INTERNET OF THINGS –

MQTT VS COAP

50.012 Networks Project

Group 8

Chua Yong Teck – 1003378…

Ong Sze Teng – …

Poh Shi Hui – 1002921

Zwe Wint Naing (Valentine) – 1002791

Table of Content

IoT Protocol – MQTT vs CoAP

The objective of our project is to find out which protocol, MQTT or CoAP, is better for Internet of Things (IoT) protocols. First and foremost, we will write an introductory passage on both of these protocols.

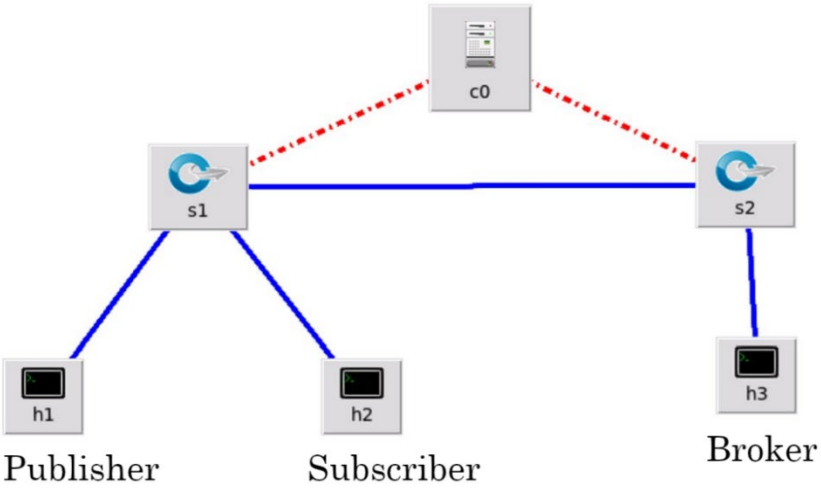
MQTT

MQTT, known as the MQ Telemetry Transport, or the Message Queuing Telemetry Transport is an OASIA standard messaging protocol for IoT. It works on a publisher-subscriber network protocol that transports messages between devices. This protocol, in particular, usually runs over TCP/IP. Its advantages are that it be scaled to connect with millions of devices, as well as having small message headers to optimize for scenarios with limited network bandwidth (fixed header of 2 bytes). In addition, MQTT has support for persistent sessions, reducing the duration of the connections over unreliable networks.

Unlike CoAP, MQTT has defined Quality of Services (QoS) levels from 0 and 2, along with their respective protocols:

- 0 – Message is delivered at most once.
    - The minimal QoS level is zero. This service level guarantees a best-effort delivery. There is no guarantee of delivery. The recipient does not acknowledge receipt of the message and the message is not stored and re-transmitted by the sender. QoS level 0 is often called “fire and forget” and provides the same guarantee as the underlying TCP protocol.
- 1 – Message is delivered at least once.
    - QoS level 1 guarantees that a message is delivered at least one time to the receiver. The sender stores the message until it gets a PUBACK packet from the receiver that acknowledges receipt of the message. It is possible for a message to be sent or delivered multiple times.
    - The sender uses the packet identifier in each packet to match the PUBLISH packet to the corresponding PUBACK packet. If the sender does not receive a PUBACK packet in a reasonable amount of time, the sender resends the PUBLISH packet. When a receiver gets a message with QoS 1, it can process it immediately. For example, if the receiver is a broker, the broker sends the message to all subscribing clients and then replies with a PUBACK packet.

    - If the publishing client sends the message again it sets a duplicate (DUP) flag. In QoS 1, this DUP flag is only used for internal purposes and is not processed by broker or client. The receiver of the message sends a PUBACK, regardless of the DUP flag.
- 2 – Message is delivered exactly once.
    - QoS 2 is the highest level of service in MQTT. This level guarantees that each message is received only once by the intended recipients. QoS 2 is the safest and slowest quality of service level. The guarantee is provided by at least two request/response flows (a four-part handshake) between the sender and the receiver. The sender and receiver use the packet identifier of the original PUBLISH message to coordinate delivery of the message.
    - When a receiver gets a QoS 2 PUBLISH packet from a sender, it processes the publish message accordingly and replies to the sender with a PUBREC packet that acknowledges the PUBLISH packet. If the sender does not get a PUBREC packet from the receiver, it sends the PUBLISH packet again with a duplicate (DUP) flag until it receives an acknowledgement.
    - Once the sender receives a PUBREC packet from the receiver, the sender can safely discard the initial PUBLISH packet. The sender stores the PUBREC packet from the receiver and responds with a PUBREL packet.
    - After the receiver gets the PUBREL packet, it can discard all stored states and answer with a PUBCOMP packet (the same is true when the sender receives the PUBCOMP). Until the receiver completes processing and sends the PUBCOMP packet back to the sender, the receiver stores a reference to the packet identifier of the original PUBLISH packet. This step is important to avoid processing the message a second time. After the sender receives the PUBCOMP packet, the packet identifier of the published message becomes available for reuse.
    - When the QoS 2 flow is complete, both parties are sure that the message is delivered and the sender has confirmation of the delivery.
    - If a packet gets lost along the way, the sender is responsible to retransmit the message within a reasonable amount of time. This is equally true if the sender is an MQTT client or an MQTT Broker. The recipient has the responsibility to respond to each command message accordingly.

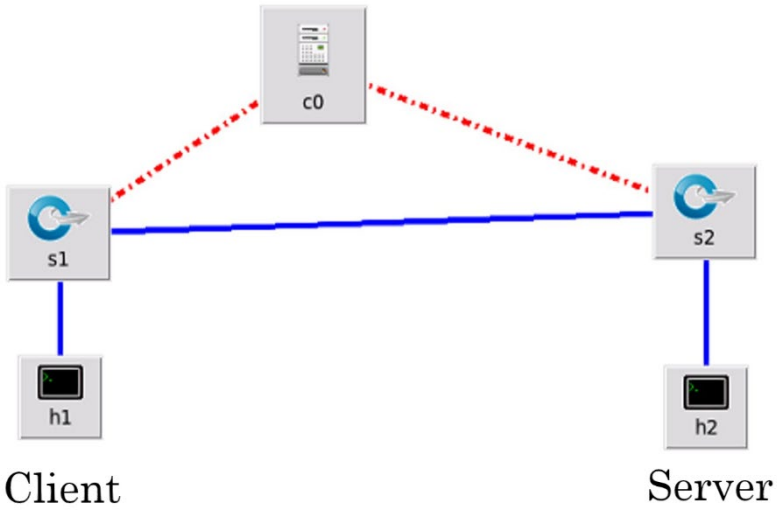


The simplest topology of MQTT is as follows on the figure above. We have a publisher that true to its name, publishes the content or the packets needed. Subscriber receives the contents, and the Broker, special to MQTT. The broker is responsible for receiving all messages, filtering the messages, determining who is subscribed to each message, and sending the message to these subscribed clients. The broker also holds the session data of all clients that have persistent sessions, including subscriptions and missed messages (more details [1]). Another responsibility of the broker is the authentication and authorization of clients. Usually, the broker is extensible, which facilitates custom authentication, authorization, and integration into backend systems. In brief, the broker is the central hub through which every message must pass. Therefore, it is important that your broker is highly scalable, easily integrable into backend systems, easy to monitor, and (of course) failure-resistant.

CoAP

On the other hand, we have CoAP, which stands for Constrained Application Protocol – a specialised web transfer protocol for use with constrained nodes and networks in the IoT. Constrained, by definition, here means the limits on power, memory, and computing resources. As a result of this, this protocol meets specialised requirements multicast support, very low overhead, and simplicity. In addition, CoAP is designed for easier translation to HTTP for simplified integration, and it is identified using hierarchical Uniform Resource Identifier (URI) schema.

The simplest topology of CoAP is as follows on the figure below, a matter of one client with one server.



Our Approach and Setup

Parameters to measure

To compare the efficiency – in terms of Round Trip Time (RTT), throughput, reliability – of the protocols, we used network simulators in order to simulate the environment that they operate in. Round Trip Time, or RTT as it was aforementioned, is defined as the amount of time it takes to get from the Host to Server, and then back to the Host. The RTT between a server and a client can typically be measured using the ping command. We will also measure other attributes such as reliability – defined by the number of packets lost (or received), over the total number of packets that were sent.

MiniNet vs Other Alternatives

Overall, our approach is to use MiniNet in order to simulate the respective systems. This is as we are familiar with the MiniNet system, as well as having been given access to such a prepared virtual machine from our Lab session. Mininet proves extremely useful as a network emulator and enabled flexibility in adjusting packet loss rate, delay and bandwidth. Additionally, other network simulators, such as NetSim [2], has three models – NetSim Professional, NetSim Standard, and NetSim Academic. As our school does not seem to have a license with them, as well as the uncertainty on the price of these licenses needed for just one project, we eventually fell back to MiniNet for our experimental setup.

Mininet allowed us to create various hosts, switches and controllers and link them up, creating a topology suitable for us to do testing. For CoAP, we have a controller connected to 2 switches that are connected to one another, each having one host. Whereas for MQTT, we have a similar setup but due to the requirements of

Tools and Experiment Setup

As a result, on top of using MiniNet, we have also made use of other systems as we will list below:

- Eclipse Mosquitto, or just Mosquitto, is "an open source (EPL/EDL licensed) message broker that implements the MQTT protocol… Mosquitto is lightweight and suitable for use on all devices from low power single board computers to full servers." [3]
    - Mosquitto to set up the MQTT system, i.e. the publisher, subscriber, and the broker, as well as establish connections that uses the MQTT protocol.
- Wireshark [4] is used in order to capture the flow of data, as well as to analyse other attributes such as RTT, throughput, and reliability as aforementioned above. It also offers "Live capture and offline analysis," which is suitable for our purpose as analysing the packet traffic between the two protocols.

Steps Taken

Some of the methods and variables that we have utilized are as follows:

- Variable QoS analysis (MQTT) – QoS levels that determines whether or not if the message is delivered at most once, at least once, or only exactly once.
- Variable message size for both protocols. 1 MB, 5 MB, and 10 MB.
- Variable number of messages for both protocols.
- Using only SSH instead of RDP to avoid lag if needed (That is, using Wireshark's CLI to start packet analysis and save the file, followed by transferring to our PC for analysis.)

Code Base

For MQTT, no editing of the source code was required, hence mostly command line tools were used, with the exception of the creation of the custom MiniNet topology required for the proper setup. The workflow is as shown below:

1. create 3 files of sizes 1MB, 5MB, 10MB

```
$ dd if=/dev/zero of=output.dat  bs=1M  count=10
```

2. set up mininet instance with 3 hosts and 2 switches using custom .py file created with miniedit.py

```
$ sudo mqtt1.py
```

3. set up the Mosquitto broker on h3 and subscriber on h2 that is listening on h3

```
$ h3 ./mosquitto/src/mosquitto &
$ h2 ./mosquitto/client/mosquitto_sub -h 10.0.0.3 -t test/q0 &
```

4. set up tshark on each host to monitor their eth0 port and save the .pcapng file with a naming convention MQTT<Host>_<file size>_<QOS level>

```
$ h1 tshark -i h1-eth0 -w /tmp/MQTT1_1_0.pcapng &
$ h2 tshark -i h2-eth0 -w /tmp/MQTT2_1_0.pcapng &
$ h3 tshark -i h3-eth0 -w /tmp/MQTT3_1_0.pcapng &
```

5. set up publisher and send file

```
$ h1 ./mosquitto/client/mosquitto_pub -h 10.0.0.3 -f hh.txt -t test/q0 -d
```

6. kill tshark

```
$ h1 killall tshark
```

7. repeat for every file, for every QOS level, and every file size of 1MB to 10MB

```
$ h2 ./mosquitto/client/mosquitto_sub -h 10.0.0.3 -t test/q1 -q 1 &
$ h1 ./mosquitto/client/mosquitto_pub -h 10.0.0.3 -f hh10.txt -t test/q1 -q
1 -d
```

Between each subscriber and publisher command, we need to set up tshark as above with the naming convention to ensure capturing of packets.

For CoAP, libcoap from https://github.com/obgm/libcoap was used to run the server and client on the host.

Topology was created using miniedit.py.

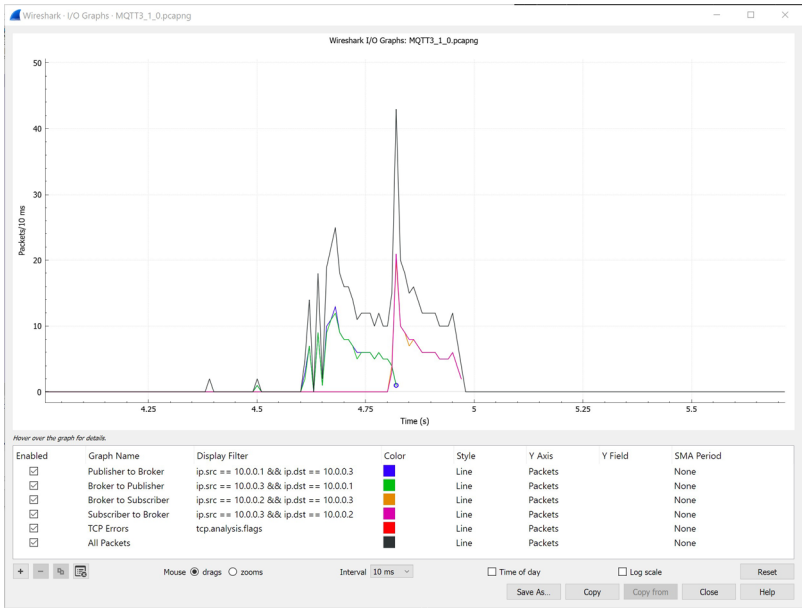
Results and Analysis

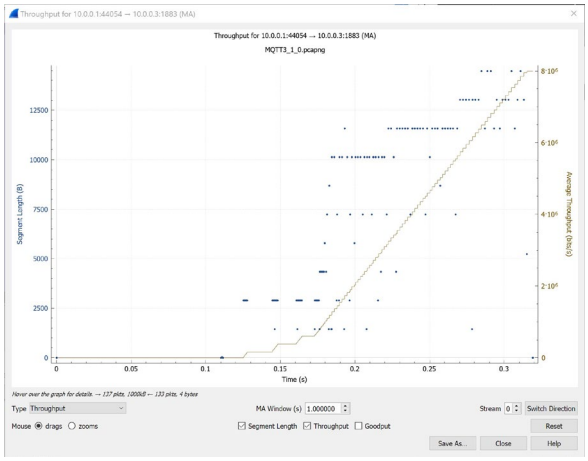
The graphs are configured such that the y-axis would show the packet flow (sent or received) every interval, while the x-axis shows the time in date and time format instead of the usual time elapsed format. We feel that this would help convey how much time has passed, and how fast the packets are received. Of note is that Wireshark has a quirk in the application whereby the Legend (detailing which line or bar graph belongs to which) disappears when the y-axis has the same unit of measurement, i.e. packets / time. On the other hand, the Legend will appear when the y-axis has different units, which then makes the y-axis disappear. This was a known bug that does not seem to have a fix as of now .

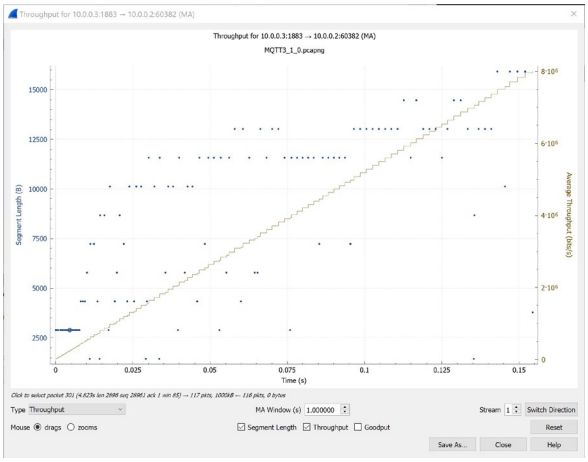
MQTT

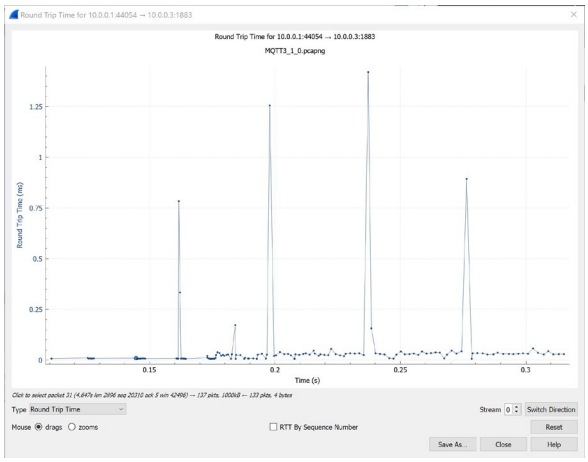
As for MQTT, there were more experiments being done since there are much more different parameters. That is because for one, there are 3 QoS (Quality of Service) settings that can be configured. Additionally, there are three Hosts to deal with and as a result, three graphs for each of the experiment. There is, however, no PUT or GET method. The filename conventions are MQTT[x]_[Message size]_[QoS setting].pcapng, whereby x = 1, 2, or 3. MQTT1 is Publisher, MQTT2 is Subscriber, MQTT3 is Broker.

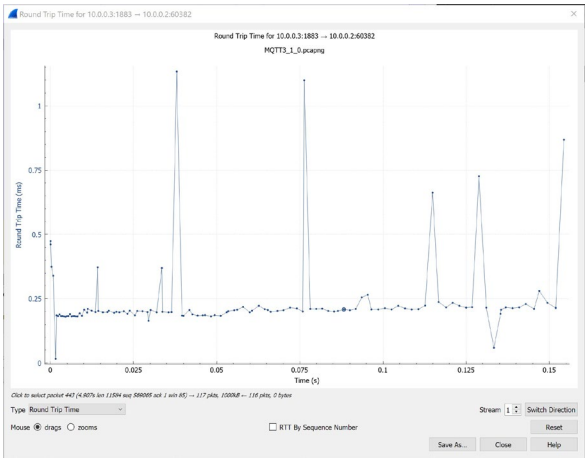
MQTT – 1 MB message size – QoS 0 (Message Delivered <= 1 time)



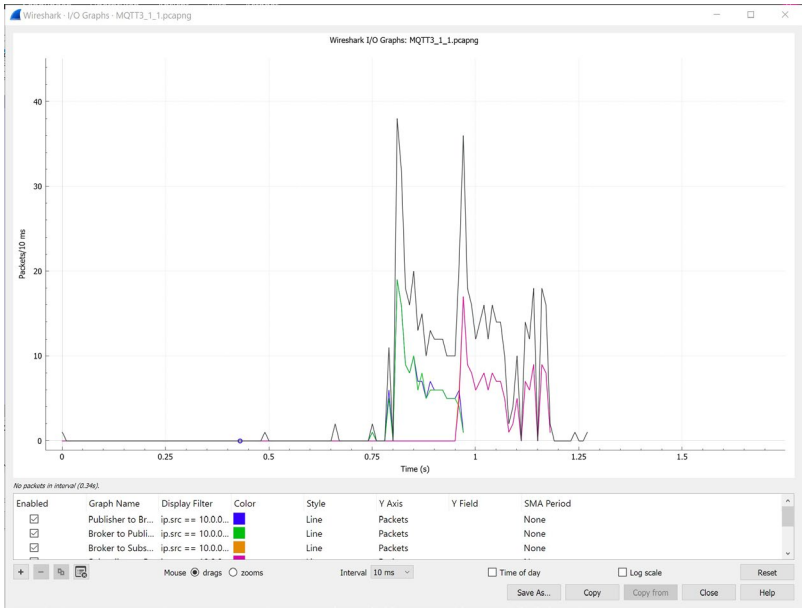


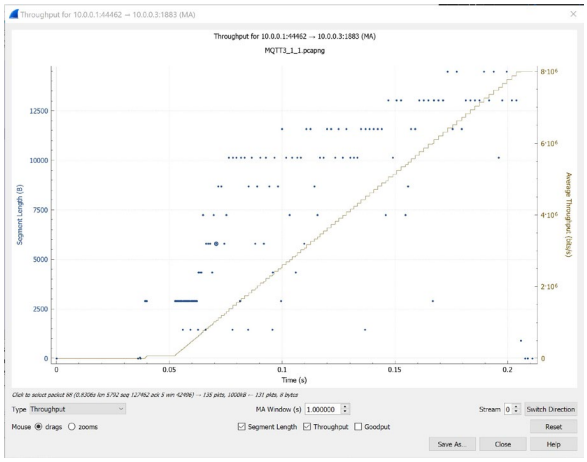


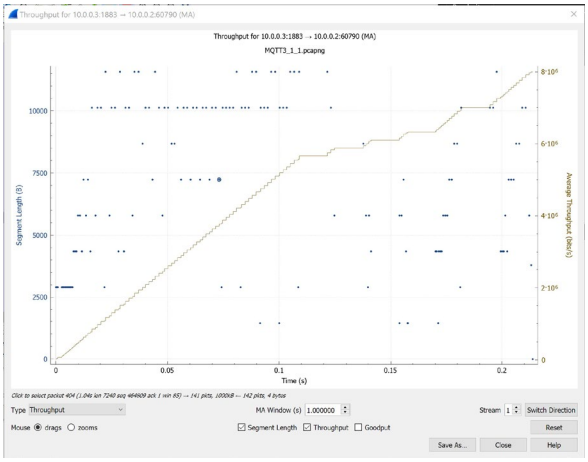


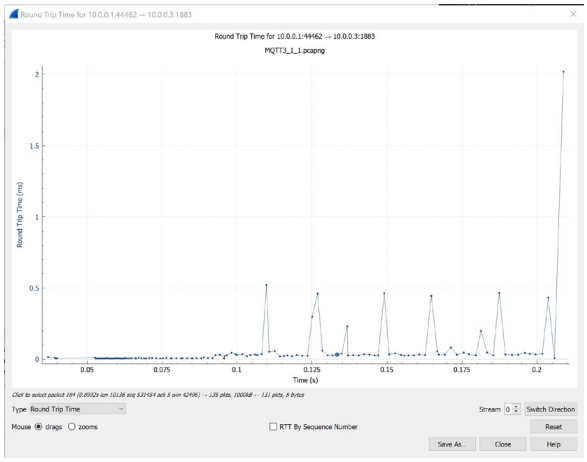


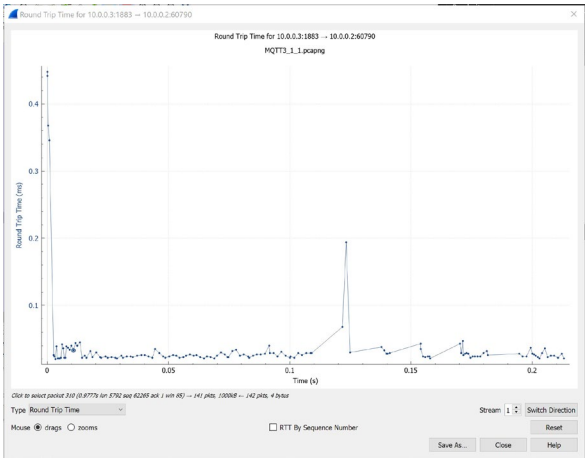
MQTT – 1 MB message size – QoS 1 (Message delivered >= 1 time)



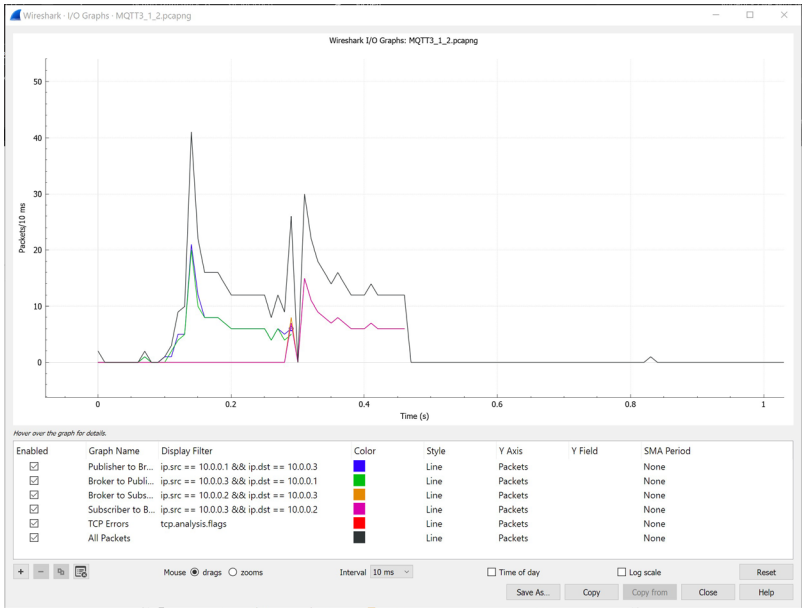


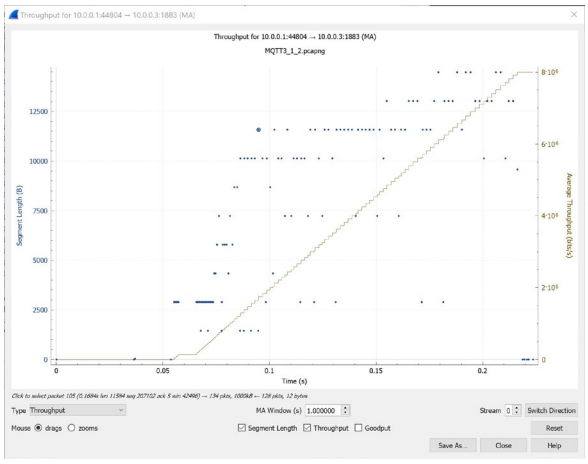


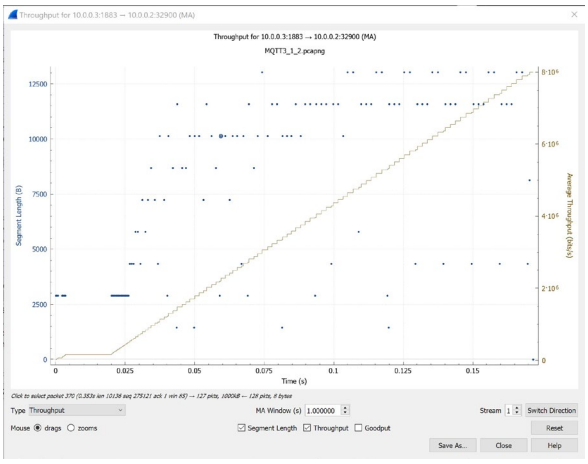


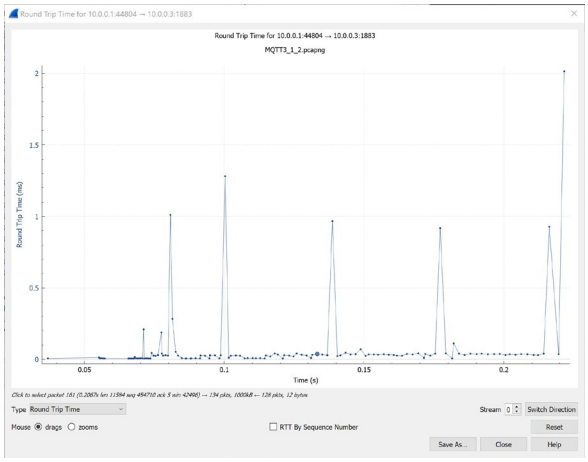


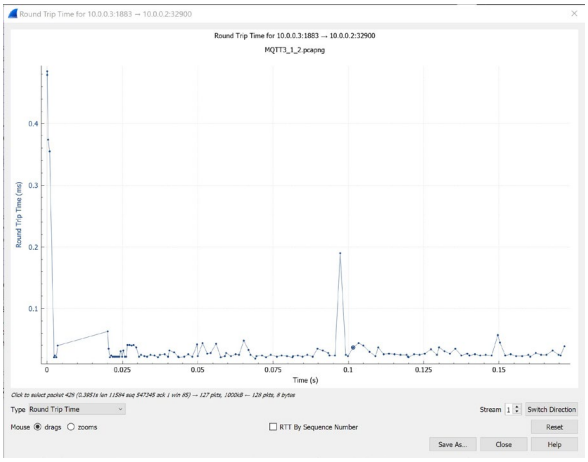
MQTT – 1 MB message size – QoS 2 (Message delivered = 1 time)



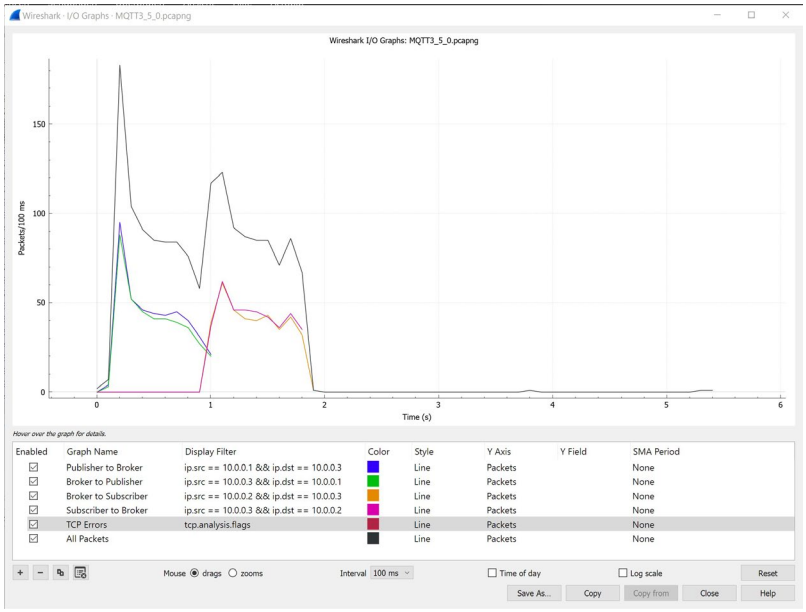


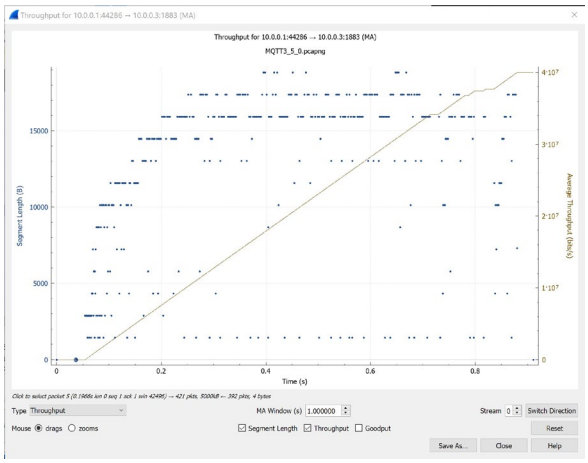


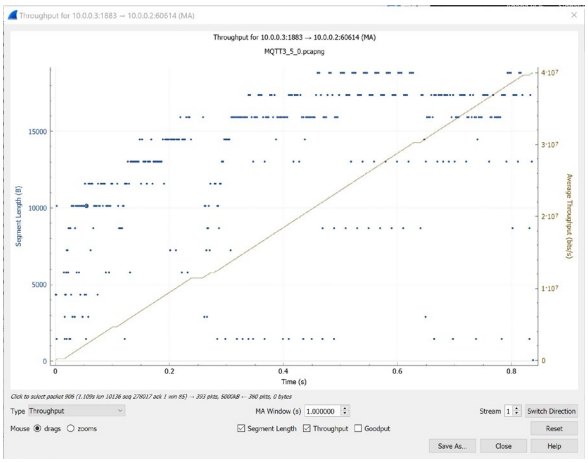


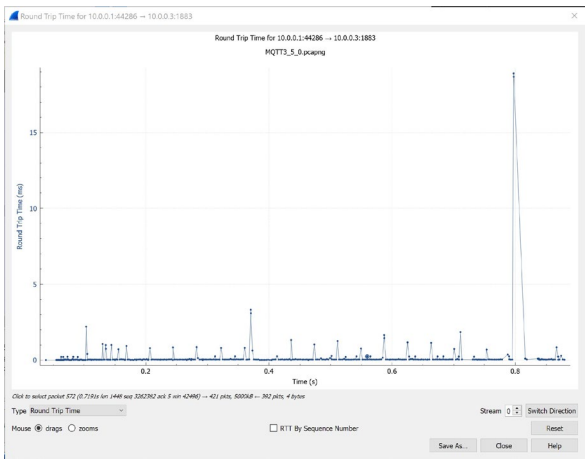


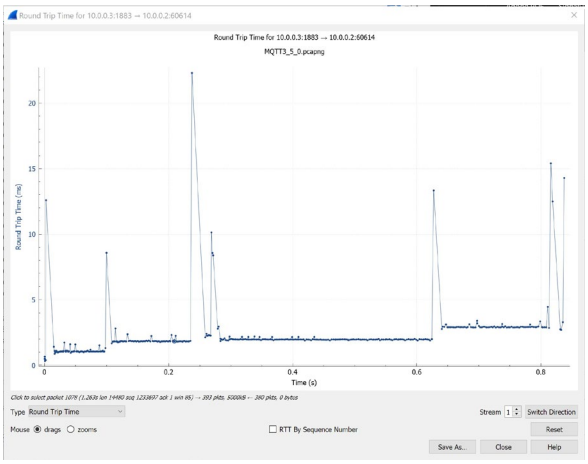
MQTT – 5 MB message size – QoS 0 (Message Delivered <= 1 time)





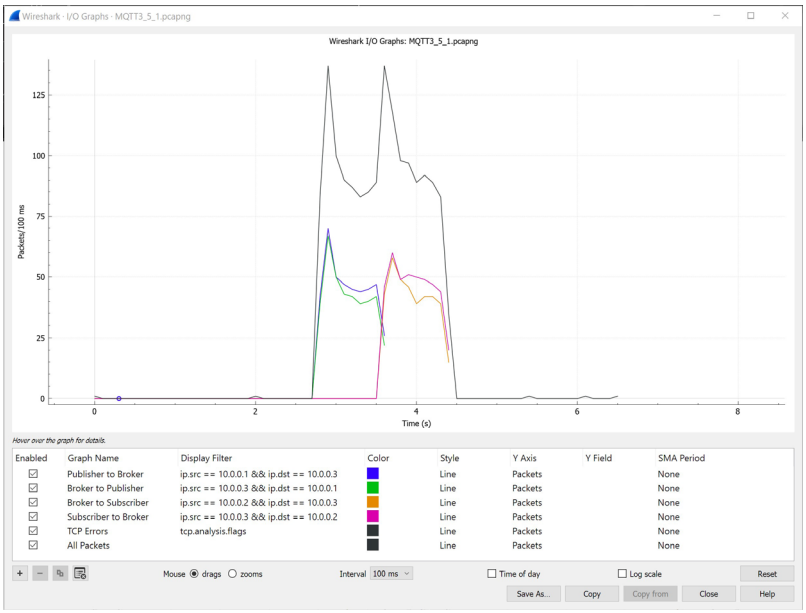


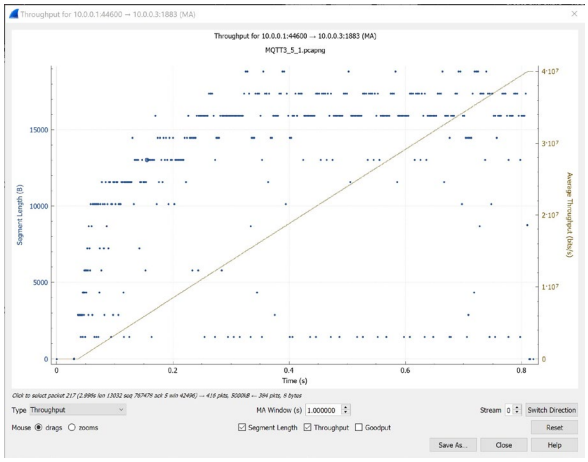


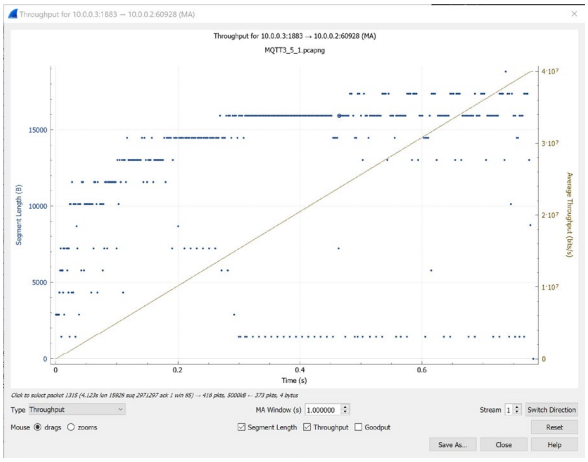


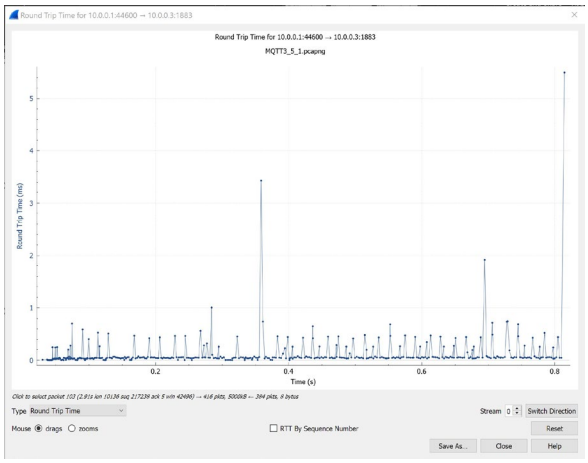
...

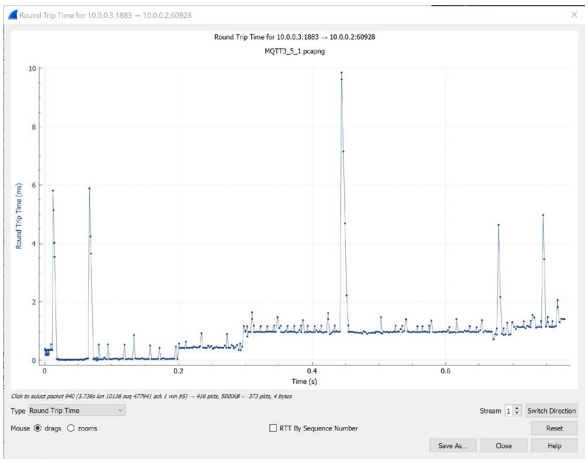
MQTT – 5 MB message size – QoS 1 (Message Delivered >= 1 time)





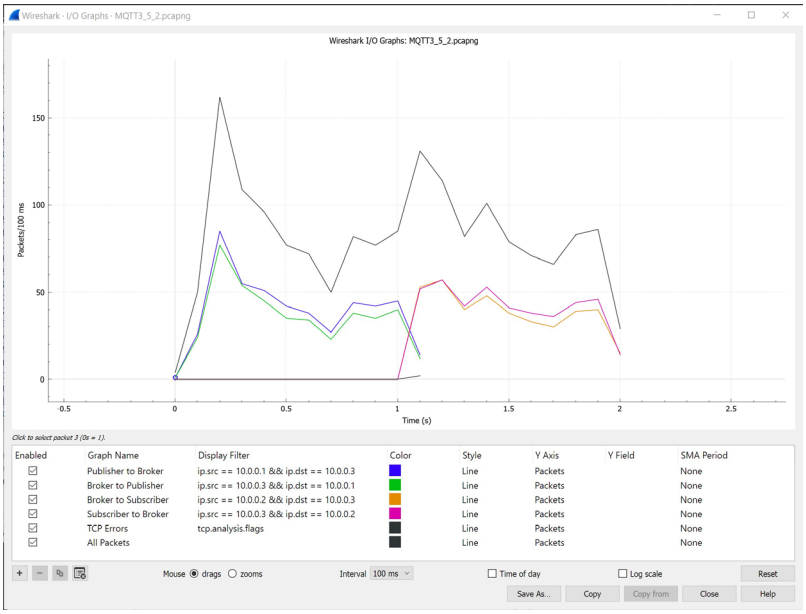


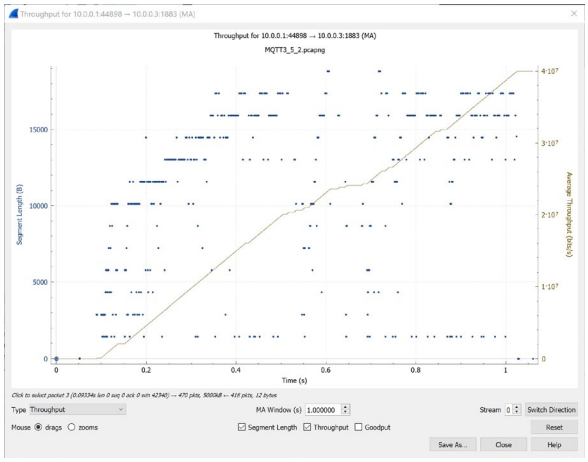


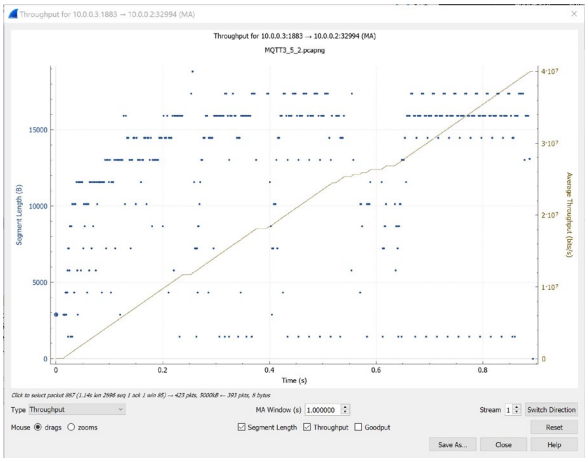


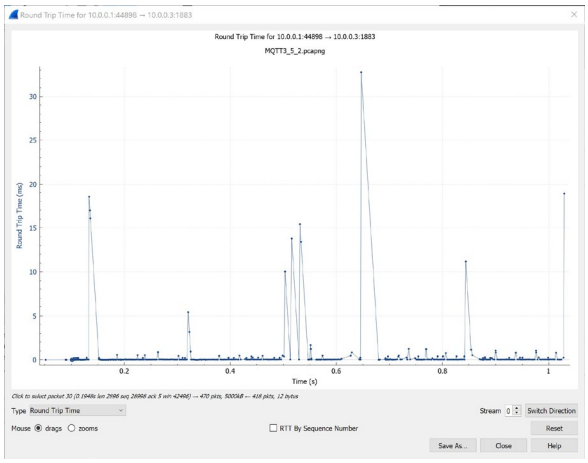
...

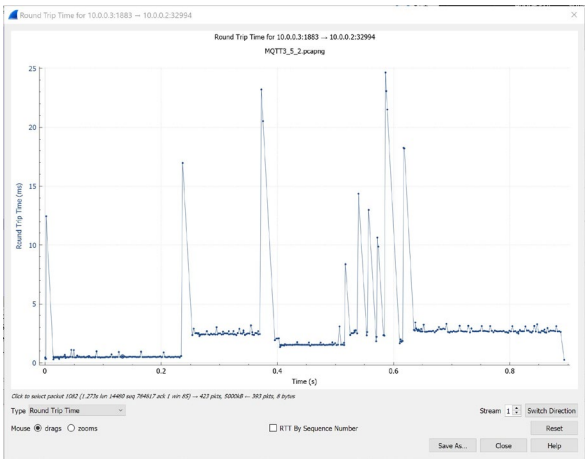
MQTT – 5 MB message size – QoS 2 (Message Delivered = 1 time)



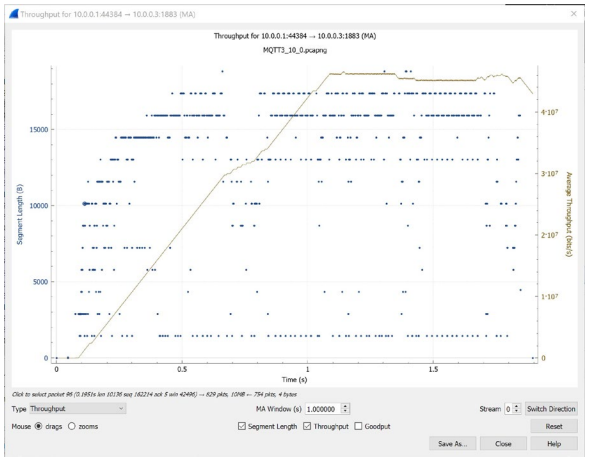


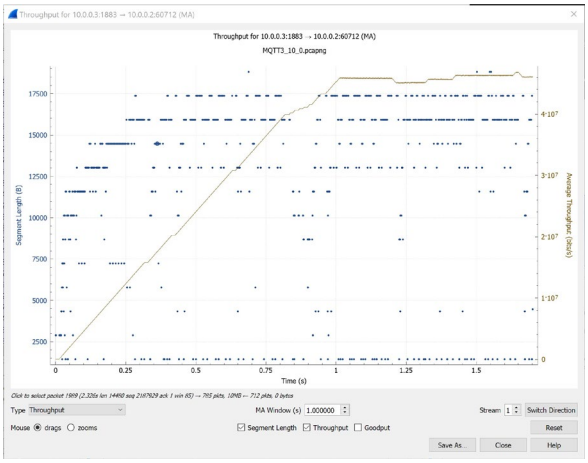


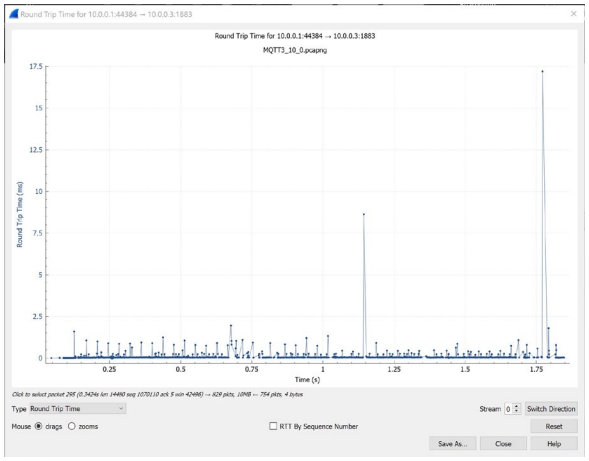


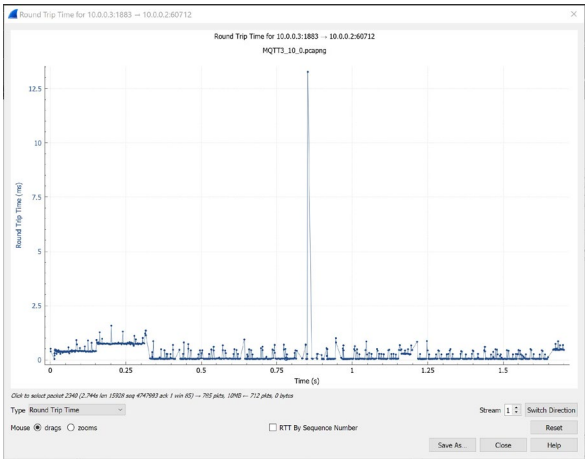


MQTT – 10 MB message size – QoS 0 (Message Delivered <= 1 time)

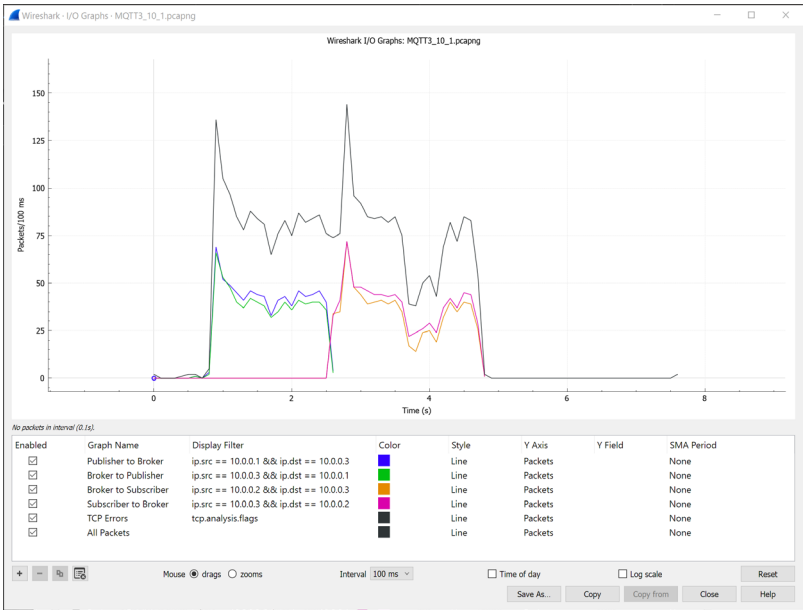


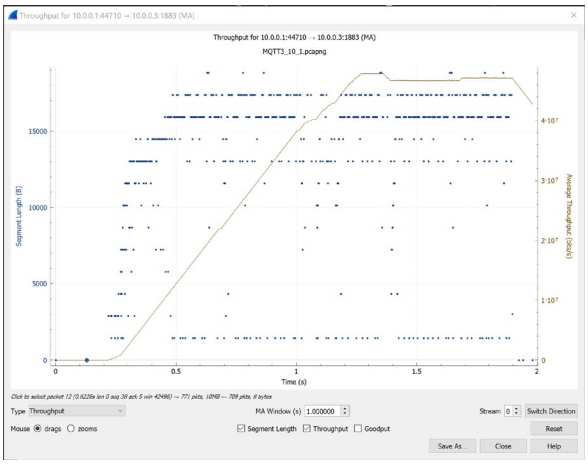


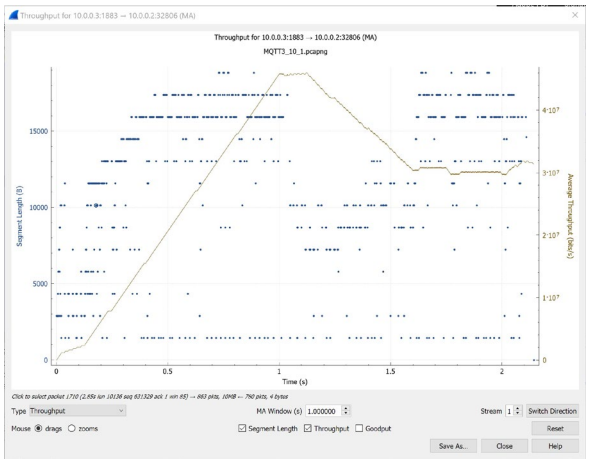


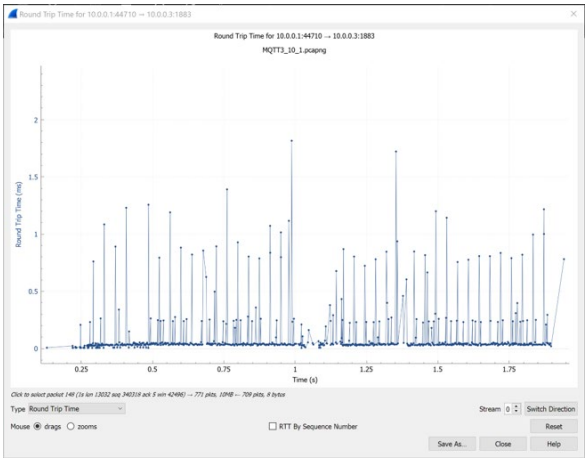


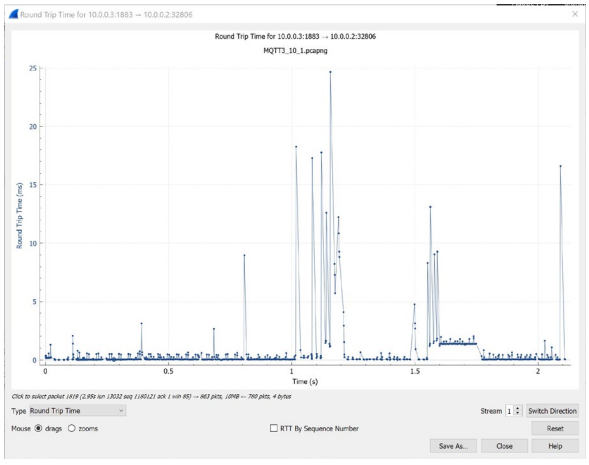
MQTT – 10 MB message size – QoS 1 (Message Delivered >= 1 time)





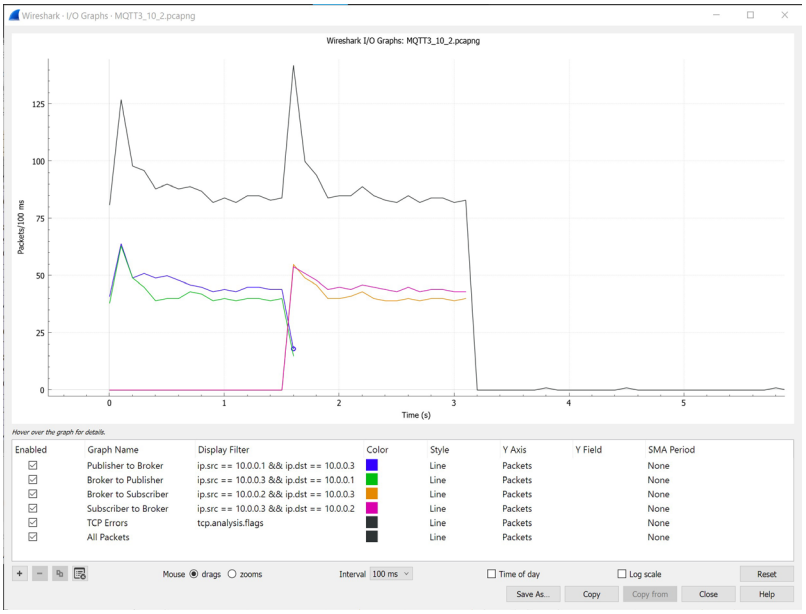


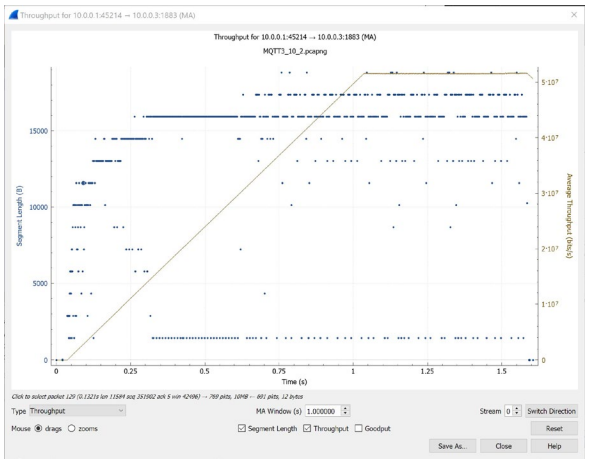


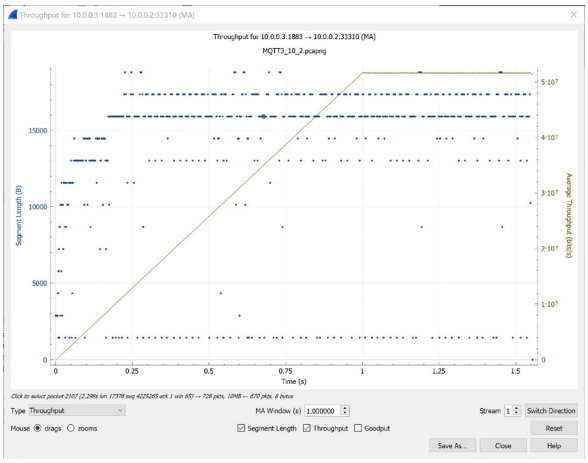


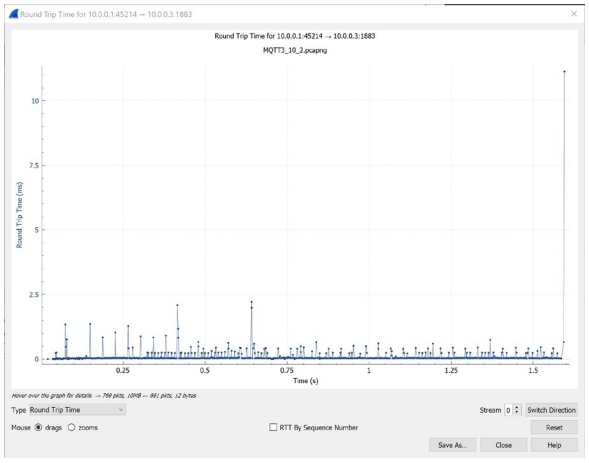
...

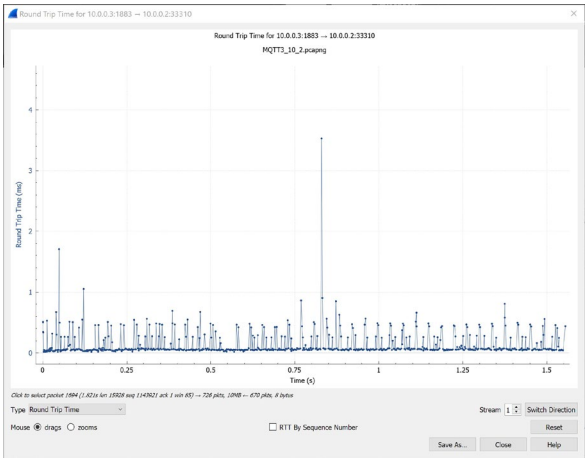
MQTT – 10 MB message size – QoS 2 (Message Delivered = 1 time)











...

Analysis of MQTT

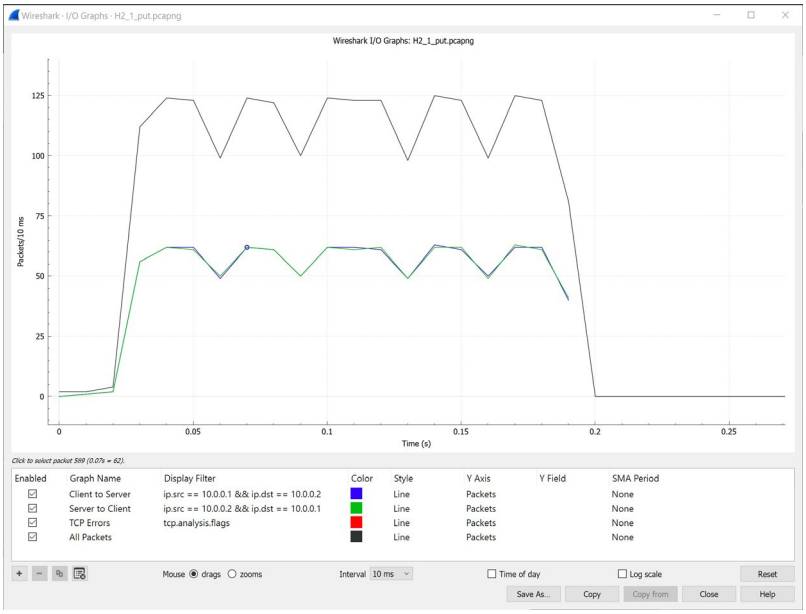
Across all the graphs measuring packet rate, 2 distinct spikes can be seen. The first spike indicates the sending of the file packets from the publisher to the broker, and the second packet indicates the forwarding of packets from the server to the subscriber.

It is interesting to note that the time taken to send all the packets from publisher to subscriber for the tested file sizes is similar for every QOS level, considering that the QOS level affects how the packets are sent for each different level. As mentioned above, the fastest and least reliable service is at QOS 0, and slowest and most reliable service is QOS 2. This is probably due to the MiniNet topology settings that defaults loss rate to be 0%, although the PUBACK, PUBREC and PUBCOMP packets has to be taken into account as well. For a file size of 1MB, the time taken is ~0.4s. For a file size of 5MB, time taken is ~2.0s. However, there is a much more noticeable deviation in the time taken to send a 10MB file for the different QOS levels. At QOS 0, ~3.6s is taken, whereas at QOS 1, ~3.8s was taken and ~3.2s was taken for QOS 2. Also, the packet rate can be seen to have fluctuated drastically across all tests, and the highest packet rate can be observed to be higher than 180 packets per 10ms.

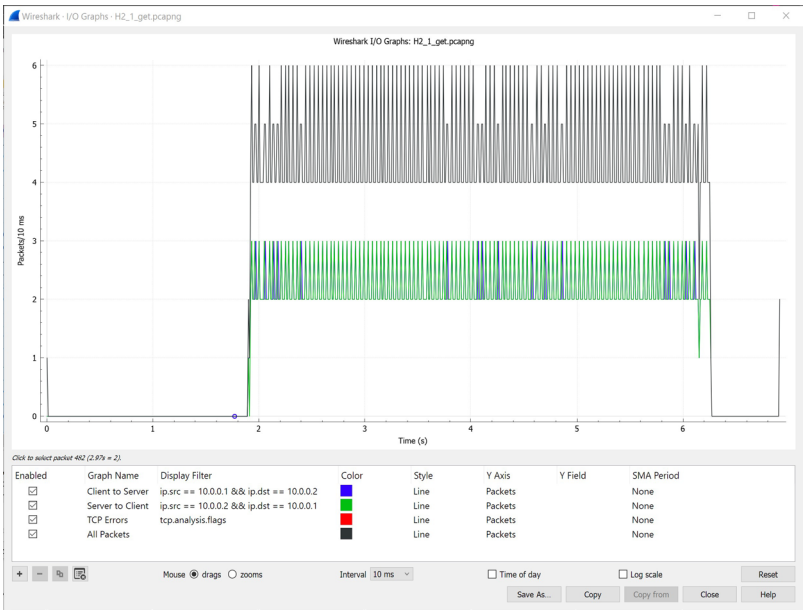
CoAP

For CoAP and three of its sections (message size 1 MB, 5 MB, and 10 MB), Host 2 (H2) is designated as the server, while Host 1 (H1) is designated as the client. The tests look at both Put and Get, whereby Put is when the client is putting the specifically-sized message onto the server, while Get is when the client is retrieving the message from the server (in other words, the server is giving out the message). The filename convention is [Host Number]_[Number of MB of a message]_put/get.pcapng.

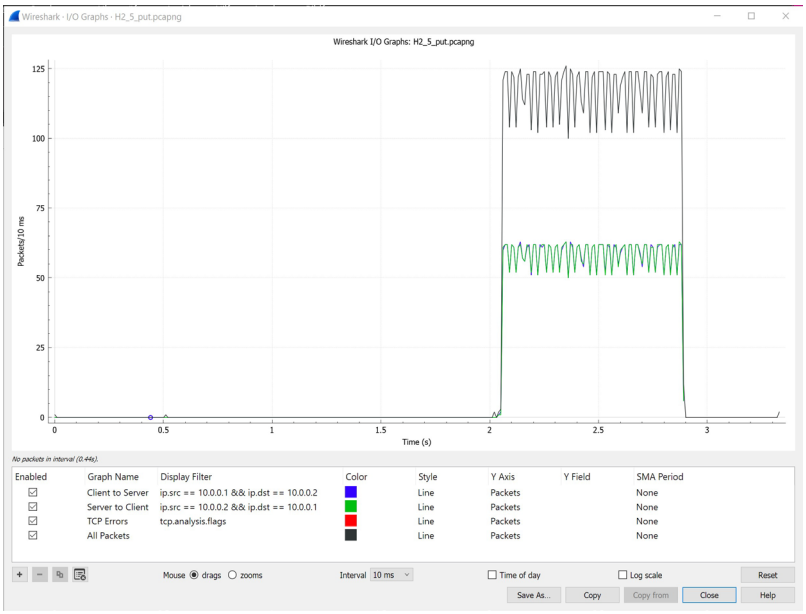
CoAP – 1 MB message size – PUT method



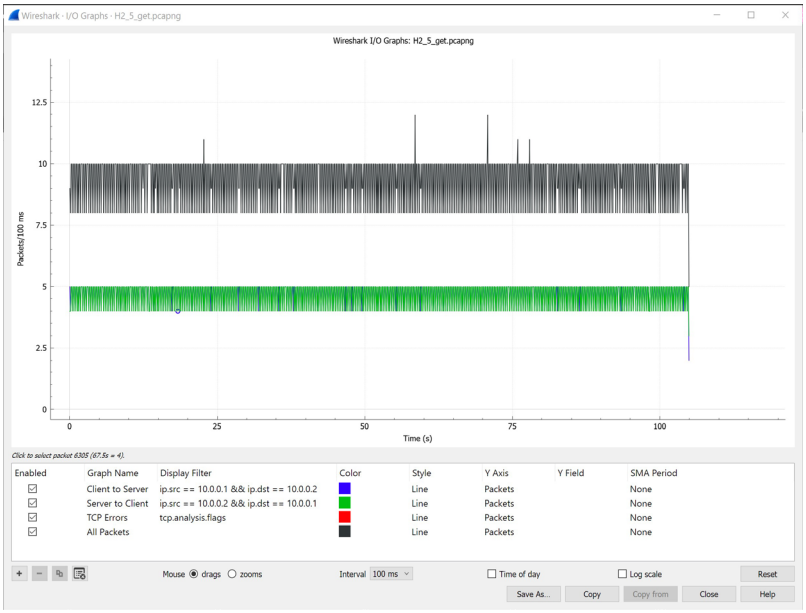
CoAP – 1 MB message size – GET method



CoAP – 5 MB message size – PUT method

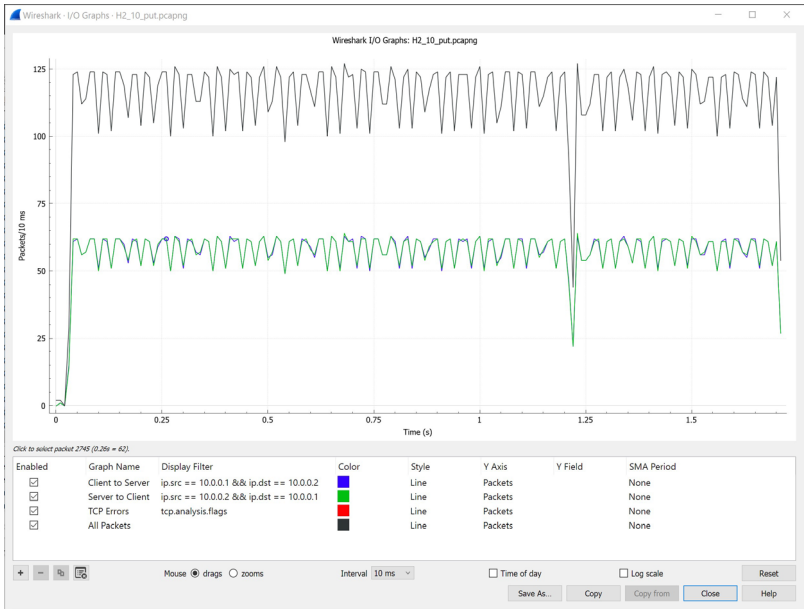


CoAP – 5 MB message size – GET method

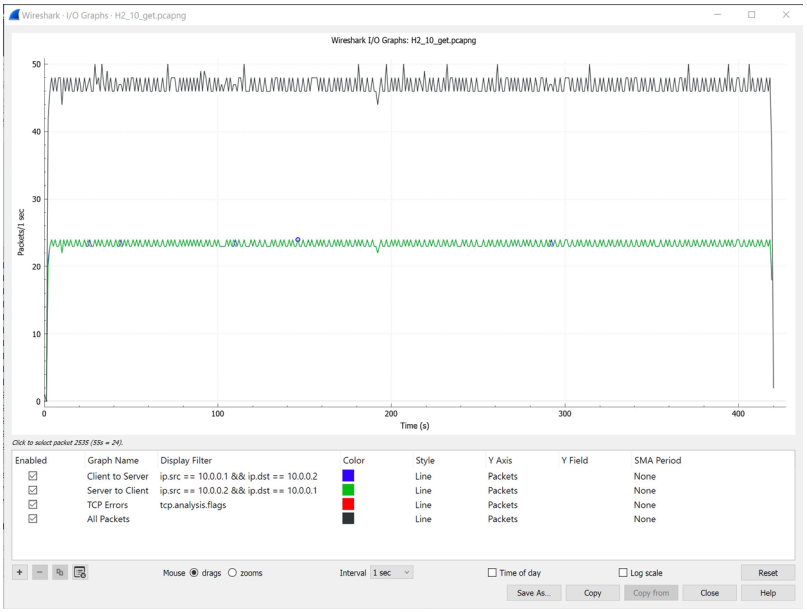


...

CoAP – 10 MB message size – PUT method



CoAP – 10 MB message size – GET method



Analysis of CoAP

On immediate look, we can observe that the GET method in general takes longer than the PUT method. The PUT method, even for the 10 MB message, would only take a second or two, while the GET method would need 4 seconds for a 1 MB message, and 400+ seconds for an 10 MB message. For both methods and for all the three message sizes, we also observe that the rate of packets sent was consistent with no drops. The packet flow between the server and the client was identical – same number of packets were sent from 10.0.0.1 to 10.0.0.2, and back from 10.0.0.2 to 10.0.0.1.

Conclusion

In conclusion, we found MQTT to be better for IoT due to its more flexible reliability. When we are sending and receiving file of a large size i.e. >100MB that requires confidentiality, it would be better to use MQTT for faster speed and increased reliability. However, files of smaller sizes However, >100MB, it would be handled better by CoAP, due to use MQTT. When we are receiving or retrieving a larger file size, however, we foundMQTT's large overhead that MQTT would fare better, since CoAP's GET method tends to yield much longer wait times.reduces speed.

Future Works

For considerable future work, we are thinking of using more hosts, or bigger message sizes, since 10 MB was still relatively small when it comes to sending messages, since most of the messaging systems nowadays have messages in order of hundreds of MBs when files (especially videos) are involved.

Additionally, we could use the WiFi systems instead of Ethernet next time in order to simulate and better assess packet loss.

Reference

[1] The HiveMQ Team, "Client, Broker / Server and Connection Establishment - MQTT Essentials: Part 3," 17 July 2019. [Online]. Available: https://www.hivemq.com/blog/mqtt-essentials-part-3-client-broker-connection-establishment/. [Accessed 1 December 2020].

[2] Tetcos, "NetSim - Network Simulator and Emulator," [Online]. Available: https://www.tetcos.com/. [Accessed 28 November 2020].

[3] Mosquitto, Eclipse Foundation, Cedalo, "Eclipse Mosquitto - An Open Source MQTT Broker," [Online]. Available: https://mosquitto.org/. [Accessed 28 November 2020].

[4] Wireshark Foundation, “Wireshark - Go Deep...,” 29 October 2020. [Online]. Available: https://www.wireshark.org/. [Accessed 28 November 2020].