Experiences And Considerations On Performance Test Of Mosquitto-Based Broker

Kitae Hwang[*1)], Jae Moon Lee[1)], In Hwan Jung[1)], Hye Jin Park[2)] and Tae Yoon Lee5[2)]

[1)] School of Computer Engineering, Faculty of Engineering, Hansung University, Seoul, Korea
[2)] Department of Computer Science, Graduate Student, Yonsei University, Seoul, Korea
[*]Corresponding author. Tel.: +82-10-4711-7446; Email address: calafk@hansung.ac.kr

Article History:Received:11 november 2020; Accepted: 27 December 2020; Published online: 05 April 2021

Abstract : MQTT is a communication protocol for exchanging messages between clients via MQTT broker and is used to build many IoT systems. Currently, various tools have been developed to help measure the performance of application systems using MQTT. However, because they are all versatile tools, a dedicated test system that meets the purpose of the application system needs to be built on their behalf. But building a test system is not that simple. In this paper we discuss a case of building a test system for measuring U-Mosquitto's performance, which has modified the open source Mosquitto broker to handle urgent messages with priority, and some issues to be considered in the process. In particular, they include selection of the server computer, the structure of the test program, the use of appropriate threads for the client, the appropriate number of clients, the appropriate workload per client, clock synchronization between clients, protocol analyzer issues, and other issues. Our experiences and considerations on these issues are expected to be a good guide for building a test system to measure the performance of MQTT application systems.

Keywords: MQTT, Mosquitto, Performance Evaluation, MQTT Broker, Message Delivery.

1. Introduction

MQTT(Message Queuing Telemetry Transport) is a communication protocol in which clients send and receive text messages in a publish/subscribe manner using a broker which relays messages[1,2]. The MQTT is relatively simple, and it is widely used in IoT(Internet of Things) application system connecting small devices, sensors, and human mobile devices because of a small burden on communication[3,4].

As with any distributed application system, the MQTT application system is also not so simple to evaluate, because many devices are connected on the network. The clocks of devices or computers connected to the network are all different, and many client computers must be mobilized to place a high workload. And if only a few computers are used and high load is taken, it may be out of the essence of distributed computing and become a distorted experiment that does not match reality. Since it is difficult to test in real situations, simulation techniques can be used. However, the simulation does not reflect the reality and it is difficult to get accurate experiment results.

Meanwhile, tools for performance testing of MQTT application systems such as mqtt-bench, JMeter, mqtt-spy, MQTT-Stresser, and MQTT Toolbox have been developed recently. These are general-purpose tools for measuring the performance of MQTT application systems and can be used for performance evaluation of relatively simple and uncomplicated MQTT application systems. In the case of MQTT application systems, that connect more than a few thousand clients or are created for a specific purpose, it is difficult to measure the correct performance with this general purpose tool[5,6]. We built a U-Mosquitto broker and application system to support urgent messages by modifying the existing Mosquitto broker[7] developed by Eclipse, and then built a dedicated test system to measure the performance.

Constructing a test system is not simple. There are a number of things to consider in creating a sufficiently large workload. This paper discusses the difficulties and solutions at each steps throughout the process in the performance test of the U-Mosquitto application system. Several issues and solutions to the performance evaluation system described in this paper are expected to help reduce trial and error in the process of measuring and evaluating the performance of MQTT application systems as well as similar distributed application systems.

2. Analysis of U-Mosquitto broker

2.1 MQTT Protocol

MQTT is a message communication protocol designed by IBM for in a publish/subscribe manner. In MQTT, data sent and received is called a message. Clients are divided into the subscriber waiting for a message and the publisher sending a message. The message is delivered via the MQTT broker and it always includes a data called the topic. The MQTT broker uses the topic to determine which subscribers should receive the message. Therefore, all subscribers must insert the topic of the message in the MQTT packet when connecting to the MQTT broker. The publisher also sends a topic together whenever they send a message to the broker. The MQTT broker sends a message to all subscribers waiting for the topic.

Figure 1 shows an example of an MQTT application system consisting of publishers, subscribers, and an MQTT broker. Three subscribers are connected to the MQTT broker for message subscription, and two of them

[*]**Corresponding author:** Kitae Hwang
School of Computer Engineering, Faculty of Engineering, Hansung University, Seoul, Korea
Email address: calafk@hansung.ac.kr

are waiting for a message of the topic 'hello'. Currently three publishers are sending messages. The second publisher sends a message of the topic 'hello', and the MQTT broker transmits the message to both the subscriber 2 and the subscriber 3 waiting for the topic 'hello'.

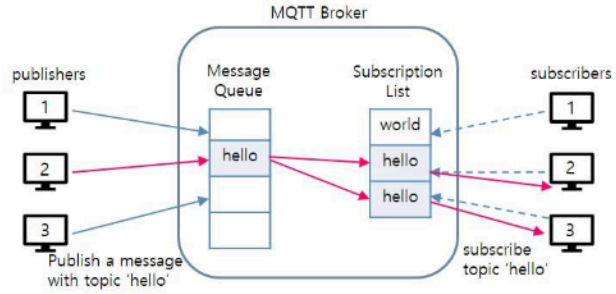

Figure 1 Topic-based message communication of MQTT

2.2 Basic process of U-Mosquitto message processing

U-Mosquitto is an MQTT broker that has been built to handle urgent messages, modifying the existing Mosquitto broker, on the basic structure of Mosquitto through our previous research. The U-Mosquitto has added an urgent message list to handle the urgent message as well as the existing normal message. The message type is divided into the normal message and the urgent message and is transmitted through the payload of the MQTT packet. U-Mosquitto uses the structure of Mosquitto which is implemented as a single thread. The main_loop in Figure 2 runs an infinite loop that is executed as a single thread. Each loop in main_loop does the same task, and what happens in each loop is as follows. First, poll () system call executes to find out which publisher sent messages(step ①). Then, for all the publishers that sent the data, only the first message is read for each publisher and distributed to the urgent message list or normal message list according to the message type (step ②). Then, messages of the urgent message list are first transmitted to the subscriber, and then messages of the normal message list are transmitted (step ③).

In the example in Figure 2, there are currently three publishers connected to U-Mosquitto, two of which sent messages and the three incoming messages appear gray. In step ②, the poll() system call returns 2. This means that data has arrived at the two publisher sockets in the broker, currently. The value 2 returned by the poll() function is called poll size. The poll size is smaller than or equal to the number of connected publishers. In Figure 2, even though two messages arrive at the first socket, only the first message is processed in the first loop and the second message is processed in the next loop.

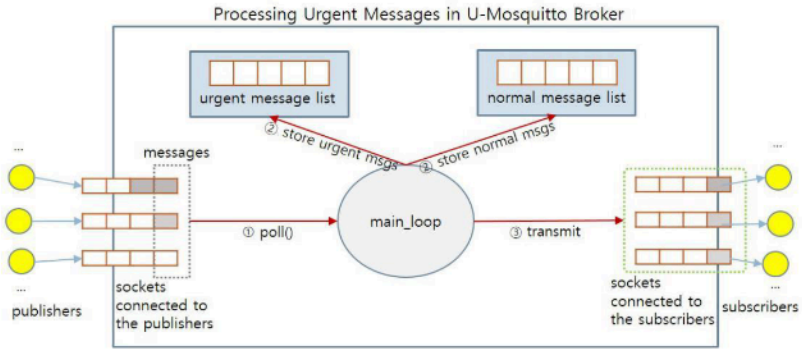

Figure 2 Message processing of U-Mosquitto

3. Performance Evaluation Issues

3.1 Basic metrics of performance evaluation

When an application system is created using an MQTT broker, the basic metrics to be measured are generally the measure of how much the MQTT broker or application system can withstand the load, how long the transmission delay from the publisher to the subscriber is, etc. In the previous research, we also constructed a test system as shown in Figure 3 and conducted various experiments to measure the performance of the application system in the process of building the U-Mosquitto and the application system. In this paper, we set the following three basic metrics in order to focus on the issues related to the construction and performance measurement of the test system.

- Message Transmission Time (T): The time it took for the message sent by the publisher to arrive at the subscriber
- CPU utilization (U): CPU utilization of U-Mosquitto Broker

- poll size (PS): Number of messages processed by U-Mosquitto at one loop

3.2 Building Test System

3.2.1 Configuring the test system

The performance of an MQTT application system depends on the MQTT broker and so in many cases the target of the evaluation is the MQTT broker. We constructed a test system as shown in Figure 3.

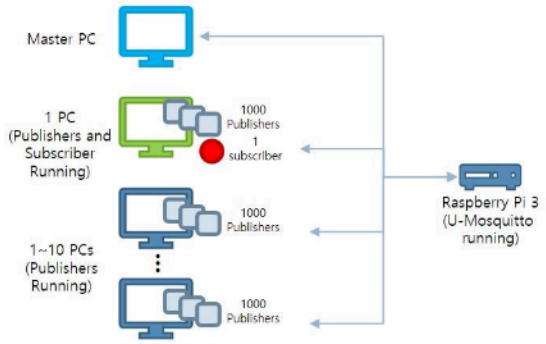

Figure 3 Test System Configuration

The test system constructed in this paper consists of 11 client PCs, 1 server computer, and 1 master PC. The U-Mosquitto broker runs on the server computer, and the publisher and the subscriber on the client PC. The Master PC initially sends the experiment parameters on the client PCs through the payload of the MQTT message as shown in Table 1 via U-Mosquitto, and makes the subscriber and the publishers start. Figure 4 shows how the command messages are sent to clients and the experiment starts, in the case of 1 subscriber and 1000 publishers.

Table1 MQTT message payload with topics 'startsub' and 'startpub'

Field	Explanation
# of PCs	Number of PCs participating in experiments
# of Publishers	Number of publishers created in a PC
# of Publishers per 100ms	Number of publishers scheduled every 100ms
Urgent ratio	Ratio of urgent messages for total messages
Duration	Total experiment time

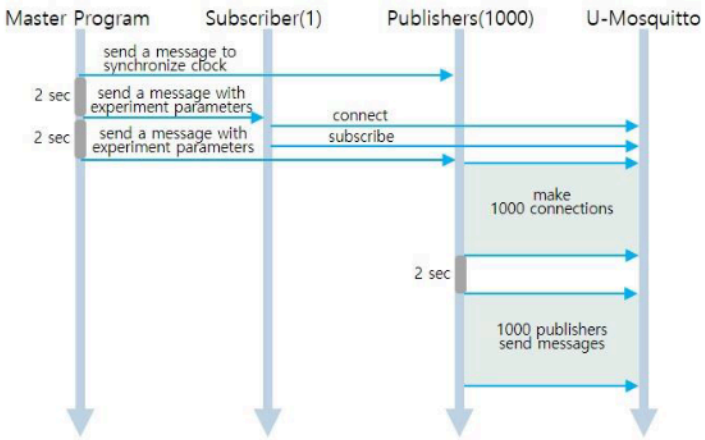

Figure 4 Process of experiment being started by master program

3.2.2 Determining server computer considering workload

If the performance of the server computer is high, a significant amount of load needs to be put on the MQTT broker to raise CPU utilization up to 100%. To do this, we increase the number of the publisher and the

subscriber, or the rate of messages sent by the publisher. To increase the number of publishers, we can increase the number of publisher running on one client PC or client PCs. But it cannot be increased infinitely. Most of the operating systems have a maximum of 65535 TCP socket connections. Another way to increase the load is to increase the publisher's message transmission rate. However, it is not realistic to send more than 10 messages per second. It is because in reality, publishers are sensors, handheld devices, or human-controlled smartphones, which do not send dozens of messages per second.

To solve this problem, we intentionally used a Raspberry Pi 3 which is a single board computer[8] with low performance as a server computer. The Raspberry Pi 3 has a Quad Core 1.2GHz Broadcom BCM2837 64bit CPU and 1GB of RAM.

3.3 Configuring the Test Program

Each MQTT client (publisher/subscriber) connects independently a TCP socket with the MQTT broker and sends or receives messages. If 1000 MQTT clients are running at the same time, the MQTT broker maintains 1000 socket connections with them. Let's take a look at the issues to consider when writing a test program to run on a client PC

3.3.1 Determining server computer considering workload

First, we have to decide whether to implement a separate thread for each MQTT client or control multiple MQTT clients with one thread. To begin with, it is not appropriate to implement one thread per MQTT client. If we run 1000 MQTT clients on one client PC, 1000 threads run on this PC, which results in scheduling overhead, which will adversely affect the experiment. In this paper, we introduce test chunk which is defined as MQTT clients controlled by a thread as a unit of experiment. Test chunk can be seen in Figure 5 and Figure 6. A thread periodically sends out messages for all publishers belonging to a single test chunk. Specifically, the publishers included in the test chunk are divided into several groups again, which are called segments in this paper. The thread divides the entire period by the number of segments to determine the period of the segment. Then, all the messages of the publishers belonging to the segment are transmitted through the segment cycle.

3.3.2 Test program implementation

The test program was written in the Java language with the flow and structure shown in Figure 5. The messageArrived () method written in the test program is a callback function that receives and processes messages with topics of 'time', 'startsub', and 'startpub' from the Master PC. This paper addresses only the process of processing the message of 'startpub' topic, which is the most important work of this function. Figure 5 shows a case where '# of PCs' is 1, '# of Publishers' is 5000, '# of Publishers per 100 ms' is 100, 'Urgent Ratio' is 10% and 'Duration' is 20 seconds, and they are published from the Master PC.

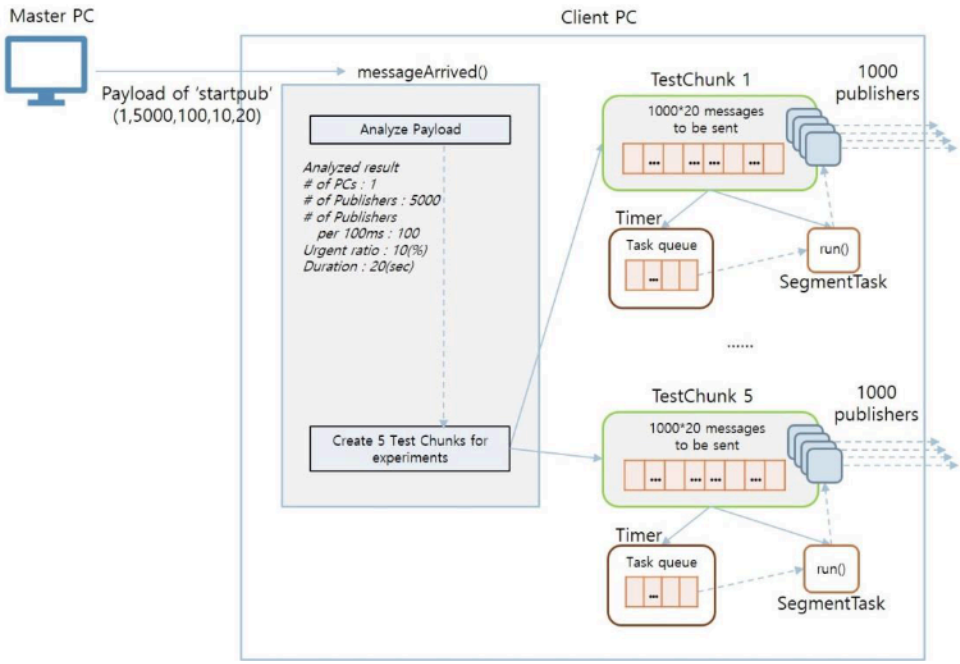

Figure 5 Process of experiment being started by master program

3.3.3 Implementing Message Delivery

To allow publishers to send messages at regular intervals, we used Java's TimerTask and Timer classes. We created a SegmentTask by inheriting the TimerTask class and overrode the run () method to send messages for the publishers belonging to the segment.

```
class SegmentTask extends TimerTask {
        public void run () {// send messages for publishers belonging to a segment
        ...
        }
}
```

The Timer class is a kind of timer thread that has a task queue internally and executes tasks inside the task queue according to a given cycle. Thus instances of the SegmentTask are put into the Timer's task queue for being executed.

Now, let's determine the number of messages sent by one SegmentTask instance. Creating one task per publisher is very inefficient. One SegmentTask deals with only one segment and implements to send messages for all publisher belonging to the segment. For example, instead of creating the SegmentTask class to send one message per 1 millisecond per publisher, the run() method can be written to be called every 100ms, causing the run() to send a message per each publisher for 100 publishers. This reduces the number of times SegmentTask's run () is called. Being called too often incurs unnecessary overhead. The following code creates an instance of Timer and one instance of SegmentTask, and calls a member method of ScheduleAtFixedRate (new SegmentTask (), 0, 100) of the Timer. As the result this method is executed, the run () method of the SegmentTask will be called every 100ms and send messages for publishers belonging to the segment. If there are multiple segments in a TestChunk, run () should be called multiple times.

The test program implemented in this paper can configure how many publishers a test chunk should contain. Let's see Figure 6 for the sake of understanding. Since 1000 publishers are made as one test chunk and 10 segments are placed in the test chunk, 100 publishers per segment are placed. To have all 1000 publishers belonging to a test chunk send one message in a second, each segment need to send 100 publisher messages per 100 milliseconds.

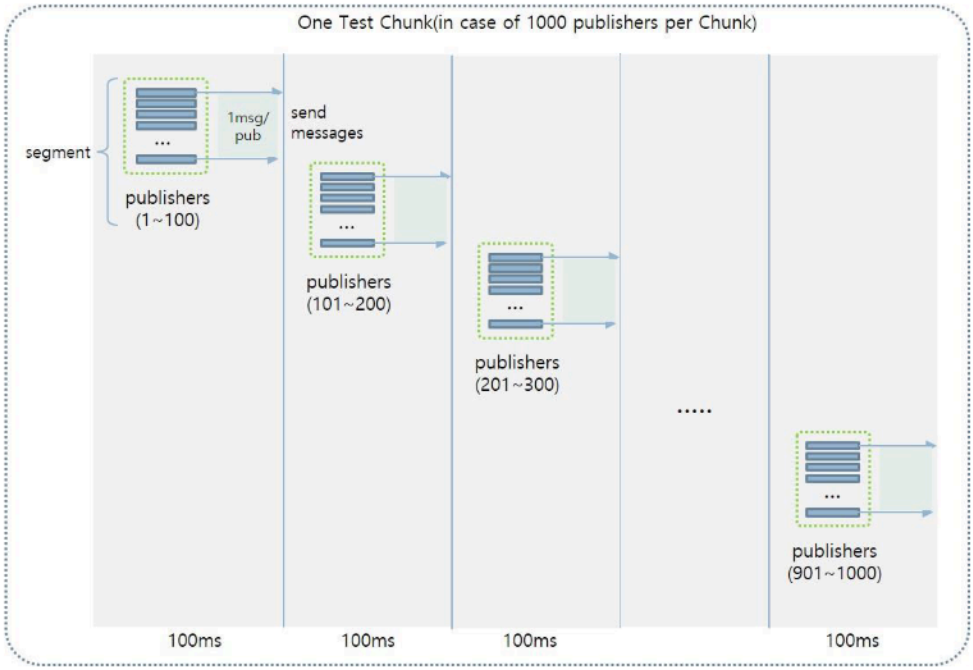

Figure 6 Message transmission process for a test chunk

3.4 MQTT Client count and load

3.4.1 Percentage of messages sent per publisher

To generate the same message traffic, we may reduce the number of publishers on a single test chunk and increase the percentage of messages sent by each publisher. However, this method causes a lot of problems.

First, if a single test chunk consists of 10 publishers and each publisher is putting a workload on sending one message per millisecond, then the client PC will have 10 socket connections. At this time, people can easily predict that client PC's operating system will send outgoing messages on 10 sockets, one message per socket in a round-robin manner, but it is not. If multiple MQTT packets are pending on one socket, the operating system

may send those packets all at once to the MQTT broker. In this case, the messages do not arrive at the MQTT broker in time order sent by 10 publishers. In some cases, when the MQTT message is transmitted, the time may be recorded in the payload of the MQTT packet by the application. For application systems that expects messages to arrive at the broker in the order in which they were sent, an unexpected error occurs. So the way the publisher sends hundreds of messages per second is bad and not realistic.

Second, as mentioned in Section 3.2.2, the publisher is actually sensor, handheld device, or human-controlled smartphone which does not send dozens of messages per second. Therefore, it is difficult to see as an experiment that reflects reality to significantly increase the message transmission rate per publisher to increase message traffic.

3.4.2 Number of publishers per client PC

It is a matter of determining how many publishers a single client PC will operate. How many test chunks should be allocated to a client PC? Suppose the publisher sends one message per second. Even if 1000 publishers is operated on a normal PC, the performance of the PC does not reach saturation.

However, when one test chunk was configured with 1000 publishers and seven test chunks were run on a client PC, the client PC's output socket began to show overload. Figure 7 shows the time it takes for all messages to leave the client PC when each publisher sends one message per second for 20 seconds on one client PC. Up to 6000 publishers, all messages left the client PC within 20 seconds, but 7000 publishers took more than 60 seconds. In this situation, U-Mosquitto's network input was bottleneck, resulting in a drop in U-Mosquitto's CPU utilization and unable to obtain normal evaluation data.

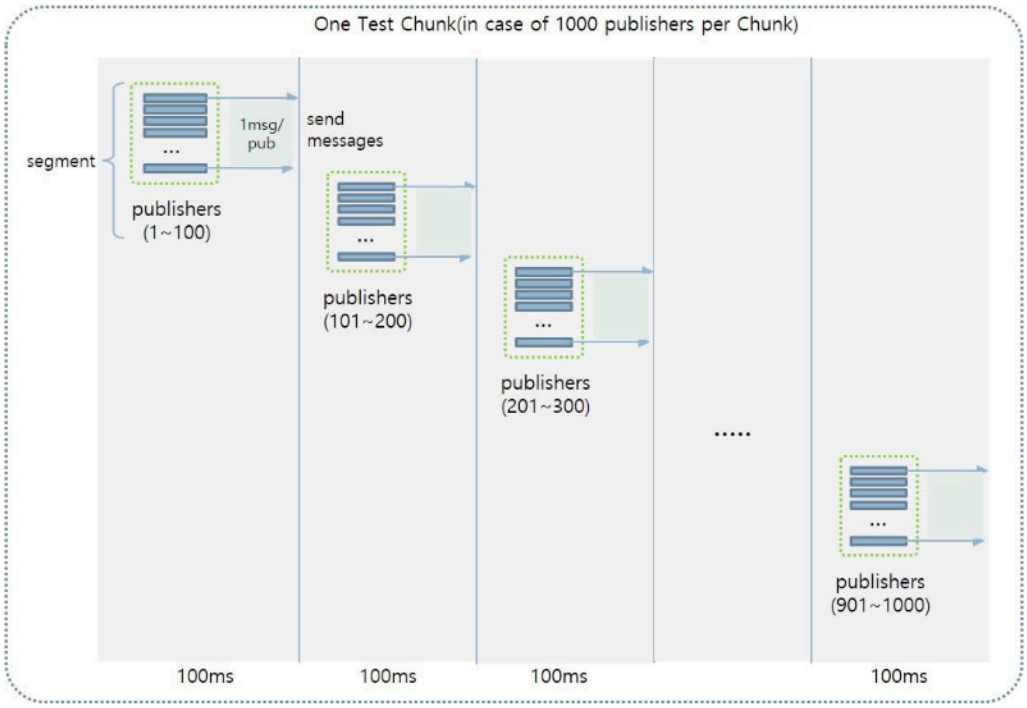

Figure 7 The time for all message to be sent according to the number of publishers

It became clear that a single client PC could not generate enough load. Our conclusion was that multiple client PCs should be used. In our study, experiments were conducted with various settings, such as utilizing a total of 11 client PCs and running 1 test chunk per client PC, or utilizing 6 client PCs and running 2 test chunks per client PC.

3.5 Clock synchronization of client PCs

In MQTT application systems as well as other distributed computing environments, clock synchronization between multiple computers connected to the network is sometimes very important. Even if the publisher puts the time the message is created in the payload of the MQTT packet, sends the message, and the subscriber receives the message and calculates the time it passes from the publisher to itself, it does not give the correct result. This is because that the clocks of the publisher and the subscriber are not the same.

In a distributed system, the Network Time Protocol (NTP) is used to synchronize the clocks of the computers connected to the network, but NTP is not a good solution for synchronizing clocks in milliseconds[9,10]. There

are many free NTP servers available on the Internet, but the reliability of the clock synchronization is low and installation is not always easy. In addition, due to the inconsistent network latency of the Internet, time errors occur between the client computers and the time error varies depending on the situation, so that it is difficult to correct. In other cases, access to an external NTP server is blocked by an internal firewall.

In this study, we had all client PCs set their clocks from Windows Timer Server, time.windows.com[11,12]. However, depending on the situation, the clock sometimes had an error of about 800 milliseconds. It was concluded that NTP is not suitable for sophisticated experimentation unless the clock is adjusted in seconds. It is also a good idea to buy and use time synchronization products like Time Tools, even if you pay for it. In this study, the publisher and the subscriber were run on the same computer to make it easier to set the clock,

3.6 Protocol analyzer issues

To check the payload of the MQTT packets. we installed and observed a protocol analyzer on the client PC and the server computer. There are various kinds of protocol analyzers such as WireShark[13] and Microsoft Message Analyzer[12]. Most of protocol analyzers are well aware of the MQTT protocol, so using them is very useful for experimentation. But we need to look at two issues.

First, it is a UI issue in the protocol analyzer. When the message traffic is low, one MQTT packet is generated as one TCP packet. However, when the message traffic gets high, multiple MQTT packets may be contained in one TCP packet. Due to UI limitations, most protocol analyzers do not show all MQTT packets within one TCP packet. Even the last MQTT packet of the TCP packet is truncated and not visible

Second, it is a matter of storing the log of the capture packet as a file. Because only what is shown on the screen is stored in the file, if the message traffic is high the MQTT packets are truncated and stored in the file.

Third, it is timestamp issue in the protocol analyzer. Depending on the protocol analyzer, the time the MQTT packet was captured is displayed on the screen, but it is not accurate. For the Microsoft Message Analyzer, we could see that the capture time was slower than the actual time. Our team created their own protocol analyzer and used it for experiments.

3.7 Open file descriptor count

The MQTT broker maintains one socket connection per MQTT client. The socket is treated as a single file by the operating system and the number of open files that can be connected at the same time is limited. We ran U-Mosquito broker on Linux on Raspberry Pi 3 and received a message saying that we could no longer open files during an experiment where 2000 publishers were connecting. For most operating systems, including Linux, the maximum open file descriptor is 65535. But for Linux, the maximum open file descriptor is initially set to 1024. The 'ulimit' command can increase this value up to 65535. However, for Mosquitto, the maximum number of clients to allow is entered in the mosquitto.conf file. So you must enter the maximum number of sockets to connect to this file.

4. Conclusion

A number of tools have been developed to help measure performance of application systems that utilize MQTT. Since these are general purpose tools, it is necessary to build a dedicated test system in accordance with the MQTT application system for that purpose. This was also the case with the MQTT application system developed with U-Mosquitto brokers to accommodate urgent messages. In the case of distributed systems including MQTT application systems, it is not so simple to construct a test system, such as requiring many clients for testing and hanging various workloads in order to get closer to the actual situation.

This paper discussed several issues related to the construction of a test system to measure the performance of MQTT application systems. Specifically, they include selection of the server computer, the structure of the test program, the appropriate thread usage on the client, the number of appropriate client PCs, the appropriate workload per client, the clock synchronization between client computers, protocol analyzer issues, and others. The consideration in this paper on these issues is expected to be good guide for building test systems for MQTT application systems.

5. Acknowledgements

This research was financially supported by Hansung University.

6. References

1. MQTT [Internet]. c2017 [cited 2019 December 1]. https://en.wikipedia.org/wiki/MQTT
2. Al-Fuqaha A, Guizani M, Mohammadi M, Aledhari M, Ayyash M. Internet of Things: A Survey on Enabling Technologies, Protocols and Applications. IEEE Communications Surveys &Tutorials. 2015 Fourthquarter;17(4):2347-76. DOI:10.1109/COMST.2015.2444095.

3. Jung IH, Lee JM, Hwang, K. An MQTT based real rime LBS system for vehicles and pedestrians. IJET. 2018 Dec.; 7(3.24):125-130. DOI: 10.14419/ijet.v7i3.24.22521
4. Sharma V, Tiwari R, A review paper on IOT & Its smart applications, IJSETR. 2016; 5(2);472-476
5. Scalagent [Internet]. c2015. Benchmark of MQTT servers; 2015 Jan. 1 [cited 2019 Dec. 1]. Available from: http://www.scalagent.com/IMG/pdf/Benchmark MQTT servers-v1-1.pdf
6. Bartnitsky J. HTTP vs MQTT performance tests [Internet]. 2017 Jan. 23 [cited 2019 Dec 5]. Available from: https://flespi.com/blog/http-vs-mqtt-performance-tests
7. Eclipse Mosquitto [Internet]. c2018. [cited 2019 March 1]. Available from: https://mosquitto.org
8. Raspberry Pi [Internet]. Raspberry Pi 3 Model B; c2018. [cited 2020 Feb. 27]. Available from: https://www.raspberrypi.org/products/raspberry-pi-3-model-b
9. Network Time Protocol [Internet]. c2018 [cited 2020 March 2]. Available from: https://en.wikipedia.org/wiki/Network_Time_Protocol
10. Time Tools [Internet]. c2018. The fundamentals of time synchronization; 2018 November 8 [cited 2020 Feb. 6]. Available from: https://timetoolsltd.com/time-sync/the-fundamentals-of-time-synchronization/
11. Time Tools [Internet]. c2018. How to synchronize Microsoft Windows to a NTP server; 2018 May 2 [cited 2020 Feb. 6]. Available from: https://timetoolsltd.com/time-sync/how-to-synchronize-microsoft-windows-to-a-ntp-server/
12. Fortunato T. Using Microsoft Message Analyzer for network troubleshooting [Internet]. c2017 [cited 2020 May 7]. Available from: https://www.networkcomputing.com/networking/using-microsoft-message-analyzer-network-troubleshooting
13. wireshark.org [Internet]. [cited 2020 May 4]. Available from: http://wireshark.org