the legitimate MQTT traffic.

7. Conclusions and Future Work

It is estimated that over 152,000 new IoT devices will connect to the Internet every minute by 2025. MQTT seems to be the most accepted messaging transport protocol for them. The introduction of so many IoT devices makes understanding their capabilities with respect to MQTT a matter of ever more critical importance.

In this paper, we set up MQTT experimental scenarios that varied the roles within the network oflow-cost devices, Raspberry Pi SBCs, and determined how each device compared not only against one another, but also against PCs of average specifications. According to our experiments, each Raspberry Pi could be a good to adequate broker for some circumstances, as they only increased transmission time by 29.26%-37.47% (RPi Zero 2 W Overclocked: 29.26%, RPi Zero 2 W: 32.90%, and RPi Zero W: 37.47%) compared to an average PC as the broker (see Section 6.4). When performing a DoS attack on the Raspberry Pi as a broker, the RPi Zero W could only handle one attacker. As shown in Section 6 of this paper, the RPi Zero 2 W and RPi 3B devices were both capable of handling the 1 and 2 attacker scenarios, at the cost of a very noticeable increase in overall transmission time. When factoring in the TLS protocol, the RPi Zero 2 W performed the best as a broker out of the Raspberry Pi devices that we tested in the experiments discussed within the scope of this paper.

The broker and client are both an essential part of an MQTT implementation, but there will only be one broker to many clients, so low-cost but capable devices become more of a priority for clients. With this in mind, we also performed other experiments with the Raspberry Pi SBCs acting as clients. We found that when varying the network bandwidth, the faster-allowed speed unsurprisingly yielded better transmission time results, at least up to the bitrate cap of the devices. Lower levels of QoS also have faster transmission times.

In general, the RPi 3B and RPi Zero 2 W often performed with similar experimental results to one another, so both could fit the same role. The RPi Zero W had the poorest performance in all circumstances but is a much cheaper option compared to the other IoT devices examined, tested, and analyzed within this paper.

In our future research work, we plan to analyze the performance of several MQTT brokers, such as VerneMQ, ActiveMQ, RabbitMQ, and Mosquitto. We also wish to investigate the performance of MQTT when using IPv6, instead of IPv4. Finally, we are interested in proposing some mathematical models that approximate the transmission time and the maximum throughput in basic MQTT scenarios.

CONFLICTS OF INTEREST

The author declares no conflict of interest.

ACKNOWLEDGMENTS

We are grateful to "Faculty Commons" and the "College of Science & Mathematics" at Jacksonville State University for partially funding this project.

REFERENCES

[1] T. Pulver, Hands-On Internet of Things with MQTT. Packt Publishing, 2019.
[2] "MQTT Homepage." https://mqtt.org
[3] "CoAP – Constrained Application Protocol Homepage." https://coap.technology
[4] Z. Shelby, K. Hartke, and C. Bormann, "The Constrained Application Protocol (CoAP)," RFC 7252. Internet Engineering Task Force (IETF), Jun. 2014.
[5] "AMQP Homepage." https://www.amqp.org
[6] "Advanced Message Queuing Protocol (AMQP) Version 1.0," OASIS Standard, Oct. 2012. http://docs.oasis-open.org/amqp/core/v1.0/os/amqp-core-complete-v1.0-os.pdf
[7] "XMPP | The Universal Messaging Standard." https://xmpp.org
[8] P. Saint-Andre, "Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence," RFC 6121. Internet Engineering Task Force (IETF), Mar. 2011.
[9] Object Management Group (OMG), "OMG Data Distribution Service (DDS) Version 1.4," Apr. 2015. https://www.omg.org/spec/DDS/1.4/PDF
[10] E. Gamess, T. N. Ford, and M. Trifas, "Performance Evaluation of a Widely used Implementation of the MQTT Protocol with Large Payloads in Normal Operation and Under a DoS Attack," in Proceedings of the 2021 ACM Southeast Conference (ACMSE 2021), May 2021, pp. 154–162. doi: 10.1145/3409334.3452067.
[11] Takanorig, "MQTT-Bench: Benchmark Tool for MQTT Broker." https://github.com/takanorig/mqtt-bench
[12] S. Imane, M. Tomader, and H. Nabil, "Comparison between CoAP and MQTT in Smart Healthcare and Some Threats," in Proceedings of the 2018 International Symposium on Advanced Electrical and Communication Technologies (ISAECT 2018), Nov. 2018, pp. 1–4. doi: 10.1109/ISAECT.2018.8618698.
[13] H. W. van der Westhuizen and G. P. Hancke, "Practical Comparison between CoAP and MQTT - Sensor to Server Level," in Proceedings of the 2018 IEEE Wireless Advanced Conference (WiAd 2018), Jun. 2018, pp. 1–6. doi: 10.1109/WIAD.2018.8588443.
[14] N. Cameron, Electronics Projects with the ESP8266 and ESP32, 1st edition. Apress, 2020. doi: 10.1007/978-1-4842-6336-5.
[15] A. Lintermann, D. Pleiter, and W. Schröder, "Performance of ODROID-MC1 for Scientific Flow Problems," Future Generation Computer Systems, vol. 95, pp. 149–162, 2019, doi: 10.1016/j.future.2018.12.059.
[16] B. H. Çorak, F. Y. Okay, M. Güzel, Ş. Murt, and S. Ozdemir, "Comparative Analysis of IoT Communication Protocols," in Proceedings of the 2018 International Symposium on Networks,

Computers and Communications (ISNCC 2018), Jun. 2018, pp. 1–6. doi: 10.1109/ISNCC.2018.8530963.

[17] R. Banno, K. Ohsawa, Y. Kitagawa, T. Takada, and T. Yoshizawa, "Measuring Performance of MQTT v5.0 Brokers with MQTTLoader," in Proceedings of the 2021 IEEE 18th Consumer Communications and Networking Conference (CCNC 2021), Jan. 2021, pp. 1–2. doi: 10.1109/CCNC49032.2021.9369467.

[18] R. A. Light, "Mosquitto: Server and Client Implementation of the MQTT Protocol," The Journal of Open Source Software, vol. 2, no. 13, pp. 1–2, 2017, doi: 10.21105/joss.00265.

[19] M. Bender, E. Kirdan, M. O. Pahl, and G. Carle, "Open-source MQTT evaluation," 2021. doi: 10.1109/CCNC49032.2021.9369499.

[20] Z. Laaroussi and O. Novo, "A Performance Analysis of the Security Communication in CoAP and MQTT," in Proceedings of the 2021 IEEE 18th Consumer Communications and Networking Conference (CCNC 2021), Jan. 2021, pp. 1–6. doi: 10.1109/CCNC49032.2021.9369565.

[21] A. Larmo, F. del Carpio, P. Arvidson, and R. Chirikov, "Comparison of CoAP and MQTT Performance over Capillary Radios," in Proceedings of the 2018 Global Internet of Things Summit (GIoTS 2018), Jun. 2018, pp. 1–6. doi: 10.1109/GIOTS.2018.8534576.

[22] W. Pipatsakulroj, V. Visoottiviseth, and R. Takano, "muMQ: A Lightweight and Scalable MQTT Broker," in Proceedings of the 2017 IEEE Workshop on Local and Metropolitan Area Networks (LANMAN 2017), Jun. 2017, vol. 2017-June, pp. 1–6. doi: 10.1109/LANMAN.2017.7972165.

[23] D. B. C. Lima, R. M. B. da Silva Lima, D. de Farias Medeiros, R. I. S. Pereira, C. P. de Souza, and O. Baiocchi, "A Performance Evaluation of Raspberry Pi Zero W Based Gateway Running MQTT Broker for IoT," in Proceedings of the 2019 IEEE 10th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON 2019), Oct. 2019, pp. 0076–0081. doi: 10.1109/IEMCON.2019.8936206.

[24] Y. Guamán, G. Ninahualpa, G. Salazar, and T. Guarda, "Comparative Performance Analysis between MQTT and CoAP Protocols for IoT with Raspberry Pi 3 in IEEE 802.11 Environments," in Proceedings of the 2020 15th Iberian Conference on Information Systems and Technologies (CISTI 2020), Jun. 2020, pp. 1–6. doi: 10.23919/CISTI49556.2020.9140905.

[25] E. Baranauskas, J. Toldinas, and B. Lozinskis, "Evaluation of the Impact on Energy Consumption of MQTT Protocol over TLS," in CEUR Workshop Proceedings, May 2019, pp. 56–60.

[26] "Hping3 Homepage." http://www.hping.org/hping3.html

[27] A. Banks and R. Gupta, "MQTT Version 3.1.1," OASIS Standard, Oct. 2014. http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.pdf

[28] ISO, "ISO/IEC 20922:2016 - Information Technology - Message Queuing Telemetry Transport (MQTT) v3.1.1," International Organization for Standardization, 2016, [Online]. Available: https://www.iso.org/standard/69466.html

[29] A. Banks, E. Briggs, K. Borgendale, and R. Gupta, "MQTT Version 5.0," OASIS Standard, Mar. 2019. https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.pdf

[30] T. Dierks and C. Allen, "The TLS Protocol Version 1.0," RFC 2246. Internet Engineering Task Force (IETF), Jan. 1999.

[31] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol - Version 1.2," RFC 5246. Internet Engineering Task Force (IETF), Aug. 2008.

[32] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3," RFC 8446. Internet Engineering Task Force (IETF), Aug. 2018.

[33] R. Grimmick, "TLS vs SSL: What's the Difference & How it Works | Varonis," Sep. 2021. https://blogvaronis2.wpengine.com/tls-vs-ssl

[34] "X.509: Information Technology - Open Systems Interconnection - The Directory: Public-key and Attribute Certificate Frameworks." https://www.itu.int/rec/T-REC-X.509

[35] S. Shapsough, F. Aloul, and I. A. Zualkernan, "Securing Low-Resource Edge Devices for IoT Systems," in Proceedings of the 2018 International Symposium in Sensing and Instrumentation in IoT Era (ISSI 2018), Sep. 2018, pp. 1–4. doi: 10.1109/ISSI.2018.8538135.

[36] T. Klosowski, "Why We Love the Raspberry Pi | Reviews by Wirecutter," Nov. 2021. https://www.nytimes.com/wirecutter/reviews/raspberry-pi

[37] "Teach, Learn, and Make with Raspberry Pi." https://www.raspberrypi.org

[38] "Raspberry Pi - About Us." https://www.raspberrypi.com/about

[39] "Raspberry Pi OS – Raspberry Pi." https://www.raspberrypi.com/software

[40] "Raspberry Pi Forums - Index Page." https://forums.raspberrypi.com

[41] "Raspberry Pi Zero W – Raspberry Pi." https://www.raspberrypi.com/products/raspberry-pi-zero-w
[42] "Raspberry Pi Zero 2 W – Raspberry Pi." https://www.raspberrypi.com/products/raspberry-pi-zero-2-w
[43] "Raspberry Pi 3 Model B – Raspberry Pi." https://www.raspberrypi.com/products/raspberry-pi-3-model-b
[44] "Eclipse Paho | The Eclipse Foundation." https://www.eclipse.org/paho