A Comparison of Publish-Subscribe and Client-Server Models for Streaming IoT Telemetry Data

Olasupo Ajayi(✉), Antoine Bagula, Joshua Bode, and Moegammad Damon

Department of Computer Science, University of the Western Cape, Cape Town, South Africa
ooajayi@uwc.ac.za

Abstract. In recent years, the Internet of Things (IoT) has become a household name both in research and commercial domains. There are numerous practical applications of IoT, ranging from sophisticated solutions in big factories to simple smart homes devices, such as temperature monitors. Common to these solutions is the need to transmit data from a sensing source to a different location where the data is read, processed, analyzed, or simply stored. Numerous models and protocols have been developed to ferry IoT data, including queueing, client-server, and publish-subscribe models, all with their merits and demerits. In this paper, a comparison of two models for transmitting live/streaming IoT telemetry data is done. The Message Queue Telemetry Transport (MQTT), a publish-subscribe model, is compared with WebSocket, a client-server model, in terms of throughput, round trip time and system utilization. Obtained results reveal that MQTT is more suited for transmitting live IoT data than WebSocket, as it had better throughput, utilized less system resource, however it was slightly slower than WebSocket.

Keywords: Client-Server · Internet of Things · MQTT · Publish-Subscribe · Round Trip Time · WebSocket

1 Introduction

Advances in technology has enabled easy communication between humans and machines (Human to machine or H2M), machines and machines (M2M), and machines and humans (M2H). The Internet of Things (IoT) is one of the primary enablers of this communication [1]. By fitting everyday objects or "things" with sensing, actuating, and communication abilities, IoT has created a new technology domain which we refer to as the "smart ecosystem". This ecosystem ranges from simple devices, such as temperature sensors, that measure and communicate ambient temperature readings, to more advanced and complex solutions capable of sensing, acting, and mimicking human decision making through artificial intelligence. Regardless of the use case, IoT devices generally collect data about/from a source and transmit same to a different (remote) destination. There are several protocols used in transferring telemetry data, including Advanced

© ICST Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2023
Published by Springer Nature Switzerland AG 2023. All Rights Reserved
M. Masinde and A. Bagula (Eds.): AFRICATEK 2022, LNICST 503, pp. 129–139, 2023.
https://doi.org/10.1007/978-3-031-35883-8_9

Message Queuing Protocol (AMQP)/RabbitMQ, Message Queuing Telemetry Transport (MQTT), Constrained Application Protocol (CoAP), HyperText Transfer Protocols (HTTP), most of which have been reviewed in [2]. These protocols can broadly be categorized into Client-Server or Publish-Subscribe architecture.

The Client-Server architecture is one in which one or more clients connect to central server. In this architecture, clients request for services or information from the server, which in turn actively waits for requests from clients and response to them. Common roles of the server include web hosting, database server, and application server. Figure 1a gives a high-level depiction of a Client-Server architecture. The Publish-Subscribe architecture on the other hand is an architecture made up of three primary components – the publisher(s), the subscriber(s), and the broker, as shown in Fig. 1b. Unlike the Client-Server where clients send request directly to the server, in publish-subscribe model the senders (publishers) are usually unaware of the type or number of receivers (subscribers), hence, does not send messages directly to them. Instead, messages from the publishers are categorized into "topics" and sent to an intermediary broker. Similarly, the subscribers are often also unaware of the publishers and simply subscribe to topics of interest on the broker. Thus, the broker serves as an intermediary between the publisher and the subscribers, handling tasks such as filtration, storage, message forwarding, and message delivery management.

In this work, the authors seek to compare the performance of the classic client-server architecture with the publish-subscribe architecture in streaming IoT telemetry data from source to destination. For the client-server architecture, the WebSocket is considered as a use case, while MQTT is considered for the publish-subscribe architecture. These protocols are briefly described in the following subsections.

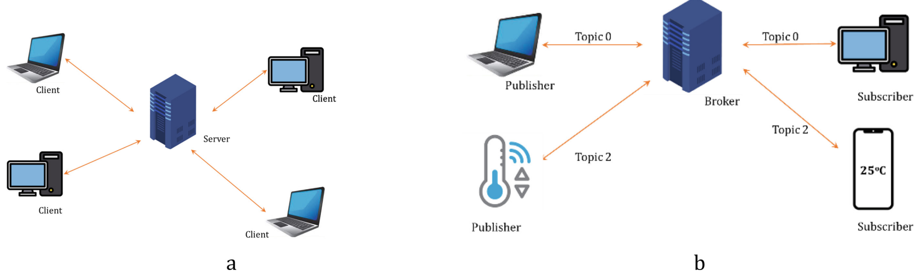

Fig. 1. a: Client-Server Architecture, b: Publish-Subscribe Architecture

1.1 WebSocket

WebSocket is a Transmission Control Protocol (TCP) based client-server model, that offers full duplex communication between clients and server [3]. Like HTTP, WebSocket is based on TCP, hence, the basic processes of creating a secure communication channel between the client and server are also observed with WebSocket. These include link establishment/handshaking, the server actively waiting for requests, clients initiating

requests, server responding, and the final acknowledgment sent by the client. However, unlike in HTTP, WebSocket connections are usually persistent, thus, avoiding the need to re-establish connections for each client's request. This significantly improves the performance of WebSockets compared to non-persistent HTTP [4]. A major disadvantage of the WebSocket is that the client and server must remain tightly coupled to each other. In essence, both the client and server must be available at the same time for information exchange to occur.

1.2 MQTT

Message Queuing Telemetry Transport (MQTT) is a lightweight messaging protocol that employs the publish-subscribe model. Since it is based on the publish-subscribe architecture, the sender and receiver are decoupled from each other and can operate independent of the other. The broker serves as a "binding" or intermediary component via which communication takes place. The broker is synonymous to a message notice board onto which publishers "write" messages under specific topics, and subscribers "read" messages under topics that interest them. Moreover, the decoupled nature of the various components of MQTT improves its scalability. Being decoupled and independent of each other there is a high risk of message loss in most publish-subscribe based protocols. These protocols define 3 levels of QoS, viz., Level 0 (which provides no guarantee of message delivery), Level 1 (which guarantees that messages sent will be received at least once during a transmission process), and Level 2 (which ensures that messages are delivered and decoded by the subscribers) [5].

2 Related Works

MQTT is a protocol that has been well used in numerous IoT related projects. In [6], the author carried out a comparative evaluation of several publicly and locally deployed MQTT brokers. For the public brokers, they considered Mosquitto, HiveMQ, and Bevywise, while ActiveMQ, Bevywise MQTT, Mosquitto, and RabbitMQ were deployed locally. Obtained results show that for the public brokers, Mosquitto performed best for Quality of Service (QoS) levels 0 and 2 in terms of message throughput and delivery times. For the local brokers, HiveMQ and Mosquitto were the top performers in terms of message delivery time. In [7], the authors compared the performance of two telemetry protocols – MQTT and CoAP, both of which are based on different architectures. MQTT which runs on Transmission Control Protocol (TCP) and utilizes the publish-subscribe architecture, while CoAP runs on User Datagram Protocol (UDP) and utilizes a Request-Response model (like a scaled down HTTP). CoAP also uses URI (Universal Resource Identifiers) rather than Topics as used in MQTT. A common gateway, compatible with both protocols, was developed by the authors and used to compare them, in terms of packet loss and delays, message throughput, and message size. The performance of both protocols was network dependent as both outperformed each other at different conditions. Ref. [8] compared MQTT to classic HTTP based on three criteria, i.) Message consumption, for which MQTT performed 25 times faster than HTTP. ii.) Data throughput, wherein MQTT was shown to be significantly faster and more frugal

on traffic utilization. iii.) Energy consumption, which was tested using a Raspberry pi, and showed that MQTT utilized 22% less energy than HTTP for the same task. In [9], AMQP was compared with MQTT in a dynamic environment using message loss, delay, jitter, and saturation boundaries as metrics. Experimental results showed that i.) AMQP delivered messages in a last in first out order, while MQTT used the first in first out order. ii.) There were no message losses across both protocols except when the publisher was overloaded. iii.) AMQP was more secure, while MQTT was more scalable. Ref. [10] compared three distributed MQTT brokers (EMQX, HiveMQ, VerneMQ) on multiple criteria including throughput, scalability, and resilience. They concluded that EMQX had the highest message throughput, while HiveMQ was the most resilient and extensible. In [11], several IoT protocols were compared including AMQP, CoAP, MQTT, REST, XMPP (Extensible Messaging and Presence Protocol), and WebSocket, based on various criteria, including security and quality of service (QoS). In term of architectures, CoAP and REST are based on Request/Response architecture, MQTT, AMQP are Publish/Subscribe, XMPP employs both Request/Response and Publish/Subscribe, while WebSocket is based on a Client/Server architecture. All protocols provide some form of QoS, except XMPP, REST, and WebSocket. The authors also reported that all protocols supported TLS (Transport Layer Security), except CoAP and REST which use Datagram Transport Layer Security (DTLS) and HTTPS respectively. Using the ESP8266, Oliveria *et al.* [12] compared the performance of MQTT and WebSocket protocols based on round-trip time (RTT). They concluded that WebSocket was better than MQTT for applications that required low RTT, while MQTT utilized lower memory space.

3 Methodology

This section presents the methodology and experimental setup used in this work for both WebSocket and MQTT. Though Fig. 1 show a depiction of generic client-server and publish-subscribe architectures, for this work, we only used subsets of these architecture as shown in Figs. 2 and 3. Our network set up consisted of only one client (or publisher) and one server (or subscriber) because we wanted to reduce the impact of uncontrollable factors, such as network delays, bandwidth congestion, and bottlenecks (network and server), which might emanate from using multiple nodes, active devices, and service providers. We believe our chosen set up allows us to compare both protocols in a "relatively controlled" environment.

For the client (or publisher), we used a Raspberry Pi version 4 B, with 4 GB RAM and 4 CPU cores running at 1.5 GHz [13]. This was chosen because of its GPIO ports (to connect sensors) and its stripped-down Linux operating system, which helps minimize the impact of complex operating system overheads. For the server (subscriber), a computer with an 8 core Intel Core i7 generation 10 CPU, 16 GB of RAM and running Windows 11 was used. This system was more than capable to act as a server for a single client.

Figure 2 shows the deployment setup for WebSocket. A Raspberry Pi was used to collect readings from the sensors, which in our use case were temperature and humidity collected from the Sensor Hat emu. Telemetry data were then sent to the WebSocket server running on the Windows PC. The PC also ran the SQLite server, onto

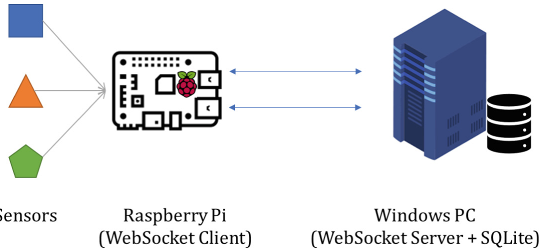

Fig. 2. WebSocket architecture

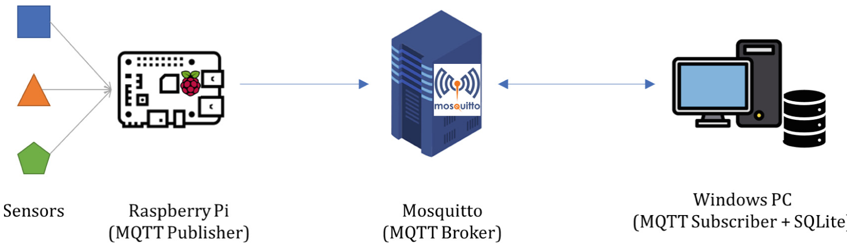

Fig. 3. MQTT architecture

which the received telemetry data were saved. Two Python scripts were written, the first "WebSocket_client.py" ran on the Raspberry Pi (WebSocket client) to capture and send telemetry data to the Windows PC (WebSocket server), while the second "WebSocket_Server.py" ran on the Windows PC, to receive data from the client and save same on the SQLite database.

Figure 3 shows the MQTT architecture used in this work and comprises of a Raspberry Pi (to collect sensor readings), a broker (Mosquitto), and a Windows PC acting as the MQTT subscriber and SQLite host. Like with the WebSocket, two Python scripts "mqtt_publisher.py" and "mqtt_subscriber.py" were ran on the Raspberry Pi and Windows PC respectively.

For each deployment scenario, the same Raspberry Pi and Windows PC were used to ensure consistent results, only the Python script being run for each experiment was changed. The Raspberry Pi was set to a local IP address of 10.0.2.15, while the Windows PC was set to 192.168.2.112. For the WebSocket, port 54786 and 12000 were opened on the client (Raspberry Pi) and server (Windows PC) respectively; while for MQTT, ports 47143 and 1883 were used for the publisher (Raspberry Pi) and subscriber (Windows PC) respectively.

In the experimental setup used to test the performance of WebSocket and MQTT for IoT data streaming, the following metrics were considered:

- Throughput, which is the rate at which data are transferred between the sender and the receiver. It is measured in bits/second. We used Wireshark statistics to calculate throughput for both models.

- Round Trip Time (RTT), the time taken for a packet to travel from a sender to the receiver and back to the sender. It is measured in milliseconds (ms). The Packet Internet Gropper (PING) messages were used to obtain RTT.
- System performance, this is simply a measure of the CPU utilization (in percentage) while using any of the protocols. We obtained this by running the System Activity Report (SAR) for a fixed period. SAR is used to monitor system resources, such as CPU usage, memory, and network utilization in a Linux operating system. For this study we focused on the "%idle" time, which is the percentage of time the CPU was idle, and "%IOwait", which is the percentage of time the CPU was idle because it was waiting for one or more IO requests to be completed and there were no other tasks to schedule.

4 Results

This section presents the obtained results of the experiments carried out. For both protocols (WebSocket and MQTT), a Raspberry Pi and a Windows PC were used.

4.1 Throughput

Data required to determine the throughput were obtained from Wireshark's conversations statistics [14]. This shows the packet count, packet size, packet direction, start and end times, and average data size transmitted across two end points. We obtained throughput by dividing the total number message size (in bits) by total time. Graphical depictions of the obtained results are shown in Figs. 4 and 5.

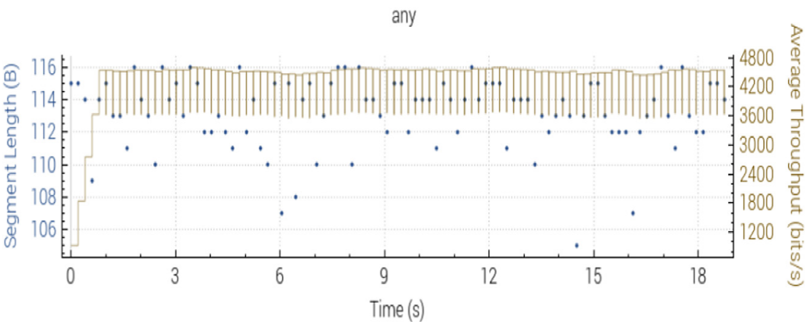

Fig. 4. Average Throughput for WebSocket

When comparing both graphs, Fig. 5 shows that MQTT had a higher throughput (5100 bits/s on the average) compared to WebSocket's average of 4500 bits/s. This implies that about 13% more packets were processed per unit time when MQTT was used versus when WebSocket was used.

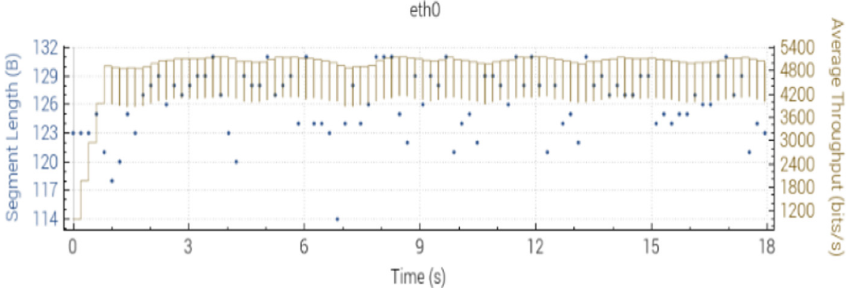

Fig. 5. Average Throughput for MQTT

4.2 Round Trip Time (RTT)

For both protocols, we measured the time it took to send messages from the Raspberry Pi to the Windows PC. We used ICMP Echo and Echo Reply messages (PING) to calculate RTT. For accuracy, we ran PING several times, with each run lasting 20 s. We then plotted the average RTT for both protocols as depicted in Figs. 6 and 7.

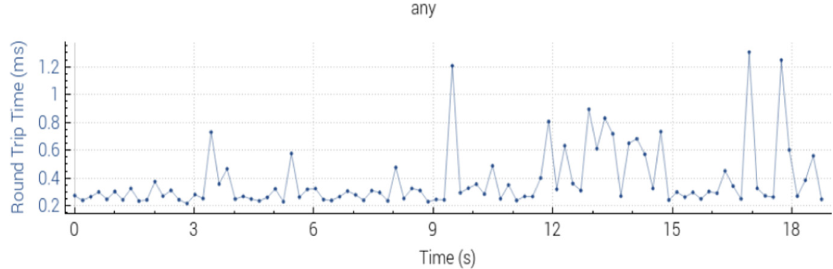

Fig. 6. RTT for WebSocket

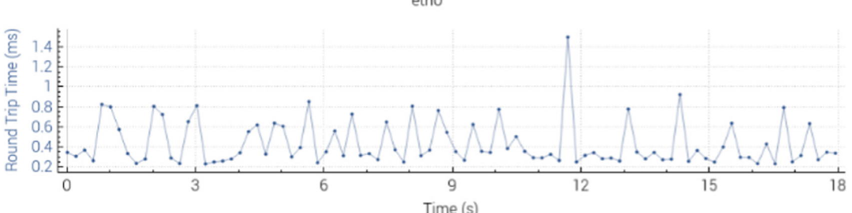

Fig. 7. RTT for MQTT

Figures 6 and 7 show the time it takes for packets to travel from the Raspberry Pi to the Windows PC and back to the Raspberry Pi, using WebSocket and MQTT respectively. The figures reveal that on the average, WebSocket is slightly faster than MQTT. This is expected as MQTT has an intermediary broker between the publisher (Raspberry Pi) and the subscriber (Windows PC), which though transparent, still runs a "stored and forward" process for all arriving messages. Despite this intermediary broker, MQTT is only marginally slower than WebSocket.

4.3 System Load

To determine the impact of both protocols on the system resources, we performed system utilization tests while running the respective protocols. Since the Raspberry Pi runs a Linux operating system, we used SAR (System Activity Report) to obtain this report. SAR (System Activity Report) is used in Linux to monitor system resource such as CPU, Memory, Disk usage, I/O activities, etc. Two metrics were of utmost importance, namely the average CPU load (%idle) and the IO wait time (%IOwait). To achieve obtained this we ran the "*sar -u 2 600*" command, which gave system reports every 2 s for 20 min and took the average values reported.

Figures 8, 9 and 10 show snapshots of the CPU load of the Raspberry Pi. The last column (in red text) represents the average load of all the Raspberry Pi's 4 CPUs, i.e., %idle, while the third to the last represents the %IOwait times.

Average:	all	0.10	0.00	0.35	0.52	0.00	99.03
Average:	all	0.36	0.00	0.36	0.07	0.00	99.21
Average:	all	0.31	0.00	0.51	0.04	0.00	99.14
Average:	all	0.37	0.00	0.66	0.02	0.00	98.96
Average:	all	0.44	0.00	0.49	0.02	0.00	99.05
Average:	all	0.49	0.00	0.38	0.02	0.00	99.11
Average:	all	0.42	0.00	0.58	0.07	0.00	98.92
Average:	all	0.20	0.00	1.48	0.02	0.00	98.30
Average:	all	0.40	0.00	0.73	0.02	0.00	98.86
Average:	all	0.27	0.00	0.49	0.02	0.00	99.22

Fig. 8. System load at idle state – CPU = 1.02% average load, Waiting for IO = 0.08% average

Average:	all	1.36	0.00	4.70	2.89	0.00	91.05
Average:	all	0.80	0.00	2.50	12.35	0.00	84.35
Average:	all	1.26	0.00	2.92	2.72	0.00	93.10
Average:	all	1.11	0.00	2.71	5.37	0.00	90.81
Average:	all	0.76	0.00	2.26	7.09	0.00	89.90
Average:	all	0.84	0.00	2.44	7.63	0.00	89.10
Average:	all	0.10	0.00	0.35	0.52	0.00	99.03
Average:	all	0.18	0.00	0.48	0.00	0.00	99.34
Average:	all	0.73	0.00	2.82	0.71	0.00	95.74
Average:	all	1.07	0.00	2.78	0.74	0.00	95.41

Fig. 9. System load when running WebSocket – CPU = 7.02% average load, Waiting for IO = 4% average

Average:	all	0.83	0.00	3.23	0.85	0.00	95.09
Average:	all	1.09	0.00	2.52	0.61	0.00	95.77
Average:	all	0.95	0.00	2.58	0.65	0.00	95.83
Average:	all	1.05	0.00	2.36	0.83	0.00	95.75
Average:	all	1.15	0.00	3.91	0.61	0.00	94.33
Average:	all	1.33	0.00	3.61	0.48	0.00	94.58
Average:	all	1.37	0.00	3.22	0.75	0.00	94.66
Average:	all	1.22	0.00	2.95	0.63	0.00	95.20
Average:	all	1.00	0.00	2.89	0.48	0.00	95.62
Average:	all	1.08	0.00	2.94	0.67	0.00	95.31

Fig. 10. System load while running MQTT – CPU = 4.78% average load, Waiting for IO = 0.66% average

The figures show that when running only the MQTT publisher script, the CPUs were less loaded at 4.78% compared to the WebSocket script's load of 7.02%. When the IO wait times were considered, MQTT is significantly better than WebSocket, as the CPUs were only idle 0.66% of the time waiting for IO request to finish, compared to 4% of the time when WebSocket was used. This is most likely because WebSocket keeps a persistent connection open with the server and must wait for responses from the server. MQTT does not have this issue as the components are decoupled as described in the Sect. 1.2. Though "%IOwait" might not only be because of the CPU waiting for packets to arrive, in our use case we tried to ensure that no other processes that might cause IO interrupt were running on the Raspberry Pi when the measurements were taken.

From the three results shown, we can conclude that MQTT is a better choice for streaming IoT telemetry data than WebSocket. This is because MQTT can transmit more data per unit time (higher throughput), uses less CPU resources, and is only marginally slower than WebSocket. Beyond this, the decoupled architecture of MQTT is much suited for IoT applications, where the sensors might be installed in a human body (e.g., heart monitors), or remote locations, such as underground or offshore in water bodies, while the broker and subscribers are in globally dispersed locations. Waiting for the client or server to be available before transmitting, as done with the WebSocket, would be detrimental.

5 Conclusion and Further Work

The objective of this paper was to compare the performance and suitability of Client-Server and Publish-Subscribe architectures for streaming IoT telemetry data. Two protocols were considered for the comparison, namely WebSocket, a client server protocol, and MQTT, a publish-subscribe protocol. The two protocols were tested using live data from sensors connected to a Raspberry Pi, using throughput, round trip time (RTT) and system load as metrics. A combination of Python scripts, Wireshark and Linux scripts were used to carry out the tests and obtained results show that MQTT performed better than WebSocket in all but RTT. The delay in RTT was very marginal and attributed to the intermediary broker.

This paper is a work-in-progress report and represents only a subset of bigger project. In future, the authors seek to expand the testing to include multiple clients, publishers, and

subscribers. The performance of single versus multiple brokers in multi-publisher-multi-subscriber scenarios is also a planned work. Moreover, real world testing in scenarios such as those described in [15, 16], using additional metrics and other protocols, such as Data Distribution Service (which is tailored for M2M communication [17]), might also be opportunities that the authors might consider exploring in future works.

References

1. Want, R., Schilit, B.N., Jenson, S.: Enabling the internet of things. Computer **48**(1), 28–35 (2015)
2. Machaka, P., Ajayi, O., Maluleke, H., Kahenga, F., Bagula, A., Kyamakya, K.: Modelling DDoS Attacks in IoT Networks Using Machine Learning (2021). arXiv preprint arXiv:2112.05477
3. Fette, I., Melnikov, A.: The WebSocket protocol (No. rfc6455) (2011)
4. Kurose J., Ross, K.: Computer Networking: A Top-Down Approach, Pearson (2016)
5. Stansberry, J.: MQTT and CoAP: Underlying Protocols for the IoT. Electron. Des, pp. 1–8 (2015)
6. Mishra, B.: Performance evaluation of MQTT broker servers. In: Gervasi, O., et al. (eds.) ICCSA 2018. LNCS, vol. 10963, pp. 599–609. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-95171-3_47
7. Thangavel, D., Ma, X., Valera, A., Tan H., Tan, C.: Performance evaluation of MQTT and CoAP via a common middleware. In: IEEE 9th International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), pp. 1–6 (2014). https://doi.org/10.1109/ISSNIP.2014.6827678
8. Bartnitsky, J.: HTTP vs MQTT performance tests (2018). https://flespi.com/blog/http-vs-mqtt-performance-tests. Accessed 25July 2022
9. Luzuriaga, J., Perez, M., Boronat, P., Cano, J., Calafate, C., Manzoni, P.: A comparative evaluation of AMQP and MQTT protocols over unstable and mobile networks. In: 12th IEEE Consumer Communications and Networking Conf. (CCNC2015), pp. 931–936 (2015).https://doi.org/10.1109/CCNC.2015.7158101
10. Koziolek, H., Grüner, S., Rückert, J.: A comparison of mqtt brokers for distributed iot edge computing. In: Jansen, A., Malavolta, I., Muccini, H., Ozkaya, I., Zimmermann, O. (eds.) Software Architecture: 14th European Conference, ECSA 2020, L'Aquila, Italy, September 14–18, 2020, Proceedings, pp. 352–368. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-58923-3_23
11. El Ouadghiri, M., Aghoutane, B., El Farissi, N.: Communication model in the Internet of Things. Procedia Comput. Sci. **1**(177), 72–77 (2020)
12. Oliveira, G., Costa, D., Cavalcanti, R., Oliveira, J., et al.: Comparison between MQTT and WebSocket protocols for Iot applications using ESP8266. In: 2018 Workshop on Metrology for Industry 4.0 and IoT 2018 Apr 16, pp. 236–241. IEEE (2018)
13. Raspberry Pi.com. Raspberry Pi 4 Tech Specs. https://www.raspberrypi.com/products/raspberry-pi-4-model-b/specifications/. Accessed 13 Nov 2022
14. Lamping, U., Warnicke, E.: Wireshark user's guide. Interface **4**(6), 1 (2004). https://www.wireshark.org/docs/wsug_html_chunked/ Accessed 13/11/2022

15. Ajayi, O.O., Bagula, A.B., Maluleke, H.C., Gaffoor, Z., Jovanovic, N., Pietersen, K.C.: Water-Net: a network for monitoring and assessing water quality for drinking and irrigation purposes. IEEE Access **10**, 48318–48337 (2022)
16. Mandava, T., Chen, S., Isafiade, O., Bagula, A.: An iot middleware for air pollution monitoring in smart cities: a situation recognition model. In: Proceedings of the IST Africa 2018 Conference, Gabarone, Botswana, pp. 9–11 (2018)
17. Kang, W., Kapitanova, K., Son, S.H.: RDDS: a real-time data distribution service for cyber-physical systems. IEEE Trans. Industr. Inf. **8**(2), 393–405 (2012)