

Performance Evaluation of MQTT Broker Servers Deployed in the Cloud

Fernando Pazos

Technology and Administration Department. National University of Avellaneda.
Mario Bravo 1460, Piñeyro, B1868, Buenos Aires, Argentina.
fapazos@undav.edu.ar

Abstract. Communication between devices on a network requires the use of protocols. On internet there are well known protocols that can be used both in the architecture of a server with multiple clients as well as in a machine to machine (M2M) communication. In Internet of Things (IoT) applications, network communication can be supervised by a server denoted as broker, and the most widely used application layer protocol for this purpose is MQTT (Message-Queuing Telemetry Transport). This paper compares the performance of eight publicly available MQTT brokers deployed in the cloud in three experiments under different stress conditions. The goal is to choose the most suitable broker to be used in the communication between a Cubesat-type nanosatellite and the land terminal.

Keywords: Internet of Things · MQTT protocol · MQTT brokers

1 Introduction

The present work is part of a larger project implemented by a consortium of Argentine universities that aims to put a nanosatellite into orbit. The objective is to promote a space laboratory for the provision of Internet of Things (IoT) services. In particular, the National University of Avellaneda is responsible for the study and recommendation of the most suitable On-Board Computer (OBC) for use in IoT, as well as the study of all the relevant aspects about the communication between the nanosatellite and the land terminal (see [13] for details).

A crucial issue to be resolved in order to put a nanosatellite into orbit is to establish reliable communication with the Earth terminal. An on-board computer (OBC) must store information collected by the sensors installed on the satellite and send it whenever requested by a client. As the connection time is only a few minutes per day according to the orbital period, the OBC must store all the data collected daily and send it securely during the connection time. This issuance is carried out via the internet [13].

There exist several architectures for connecting an IoT device with another device, either client or server. In the present project a machine-to-machine (M2M) communication between the satellite and the land terminal, which must receive and store the data sent for further analysis, must be implemented.

In IoT applications based on the TCP/IP model, some of the most commonly used application layer protocols include CoAP, AMQP, XMPP and HTTP, however, in M2M communication the most widely used is MQTT (Message-Queuing Telemetry Transport) [18, 2]. It is based on the asynchronous publishing/subscribing topology of small messages, typically of a few bytes, which makes this protocol suitable for connecting remote devices. It can be implemented with an extremely lightweight code, requires little bandwidth and is very energy efficient, so it can be deployed on microcontrollers and devices having limited processing capabilities and memory [21]. This protocol is widely used in a number of industries such as automotive, oil and gas among many others (see details in [18, 2, 4, 21]).

1.1 The MQTT protocol

The implementation of a communication network using MQTT requires the control and management by a back-end server on the internet. This server is denoted as *broker*. The broker is responsible

for receiving and delivering messages sent by the clients. The architecture of the network managed by the broker is a star configuration. The clients connected to the broker play subscriber and publisher roles. The publishers send messages on a topic head to the broker, which delivers them to the subscribers that have previously subscribed to that topic [21]. Of course, communication is bidirectional, so the clients that publish in a topic can be subscribed to other topics, thus receiving the messages published by other clients.

The message format is **'topic': 'payload'**, where

- Topic: “key” or identification of the message published. The topic name is an UTF-8 encoded string used to deliver the message to the clients subscribed to it.
- Payload: string containing the message itself formatted as an array of characters.

The topic can have subtopics (separated by a slash) in a hierarchical structure. Subscribers can subscribe to an individual topic or to a set of topic using “wildcards” [4].

Fig. 1 shows a simplified star scheme of the publisher/subscriber model, where a temperature sensor publishes the measured data with the topic “temperature”; two clients subscribed to this topic receive the message.

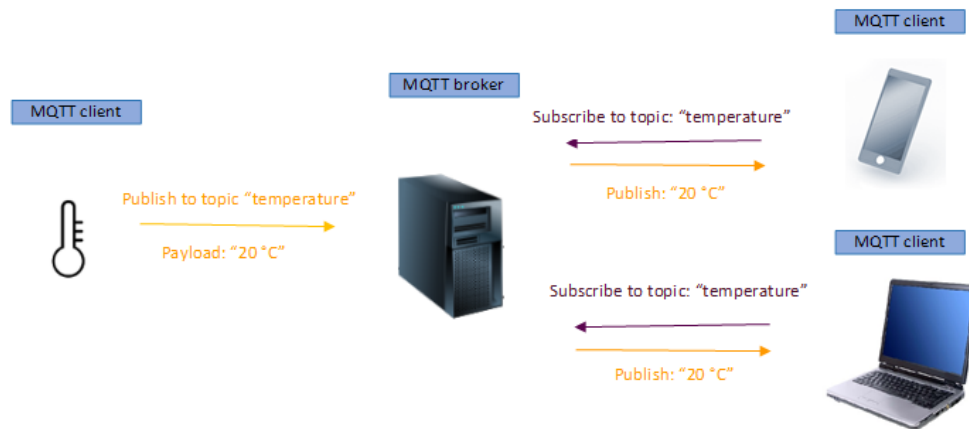


Fig. 1: Example of the publisher/subscriber configuration

MQTT runs on a TCP/IP transport layer socket using ports 1883 and 8883 for non-encrypted and encrypted communication, respectively.

MQTT supports three levels of quality of service (QoS) to ensure message transport reliability described as follows [4, 11].

- QoS 0 (At most once, or “fire and forget”): Messages are sent at most once and it does not provide guarantee delivery of a message. The sender sends the message and does not store it. The receiver does not acknowledge its receiving. Messages can be lost; there is no retransmission.
- QoS 1 (At least once): Messages are sent at least once. The sender sends a message and expects to receive an acknowledgment from the receiver. If the receiver does not acknowledge receipt, or the message is lost, the sender resends the message until acknowledgment is obtained.

- QoS 2 (Exactly once): Messages are sent exactly once by using 4-way handshaking. This is the slowest of all the levels and increases the communication load but is the best option when message duplication is not allowed.

Other parameters to be set when using the MQTT protocol include

- *keep alive* interval: maximum time for which a client must publish a message or to send a “ping” request to the broker in order not to be disconnected from the network.
- *retain-flag*: indicates whether the broker should store a message for delivery to clients that later subscribe to that topic.
- *SSL certificate*: indicates whether the communication is encrypted using the TLS (Transport Layer Security) protocol.
- *last will and testament*: topic and payload set by a client when connected. The payload is delivered to the clients subscribed to this topic when the connection with the sender is lost.

More information about the MQTT protocol can be found in [4, 2, 16, 21, 22].

1.2 MQTT Brokers

There exists a number of publicly available MQTT brokers with very diverse configurations and features. In [3] some of them are listed.

Some brokers require registration and credentials (username and password) must be provided for connection. Some of them are deployed in the cloud, whereas others require the installation of a software on a local server which will be used as broker to manage the communication between clients. Some of them present a *dashboard* on a web page where all the messages received and delivered by the broker are printed. Some brokers allow the clients to establish encrypted communication with an SSL/TLS certificate, whereas others do not.

There exist many studies comparing the efficiency of the MQTT protocol in IoT applications against that presented by other protocols. Also, many works compare the performance of several MQTT brokers under different conditions (see [14–16] and references therein).

In [10] seven brokers are analyzed from a security point of view by performing a DOS attack and information gathering techniques on the broker, and comparing the vulnerability of each one.

In [11] the performances of wired and wireless networks using the broker *mosquitto* are analyzed by measuring the end-to-end delays and message loss as a function of the QoS and the size of the payload.

In [16] the performances of six brokers (*mosquitto*, *active-MQ*, *hivemq*, *bevywise*, *verneMQ*, and *emqx*) are evaluated in terms of message processing rates. The performances are quantified in terms of *latency*, *scalability* and *availability* (see definitions in [16, s. 2.4]).

In [14] two experiments are carried out. In the first one a Raspberry Pi board publishes analogical data and a local computer receives the information. The performances of three MQTT brokers deployed in the cloud are compared by measuring the mean latency. In the second test five MQTT brokers are deployed in a local computer which also plays the role of subscriber. Another local computer is used as publisher. The performances of the brokers are compared by measuring the mean latency as a function of the QoS and the size of the payload.

The present study aims to test the performance of the services offered by some brokers, such as the transmission reliability and the response time, in order to determine the most suitable to be applied in the communication between a nanosatellite and the land terminal.

A total of twenty brokers available on internet were tested, but as they present very different configurations and in order to test under the same conditions, we choose those that meet the following criteria: being free, being deployed in the cloud, and allowing M2M communication without the mandatory use of a dashboard on a web page.

The brokers chosen were:

- saas.theakiro.com [1]
- mqtt.flespi.io [6]
- test.mosquitto.org [17]
- broker.hivemq.org [8]
- mqtt.fluux.io [7]
- broker.emqx.io [5]
- broker.mqttdashboard.com [19]
- ioticos.org [9]

Some of these brokers are scalables, which means that they can increase the capacity of the service with the increase in load. The strategies for expanding the service are clustering, or the ability to share the service across several servers or cores, and bridging, where the messages are delivered to other brokers when the processing time exceeds a maximum limit [4, 16]. For example *hivemq* and *emqx* are scalables, whereas *mosquitto* is not.

In this work, the results obtained from three experiments carried out under different conditions are presented in order to quantify the performance of the eight MQTT brokers chosen. The performance evaluation considers the *latency* (time elapsed between the delivery of a message and its reception) and the *reliability* (ability to deliver all received messages, without message loss). The goal is to determine the most suitable broker to be used in the communication between a nanosatellite and the land terminal.

The main contribution of the present work is to compare the performance of some brokers not mentioned in the references cited (such as *ioticos* and *akiro*) making the appropriate tests for the project for which the selected broker will be used.

2 Experimental results

In this section, the results of three experiments carried out in order to quantify the performance of the brokers are presented.

The experiments used a WiFi internet connection of 5 GHz, with ping of *53ms* and *54mbps* and *78mbps* of download/upload speed, respectively.

In all the test performed, a local computer executes a VCL application specially developed for these tests in the RAD Studio 10.2 IDE. The MQTT protocol has been provided using TMS MQTT components [22] in the script. Fig. 2 shows the screen of the application.

2.1 First test: *jitter* evaluation

The first test aims to reproduce the project framework, i.e. an IoT device sending data to a land terminal. The IoT board publishes analogical temperature values on a topic head and a local computer is subscribed to this topic. The IoT device publishes 100 strings at predetermined intervals of *100ms*, *500ms*, and *1000ms*, respectively.

The temperature sensor used is the TMP36. This sensor is able to measure temperatures between $-40^{\circ}C$ and $125^{\circ}C$ and has an output scale factor of $10mV/^{\circ}C$.

A NodeMCU v.1 board will be used as IoT device. It is based on the ESP8266 microcontroller, which allows WiFi connections at 2.4 GHz with 802.11 b/g/n protocol. Its A/D converter supports input signals between 0V and 3.3V and has a resolution of 10 bits.

The subscriber is a Dell Inspiron 5557 computer with processor Intel(R) Core(TM) i7-6500U CPU @ 2.50GHz, 8GB of RAM memory, and a 64 bits Windows 10 Pro as operating system.

Fig. 3 shows the layout of the system used in this test.

The MQTT protocol on the IoT board is provided by the library Pubsubclient.h 2.8 [20]. This is very simple to use. Clients can publish QoS 0 messages and can subscribe at QoS 0 or QoS 1. The maximum message size, including the header is 256 bytes (although it is configurable). It supports last

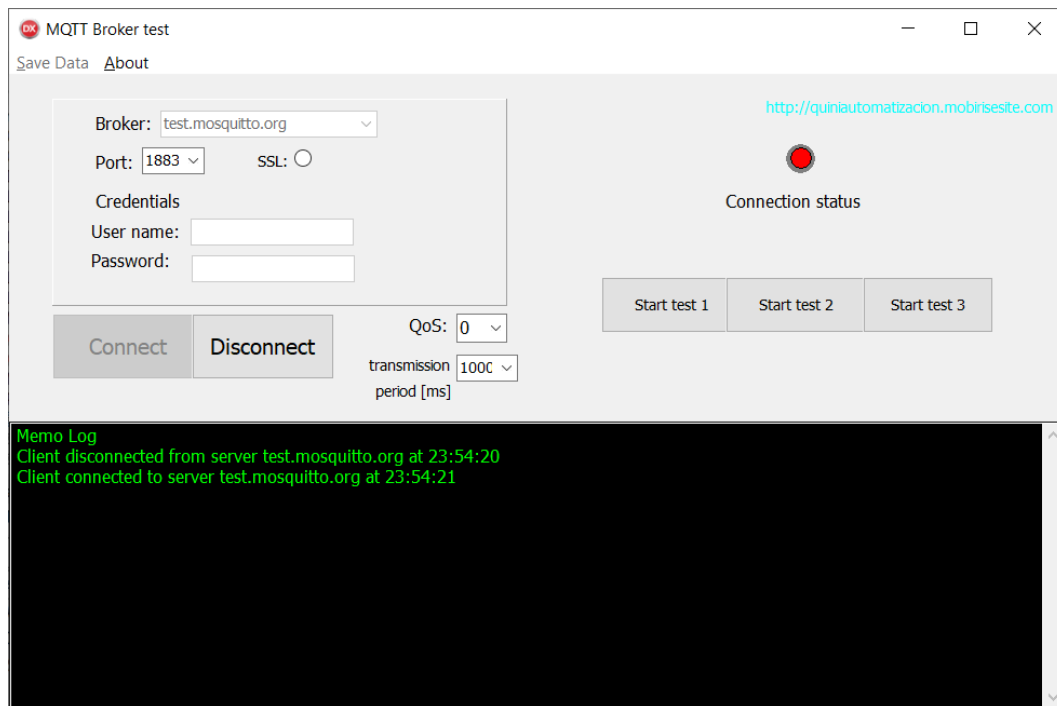


Fig. 2: Application executed by the computer in the tests performed

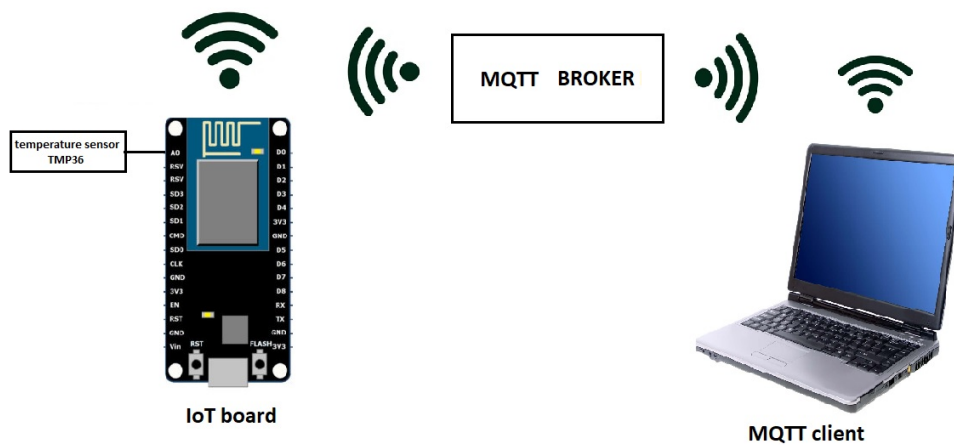


Fig. 3: System layout in the first and second tests

will and testament, and allows to set the retain flag and to configure the keep alive interval (which is 15 seconds by default). In this first test we use QoS 0, port number 1883 and no testament.

In this experiment, the computer first publishes a message indicating the required transmission interval ($100ms$, $500ms$, or $1000ms$) followed by a start command. Once the IoT board receives these messages, it starts to publish 100 temperature measurements at the specified intervals.

Due to the asynchrony between the internal clocks of the clients, it is not possible to obtain an accurate latency value. Instead, the time between the messages received (denoted as *jitter* in [12]) is measured. This is not affected by the asynchrony between the clients, since we take as the reference clock only one of them, namely the subscriber client clock [12].

Table 1 shows the results obtained in this test.

Table 1: **Test 1.** Mean time between messages reception and standard deviation for each transmission interval [ms]

time interval [ms]	1000		500		100	
	mean	std	mean	std	mean	std
saas.theakiro.com	1000.8	185.87	500.99	26.52	392.08	55.57
mqtt.flespi.io	1001.2	188.47	501.12	18.02	102.22	119.45
test.mosquitto.org	1001.1	18.50	501.01	15.62	100.89	110.7
broker.hivemq.org	1000.9	13.84	500.97	10.90	103.11	116.95
mqtt.fluux.io	1001	11.35	500.89	13.03	101.43	77.34
broker.emqx.io	1001.11	7.79	500.9	10.74	190.41	178.79
broker.mqttdashboard.com	1000.9	10.53	500.97	15.74	102.19	114.86
ioticos.org	1001.5	90.12	500.25	67.84	101.4	48.06

In this test there was no message loss. Regularity in the message reception indicates the broker's ability to handle messages at these intervals, which in turn indicates the reliability of the communication between the board and the local client. With the smallest transmission interval, $100ms$, some brokers like *mqttdashboard* seem to block sometimes, so they present large standard deviations. The broker *fluux* presents the smallest standard deviation on average in the three intervals tested.

2.2 Second test: mean latency between an IoT board and a client

In the second test the latency presented by the brokers will be measured. With this purpose, the computer publishes 100 short messages at predetermined intervals on a topic head. The IoT board receives the messages and simply republishes them on another topic head, which the computer is subscribed to. Latency is measured as the time elapsed from the publication of the message to its reception by the computer. Note that asynchrony between both of the clients is irrelevant in this context.

The devices and the system layout are the same that those used in the first test and shown in Fig. 3.

The publication intervals also are $100ms$, $500ms$, and $1000ms$. The test will be carried out as a function of the QoS with which the computer will publish and receive the messages, while the IoT board will continue to use QoS 0.

The payload of the messages are

message n. #n°

so they have from 12 to 14 characters, depending on the number of digits of the message number. The other parameters are the same that those used in the former test.

Table 2 shows the results obtained in this test.

Table 2: **Test2**. Mean latency (top rows) and standard deviation (bottom rows) for each publication interval and each QoS [ms]. X: system collapse.

QoS time interval [ms]	QoS 0			QoS 1			QoS 2		
	1000	500	100	1000	500	100	1000	500	100
saas.theakiro.com	37544	74698	105100	35569	71290	96813	31957	66659	94927
	18084	30955	27985	17874	27611	38708	17317	32379	38810
mqtt.flespi.io	552.78	552.73	1242.4	547.15	558.05	8474.5	612.07	1601.1	11902
	15.18	24	305.15	18.67	16	4802	256.45	625.11	7460
test.mosquitto.org	544.06	552.62	1424.4	507.07	685.63	7845	778.8	932.99	X
	186.1	173.75	605.12	15.27	455.49	4419	19.2	113.71	X
broker.hivemq.org	501.74	508.3	1177.2	512.41	530.39	7178.4	518.26	550.04	1067.5
	16.65	15.01	284.01	19.09	15.57	4084.6	23.57	238.36	6523.8
mqtt.fluux.io	383.78	373.27	974.18	368.46	380.05	3989	398.86	357.17	7217.8
	143.35	18.49	272.83	19.24	26.94	2120	180.31	24.61	4278.9
broker.emqx.io	568.87	575.67	11005	579.75	562.49	8392	568.41	1858	11734
	21.38	14.20	14513	19.12	13.98	4880.9	14.19	762.25	7284
broker.mqtdashboard.com	569.57	579.35	1128.8	559	594.69	8103.3	568.12	845.04	11029
	124.70S	29.83	291	23.25	221.3	4564.4	15.1	186.18	6845.1
ioticos.org	393.35	410.21	762.9	218.6	173.2	932.83	126.66	269.68	751.1
	898.75	794.4	472.64	307.56	204.60	661.73	17.27	376.139	490.06

The broker *akiro* presented the largest end-to-end delays.

When the elapsed time is greater than the publication interval, it increases monotonically with the message number because the messages accumulate in the queue waiting to be processed by the broker, both when the computer sends the message to the IoT board and when the IoT board returns the message back to the computer. For example, with the broker *hivemq*, QoS 1, $T=100ms$, the delays between the messages 17 and 23 were 2371 2522 2686 2866 2986 3122 3255 [ms].

With QoS 2 and transmission interval equal to $100ms$, the broker *mosquitto* collapsed.

Sometimes, some brokers temporarily block. For example, with the broker *ioticos*, QoS 0 and $T=500ms$, the message 41 took $4487ms$, a much greater value than the average presented by this broker in this test ($410.21ms$).

Fig. 4 shows the delays presented by the brokers tested with interval equal to $500ms$. The plots of the broker *akiro* are not shown because its minimum elapsed times were $4883ms$, $3558ms$ and $4592ms$ for each QoS respectively. Note that with QoS 2, the brokers that cannot process the delivery of messages in a time less than the transmission interval have monotonically increasing delays on average.

Fig. 5 shows the minimum, mean and maximum elapsed times presented by the brokers tested for each transmission interval and each QoS used. Also here, the bars corresponding to the broker *akiro* are not shown because its delays were greater than those plotted.

The broker *ioticos*, although sometimes blocks, presents the smallest mean latency. The broker *flux* has more regularity presenting the smallest standard deviation on average.

2.3 Third test: mean latency between a client and itself

In this test the latency presented by the brokers will be measured independently of the speed of response of the IoT board, which will not be used in this test. For that reason, the local computer will be publisher and also subscriber, so it must subscribe the topic to which it publishes the messages.

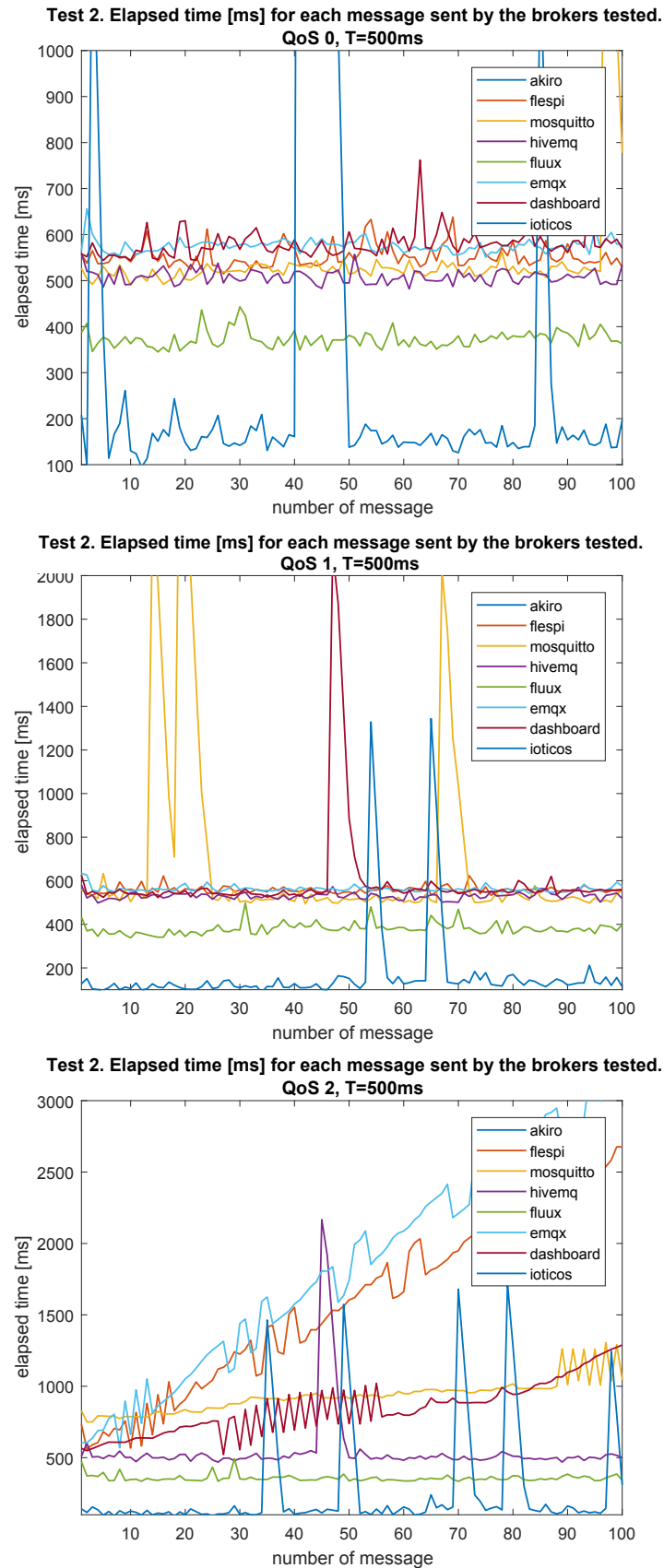


Fig. 4: Elapsed times of the brokers tested in the tests developed for QoS 0 (top), QoS 1 (middle) and QoS 2 (bottom)

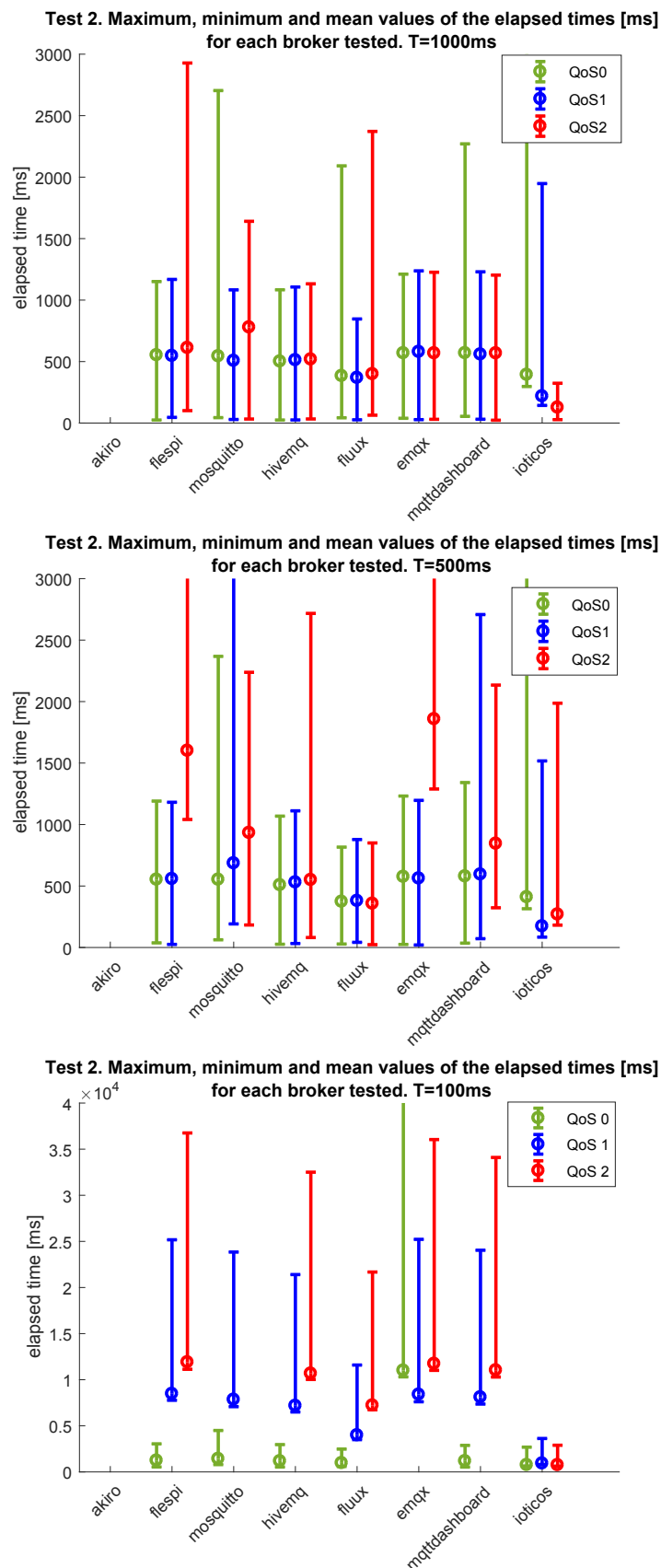


Fig. 5: Minimum, mean and maximum values of the elapsed times of the second test with transmission intervals of 1000ms (top), 500ms (middle) and 100ms (bottom) for each QoS tested

The latency will be measured as the end-to-end delay between the transmission and the reception of a message. The measurements will be made as a function of the QoS. The transmission intervals will be the same that those used in the two former tests. The payload also is the same that the used in the second test, as well as the other parameters.

Table 3 shows the results of this test.

Table 3: **Test3**: Mean latency (top rows) and standard deviation (bottom rows) for each publication interval and each QoS [ms]. X: system collapse. L.D.= lost data.

QoS time interval [ms]	QoS 0			QoS 1			QoS 2		
	1000	500	100	1000	500	100	1000	500	100
saas.theakiro.com	1185.7	1397.7	31035	1194.8	11421	35168	1174.4	14381	32598
	51.05	7292.8	18185	104.09	6381	19027	63.23	7888	17927
mqtt.flespi.io	290.77	295.1	908.2	321.63	292.71	8474	297.86	1662	11741
	12.17	53.90	288.87	14.23	13.31	4965	12.95	774.46	7332
test.mosquitto.org	277.85	287.93	880.75	286.56	279	7642	551.66	1056	X
	12.95	13	288.55	17.48	13.82	4439	17.79	313.95	
broker.hivemq.org	313.02	308.89	907.07	294.93	302.69	8758	297.99	1703.4	1178
	14.37	12.32	285.67	13	13.66	5109	11.76	807.42	7375
mqtt.fluux.io	203.77	204.86	748.03	190.58	202.99	3935.5	213.16	224.95	8682.1
	12.77	11.98	266.69	14.83	12.77	2195	15.18	16	4865
broker.emqx.io	301.2	302.02	917.81	321.2	429.76	7974	423.1	549.15	11038
	10.54	10.96	298.82	16.81	88.26	4666	86.43	135.52	6906
broker.mqttdashboard.com	309.68	297.29	906.21	302.88	312.35	9050	329.57	1846	12110
	15.55	14.78	299.36	25.84	14.54	5279	20.14	904.33	7603
ioticos.org	71.25	87.08	546.07	92.67	93.1	542.79	79.1	72.74	540.75
	10.82	17.09	276.85	16.78	14.78	277.88	12.09	12.68	276.23

The results obtained in this test are not significantly different from those obtained in the second test, except for the fact that the mean latencies are lower than those presented in the former test. It was expectable, because in this test the messages are not sent to the IoT board, the path taken is from the computer to the broker and back to the computer.

Here again the broker `mosquitto` collapsed when QoS 2 and transmission interval equal to 100ms were used. The broker `emqx` lost 15 messages when QoS 0 and interval equal to 100ms were used.

Here again, the broker `ioticos` presented the smallest mean latency, while the broker `fluux` presented the smallest standard deviation on average.

3 Conclusions

The measurements obtained in the test performed allow us to reach some important conclusions.

The brokers `saas.theakiro.com` and `broker.emqx.io` presented higher latency than their competitors. The latter lost data when subjected to a slightly increased stress condition.

In all the tests carried out here, the smallest latency was presented by the broker `ioticos.org`. Even when messages accumulate in the queue, this broker is the fastest one to process them. This broker is not tested in any of the references cited, which is a contribution of the present study.

The broker `mqtt.fluux.io` presented the smallest standard deviation on average, which means that it is the one that more regularly handles the data transmission.

The broker `ioticos.org` requires the use of credentials, for which users must sign up. It also has the particularity of providing a root topic when a project node is created in the web page. The topic of all the messages sent by the publishers and all the topics subscribed by the subscribers in the network must begin with this root topic. Of course, subtopics can be added at the end of the root topic. Using the root topic prevents devices outside the network from exchanging messages with devices used in the project.

Unlike the test carried out in [12, 11, 14], here we are not interested in evaluating the performance of the brokers as a function of the payload size, because in our application the messages sent by the nanosatellite to the land terminal will be short strings containing data measured.

As a future work we propose to perform tests closer to the real need for communication with a nanosatellite, for example using the sensors that will effectively be used in the project and the transmission interval necessary to send all the data collected daily during the short connection time.

References

1. Akiro: `saas.theakiro.com`. Available at <https://www.akiroio.com/> (2022), last accessed October 3, 2022
2. Banks, A., Gupta, R.: MQTT version 3.1.1. Oasis standard. <http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html> (2014)
3. Chilukuri, R.T.: Public brokers. Available at https://github.com/mqtt/mqtt.org/wiki/public_brokers (2021)
4. Crespo, E.: Aprendiendo Arduino. MQTT. Available at <https://aprendiendoarduino.wordpress.com/2018/11/19/mqtt/> (2018)
5. Emqx: `broker.emqx.io`. An open-source, cloud-native, distributed MQTT broker for IoT. Available at <https://www.emqx.io/>, last accessed October 3, 2022
6. Flespi: `mqtt.flespi.io`. MQTT broker. Available at <https://flespi.com/mqtt-broker>, last accessed October 3, 2022
7. Fluux: `mqtt.fluux.io`. Available at <https://flespi.com/mqttbroker>, last accessed October 3, 2022
8. Hivemq: `broker.hivemq.com`. Available at <https://www.hivemq.com/downloads/>, last accessed October 3, 2022
9. Ioticos: `ioticos.org`. Available at <https://www.ioticos.org>, last accessed October 3, 2022
10. Kotak, J., Shah, A., Shah, A., Rajdev, P.: A comparative analysis on security of MQTT brokers. In: Proceedings of the 2nd Smart Cities Symposium (SCS 2019). pp. 1–5. Bahrain (2019). <https://doi.org/10.1049/cp.2019.0180>
11. Lee, S., Kim, H., Hong, D., Ju, H.: Correlation analysis of MQTT loss and delay according to QoS level. In: Proceedings of the International Conference on Information Networking (ICOIN). Bangkok, Thailand (2013). <https://doi.org/10.1109/icoin.2013.6496715>
12. Luzuriaga, J.E., Cano, J.C., Calafate, C., Manzoni, P., Perez, M., Boronat, P.: Handling mobility in IoT applications using the MQTT protocol. In: Proceedings of the 2015 Internet Technologies and Applications (ITA). pp. 245–250. Wrexham, UK (September 2015). <https://doi.org/10.1109/ITechA.2015.7317403>
13. Mayer, R., D’Angiolo, F., Caporaletti, G., Contreras, D., Perez, H., Collado, F., Loiseau, M.: Estudio y recomendación de computadoras de abordo para Cubesat. Tech. report, Technology and Administration Department. National University of Avellaneda, Avellaneda, Argentina (2022)
14. Mishra, B.: Performance evaluation of MQTT broker servers, Lecture Notes in Computer Science, vol. 10963, pp. 599–609. Springer International Publishing (2018). https://doi.org/10.1007/978-3-319-95171-3_47
15. Mishra, B., Kertesz, A.: The use of MQTT in M2M and IoT systems: a survey. *IEEE Access* **8** (2020). <https://doi.org/10.1109/ACCESS.2020.3035849>
16. Mishra, B., Mishra, B., Kertesz, A.: Stress-testing MQTT brokers: A comparative analysis of performance measurements. *Energy* **14**(18) (2021). <https://doi.org/10.3390/en14185817>
17. Mosquitto: `test.mosquitto.org`. MQTT broker. Available at <http://test.mosquitto.org/>, last accessed October 3, 2022
18. MQTT: The standard for IoT messaging. Available at <https://mqtt.org/>
19. MqttDashboard: `broker.mqttdashboard.com`. Available at <http://www.mqtt-dashboard.com/>, last accessed October 3, 2022
20. O’Leary, N.: Pubsubclient: Arduino client for MQTT. Available at <https://github.com/knolleary/pubsubclient> (2020)

21. Soni, D., Makwana, A.: A survey on MQTT: A protocol of Internet of Things (IoT). In: Proceedings of the International Conference on Telecommunication, Power Analysis and Computing Techniques (ICTPACT). Chennai, India (2017)
22. TMS: MQTT software. Developer guide. Available at <https://download.tmssoftware.com/Download/Manuals/TMSMQTTDevGuide.pdf> (2020)