Performance Evaluation of MQTT Broker Servers

Biswajeeban Mishra(✉)

Department of Software Engineering, Dugonicster 13, Szeged 6720, Hungary
mishra@inf.u-szeged.hu

Abstract. Internet of Things (IoT) is a rapidly growing research field, which has enormous potential to enrich our lives for a smarter and better world. Significant improvements in telemetry technology make it possible to quickly connect things (i.e. different smart devices) situated at different geographical locations. Telemetry technology helps to monitor and measure the devices from remote locations, making them even more useful and productive at a low cost of management. MQTT (MQ Telemetry Transport) is a lightweight messaging protocol that meets today's smarter communication needs. The protocol is used for machine-to-machine communication and plays a pivotal role in IoT. In case the network bandwidth is low, or a network has high latency, and for devices having limited processing capabilities and memory, MQTT is able to distribute telemetry information using a publish/subscribe communication pattern. It enables IoT devices to send or publish information on a topic head to a server (i.e. MQTT broker), then it sends the information out to those clients that have previously subscribed to that topic. This paper puts several publicly available brokers and locally deployed brokers into experiment and compares their performance by subscription throughput i.e., in how much time a broker pushes a data packet to the client (the subscriber) or how much time a data packet takes to reach the client (the subscriber) from the broker. MQTT brokers based on the latest MQTT v3.1.1 version were evaluated. The paper also includes mqtt-stresser and mqtt-bench stress test results of both locally and publicly deployed brokers.

Keywords: Internet of Things · MQTT · MQTT brokers · Cloud computing

1 Introduction

With the rise of the Internet of Things (IoT), billions of embedded smart devices and sensors are interconnected, and they exchange data using the existing Internet infrastructure. They enormously impact and improve our life. In today's fast-growing world, many IoT application areas exist, starting from manufacturing, automobile, agriculture, energy management, environmental monitoring to smart cities and defense sector. For example, a patient's real-time health data, such as blood pressure, heart-bit rate, etc. can be monitored by a doctor from a far distant location. The supplying company can trace leakage in oil and gas pipelines from a central control room, and supply can immediately be stopped to avoid accidents [1]. Trespassing events across the borders of a

O. Gervasi et al. (Eds.): ICCSA 2018, LNCS 10963, pp. 599–609, 2018.
https://doi.org/10.1007/978-3-319-95171-3_47

nation can be detected and notified to the appropriate authorities for necessary action. All these IoT networks use several radio technologies such as RFID (radio-frequency identification), WLAN (wireless local area network), WPAN (wireless personal area network) or WMAN (wireless metropolitan area networks)[1] to create a Machine-to-Machine (M2M) network [2]. No matter which radio technology is used to operate an M2M network, the end device or machine must make their data available to the Internet [2, 3]. Many M2M data transfer protocols are available for IoT systems, amongst them, MQTT, CoAP, AMQP, and HTTP are the widely accepted ones [4–7]. Considering message size vs. message overhead, power consumption vs. resource requirement, reliability/QoS vs. interoperability, bandwidth vs. latency, security vs. provisioning, or M2M/IoT usage vs. standardization, MQTT stands tall among all [2]. In this paper we have investigated the performance of several MQTT brokers, and compared their properties concerning subscription throughput. The research question was- *In standard domestic deployment use case, is there any difference in performance of different MQTT broker distributions at standard TCP/IP level? And how the same brokers' performance varies when they are put under stress test?* In the following section we discuss a bit more about MQTT protocol. In Sect. 3, details of the test scenario for public and locally deployed brokers are given. Evaluation results are discussed in Sect. 4. Finally, the paper is concluded in Sect. 5.

2 More About MQTT

In this section we briefly discuss the basic components of MQTT. First subsection talks about MQTT protocol and MQTT Packet format. Second gives information about MQTT Quality of Service levels and the MQTT control packets associated with various QoS levels.

2.1 MQTT Packet Format

MQTT works over IANA registered TCP port numbers 8883 and 1883. The port number 8883 is used for using MQTT over SSL/TSL, and 1883 is used for non-TLS communications [11]. In MQTT, MQTT brokers play a major role. A client otherwise called as a publisher usually sends its messages to a broker (MQTT Server) on different topic-tags and consumers otherwise known as receivers subscribe to these topics to receive the messages. An MQTT Broker can handle up to 45 thousand concurrent connected MQTT clients [13] and is responsible for authenticating (publishers and subscribers) receiving, filtering, sorting and sending messages to clients. A plaintext MQTT message has a fixed-length header of 2 Bytes: an optional message-length header, and a message payload header. The first eight bits of MQTT TCP Packet Format is used as fixed MQTT header, where the first four bits represent the Message Type, the fifth bit is used for the duplicate (DUP) flag, the sixth and seventh bits are

[1] The IEEE 802.16 Working Group on Broadband Wireless Access Standards, http://www.ieee802.org/16/.

used for the QoS (Quality of Service) level and the eighth bit is used for "retain" message service. The second 8 bits are reserved for the variable length header, and the optional header which can be used for TLS payloads [11, 12] See Fig. 1.

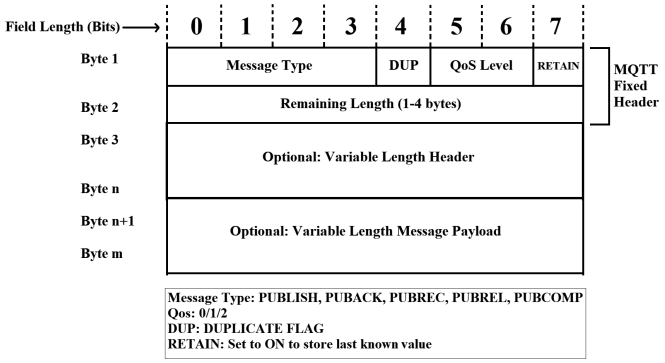

Fig. 1. TCP MQTT Packet Format

2.2 Quality of Service

MQTT provides three 'Quality of service' levels for delivering messages to an MQTT Broker and to any client (ranging from 0 to 2). In QoS 0, a message will not be acknowledged by the receiver or stored and delivered by the sender, only the payload is sent or received. This is often called "fire and forget." It is the minimal level and guarantees the best delivery effort. Only PUBLISH MQTT Control Packet is used to publish messages. In QoS 1 acknowledgment is assured. Data loss may occur. At least once delivery is guaranteed. Here PUBACK (Publish Acknowledgement) control packet is used. The client keeps a copy of the message until it receives PUBACK- Control Packet from the server for Publication Acknowledgement. If PUBACK does not get received within a stipulated time, the client resends the PUBLISH packet, and the duplicate (DUP) flag is set to 1. The (DUP) flag is used only for internal purposes by the programmers, and the broker continues to send PUBACK control packet regardless of the status of the DUP flag. In QoS 2, data delivery is assured. Exactly once delivery is guaranteed. QoS uses PUBLISH- Publish messages, PUBREC - Publish Received, PUBREL - Publish Release, and PUBCOMP - Publish Complete MQTT control packets. When the broker receives a QoS 2 PUBLISH packet, it processes the request and acknowledges the publishing client with a PUBREC message. After the client receives the PUBREC, it discards previously stored PUBLISH packet as PUBREC packet confirms the client that broker has successfully received the message and responds the server with a PUBREL packet. As the PUBREL packet reaches the server, it discards all the stored information about the publishing partner and sends a PUBCOMP packet to it to mark the completion of the process. The server stores an identifying reference to the publishing partner until it sends PUBCOMP message. This process eliminates the possibility of duplicate delivery of messages [10–12].

3 Test Scenario

3.1 For Publicly Available Brokers

In this section we introduce the test scenario for evaluation of publicly available MQTT brokers [4], the considered ones are mentioned in Table 1. In our experiment See 0, live environment event data were sent from a Raspberry PI3 Board to the cloud using the MQTT Protocol. So, on Raspberry PI Board side, an MQTT client, called "publisher" was created using a Node-Red programming language See Fig. 2. to read environment event values from onboard temperature, humidity, and pressure sensors, and publish them on a given topic-tag to an MQTT message broker server at a rate of approximately one message per second. On the receiving end, another MQTT client, called "subscriber" was created to subscribe to the publishing topic, and receive data See Fig. 3. Eclipse Foundation recommended MQTT data capture tool 'MQTT Spy' [16] was used to capture, save and analyze the received data. The goal was to evaluate overall topic-specific message load and broker payload of each broker. The evaluation parameters are depicted in Table 2.

Table 1. Publicly hosted MQTT Brokers

Type	Mosquitto	HiveMQ	Bevywise
Address	test.mosquitto.org	broker.mqttdashboard.com	mqttserver.com
Port	1883	1883	1883 (TCP)
Sign-up needs	No	No	Yes

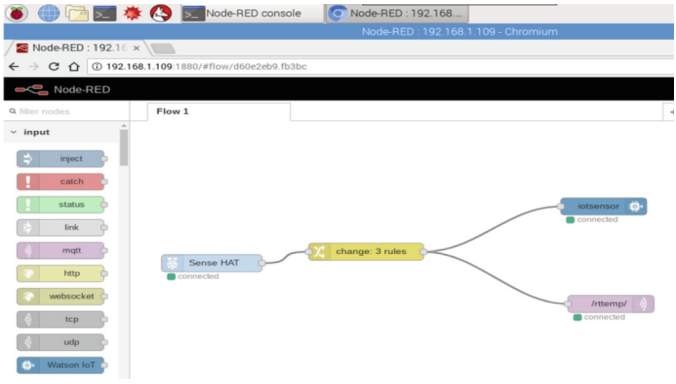

Fig. 2. NodeRed program flow to send RPi sensor data to public Brokers

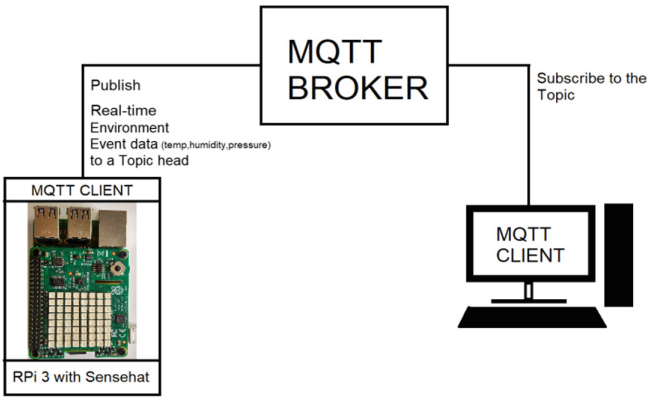

Fig. 3. Public broker experiment scenario

Table 2. Published message properties and publishing conditions for public Brokers

QoS	0/1/2
Payload	14 Bytes
Keep Alive Time (in seconds)	60
Clean Session	True
Total Number of message published	1000

3.2 For Locally Deployed Brokers

The following MQTT brokers are deployed in a local computing environment with default server configurations: ActiveMQ v5.15.2[2], Bevywise MQTT Route[3], HiveMQ 3.3[4], Mosquitto-1.4.14[5], and RabbitMQ v3.7.2[6]. Concerning the local test environment, all servers (broker and client instances) were deployed in a hardware and software setup shown in Table 3.

Table 3. Local hardware and software configurations

Hardware		Software	
Processor	Intel® Core™i5-5200U CPU@ 2.20 GHz × 4	OS	Ubuntu 16.04.3
		OS TYPE	64-bit
RAM	8 GB	KERNEL VERSION	4.10.0-42-generic
Disk	SATA 3.0, 6.0 Gb/s 5400 rpm	JAVA VERSION	Java version "9.0.1"

[2] http://activemq.apache.org.
[3] https://www.bevywise.com/mqtt-broker/.
[4] https://www.hivemq.com.
[5] https://mosquitto.org.
[6] https://www.rabbitmq.com.

In the locally deployed MQTT brokers' test scenario, See Fig. 4, the goal was to evaluate overall message rate and broker payload in a given context (QoS 0/1/2, 1 topics, 1 client); the parameters are same as shown in Table 2, only the payload were raised to 31 Bytes. A JavaScript program was created to simulate the sensors in a house. The script simulated and published temperature and humidity sensor values of a room on a unique topic, 1000 times (1 message/second) and thus a total number of 1000 messages were taken into account to calculate performance statistics of a given broker server.

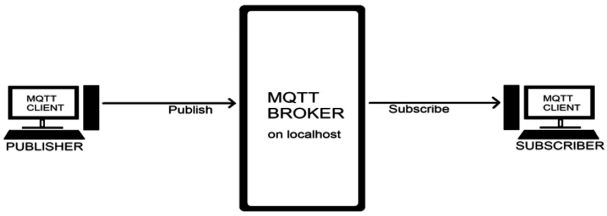

Fig. 4. Locally deployed broker servers' experiment scenario

Publishing JavaScript

```
function publish()
{
  var Thread = Java.type("java.lang.Thread");

  for (i = 0; i< 1000; i++)
  {
    mqttspy.publish("home/bedroom1",
        "{ " +
        "temp: " + (19 + Math.floor((Math.random() * 10) +
1) / 10) + ", " +
        "humidity: " + (59 + Math.floor((Math.random() *
10) + 1) / 10) + ", " +
        "}", 1, false);
    try
    {
      Thread.sleep(1000);
    }
    catch(err)
    {
      return false;
    }
  }
  return true;
}
publish();
```

Keep Alive Time. Keep Alive Time is usually measured in seconds. To call the connect function of an MQTT client, it has to be provided with four parameters as shown below:

```
connect(host, port=1883, keepalive=60, bind_address="")
```

The purpose of keep alive parameter is to ensure that the underlying connection between client and broker is still open to communicate. It is the client's responsibility to confirm that the interval between Control Packets being sent should not exceed the Keep-Alive value. If there is no data flow over an open Connection during specified keep alive period, then the client will send a PINGREQ packet and wait to receive a PINGRESP from the broker. This exchange of message affirms that the connection is open and working. The Keep-Alive is expressed as a 16-bit word and measured in seconds [11].

4 Evaluation Results

4.1 Public Brokers (Public Test Case Scenario Results)

In our public test case scenario, we found that at standard Transport Layer level over TCP/IP, there is a very small difference in performance between different publicly deployed MQTT Broker, but at QoS 0 and QoS 2 level the message load rate See Fig. 5 of Mosquitto Broker was significantly higher than other public brokers.

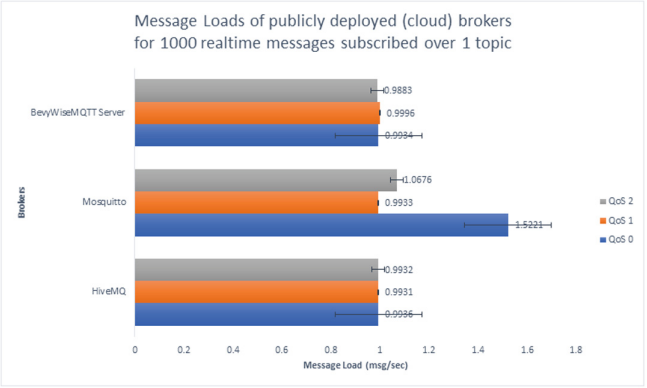

Fig. 5. Message load rate of public brokers

If we sort the brokers according to the server to client message delivery time, at QoS 0 Mosquitto consumed least time to deliver the messages to the client, then came HiveMQ and then MQTT Server. At QoS 1 level MQTT Server performed the best, Mosquitto came second, and HiveMQ reserved the third position. At QoS 2 level Mosquitto again topped the list, HiveMQ was on the second position, MQTT Server secured the third place. See Fig. 6.

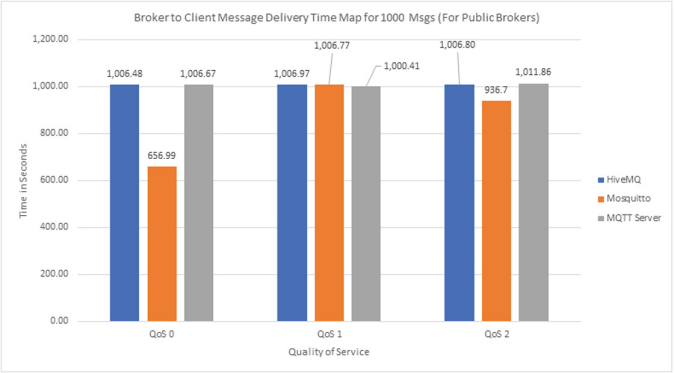

Fig. 6. Broker to client message delivery time map (Public Brokers)

4.2 Locally Deployed Brokers

Local Test Case Scenario Results. In our local test case scenario, we found very little difference in performance between the locally deployed brokers overstandard TCP/IP. We also found that RabbitMQ does not support QoS2 subscriptions. RabbitMQ automatically downgrades QoS 2 [11], publishes and subscribes to QoS 1See Fig. 7. If we sort the brokers in an ascending order according to the server to client message delivery time, at QoS 0, HiveMQ delivered the messages first and then RabbitMQ, Mosquitto, MQTT Route, and Active MQ followed accordingly. At QoS 1 HiveMQ topped the list and Mosquitto, RabbitMQ, MQTT Route and Active MQ occupied the second, third, fourth and fifth position respectively. At QoS 2 HiveMQ again secured the first position, then came Mosquitto, MQTT Route, and ActiveMQ one after another. See Figs. 8, 9 and 10.

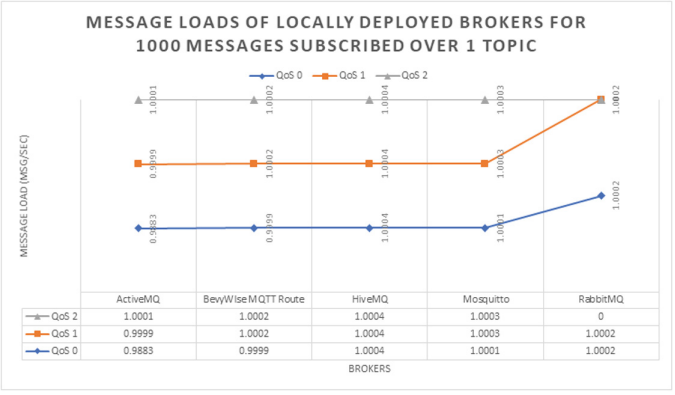

Fig. 7. Message load rate, locally deployed brokers.

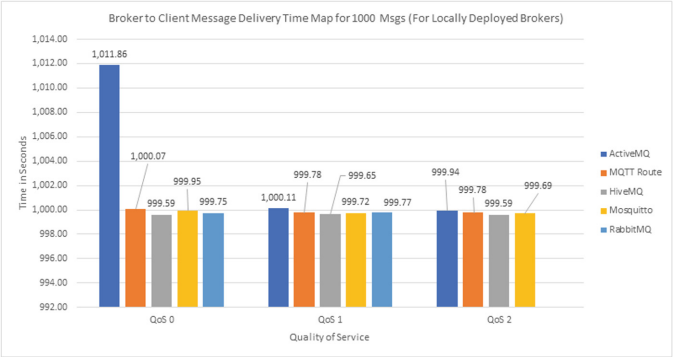

Fig. 8. Broker to client message delivery time map (Local Brokers)

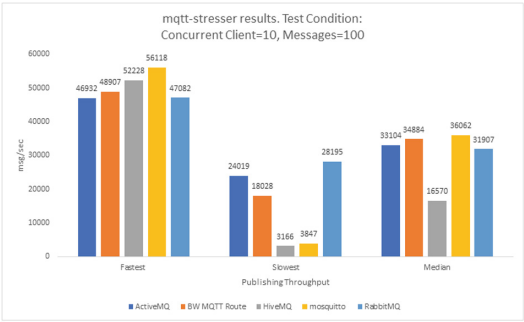

Fig. 9. Broker to client message delivery time map (Public Brokers)

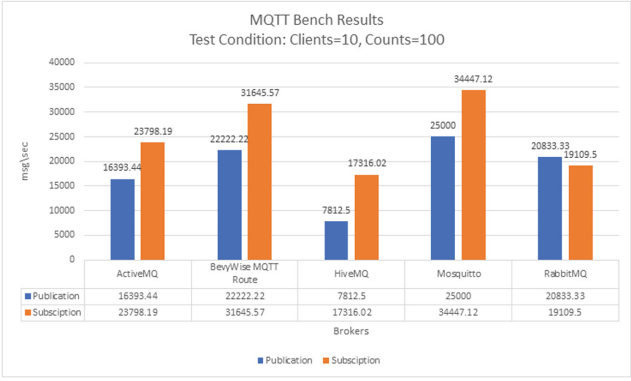

Fig. 10. Broker to client message delivery time map (Public Brokers)

Benchmark Results:

Benchmark tool: mqtt-stresser [17]
Concurrent Clients: 10
Number of Messages: 100
Benchmark tool: mqtt-bench [18]
Concurrent Clients: 10
Number of Messages: 100

5 Conclusion

The Internet of Things paradigm is a rapidly growing research field, which has a great potential to ease our lives and to create a better world. There are many M2 M data transfer protocols available for IoT systems, including MQTT, CoAP, AMQP, and HTTP. In this paper we investigated the performance of MQTT brokers and compared their properties by subscription throughput under a specific publishing condition. Our results showed very small difference in the performance of MQTT brokers in standard domestic deployment use case. On the other hand, we found stress testing results of MQTT brokers very inspiring, and we plan to continue this research line in the future with increased number of load conditions.

Acknowledgements. The author would like to show his gratitude to his colleagues: Attila Kertesz, Tamás Pflanzner and Tibor Gyimothy for their constructive comments and suggestions that made this research work possible.

References

1. Lampkin, V., et al.: Building Smarter Planet Solutions with MQTT and IBM WebSphere MQ, Telemetry. IBM Redbooks publication, 268 p. (2012)
2. Naik, N.:Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP. In: IEEE International Systems Engineering Symposium, Vienna, pp. 1–7 (2017)
3. Karagiannis, V., Chatzimisios, P., Vazquez-Gallego, F., Alonso- Zarate, J.: A survey on application layer protocols for the Internet of Things. Trans. IoT Cloud Comput. **3**(1), 11–17 (2015)
4. Bellavista, P., Zanni, A.: Towards better scalability for IoT-cloud interactions via combined exploitation of MQTT and CoAP. In: IEEE 2nd International Forum on Research and Technologies for Society and Industry Leveraging a better tomorrow (RTSI) (2016)
5. Gazis, V., Gortz, M., Huber, M., Leonardi, A., Mathioudakis, K., Wiesmaier, A., Zeiger, F., Vasilomanolakis, E.: A survey of technologies for the Internet of Things. In: Proceedings of 2015 IEEE International Wireless Communications and Mobile Computing Conference, pp. 1090–1095 (2015)
6. Foster, A.: Messaging technologies for the industrial internet and the Internet of Things whitepaper PrismTech. (2015)
7. Kovatsch, M., Lanter, M., Shelby, Z.: Californium: scalable cloud services for the Internet of Things with CoAP. In: IEEE International Conference on Internet of Things, pp. 1–6 (2014)

8. Stephen, N.: Power Profiling: HTTPS Long Polling vs. MQTT with SSL, on Android (2015). http://stephendnicholas.com/archives/1217. Accessed 04 Apr 2018
9. Lee, S., Kim, H., Hong, D.K., Ju, H.: Correlation analysis of MQTT loss and delay according to Qos level. In: The International Conference on Information Networking (ICOIN), pp. 714–717, January 2013
10. Pahomqtt c client library: Quality of service. http://www.eclipse.org/paho/files/mqttdoc/MQTTClient/html/qos.html. Accessed 19 June 2017
11. MQTT Version 3.1.1, 28 October 2018. http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html. Accessed 04 Apr 2018
12. Fehrenbach, P.: Messaging Queues in the IoT Under Pressure-Stress Testing the Mosquitto MQTT Broker, FakultätInformatik Hochschule Furtwangen University, 10 May 2017. https://blog.it-securityguard.com/wp-content/uploads/2017/10/IOT_Mosquitto_Pfehrenbach.pdf Accessed 04 Apr 2018
13. Nistsp 800-132, recommendation for password-based key derivation part1: Storage applications. http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-132.pdf. Accessed 04 Mar 2018
14. Public Brokers. https://github.com/mqtt/mqtt.github.io/wiki/public_brokers. Accessed 04 Apr 2018
15. Github- mqtt-spy. https://github.com/eclipse/paho.mqtt-spy. Accessed 15 Mar 2018
16. RabbitMQ Documentation. https://www.rabbitmq.com/documentation.html. Accessed 04 Apr 2018
17. Github- mqttstresser. https://github.com/inovex/mqtt-stresser. Accessed 15 Mar 2018
18. MQTT Bench mqtt-bench. https://github.com/takanorig/mqtt-bench. Accessed 15 Mar 2018