Open-Source MQTT Evaluation

Melvin Bender*, Erkin Kirdan†*, Marc-Oliver Pahl‡*, Georg Carle*
*Technical University of Munich, †Covalion, Framatome, ‡IMT Atlantique
*{melvin.bender,erkin.kirdan,pahl,carle}@tum.de, † erkin.kirdan@framatome.com, ‡ marc-oliver.pahl@imt-atlantique.fr

2021 IEEE 18th Annual Consumer Communications & Networking Conference (CCNC) | 978-1-7281-9794-4/21/$31.00 ©2021 IEEE | DOI: 10.1109/CCNC49032.2021.9369499

***Abstract*—MQTT is a lightweight publish-subscribe protocol used in the Internet of Things. Its popularity leads to several implementations in different languages. In this paper, we evaluate the most popular open-source MQTT implementations, Mosquitto, HiveMQ, EMQX, VerneMQ, MQTT.js and Paho. Our evaluation includes interoperability, resource consumption and latency. We create a generic test framework independent of any MQTT implementation or language. According to our interoperability results, major client and server implementations conform to the base requirements of the standard and thus can interoperate. The language of implementation is a crucial factor for resource consumption and causes considerable differences in scalability. Our results show that latencies between implementations in lossy networks differ. Overall, there is a trade-off between resource consumption and network latency.**

***Index Terms*—mqtt, open-source, performance**

I. Introduction

The Internet of Things (IoT) covers diverse applications and use-cases. Its communication protocols and patterns continuously develop and diversify [1]. The publish-subscribe pattern is a common approach to handle the increasing amount of devices in the IoT in an efficient and scalable manner. Publish-subscribe relies on multi-cast messaging. A publisher sends a message to an event channel. The event channel handles the forwarding of the message to each interested subscriber. This provides a decoupling in space, time and synchronization [2].

MQTT is a lightweight publish-subscribe protocol used for machine-to-machine communication [3]. It offers dynamic scalability, resource efficiency, and easy implementation. It implements the publish-subscribe pattern by using *topics*. An MQTT client connects to an MQTT broker and publishes a message related to a topic. The broker forwards the message to the clients that have subscribed to the respective topic (see Figure 1).

A typical IoT scenario consists of several electronic devices communicating with each other. These devices mostly have constraints on power, memory, and processing resources. They create constrained-node networks [4]. Any software running on such devices should spend memory, power and network bandwidth carefully to provide long battery life.

Considering the reasons above, MQTT is commonly used in the IoT where devices can be highly limited in their networking resources. Due to its popularity, there are several implementations of the protocol. Implementation selection is often made based on familiarity or language preference.

This research was funded by the German Federal Ministry of Economic Affairs and Energy (BMWI) in DECENT (0350024) and the German-French Academy in SCHEIF.

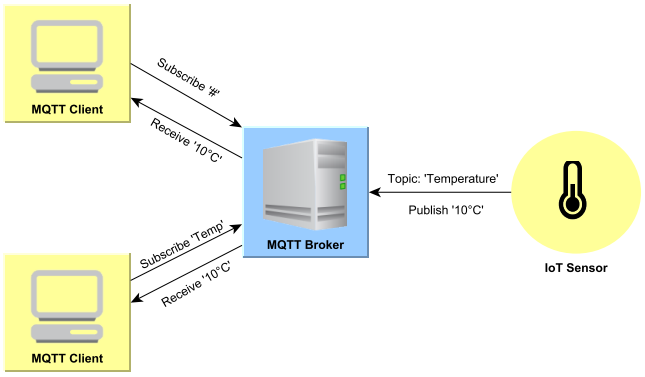

Fig. 1: Publish-Subscribe Pattern in MQTT

Moreover, a comparative evaluation of implementations is missing. With this paper, we aim at filling that gap and provide an overview of the most popular MQTT implementations.

Section II gives an overview of the state of the art. Section III presents the selection criteria for the evaluated open-source implementations. Section IV introduces the test framework. Section V discusses the results of the evaluation.

II. Related Work

We survey research efforts that analyze various IoT protocols in terms of performance and scalability. Our goal is to make an in-depth comparison of MQTT brokers that makes use of the measurements and experiment parameters we observe.

In [5], the Quality of Service level in the MQTT is thoroughly tested in a wired and wireless environment. They find that there is a strong positive correlation between latency and message loss for each QoS level.

In [6], a smartphone-based sensor platform is considered. The authors consider the MQTT and CoAP protocol for the usage in their smartphone application scenario. They state that MQTT is more reliable and has better congestion control mechanisms.

In [7], the authors compare the performance of MQTT and the Advanced Message Queuing Protocol (AMQP) in mobile networks. They recommend the MQTT protocol for constrained networks because of its better energy efficiency.

In [8], the authors compare MQTT brokers as a preliminary experiment. They do a preliminary evaluation of different brokers to determine a reference broker for their main experiment. They find that Mosquitto has the best results among them.

III. Open-Source Implementations

Following, we give an overview of popular MQTT clients and brokers. For an implementation to be considered for our research, we define three criteria.

1) **Open-source:** An MQTT implementation shall be open-source to be considered for our experiments. Having access to the codes gives us better insight into how the implementation works.
2) **Active:** Moreover, an MQTT client or broker shall be actively maintained. If it actively being developed, it is more likely it implements the latest MQTT features.
3) **Common:** Last, a broker or client shall be commonly used. By looking at the star counter in the Github repositories, and web search engine trends, we can gain insight about into popularity.

The selected implementations are given in Table I. The table is filled with information from 31.08.2020.

A. MQTT Brokers

There is a large variety of MQTT brokers available. All of them support MQTT version 3.1.1, widespread support for MQTT 5 is still to come. Furthermore, all of them were released in 2020. We observe that Mosquitto is the most popular broker in the research community. The C-built broker is light-weight and has a simple command interface that allows for quick creation of MQTT networks. The EMQX Broker is a popular broker developed in Erlang. Its developers advertise that the broker can be used in distributed infrastructures. The developers of VerneMQ claim that their application scales especially well.

B. MQTT Clients

We survey open-source MQTT client implementations. We choose the Mosquitto client library because of its maturity and good usability via its console commands. The Paho MQTT client is also popular and appears in many of the research papers we survey. The MQTT.js, written in JavaScript, is also chosen because of its popularity.

TABLE I: Overview of Open-Source MQTT Implementations

Name	Stars	Language	Version	Rel. work
RabbitMQ	7500	Erlang	3.1.1	2
EMQ X Broker	6500	Erlang	3.1, 3.1.1, 5	-
ejabberd	4611	Erlang	3.1, 3.1.1, 5	-
Mosquitto	3958	C/C++	3.1, 3.1.1, 5	6
VerneMQ	2196	Erlang	3.1, 3.1.1, 5	-
MQTTnet	1636	C#	3.1, 3.1.1, 5	-
ActiveMQ	1788	Java	3.1	3
HiveMQ CE	450	Java	3.1, 3.1.1, 5	2
MQTT.js	5700	JavaScript	3.1, 3.1.1	1
Paho MQTT	1160	Various	3.1., 3.1.1	4
Aedes	777	JavaScript	3.1, 3.1.1	-
esp-mqtt	365	C	3.1	-
HiveMQ Client	330	Java	3.1.1, 5.0	-

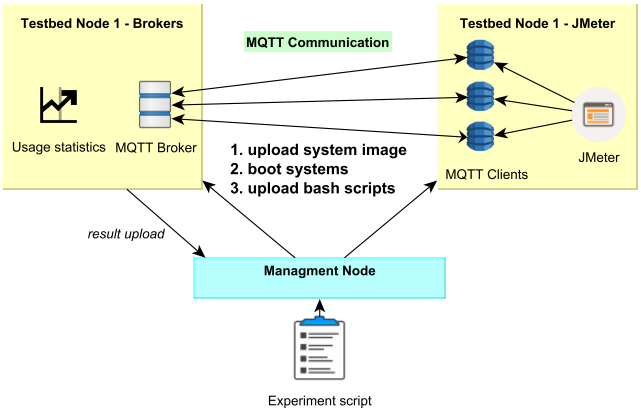

Fig. 2: Experiment Testbed

IV. Framework

We design a framework to test MQTT implementations in a reproducible manner. We present the test tools we use to create our test framework. Moreover, we discuss the attributes of our environment that enable us to do reproducible tests.

A. Tools

Using testing software for MQTT networks allows a systematic and freely configurable benchmark of MQTT brokers and clients. By changing the parameters, we can simulate different network conditions such as packet drop rate. We use *Apache JMeter* and its MQTT plugin. JMeter allows us to configure various MQTT scenarios dynamically. It handles latency measuring of MQTT messages. For measuring resource usage, we use the Linux tool *atop*. This monitoring service runs as a background service on the system and records CPU usage in short intervals.

B. Environment

Random system background activity causes variations in the measurements. Network activity can be random. To eliminate such influences, we run several tests to get reproducible results.

Another important aspect is the *common environment* for all tests. Our testbed has remote server nodes that we can use for the clear separation between MQTT brokers and clients (see Figure 2). We deploy Debian 9 since it offers a stable system. A test run for our experiments has a fixed process flow. The server gets reset. A clean system image is uploaded. Then the individual test scripts are uploaded to the two testbed nodes. One node handles the installing and measuring of the MQTT client or broker to be evaluated. The other testbed node handles the installation of the Apache JMeter and the creation of the network test load.

V. Evaluation

The evaluation is done in four categories: interoperability, resource usage, latency and security. The MQTT clients we test are the Mosquitto client library, MQTT.js and Paho

MQTT. The MQTT brokers we test are Mosquitto, EMQX, HiveMQ and VerneMQ.

A. Interoperability

As a first step, a crosswise broker-client test is made. If the payload of the publishing client is successfully forwarded to the subscribing client, the test is considered to be successful. All of the MQTT clients are able to work with all of the brokers pairwise.

As a second test, the Paho Testing Utilities are used to test if MQTT broker implementations conform to the MQTT 3.1.1. specification. If they stay true to the specification, it is more likely that they will interoperate with the other brokers. All four brokers pass all the necessary tests (see Table II). There are only optional tests that are failed by the chosen brokers. No broker uses negative acknowledgements which should be sent when a client tries to subscribe to a topic that is not allowed. VerneMQ and HiveMQ fail the $ topics test, which means that they pass broker statistics for wildcard subscriptions.

TABLE II: MQTT 3.1.1 Conformance Test

feature	Mosquitto	EMQX	VerneMQ	HiveMQ
Basic test	✓	✓	✓	✓
Retained messages	✓	✓	✓	✓
Will messages	✓	✓	✓	✓
Zero length client IDs	✓	✓	✓	✓
Offline message queuing	✓	✓	✓	✓
Overlapping subscriptions	✓	✓	✓	✓
Keepalive	✓	✓	✓	✓
Redelivery on reconnect	✓	✓	✓	✓
Subscribe failure	-	-	-	-
$ Topics	✓	✓	-	-
Unsetting retainers	✓	✓	✓	✓

B. Resource Usage

Devices that use MQTT technology may be heavily limited in resources. This limitation can restrict their battery life or the amount of MQTT messages they can handle at once. Two essential resources for any computational device are the *CPU* and *memory*. We deploy MQTT network scenarios and measure the CPU and memory usage during execution.

1) Clients: For our client test, we publish an increasing amount of messages using the client to be evaluated. We use Mosquitto as the MQTT broker to be used for this experiment. The client connects to the broker and publishes messages to a test topic with short, arbitrary strings as the payload. The peak memory used by clients does not increase with the number of messages. We observe that the JavaScript-built MQTT.js has 20 times higher memory usage than Paho or Mosquitto. MQTT.js uses around 850 MB of memory, whereas Paho uses around 50 MB. Mosquitto has the lowest memory requirements at around 16 MB.

We measure how many CPU cycles are used during the publishing operations . The resource usage increases with the number of messages a client sends. We observe that MQTT.js and Paho have significantly higher resource usage than Mosquitto. Mosquitto uses up to 300 Million cycles, while Paho and MQTT.js use around 3 Billion Cycles.

2) Brokers: To test the brokers, we create the following four scenarios.

1) *increasing number of subscriptions by single user*: A client connects to the broker and subscribes to an increasing number of topics. There are no publishers. After the subscriptions are done, the broker's resource usage is measured for a 30-second interval.
2) *increasing number of subscribers for a single topic*: An increasing number of clients connect to the broker, and all subscribe to the same topic. There are no publishing clients. After the subscriptions are done, the broker's resource usage is measured for a 30-second interval.
3) *increasing number of messages without subscribers*: We increase the number of messages a client publishes to the broker. There are no subscribers. Therefore, there is no MQTT message forwarding. We measure the resource statistics until all messages have been published.
4) *increasing number of subscribers*: We gradually increase the number of clients connecting to the broker and subscribing to the same topic. There is a single client that publishes short, arbitrary strings to that topic. After all the clients are subscribed, the broker's statistics rare measured for a 60-second interval.

The memory usage of the brokers increases slightly for an increasing amount of topics (see Table III). We observe that the memory usage of the brokers increases significantly with the increasing amount of clients. Mosquitto has the lowest memory requirements, only using around 11 MB at most. There is a significant gap in memory usage between Mosquitto and the other brokers. VerneMQ uses up to 135 MB. EMQX follows this with a maximum memory usage of around 200 MB. HiveMQ has the highest memory usage with up to 825 MB.

TABLE III: Broker Test - Memory Usage

Implementation	Peak memory in scenarios [MB]			
	1	2	3	4
Mosquitto	$5.03 - 10.6$	$3.5 - 5.5$	3	$4.1 - 6.8$
VerneMQ	$65 - 125$	$77 - 96$	$55 - 58$	$68 - 135$
EMQX	$120 - 148$	$120 - 124$	118	$126 - 197$
HiveMQ	$311 - 366$	$579 - 718$	$301 - 310$	$661 - 825$

We observe that the CPU usage increases with the message publishing. The CPU usage results of the brokers highly vary among the different scenarios. Mosquitto uses the lowest amount of CPU cycles, except for the second scenario. The broker shows worse usage when there are a lot of connected subscribers and no message forwarding. Overall, we observe, that the brokers that make use of a Virtual Machine for operation have significantly higher resource usage values. Erlang uses BEAM, while the Java environment makes use of the Java Virtual Machine. VerneMQ shows higher CPU usage when there are many clients present. If there is a large message forwarding load, HiveMQ and EMQX show better results than VerneMQ.

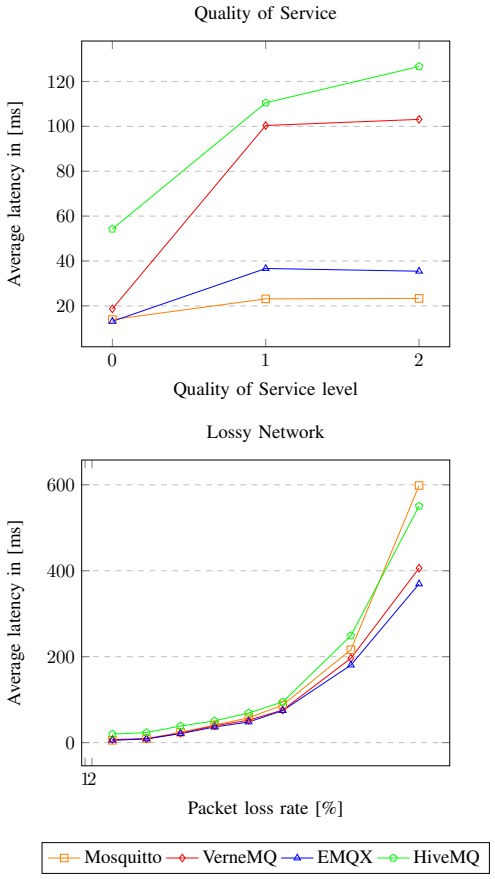

Fig. 3: Latency Experiment Results

C. Latency

The time it takes for an MQTT message to travel from producer to consumer is one of the measurements most widely used in protocol evaluation. For various network domains, we determine which MQTT implementations achieve the lowest latency. We deploy two types of latency experiments. In the first test, we investigate the correlation between Quality of Service level and average broker latency. In the second test, we survey the effect of an increasing average packet drop rate on the broker's performance.

1) Quality of Service level: In this scenario, we test how different Quality of Service levels affect the latency for the various brokers. We deploy 2,000 clients that each subscribes to a test topic and publishes a message 20 times. We observe that messages on QoS level 2 do mostly not take longer to arrive than on level 1. On average, the Mosquitto broker shows the best results for a lossless scenario with a low amount of traffic (see Figure 3). The HiveMQ broker has the worst results, reaching a maximum latency of around 125 milliseconds.

2) Lossy Networks: In this scenario, packets get dropped randomly on their way from broker to the client at random rates ranging from 5% up to 50%. In our scenario, 200 clients are publishing and subscribing to one topic. The EMQX broker performs best across all packet loss drop rates (see Figure 3). For higher packet drop rates, the difference in performance between the brokers' increases. Mosquitto has the best performance for the 5% packet loss. We observe that the two Erlang-based brokers, VerneMQ and EMQX, perform best for lossy networks.

VI. Conclusion

We inspected a selection of MQTT client libraries. Our results indicate that the Mosquitto client has the best scalability features among them. Even for a large number of incoming messages, the number of CPU cycles remains small. The system memory used also remains small, which allows for the efficient implementation of the library in resource-constrained IoT devices. Our research indicates that the resource usage and performance of MQTT is highly dependent on the choice of broker. The results of the resource usage tests show that the Mosquitto broker implementation always has the lowest requirements for CPU and memory. However, the performance of Mosquitto is significantly worse for networks with packet losses and a high number of clients. VerneMQ has also shown to have relatively high requirements for specific scenarios. The resource usage results are similar for the EMQ X broker.

We show that Mosquitto can be used on a resource-constrained device for thousands of clients. However, better latency results are reached by the Erlang-based brokers for specific scenarios. This applies to networks with packet losses, like in a cellular environment. The EMQX or VerneMQ brokers show better latency results for networks with thousands of publishers and packet losses occurrences. EMQX is the broker with the consistently best result of our experiments. VerneMQ and EMQX both require a large amount of memory and CPU power but produce the best MQTT latency results. Therefore, we observe that there is a trade-off between resource usage and network latency.

References

[1] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash, "Internet of things: A survey on enabling technologies, protocols, and applications," *IEEE Communications Surveys and Tutorials*, vol. 17, no. 4, pp. 2347–2376, 2015.

[2] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM Computing Surveys*, vol. 35, no. 2, pp. 114–131, 2003.

[3] A. Banks, E. Briggs, K. Borgendale, and R. Gupta, *MQTT Version 5.0*, 7 Mar. 2019. OASIS Standard.

[4] A. Keranen, M. Ersue, and C. Bormann, "Terminology for constrained-node networks," *RFC*, vol. 7228, pp. 1–17, 2014.

[5] S. Lee, H. Kim, D. kweon Hong, and H. Ju, "Correlation analysis of mqtt loss and delay according to qos level," in *The International Conference on Information Networking 2013 (ICOIN)*, pp. 714–717, 2013.

[6] N. D. Caro, W. Colitti, K. Steenhaut, G. Mangino, and G. Reali, "Comparison of two lightweight protocols for smartphone-based sensing," in *2013 IEEE 20th Symposium on Communications and Vehicular Technology in the Benelux (SCVT)*, pp. 1–6, 2013.

[7] J. E. Luzuriaga, M. Perez, P. Boronat, J. C. Cano, C. Calafate, and P. Manzoni, "A comparative evaluation of amqp and mqtt protocols over unstable and mobile networks," in *2015 12th Annual IEEE Consumer Communications and Networking Conference (CCNC)*, pp. 931–936, 2015.

[8] R. Banno, J. Sun, M. Fujita, S. Takeuchi, and K. Shudo, "Dissemination of edge-heavy data on heterogeneous mqtt brokers," in *2017 IEEE 6th International Conference on Cloud Networking (CloudNet)*, pp. 1–7, 2017.