Article

Stress Testing MQTT Brokers: A Comparative Analysis of Performance Measurements

Biswajeeban Mishra [1,†,‡,*], Biswaranjan Mishra [2,‡] and Attila Kertesz [1,†,‡]

1 University of Szeged, Hungary - 6720; mishra@inf.u-szeged.hu, keratt@inf.u-szeged.hu
2 #34b, 5a cross, NRI Layout, Bengaluru, India - 560043; biswaranjan.mishra@live.com
* Correspondence: mishra@inf.u-szeged.hu
† Current address: Department of Software Engineering, University of Szeged, Szeged, Hungary - 6720
‡ These authors contributed equally to this work.

Abstract: Nowadays, Internet of Things (IoT) protocols are at the heart of Machine-to-Machine (M2M) communication. Irrespective of the radio technologies used for deploying an IoT/M2M network, all independent data generated by IoT devices (sensors and actuators) rely heavily on the special messaging protocols used for M2M communication in IoT applications. As the demand for IoT services is growing, the need for reduced power consumption of IoT devices and services is also growing to ensure a sustainable environment for future generations. The Message Queuing Telemetry Transport or in short MQTT is a widely used IoT protocol. It is a low resource consuming messaging solution based on the publish-subscribe type communication model. This paper aims to assess the performance of several MQTT Broker implementations (also called as MQTT Servers) using stress testing, and to analyze their relationship with system design. The evaluation of the brokers is performed by a realistic test scenario, and the analysis of the test results is done with three different metrics: CPU usage, latency, and message rate. As the main contribution of this work, we analyzed six MQTT brokers (Mosquitto, Active-MQ, Hivemq, Bevywise, VerneMQ, and EMQ X) in detail, and classified them using their main properties. Our results showed that Mosquitto outperforms the other considered solutions in most metrics, however, ActiveMQ is the best performing one in terms of scalability due to its multi-threaded implementation, while Bevywise has promising results for resource-constrained scenarios.

Keywords: Internet of Things; Messaging Protocol; MQTT; MQTT Brokers; Performance Evaluation; Stress testing

Citation: Mishra, B.; Mishra, B.; Kertesz, A. Title. *Energies* **2021**, *1*, 0. https://doi.org/

Received:
Accepted:
Published:

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Copyright: © 2021 by the authors. Submitted to *Energies* for possible open access publication under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/ 4.0/).

1. Introduction

In recent times, as the cost of sensors and actuators is continuing to fall, the number of Internet of Things (IoT) devices is rapidly growing and becoming parts of our lives. As a result, the IoT footprint is significantly noticeable everywhere. It is hard to find any industry that does not get revolutionized with the advent of this promising technology. A recent report [1] states that there would be around 125 billion IoT devices connected to the Internet by 2030. IoT networks use several radio technologies such as WLAN, WPAN, etc. for communication at a lower layer. Regardless of the radio technology used, to create an M2M network, the end device or machine (IoT device) must make their data accessible through the Internet [2,3]. IoT devices are usually resource-constrained. It means that they operate with limited computation, memory, storage, energy storage (battery), and networking capabilities [4], [6]. Hence, the efficiency of M2M communications largely depends on the underlying special messaging protocols designed for M2M communication in IoT applications. MQTT (Message Queuing Telemetry Transport) [5], CoAP (Constrained Application Protocol), AMQP (Advanced Message Queuing Protocol), and HTTP (Hypertext Transfer Protocol) are the few to name in the M2M Communication Protocol segment [4,6]. Among these IoT Protocols, MQTT is a free, simple to deploy, lightweight, and energy-efficient application layer protocol. These properties make MQTT an ideal communication solution for IoT systems [8–10]. As Green Computing primarily focuses on implementing energy-saving technologies that help reduce the greenhouse impact on the environment [11], ideal design and implementation of MQTT based solutions can be immensely helpful in realizing the goals of a sustainable future. MQTT is a topic-based publish/subscribe type protocol that runs on TCP/IP using ports 1883 and 8883 for non-encrypted and encrypted communication, respectively. There are two types of network entities in the MQTT protocol: a message broker, also called as the server, and the client, which actually play publisher and subscriber roles). A publisher sends messages with a topic head to a server, then it delivers the messages to the subscribers listening that topic [8]. Currently, we have many MQTT based broker (server) distributions available in the market from various vendors.

Our main goal in this research is to answer the following question: *How does a scalable or a non-scalable broker implementation perform in a single-core and multi-core CPU test-bed, when it is put under stress-conditions?* The main contribution of this work is analyzing and comparing the performance of considered scalable and non-scalable brokers based on the following metrics: maximum message rate, average process CPU Usage in percentage at maximum message rate, normalized message rate at 100% CPU usage, and average latency. This work is a revised and significantly extended version of the short paper [12]. It highlights the relationship of a MQTT broker system design and its performance under stress-testing. The MQTT protocol has many application areas such as healthcare, logistics, smart city services etc [13]. Each application area has a different set of MQTT-based requirements. In this experiment, we are not evaluating MQTT brokers against those specific set of requirements rather we are conducting a system test of MQTT servers to analyze their message handling capability, the robustness of implementation, and efficient resource utilization potential. To achieve this, we send a high volume of short messages (low payload) with a limited set of publishers and subscribers.

The remainder of this paper is organized as follows. Section 2 introduces background of this study. Section 3 summarizes some notable related works. Section 4 describes the test environment, evaluation parameters and test results in detail. In Section 5, we discuss the evaluation results, and finally, with Section 6 we conclude the paper.

2. Background

2.1. Basics of a publish/subscribe Messaging Service

These are the terms we often come across while working with a publish/subscribe or Pub/Sub System. "Message" refers to the data that flows through the system. "Topic" is an object that presents a message stream. "Publisher" creates messages and sends them to the messaging service on a particular Topic head. The act of sending messages to the messaging service is called "Publishing". A publisher is also referred to as a Producer. "Subscriber", otherwise known as "Consumer", receives the messages on a specific subscription. "Subscription" refers to an interest in receiving messages on a particular topic. In a Pub/Sub system, producers of the event-driven data are usually decoupled from the consumers of the data [14,15]: meaning publishers and subscribers are independent components that share information by publishing event-driven messages and by subscribing to event-driven messages of choice [14]. The central component of this system is called an event broker. It keeps a record of all the subscriptions. A publisher usually sends a message to the Event broker on a specific topic head and then the event broker sends it to all the subscribers that previously subscribed to that topic. The event broker basically acts as a postmaster to match, notify, and deliver events to the corresponding subscribers. Fig. 1 describes the overall architecture of a Pub/Sub system [16].

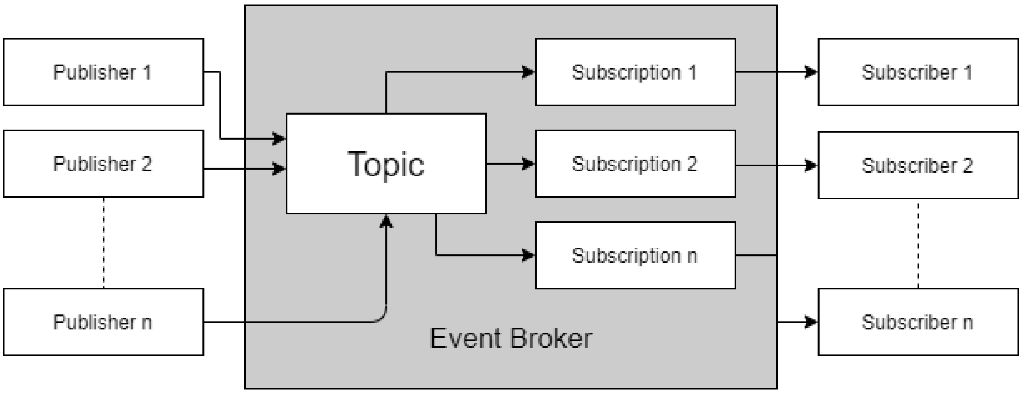

Figure 1. Overall architecture of a Pub/Sub system

2.2. Overview of MQTT Architecture

MQTT is a simple, lightweight, TCP/IP based Pub/Sub type messaging protocol [11]. MQTT supports one-to-many, two-way, asynchronous communication [7]. Having a binary header makes MQTT a lightweight protocol to carry telemetry data transmission between constraint devices [17,18] over unreliable networks [19]. It has three constituent components:

- A Publisher or Producer (An MQTT client).
- A Broker (An MQTT server).
- A Consumer or Subscriber (An MQTT client).

In MQTT, a client that is responsible for opening a network connection, creating and sending messages to the server is called a publisher. The subscriber is a client that subscribes to a topic of interest in advance in order to receive messages. It can also unsubscribe from a topic in order to delete a request for application messages and close network connection to the server [20] as needed. The server is otherwise known as a broker acts as a post office between the publisher and the subscriber. It receives messages from the publishers and forwards them to all the subscribers. Fig. 2 presents a basic model of MQTT [21].

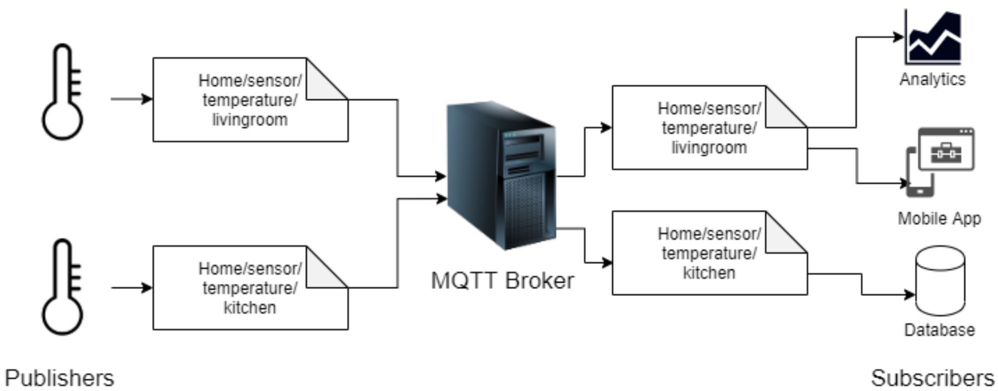

Figure 2. MQTT Components: Publisher, MQTT Broker, and Subscriber.

Any application message carried by the MQTT protocol across the network to its destination contains a quality of service (QoS), payload data, a topic name [22], and a collection of properties. An application message can carry a payload up to the maximum size of 256 MB [3]. A topic is usually a label attached to all messages. Topic names are UTF-8 encoded strings and can be freely chosen [22]. Topic names can represent a multilevel hierarchy of information using a forward slash (/). For example, this topic name can represent a humidity sensor in the kitchen room: "home/ sensor/humidity/kitchen". We can have other topic names for other sensors that are present in other rooms: "home/sensor/temperature/livingroom", and "home/ sensor/temperature/kitchen" etc. Fig. 3 shows an example of a topic tree.

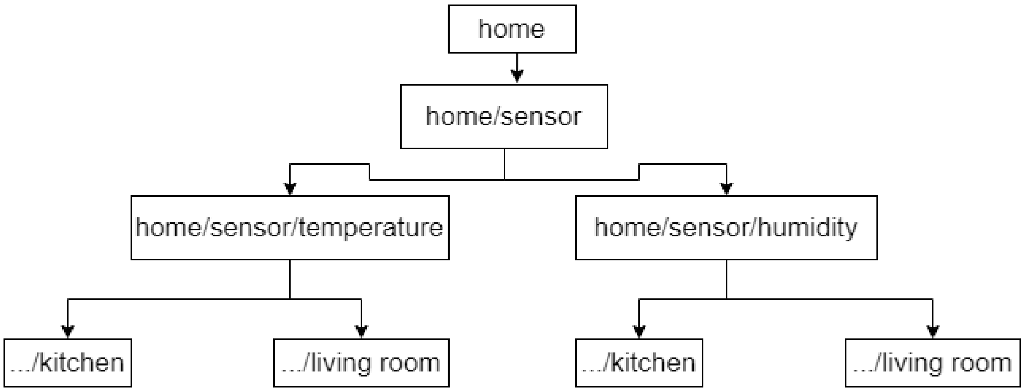

Figure 3. Topic tree hierarchy.

MQTT offers three types of QoS (Quality of Service) levels to send messages to an MQTT Broker or a client. It ranges from 0 to 2, see Fig. 4. By using QoS level 0: the sender does not store the message, and the received does not acknowledge its receiving. This method requires only one message and once the message is sent to the broker by the client it gets deleted from the message queue. Therefore QoS 0 nullifies the chances of duplicate messages, that is why it is also called as the "fire and forget" method. It provides a minimal and most unreliable message transmission level that offers the fastest delivery effort. Using QoS 1, the delivery of a message is guaranteed (at least once, but the message may be sent more than once , if necessary). This method needs two messages. Here, the sender sends a message and waits to receive an acknowledgment (PUBACK message) to receive. If it receives an acknowledgment from the client then it deletes the message from the outward-bound queue. In case, it does not receive a PUBACK message, it resends the message with the duplicate flag (DUP flag) enabled. The QoS 2 level setting guarantees exactly-once delivery of a message. This is the slowest of all the levels and needs four messages. In this level, the sender sends a message and waits for an acknowledgment (PUBREC message). The receiver also sends a PUBREC message. If the sender of the message fails to receive an acknowledgment (PUBREC), it sends the message again with the DUP flag enabled. Upon receiving the acknowledgment message PUBREC, the sender transmits the message release message (PUBREL). If the receiver does not get the PUBREL message it resends the PUBREC message. Once the receiver receives the PUBREL message, It forwards the message to all the subscribing clients. Thereafter the receiver sends a publish complete (PUBCOMP) message. In case the sender does not get the PUBCOMP message, it resends the PUBREL message. Once the sending client receives the PUBCOMP message, the transmission process gets marked as completed and the message can be deleted from the outbound queue [13].

2.3. Scalability and types of MQTT Broker Implementations

System scalability can be defined as the ability to expand to meet increasing workload [23]. Scalability enhancement of any message broker depends on two prime factors; the first one is to enhance a single system performance, while the second one is to use clustering. In case of an MQTT message broker deployment, the performance of an MQTT broker using a single system can be improved by using event-driven I/O mechanism for the CPU cores during dispatching TCP connections from MQTT clients [22]. The other way of achieving better scalability is clustering, when an MQTT broker cluster is used in a distributed fashion. In this case it seems to be a single logical broker for the user, but in reality, multiple physical MQTT brokers share the same workload [24].

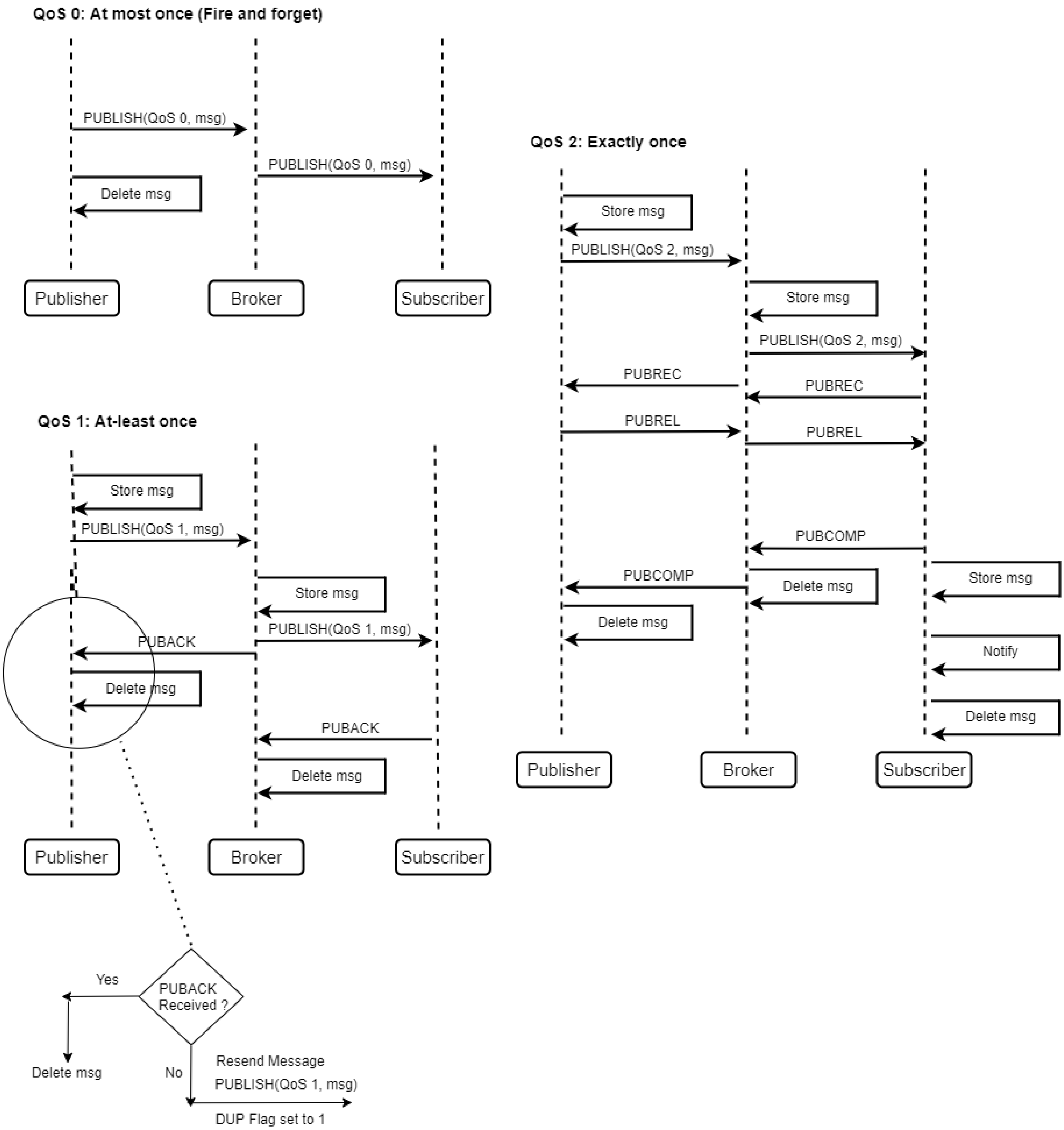

Figure 4. Different QoS levels.

There are two types of message broker implementations: single or fixed number of threads non-scalable broker implementations and multi-thread or multi-process scalable broker implementations that can efficiently utilize all available resources in a system [16]. For example, Mosquitto and Bevywise MQTT Route are non-scalable broker implementations that cannot utilize all system resources, and broker implementations like ActiveMQ, HiveMQ, VerneMQ, and EMQ X are scalable [24]. It is be noted that Mosquitto provides a "bridge mode" that can be used to form a cluster of message brokers. In this mode, multiple cores get utilized according to the number of Mosquitto processes running in the cluster. However, the drawback of this mode is the communication overhead between the processes inside the cluster results in the poorer overall performance of the system [16].

2.4. Evaluating the Performance of a Messaging Service

Google in its "Cloud Pub/Sub" product guide [14] nicely narrates the parameters to judge the performance of any publish/subscribe type messaging service. The performance of a publish/subscribe type messaging services can be measured in three factors "latency", "scalability", and "availability". However, these three aspects frequently contradict each other and involve compromises on one to boost the other two. The following paragraphs put some light on these terms in a pub/sub type messaging service prospective.

2.4.1. Latency

Latency is a time-based metric for evaluating the performance of a system. A good messaging service has to optimize and reduce latency, wherever it is possible. The latency metric can be defined for a publish/subscribe service in the following: it denotes the time the service takes to acknowledge a sent message, or the time the service takes to send a published message to its subscriber. Latency can also be defined as the time taken by a messaging service to send a message from the publisher to the subscriber [14].

2.4.2. Scalability

Scalability usually refers to the ability to scale up with the increase in load. A robust scalable service can handle the increased load without an observable change in latency or availability. One can define load in a publish/subscribe type service by referring to the number of topics, publishers, subscribers, subscriptions or messages, as well as to the size of messages or the payload, and to the rate of sent messages, called throughput [14].

2.4.3. Availability

Systems can fail. It has many reasons. It may occur due to a human error while building or deploying software or configurations or it may be caused due to hardware failures such as disk drives malfunctioning or network connectivity issues. Sometimes a sudden increase in load results in resource shortage and thus causes a system failure. When we say "sound availability of a system" - it usually refers to the ability of the system to handle a different type of failure in such a manner that is unobservable at the customer's end [14].

3. Related Work

There have been numerous works around the performance evaluation of various IoT communication protocols. In this section, we briefly summarize some of the notable works published in recent years. Table 1 presents a comparison of related works according to their main contributions.

In 2014, Thangavel, Dinesh, et al. [25], conducted multiple experiments using a common middleware, to test MQTT and CoAP protocols, bandwidth consumption and end-to-end delay. Their results showed that using CoAP messages showed higher delay and packet loss rates than using MQTT messages.

Chen, Y., and Kunz, T., in 2016 [26], evaluated in a medical test environment MQTT, CoAP, and DDS (Data Distribution Service) performance, compared to a custom, UDP-based protocol. They used a network emulator, and their findings showed that DDS consumes higher bandwidth than MQTT, but it performs significantly better for data latency and reliability. DDS and MQTT, being TCP-based protocols, produced zero packet loss under degraded network conditions. The custom UDP and UDP-based CoAP showed significant data loss under similar test conditions.

Mishra, B., in 2019 [20], investigated the performance of several public and locally deployed MQTT brokers, in terms of subscription throughput. The performance of MQTT brokers was analyzed under normal and stressed conditions. The test results showed that there is an insignificant difference between the performance of several MQTT brokers in normal deployment cases, but the performance of various MQTT brokers significantly varied from each other under the stressed conditions.

Pham, M. L., Nguyen, et al., in 2019 [27], introduced an MQTT benchmarking tool named MQTTBrokerBench. The tool is useful to analyze the performance of MQTT brokers by manually specifying load saturation points for the brokers.

Bertrand-Martinez, Eddas, et al [28], in 2020, proposed a method for the classification and evaluation of IoT brokers. They performed qualitative evaluation using the ISO/IEC 25000 (SQuaRE) set of standards and the Jain's process for performance evaluation. The authors have validated the feasibility of their methodological approach with a case study on 12 different open source brokers.

Koziolek H, Grüner S, et al. [29], in 2020 compared three distributed MQTT brokers in terms of scalability, performance, extensibility, resilience, usability, and security. In their edge gateway, the cluster-based test scenario showed that EMQX had the best performance, while HiveMQ showed no message loss, while VerneMQ managed to deliver up to 10K msg/s, respectively. The authors also proposed six decision points to be taken into account by software architects for deploying MQTT brokers.

Referring back to this work of ours, we compare both scalable and non-scalable MQTT brokers and analyze the performance of six MQTT brokers in terms of message processing rate at 100% process/system CPU utilization, normalized message rate at unrestricted resource (CPU) usage, and average latency. We also analyze how each broker performs in a single-core and multi-core processor environment. For a better analysis of the performances of MQTT brokers, we conducted this experiment in a low-end local testing environment as well as in a comparatively high-end cloud-based testing environment. This experiment deals with an important problem of the relation of MQTT broker system design and its performance under stress testing. Although It is a well-known fact that modular systems better perform on scalable and elastic requirements but we lack experiment-based information about that relationship. So, results obtained in this study would be immensely helpful to developers of real-time systems and services.

4. Local and Cloud Test Environment Settings and Benchmarking Results

This section presents the setup of our realistic testbed in detail. To conduct stress tests on various MQTT Brokers, we have built two emulated IoT environments:

- one is a local testing environment, and
- the other one is a cloud-based testing environment.

The local testbed was created using an Intel NUC (NUC7i5BNB), a Toshiba Satellite B40-A laptop PC, and an Ideapad 330-15ARR laptop PC. To diminish network bottleneck issues, the devices were connected through a Gigabit Ethernet switch. The Intel NUC7i5BNB was configured as a server running an MQTT Broker, the Ideapad 330-15ARR laptop was used as a publisher machine, and the Toshiba, Satellite B40-A was used as a subscriber machine. The Ideapad 330-15ARR (publisher machine), with 8 hardware threads, is capable enough of firing messages at higher rates. Table 2 presents a summary of the specifications of the hardware and software used to build our local evaluation environment.

The cloud testbed was configured on Google Cloud Platform (GCP) [30]. We created three c2-standard-8 virtual machine (VM) instances that are having 8 vCPUs, 32 GB of memory, and 30 GB local SSD each to act as publisher, subscriber, and server respectively. All the VM instances are placed within a Virtual Private Cloud (VPC) Network subnet using Google's high-performing premium tier network service [31]. Table 3 presents a summary of the specifications of our cloud test environment [32].

Table 1: Comparison of related works according to their main contributions.

Paper	Publication Year	Aim
Thangavel, Dinesh, et al. [25]	2014	testing performance of MQTT and CoAP protocols in terms of end-to-end delay and bandwidth consumption
Chen, Y., & Kunz, T. [26]	2016	evaluating performance of MQTT, CoAP, DDS and a custom UDP-based protocol in a medical test environment
Mishra, B. [20]	2018	comparing performance of MQTT brokers under basic domestic use condition
Pham, M. L., Nguyen, et al. [27]	2019	introduced an MQTT benchmarking tool named MQTTBrokerBench.
Bertrand-Martinez, Eddas, et al. [28]	2020	proposed a methodology for the classification and evaluation of IoT brokers.
Koziolek H, Grüner S, et al. [29]	2020	compared performance of three distributed MQTT brokers
Our work	2020	comparing and analyzing performance of MQTT brokers by them under stress test, both scalable and non-scalable brokers taken into consideration

In this experiment we used a higher message publishing rate with multiple publishers, and the overall CPU usage we experienced stayed below 70% on the publisher machine. On the other hand, we also noticed that CPU usage on the subscriber side did not exceed 80%. We experienced no swap usage at the subscriber, broker or publisher machines during the evaluation.

For this experiment, we have developed a Paho Python MQTT library [33] based benchmarking tool called MQTT Blaster [34] from scratch to send messages at very high rates to the MQTT server from the publisher machine. The subscriber machine used the "mosquitto_sub" command line subscribers, which is an MQTT client for subscribing to topics and printing the received messages. During this empirical evaluation, the "mosquitto_sub" output was redirected to the null device (/dev/null). In this way we could ensure that resources are not consumed to write messages, and each subscriber was configured to subscribe to the available published topics. In this way we made the server reaching its threshold at reasonable message publishing rates. Fig. 5 presents the evaluation environment topology.

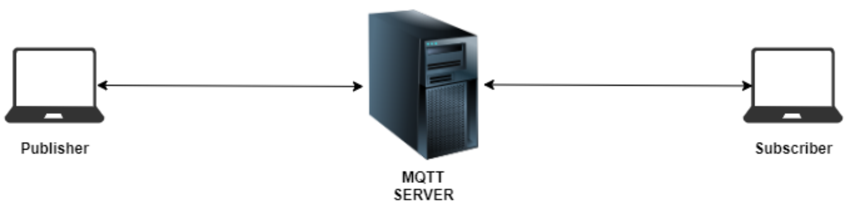

Figure 5. The evaluation environment topology.

4.1. Evaluation Scenario

This experiment was conducted on four widely used scalable and two non-scalable MQTT broker implementations. The other criteria for the selection of brokers were ease of availability and configurability. The tested brokers are: "Mosquitto 1.4.15" [35], "Bevywise MQTT Route 2.0" [36], "ActiveMQ 5.15.8" [37], "HiveMQ CE 2020.2" [38], "VerneMQ 1.10.2" [39] and "EMQ X 4.0.8" [40]. Out of these MQTT brokers, Mosquitto and Bevywise MQTT Route are non-scalable implementations, and the rest are scalable in nature. It is to be mentioned that Mosquitto is a single-threaded implementation, and Bevywise MQTT Route uses a dual thread approach, in which the first thread acts as an initiator of the second that processes messages. Table 4 presents an overview of the brokers.

Table 2: Hardware and software details of the local testing environment.

HW/SW Details	Publisher	MQTT Broker	Subscriber
CPU	64 bit AMD Ryzen 5 2500U @3.6 GHz	64 bit An Intel(R) Core(TM) i5-7260U CPU @2.20GHz	Intel(R) Pentium(R) CPU 2020M @2.40GHz
Memory	8GB, SODIMM DDR4 Synchronous Unbuffered (Unregistered) 2400 MHz (0.4 ns)	8GB, SODIMM DDR4 Synchronous Unbuffered (Unregistered) 2400 MHz (0.4 ns)	2GB, SODIMM DDR3 Synchronous 1600 MHz (0.6 ns)
Network	1Gbit/s, RTL8111/8168/8411 PCI Express Gigabit Ethernet Controller	1Gbit/s, Intel Ethernet Connection (4) I219-V	AR8161 Gigabit Ethernet, speed=1Gbit/s
HDD	WDC WD10SPZX-24Z (5400rpm), 1TB, connected over SATA 6gbps interface	WDC WD5000LPCX-2 (5400rpm), 500TB, connected over SATA 6gbps interface	HGST HTS545050A7
OS, Kernel	Elementary OS 5.1.4, Kernel 4.15.0-74-generic	Elementary OS 5.1.4, Kernel 4.15.0-74-generic	Linux Mint 19, Kernel 4.15.0-20-generic

Table 4: A bird's-eye view of the tested brokers.

MQTT Brokers	Mosquitto	Bevywise MQTT Route	ActiveMQ	HiveMQ CE	VerneMQ	EMQ X
OpenSource	Yes	No	Yes	Yes	Yes	Yes
Written in (prime programming language)	C	C, Python	Java	Java	Erlang	Erlang
MQTT Version	3.1.1, 5.0	3.x, 5.0	3.1	3.x, 5.1	3.x, 5.0	3.1.1
QoS Support	0, 1, 2	0, 1, 2	0, 1, 2	0, 1, 2	0, 1, 2	0, 1, 2
Operating System Support	Linux, Mac, Windows	Windows, Windows server, Linux, Mac and Raspberry Pi	Windows, Unix/ Linux/Cygwin	Linux, Mac, Windows	Linux, Mac OS X	Linux, Mac, Windows, BSD

Table 3: Hardware and software details of the cloud testing environment.

HW/SW details	Publisher/Subscriber/Server
Machine type	c2-standard-8 [30]
CPU	8 vCPUs
Memory	32 GB
Disk size	local 30 GB SSD
Disk type	Standard persistent disk
Network Tier	Premium
OS, Kernel	18.04.1-Ubuntu SMP x86_64 GNU/Linux, 5.4.0-1038-gcp

4.1.1. Evaluation conditions

All the brokers were configured to run on these test conditions, see Table 5, without authentication method enabled and RETAIN flag set to true. It is to be noted that with increase in the number of subscribers or the number of topics or message rate results in an increased load on the broker. In our test environment, with the combination of 3 different publishing threads (1 topic per thread) and 15 subscribers, we were able to push the broker to 100% process usage and limit the CPU usage on publisher and subscriber machines below 70% and 80% respectively.

Table 5: Test conditions for the experiment.

Number of topics:	3 (via 3 publishers threads)
Number of publishers:	3
Number of subscribers:	15 (subscribing to all 3 topics)
Payload:	64 bytes
Topic names used to publish large number of messages:	'topic/0', 'topic/1', 'topic/2'
Topic used to calculate latency	'topic/latency'

4.1.2. Latency Calculations

Latency is defined as the time taken by a system to transmit a message from a publisher to a subscriber [13]. This experiment tries to simulate a realistic scenario of a client trying to publish a message, when the broker is overloaded with a large number of messages on various topics from different clients. To achieve this, a different topic was used to send messages for latency calculations from the topics on which messages were fired to overload the system. It is noteworthy that an ideal broker implementation should always be able to efficiently process messages irrespective of the rate of messages fired to it.

4.1.3. Message Payload

By using the MQTT protocol, all messages are transferred using a single telemetry parameter [9]. Baring this in mind, we utilized a small payload size not to overload the server memory. Concerning the message payload size setting, we used 64 bytes for the entire testing.

4.2. Benchmarking Results

We separate our experimental results into three distinct segments for better interpretation and understanding. We had taken 3 samples for each QoS in each segment and the best result with the maximum rate of message delivery, and zero message drop was considered for comparison. The three different categories are:

1. Projected message processing rates of non-scalable brokers at 100% process CPU usage. See Table 6, 9.
2. Projected message processing rates of scalable brokers at 100% system CPU usage. See Table 7, 10.
3. Latency comparison of all the brokers (both scalable and non-scalable brokers) – see Table 8, 11.

Table 8: Latency comparison of all the brokers in local test environment.

Brokers	Average latency in ms.		
	QoS0	QoS1	QoS2
Mosquitto1.4.15	1.65	0.74	1.38
Bevywise MQTT Route 2.0	1.13	0.96	1.53
ActiveMQ 5.15.8	2.33	1.38	2.14
HiveMQ CE 2020.2	7.69	58.48	3.66
VerneMQ 1.10.2	1.53	1.98	2.89
EMQ X 4.0.8	1.34	0.87	3.68

Table 6: Projected message processing rates of non-scalable brokers at 100% process CPU usage (local test results).

QoS	**QoS0**		**QoS1**		**QoS2**	
Observations/Broker (non-scalable)	**Mosquitto 1.4.15**	**Bevywise MQTT Route 2.0**	**Mosquitto 1.4.15**	**Bevywise MQTT Route 2.0**	**Mosquitto 1.4.15**	**Bevywise MQTT Route 2.0**
Peak message rate (msgs/sec)	32016.00	32839.00	9488.00	3542.49	6585.00	2649.00
Average process CPU usage(%) at above message rate	84.29	97.93	89.00	95.79	96.73	98.20
Projected message processing rate at 100% process CPU usage	37983.15	33533.14	10660.67	3698.18	6807.61	2697.56
Average latency (in ms)	1.65	1.13	0.74	0.96	1.38	1.53
Important note*	Mosquitto and Bevywise are fixed thread/single thread broker implementations. They cannot scale up to utilize all cores available in the system.					

Table 7: Projected message processing rates of scalable brokers at 100% system CPU usage (local test results).

QoS	**QoS0**				**QoS1**				**QoS2**			
Observations/Broker (scalable)	**ActiveMQ 5.15.8**	**HiveMQ CE 2020.2**	**VerneMQ 1.10.2**	**EMQ X 4.0.8**	**ActiveMQ 5.15.8**	**HiveMQ CE 2020.2**	**VerneMQ 1.10.2**	**EMQ X 4.0.8**	**ActiveMQ 5.15.8**	**HiveMQ CE 2020.2**	**VerneMQ 1.10.2**	**EMQ X 4.0.8**
Peak message rate (msgs/sec)	39479.00	8748.00	11760.00	18034.00	12873.00	708.00	4655.00	4633.41	10508.00	579.00	2614.00	2627.31
Average system CPU usage(%) at above message rate	91.78	97.93	96.51	98.71	92.56	63.44	97.34	96.82	90.91	64.28	96.79	95.54
Projected message processing rate at 100% system CPU usage	43014.82	8932.91	12185.27	18269.68	13907.74	1116.02	4782.21	4785.59	11558.68	900.75	2700.69	2749.96
Average latency (in ms)	2.33	7.69	1.53	1.34	1.38	58.48	1.98	0.87	2.14	3.66	2.89	3.68
Important note*	All the brokers listed in this table are scalable in nature and can utilize all cores available in the system.											

5. Discussion

5.1. Local evaluation results

In Table 6, we present a comparative performance analysis of non-scalable MQTT brokers. For non-scalable brokers like Mosquitto and Bevywise MQTT Route, the projected message rate at 100% CPU usage (R_{ns}) can be calculated with the below Equation 1:

$$R_{ns} = \frac{Peak\ Message\ rate}{Average\ Process\ CPU\ Usage} * 100 \tag{1}$$

Note: *The CPU usage of a process (process CPU usage) is a measure of how much (in percentage) of the CPU's cycles are committed to the process that is currently running. Average process CPU utilization indicates the observed average of CPU utilization by the process during the experiment [41].*

In this segment, Mosquitto 1.4.15 beats Bevywise MQTT Route 2.0 in terms of projected message processing rate at approximately 100% process CPU usage across all the QoS categories. See Figure 6.

It is to be mentioned that being non-scalable Mosquitto and Bevywise MQTT Route cannot make use of all available cores on the system. In terms of average latency (round trip time), we found that at QoS0 Bevywise MQTT Route 2.0 leads the race, while in all other QoS categories (QoS1 and QoS2), Mosquitto 1.4.15 occupies the top spot. See Figure 7.

Table 7 shows the benchmarking results of scalable broker implementations. In this comparison, ActiveMQ 5.15.8 beats all other broker implementations (HiveMQ CE 2020.2, VerneMQ 1.10.2, EMQ X 4.0.8) in terms of "average latency" across all QoS categories. See Figure 7.

In a multi-core or distributed environment, a scalable broker implementation would scale up to utilize the maximum system resources available. Hence, the CPU utilization data sum up the CPU utilization by the process group consisting of all sub-processes/threads. The process group CPU utilization for scalable brokers can reach up to 100*n% (where n = the number of cores available in the system). Here, in this test environment as n = 4, the CPU utilization percent for the deployed brokers could go up to 400%. This comparison gives a fair idea of how various brokers scale up and

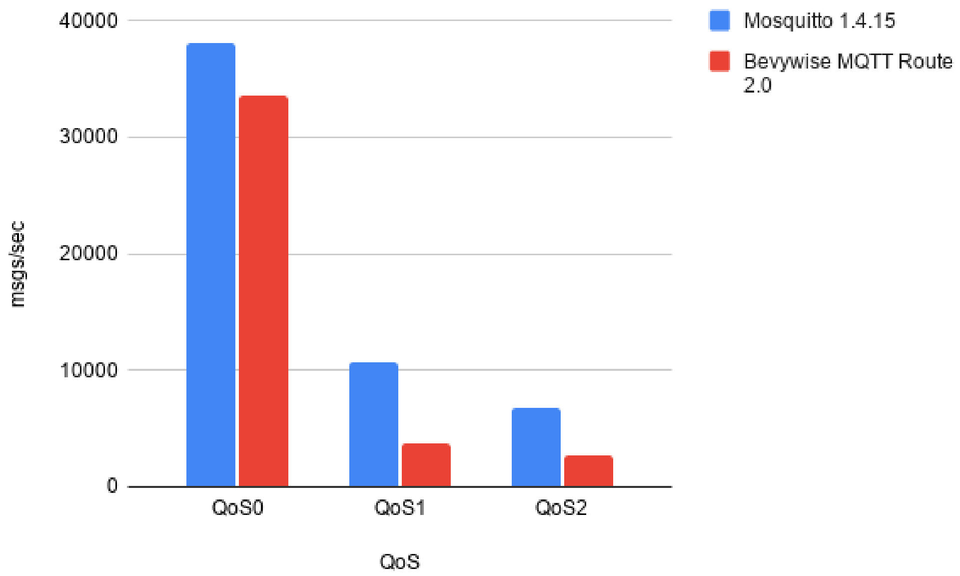

Figure 6. Projected message rate (msgs/sec) of non-scalable brokers at ~100% process CPU usage in the local evaluation environment.

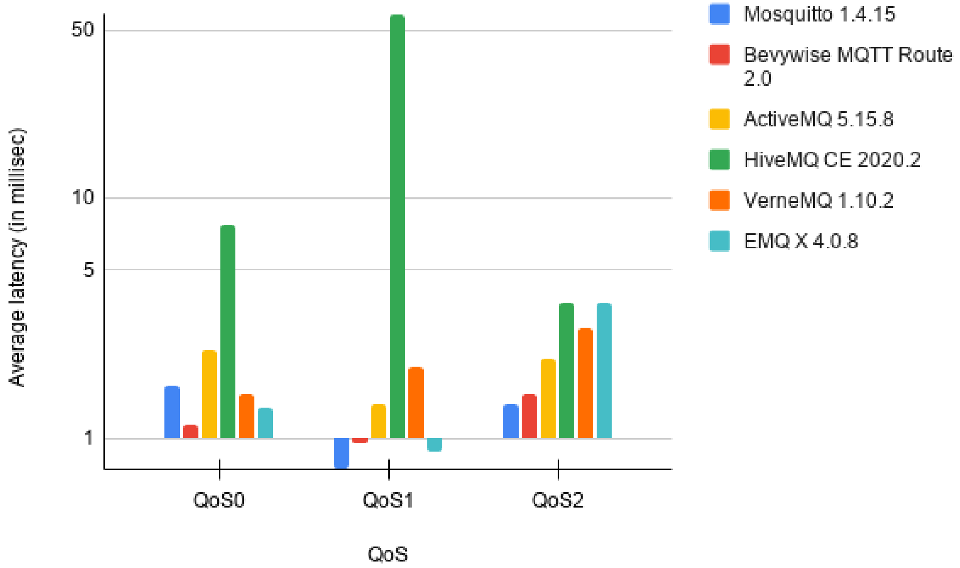

Figure 7. A comparison of average latency of all scalable and non-scalable brokers in the local evaluation environment.

perform when they are deployed on a multi-core set-up. For scalable brokers, Equation 2 calculates the projected message rate at the unrestricted resource (CPU) (R_s):

$$R_s = \frac{Peak\ Message\ rate}{Average\ System\ CPU\ Usage} * 100 \tag{2}$$

Note: *System CPU usage refers to how the available processors whether real or virtual in a System are being utilized. Average System CPU usage refers to the observed average system CPU utilization by the process during the experiment [42].*

At QoS0, in terms of the projected message processing rate at 100% system CPU usage, EMQ X leads the race, at QoS1 and QoS2 ActiveMQ seems to be showing the best performance among all the brokers put to test; see Figure 8.

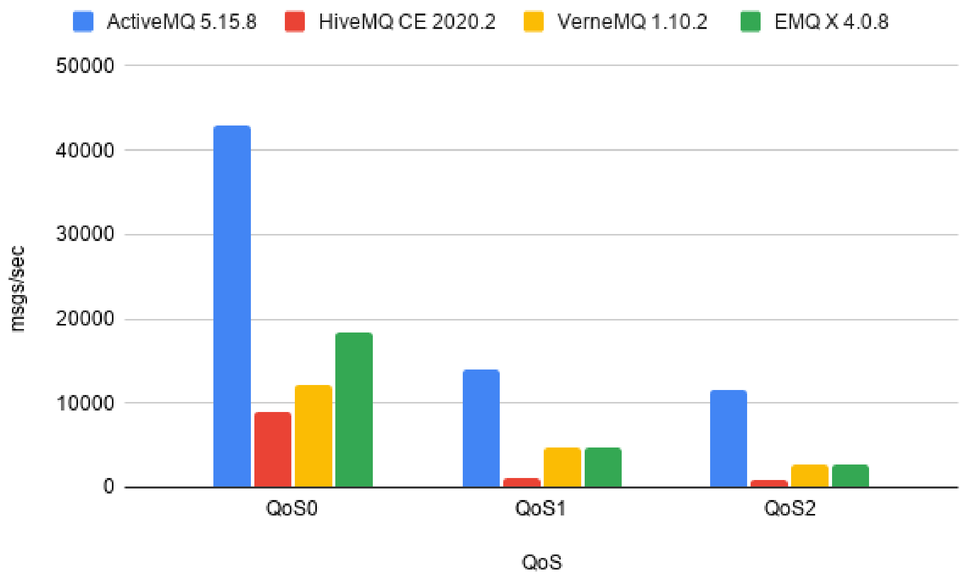

Figure 8. Projected message rate (msgs/sec) of scalable brokers at 1̃00% system CPU usage in the local evaluation environment.

Sorting all the MQTT brokers according to the message processing capability with full system resource utilization (from highest to lowest: left to right) - At QoS0: ActiveMQ, Mosquitto, Bevywise MQTT Route, EMQ X, VerneMQ, HiveMQ CE. At QoS1: ActiveMQ, Mosquitto, EMQ X, VerneMQ, Bevywise MQTT Route, HiveMQ CE. At QoS2: ActiveMQ, Mosquitto, VerneMQ, Bevywise MQTT Route, HiveMQ CE.

Table 7 shows a side-by-side comparison of both scalable and non-scalable brokers in terms of average latency recorded. Sorting all the tested brokers according to the average latency recorded (from lowest to highest: left to right) - At QoS0: Bevywise MQTT Route, EMQ X, VerneMQ, Mosquitto, ActiveMQ, HiveMQ CE. At QoS1: Mosquitto, EMQ X, Bevywise MQTT Route, ActiveMQ, VerneMQ, HiveMQ CE. At QoS2: EMQ X, Mosquitto, Bevywise MQTT Route, HiveMQ CE, VerneMQ, ActiveMQ.

5.2. Cloud-based evaluation results

In this subsection, we discuss the performance of MQTT brokers on the Google Cloud test environment. It is to be mentioned that the stress testing on MQTT brokers in the cloud environment is done with the latest versions of the brokers available. Table 9 lists average latency and projected message processing rates of non-scalable brokers at 100% CPU usage. In terms of projected message processing rate and average latency recorded Mosquitto 2.0.7 beats Bevywise MQTT 3.1- build 0221-01; see Figure 9 and 10.

Table 10 shows the benchmarking results of scalable broker implementations. In this comparison, ActiveMQ 5.16.1 beats all other broker implementations (HiiveMQ CE 2020.2, VerneMQ 1.11.0, EMQX 4.2.7) in terms of the projected message processing rate at 100% system CPU usage across all QoS categories. Concerning the average latency recorded, EMQX 4.2.7 leads the race at QoS0, VerneMQ 1.11.0 tops at QoS1, and ActiveMQ 5.16.1 leads at QoS2 among all the scalable brokers put to test. See Figure 11 and 10.

Sorting all the tested MQTT brokers according to the message processing capability with full system resource utilization (from highest to lowest: left to right) - At QoS0: ActiveMQ, EMQX, Mosquitto, HiveMQ, VerneMQ, Bevywise MQTT Route. At QoS1:

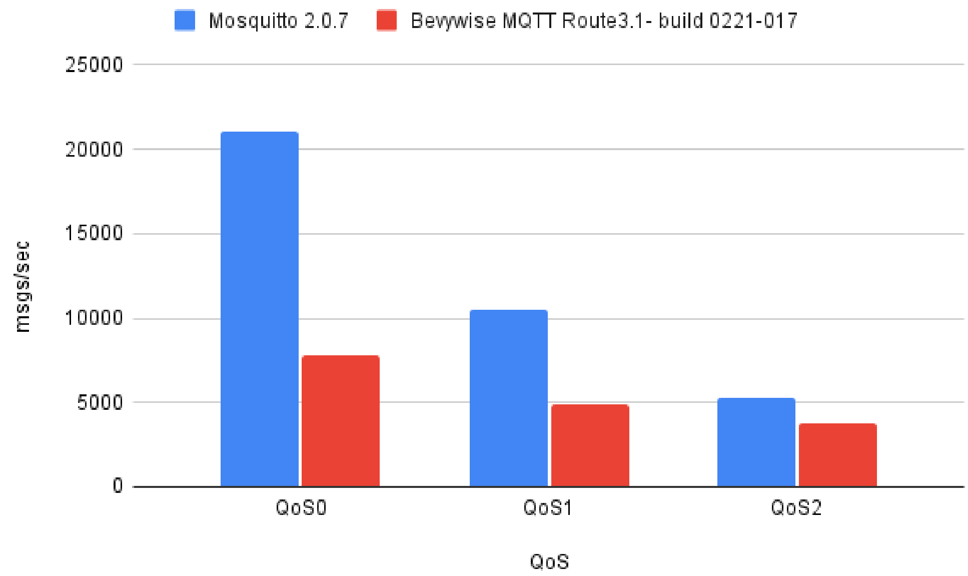

Figure 9. Projected message rate (msgs/sec) of non-scalable brokers at ~100% process CPU usage in the cloud evaluation environment.

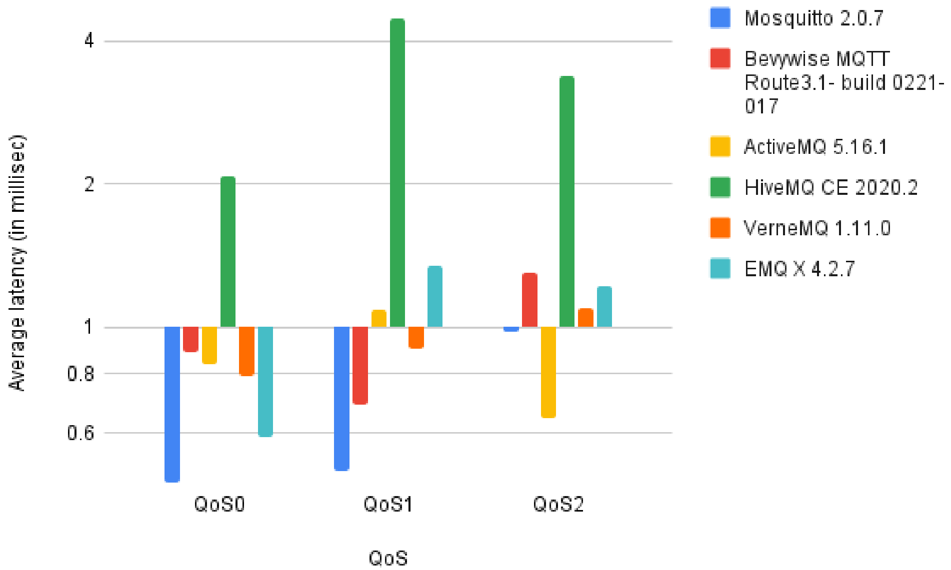

Figure 10. A comparison of average latency of all scalable and non-scalable brokers in the cloud evaluation environment.

ActiveMQ, EMQX, HiveMQ, Mosquitto, Bevywise MQTT Route, VerneMQ. At QoS2: ActiveMQ, EMQX, HiveMQ, Mosquitto, Bevywise MQTT Route, VerneMQ.

Table 10 shows a side-by-side comparison of both scalable and non-scalable brokers in terms of average latency recorded. Sorting all the tested brokers according to the average latency recorded (from lowest to highest: left to right)- At QoS0: Mosquitto, EMQ X, VerneMQ, Bevywise MQTT Route, HiveMQ. At QoS1: Mosquitto, Bevywise MQTT Route, VerneMQ, EMQX, HiveMQ. At QoS2: ActiveMQ, Mosquitto, VerneMQ, EMQ X, Bevywise MQTT Route, HiveMQ.

To summarize our evaluation experiments, we can state that ActiveMQ scales well to beat all other brokers' performance on our local testbed (using a 4 core/8GB

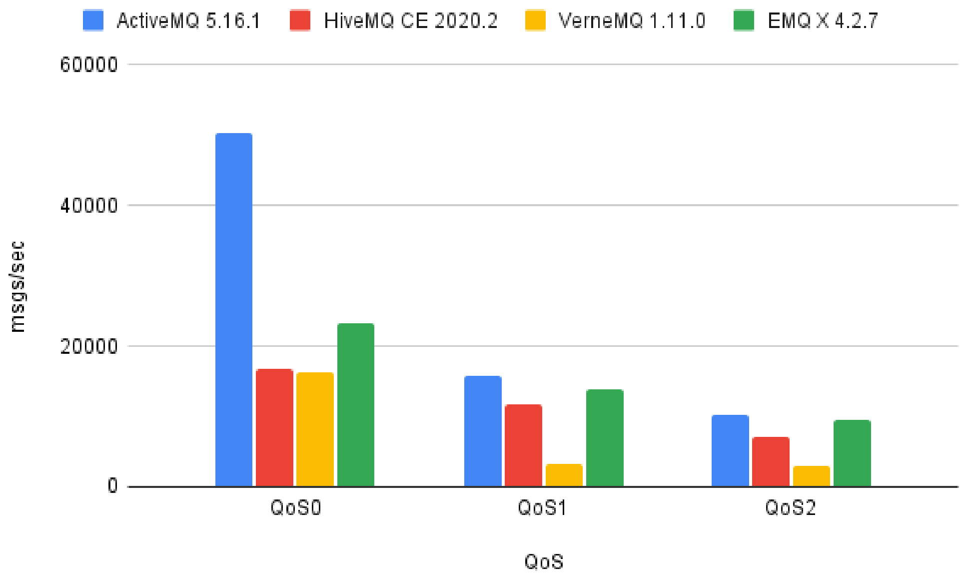

Figure 11. Projected message rate (msgs/sec) of scalable brokers at ~100% system CPU usage in cloud evaluation environment.

machine), and cloud testbed (on an 8 vCPU/32GB machine). It is the best scalable broker implementation we have tested so far. EMQ X, VerneMQ, HiveMQ CE also perform reasonably well in our test environment. On the other hand, if the hardware is resource-constrained (CPU/Memory/IO/Performance) or having a lower specification, than the local testbed used in this experiment, then Mosquitto or Bevywise MQTT Route can be taken as better choices over other scalable brokers. Another important point to observe is that when we moved from a local testing environment to a cloud testing environment having stronger hardware specification in terms of number of cores and memory, significant improvement in latency is shown by each of the brokers.

6. Conclusion

M2M protocols are the foundation of Internet of Things communication. There are many M2M communication protocols such as MQTT, CoAP, AMQP, and HTTP, are available. In this work, we reviewed and evaluated the performance of six MQTT brokers in terms of message processing rate at 100% process group CPU utilization, normalized message rate at unrestricted resource (CPU) usage, and average latency by putting the brokers under stress test.

Our results showed that broker implementations like Mosquitto and Bevywise could not scale up automatically to make use of the available resources, yet they performed better than other scalable brokers on a resource-constrained environment. Mosquitto was the best performing broker in the first evaluation scenario, followed by Bevywise. However, in a distributed/multi-core environment, ActiveMQ performed the best. It scaled well, and showed better results than all other scalable brokers we put to test. The findings of this research highlight the significance of the relationship between MQTT broker system design and its performance under stress testing. It aims to fill the gap of lack of test-driven information on the topic, and helps real-time system developers to a great extent in building and deploying smart IoT solutions.

In the future, we would like to continue our evaluations in a more heterogeneous cloud deployment, and further study the scalability aspects of bridged MQTT broker implementations.

Table 9: Projected message processing rates of non-scalable brokers at 100% process CPU usage (cloud test results)

QoS	QoS0		QoS1		QoS2	
Observations/ Broker (non-scalable)	Mosquitto 2.0.7	Bevywise MQTT Route 3.1- build 0221-017	Mosquitto 2.0.7	Bevywise MQTT Route 3.1- build 0221-017	Mosquitto 2.0.7	Bevywise MQTT Route 3.1- build 0221-017
Peak message rate (msgs/sec)	17946.00	7815.00	8927.00	4861.00	4423.00	3688.00
Average process CPU usage(%) at above message rate	85.12	100.31	84.70	100.34	83.24	97.91
Projected message processing rate at 100% process CPU usage	21083.18	7790.85	10539.55	4844.53	5313.55	3766.72
Average latency (in ms)	0.47	0.89	0.50	0.69	0.98	1.30
Important note*	Mosquitto and Bevywise are fixed thread/single thread broker implementations. They cannot scale up to utilize all the cores available in the system.					

Table 10: Projected message processing rates of scalable brokers at 100% system CPU usage (cloud test results)

QoS	QoS0				QoS1				QoS2			
Observations/ Broker (non-scalable)	ActiveMQ 5.16.1	HiveMQ CE 2020.2	VerneMQ 1.11.0	EMQX Broker 4.2.7	ActiveMQ 5.16.1	HiveMQ CE 2020.2	VerneMQ 1.11.0	EMQX Broker 4.2.7	ActiveMQ 5.16.1	HiveMQ CE 2020.2	VerneMQ 1.11.0	EMQX Broker 4.2.7
Peak message rate (msgs/sec)	41697.00	13338.00	14332.00	17838.00	9663.00	8188.00	2622.00	11054.00	6196.00	4887.00	2240.00	7342
Average process CPU usage (%) at above message rate	82.77	80.09	88.29	76.83	60.73	70.43	82.16	79.28	59.97	68.32	72.70	76.84
Projected message rate at 100% system CPU usage	50376.95	16653.76	16232.87	23217.49	15911.41	11625.73	3191.33	13942.99	10331.83	7153.10	3081.16	9554.92
Average latency (in ms)	0.83	2.07	0.79	0.59	1.09	4.48	0.90	1.35	0.64	3.38	1.10	1.22
Important note*	All the brokers listed in this table are scalable in nature and can utilize all cores available in the system.											

Table 11: Latency comparison of all the brokers in the cloud evaluation environment.

Brokers	**Average latency in ms.**		
	QoS0	**QoS1**	**QoS2**
Mosquitto 2.0.7	0.47	0.50	0.98
Bevywise MQTT Route 3.1- build 0221-017	0.89	0.69	1.30
ActiveMQ 5.16.1	0.83	1.09	0.64
HiveMQ CE 2020.2	2.07	4.48	3.38
VerneMQ 1.11.0	0.79	0.90	1.10
EMQ X 4.2.7	0.59	1.35	1.22

Author Contributions: Conceptualization, A.K., B.M and B.M.; methodology, A.K, B.M and B.M; software, B.M and B.M; validation, A.K., B.M and B.M.; formal analysis,A.K., B.M and B.M; investigation, B.M and B.M.; resources, B.M and B.M.; data curation, B.M and B.M.; writing—original draft preparation, B.M.; writing—review and editing, A.K., B.M.; visualization, B.M; supervision, A.K.; project administration, A.K.; funding acquisition, A.K. All authors have read and agreed to the published version of the manuscript.

Funding: The research leading to these results was supported by the Hungarian Government and the European Regional Development Fund under the grant number GINOP-2.3.2-15-2016-00037 ("Internet of Living Things"). The experiments presented in this paper are based upon work supported by Google Cloud.

Data Availability Statement: The source code of our benchmarking tool called MQTT Blaster we used for the analysis is available on GitHub [29]. The measurement data we gathered during the evaluation are shared in the tables and figures of this paper.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. IHS Markit, Number of Connected IoT Devices Will Surge to 125 Billion by 2030. Available at: https://technology.ihs.com/596542/. Accessed Aug. 06, 2020
2. Karagiannis V, Chatzimisios P, Vazquez-Gallego F, Alonso-Zarate J. "A survey on application layer protocols for the internetof things," Transaction on IoT and Cloud computing 015; 3(1): 11–17. 2015
3. N. Naik, “Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP”, 2017 IEEE International Systems Engineering Symposium (ISSE), 2017, doi: 10.1109/syseng.2017.8088251
4. S. Bandyopadhyay and A. Bhattacharyya, “Lightweight Internet protocols for web enablement of sensors using constrained gateway devices”, 2013 International Conference on Computing, Networking and Communications (ICNC), 2013, doi: 10.1109/iccnc.2013.6504105

5. “MQTT v5.0”. Available at: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html. Accessed Aug. 06, 2020)
6. “Messaging Technologies for the Industrial Internet and the Internet Of things”. Available at: https://www.smartindustry.com/assets/Uploads/SI-WP-Prismtech-Messaging-Tech.pdf. Accessed Aug. 06, 2020
7. N. S. Han, "Semantic service provisioning for 6lowpan: powering internet of things applications on web", Ph.D. thesis (2015)
8. R. Kawaguchi and M. Bandai, “Edge Based MQTT Broker Architecture for Geographical IoT Applications”, 2020 International Conference on Information Networking (ICOIN), 2020, doi: 10.1109/icoin48656.2020.9016528
9. Y. Sasaki and T. Yokotani, “Performance Evaluation of MQTT as a Communication Protocol for IoT and Prototyping,” Advances in Technology Innovation, vol. 4, no. 1, pp. 21 - 29, 2019
10. A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari and M. Ayyash, "Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications," in IEEE Communications Surveys & Tutorials, vol. 17, no. 4, pp. 2347-2376, Fourthquarter 2015, doi: 10.1109/COMST.2015.2444095
11. Farhan, L., Kharel, R., Kaiwartya, O., Hammoudeh, M., and Adebisi, B. (2018). Towards green computing for Internet of things: Energy oriented path and message scheduling approach. Sustainable Cities and Society, 38, 195-204.
12. B. Mishra and B. Mishra, "Evaluating and Analyzing MQTT Brokers with stress testing," in The 12th Conference of PHD Students in Computer Science, CSCS 2020, Szeged, Hungary, June 24 – June 26, 2020, pp. 32-35. Available at: http://www.inf.u-szeged.hu/~cscs/pdf/cscs2020.pdf. Accessed Jan. 06, 2021
13. “Mishra, B., and Kertesz, A. (2020). The use of MQTT in M2M and IoT systems: A survey. IEEE Access, 8, 201071-201086.)
14. “Pub/Sub: A Google-Scale Messaging Service | Google Cloud”. Available at: https://cloud.google.com/pubsub/architecture. Accessed Aug. 06, 2020
15. C. Wenzhi, Liubai and F. Zhenzhu, "Bayesian Network Based Behavior Prediction Model for Intelligent Location Based Services," 2006 2nd IEEE/ASME International Conference on Mechatronics and Embedded Systems and Applications, Beijing, 2006, pp. 1-6, doi: 10.1109/MESA.2006.296936
16. P. Jutadhamakorn, T. Pillavas, V. Visoottiviseth, R. Takano, J. Haga, and D. Kobayashi, “A scalable and low-cost MQTT broker clustering system”, 2017 2nd International Conference on Information Technology (INCIT), 2017, doi: 10.1109/incit.2017.8257870
17. “MQTT and CoAP, IoT Protocols | The Eclipse Foundation”. Available at: https://www.eclipse.org/community/eclipse_newsletter/2014/
february/article2.php. Accessed June 10, 2021
18. Foster, Andrew. "Messaging technologies for the industrial internet and the internet of things." PrismTech Whitepaper (2015): 21
19. B.Mishra. "Performance evaluation of MQTT broker servers." In International Conference on Computational Science and Its Applications, pp. 599-609. Springer, Cham, 2018
20. 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
21. S. Lee, H. Kim, D. Hong and H. Ju, "Correlation analysis of MQTT loss and delay according to QoS level," The International Conference on Information Networking 2013 (ICOIN), Bangkok, 2013, pp. 714-717, doi: 10.1109/ICOIN.2013.6496715
22. W. Pipatsakulroj, V. Visoottiviseth, and R. Takano, “muMQ: A lightweight and scalable MQTT broker”, 2017 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN), 2017, doi: 10.1109/lanman.2017.7972165
23. Bondi, André B. "Characteristics of scalability and their impact on performance." Proceedings of the 2nd international workshop on Software and performance. 2000
24. A. Detti, L. Funari and N. Blefari-Melazzi, "Sub-linear Scalability of MQTT Clusters in Topic-based Publish-subscribe Applications," in IEEE Transactions on Network and Service Management, doi: 10.1109/TNSM.2020.3003535
25. Thangavel, Dinesh, et al. "Performance evaluation of MQTT and CoAP via a common middleware." 2014 IEEE ninth international conference on intelligent sensors, sensor networks and information processing (ISSNIP). IEEE, 2014
26. Chen, Yuang, and Thomas Kunz. "Performance evaluation of IoT protocols under a constrained wireless access network." 2016 International Conference on Selected Topics in Mobile & Wireless Networking (MoWNeT). IEEE, 2016
27. Pham, Manh Linh, Truong Thang Nguyen, and Manh Dong Tran. "A Benchmarking Tool for Elastic MQTT Brokers in IoT Applications." International Journal of Information and Communication Sciences 4.4 (2019): 70-78
28. Bertrand-Martinez, Eddas, et al. "Classification and evaluation of IoT brokers: A methodology." International Journal of Network Management (2020): e2115
29. Koziolek, Heiko, Sten Grüner, and Julius Rückert. "A Comparison of MQTT Brokers for Distributed IoT Edge Computing." European Conference on Software Architecture. Springer, Cham, 2020
30. “Documentation | Google Cloud”. Available at: https://cloud.google.com/docs. Accessed 24 Mar. 2021
31. “Using Network Service Tiers | Google Cloud.”. Available at: https://cloud.google.com/network-tiers/docs/using-network-service-tiers. Accessed 24 Mar. 2021
32. “Machine Types | Compute Engine Documentation | Google Cloud”. Available at: Google Cloud, https://cloud.google.com/compute/docs/machine-types. Accessed 24 Mar. 2021
33. “paho-mqtt PyPI”. Available at: https://pypi.org/project/paho-mqtt/. Accessed Aug. 10, 2020
34. "MQTT Blaster". Available at: https://github.com/MQTTBlaster/MQTTBlaster. Accessed June 28, 2021
35. “Mosquitto man page | Eclipse Mosquitto”. Available at: https://mosquitto.org/man/mosquitto-8.html. Accessed Aug. 10, 2020

36. "MQTT Broker Developer documentation-MQTT Broker-Bevywise". Available at: https://www.bevywise.com/mqtt-broker/developer-guide.html. Accessed Aug. 10, 2020
37. "ActiveMQ Classic". Available at: https://activemq.apache.org/components/classic/. Accessed Aug. 10, 2020
38. "HiveMQ Community Edition 2020.3 is released". Available at: https://www.hivemq.com/blog/hivemq-ce-2020-3-released/. Accessed Aug. 10, 2020
39. "Getting Started - VerneMQ". Available at: https://docs.vernemq.com/getting-started. Accessed Aug. 10, 2020
40. "MQTT Broker for IoT in 5G Era | EMQ". Available at: https://www.EMQX.io/. Accessed Aug. 10, 2020
41. "Understanding CPU Usage in Linux | OpsDash". Available at: https://www.opsdash.com/blog/cpu-usage-linux.html/. Accessed Sep. 03, 2021
42. "System CPU Utilization workspace | IBM". Available at: https://www.ibm.com/docs/en/om-zos/5.6.0?topic=workspaces-system-cpu-utilization-workspace/. Accessed Sep. 03, 2021