Computer Networks Assignment 3 - Testing MQTT

Name: Linxiang Hu
ANU ID: u7633783

May 2024

1 Introduction about MQTT

MQTT is a standard-based messaging protocol or set of rules used for machine-to-machine communication. Smart sensors, wearable devices, and other Internet of Things (IoT) devices often need to transmit and receive data over networks with limited bandwidth and constrained resources. These IoT devices use MQTT for data transmission because it is easy to implement and can efficiently transmit IoT data. MQTT supports messaging from devices to the cloud and from the cloud to devices.
Many IoT devices connect via unreliable cellular networks that have low bandwidth and high latency. MQTT has built-in features that reduce the time it takes for IoT devices to reconnect to the cloud. It also defines three different quality of service levels to ensure reliability for IoT use cases—At Most Once (0), which is QoS 0, At Least Once (1), which is QoS 1 and Exactly Once (2), which is QoS 2.

1.1 About QoS

QoS always consists of two different message deliveries: from the client to the broker, and from the broker to the subscribed client. They have subtle differences, which means we need to discuss them separately. The QoS level for a client pushing to a broker depends on the QoS level set by the client for a specific message. More intuitively, when a client pushes a message to the broker, the QoS level is determined by the client. When the broker sends a message to a subscribed client, it uses the QoS level previously set by that client.

1.1.1 QoS 0

Figure 1: Sketch diagram of QoS 0

1. Under QoS level 0, messages are delivered on a best-effort basis, which means that they are sent but without a guarantee of successful delivery. This level does not attempt to retry message transmission, nor does it confirm whether the message has been received by the broker.

2. A message will not be acknowledged by the broker, nor will it be stored and resent by the client: at this level, once the sender dispatches the message, it does not store a copy to wait for an acknowledgment from the broker. Upon receiving the message, the broker also does not send any form of acknowledgment to confirm receipt.

3. This is also referred to as ”fire-and-forget”: As the message is not tracked or confirmed after sending, this mode is sometimes colloquially called as ”fire-and-forget.” If the message is lost during transmission, the client remains unaware and will not attempt any retries.

4. Although the MQTT protocol typically operates over TCP/IP, which is a reliable transport protocol, the application of QoS level 0 in MQTT does not leverage all the reliability features of TCP. Even though TCP ensures that packets are correctly delivered from one network endpoint to another, MQTT’s QoS level 0 does not involve additional message confirmation or retransmission, thus maintaining a ”best-effort” service.

1.1.2 QoS 1

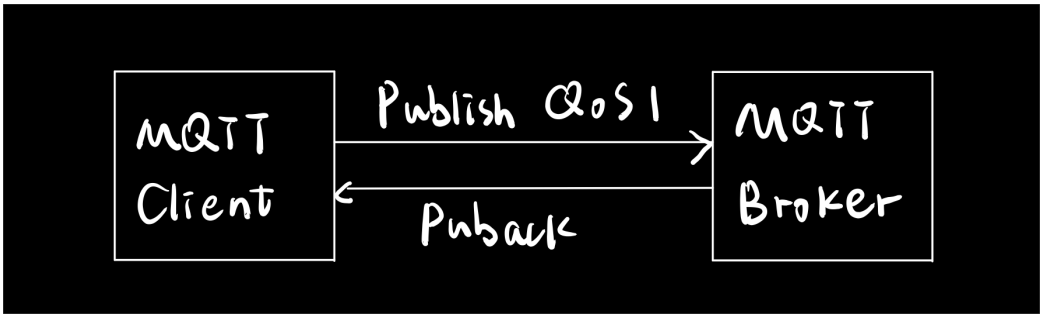

Figure 2: Sketch diagram of QoS 1

When using the Quality of Service level 1 (QoS 1) in MQTT, this level provides an ”at least once” delivery guarantee. This means:

1. Messages are sent at least once: At this level, the system ensures that messages are delivered to the broker at least once, reducing the risk of message loss during transmission.

2. Potential for duplicate messages: Although the system guarantees that messages will be delivered at least once, in some scenarios, messages might be sent multiple times. For instance, if the client does not receive a timely acknowledgment from the beoker due to network instability, it may resend the message. Therefore, message delivery using QoS level 1 could result in duplicates.

3. Message storage and acknowledgment: In this level, the sender stores a copy of the message after sending it. This copy is retained until the client receives a confirmation from the broker. The broker confirms the receipt of the message by sending a response in a specific format known as PUBACK.

Overall, QoS level 1 offers a more reliable method of message transmission compared to QoS level 0, as it reduces message loss by requiring confirmation from the receiver. However, it may also lead to the resending of messages, so it is necessary for the receiver to handle potential duplicates when using this level.

1.1.3 QoS 2

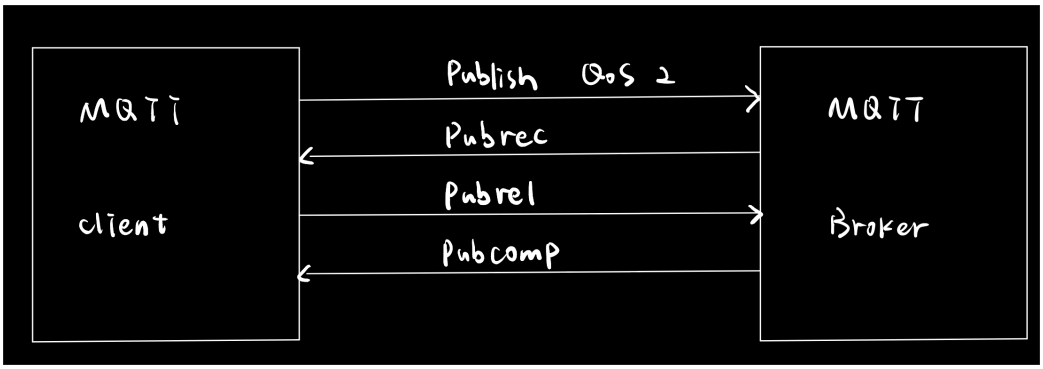

Figure 3: Sketch diagram of QoS 2

In the MQTT protocol, Quality of Service level 2 (QoS 2) provides the highest level of message delivery assurance. This level guarantees that each message is received exactly once, making it the most reliable service level, which means:

1. Guaranteed single reception: When the broker receives a QoS 2 PUBLISH message, it processes the message and then acknowledges it by sending a PUBREC message back to the client. This acknowledgment is the initial step in the process.

2. Prevention of message duplication: After sending the PUBREC message, the broker tracks the packet identifier of the message until it sends a PUBCOMP message. This ensures that each PUBLISH packet is processed only once, preventing duplicate processing.

3. Intermediate acknowledgment and final completion: Once the client receives the PUBREC message, it knows that the message has been received by the broker and can discontinue retaining the original PUBLISH message. The client then saves the PUBREC information and sends a PUBREL message. When the broker receives PUBREL, it discards all stored states for that message and responds with a PUBCOMP message. Finally, when the client receives the PUBCOMP message, it also performs the same actions, marking the completion of the message delivery process.

4. Handling message loss: At any point, if a message packet is lost, the client is responsible for resending the last transmitted message after a specific

time. Similarly, the broker is obligated to respond to every control message received.

In summary, QoS level 2 offers the highest level of reliability in message delivery, ensuring each message is accurately received and processed only once through a series of acknowledgments and state tracking, thus preventing any loss or duplication of messages. This level is suitable for applications that demand high reliability in data transmission.

2 Code Part

2.1 Publisher

This part is mainly used to dynamically manage and send messages to MQTT brokers.

1. **Main Data Structure**

ExecutorService: Java's ExecutorService is used to manage the thread pool, which makes it possible to execute tasks on multiple publisher instances in parallel. Just like what the **Fig 4** shows.

```
private static final ExecutorService executorService = Executors.newCachedThreadPool();
```

Figure 4: Code about thread pool

```
@Override
public void messageArrived(String topic, MqttMessage message) throws Exception {
    if ("request/instancecount".equals(topic)) {
        int requestedCount = Integer.parseInt(new String(message.getPayload()));
        int currentCount = currentInstanceCount.get();
        // Start the corresponding number of instances to publish messages
        while (currentCount < requestedCount) {
            System.out.println("Create New Instance");
            int newInstanceId = currentCount + 1;
            executorService.submit(() -> runPublisher(newInstanceId));
            currentInstanceCount.incrementAndGet();
            currentCount++;
        }
```

Figure 5: How it achieves multiple publishers publishing

When the publisher received a updated instance count value from the analyzer, the thread pool allowed it to start multiple threads to start publishing just like in **Fig**

5.
AtomicInteger: AtomicIntegers are used to safely manage instance counting in a concurrent environment and ensure the correct number of instances. When we want to start multiple publishers, due to the asynchronous nature of threads, we may end up launching too many instances. Using synchronization ensures state consistency and prevents us from starting too many or too few instances just like the **Fig 6**.

```
private static final AtomicInteger currentInstanceCount = new AtomicInteger( initialValue: 0);
```

Figure 6: AtomicIntegers

HashMap: A HashMap, named sharedSettings, is utilized to store shared configuration settings such as delay (delay) and Quality of Service (QoS). This allows for efficient and quick access to these settings across various threads that may need to use or update this information concurrently. When the analyzer updates the settings about the delay and quality of service. The code will update the updated parameters to it. Before publishers send messages, they will retrieve the control parameters. **Fig 7** shows the HashMap for storing setting information.

```
// Stores settings for delay and qos
private static final Map<String, String> sharedSettings = new HashMap<>();
```

Figure 7: HashMap for storing settings

Object lock and synchronization: The use of an object lock along with synchronization blocks serves to manage concurrency among threads. By employing a lock and synchronized sections within the code, we ensure that updates to the shared settings are done atomically and consistently. This mechanism is critical in maintaining the integrity of data and preventing race conditions where multiple threads may attempt to modify the same data at the same time, potentially leading to inconsistent or erroneous states.

```
private static void runPublisher(int instanceId) {
    try {
        System.out.println("Enter runPublisher Method");
        MqttClient client = new MqttClient( serverURI: "tcp://localhost:1883", MqttClient.gen
        // Connection setting
        MqttConnectOptions options = new MqttConnectOptions();
        options.setCleanSession(true);
        client.connect(options);
        client.subscribe( topicFilter: "request/#");

        while (true) {
            synchronized (lock) {
                while (!settingsUpdated) {
                    // Use while loop to avoid spurious wakeup
                    // Wait until all settings are updated
                    lock.wait();
                }
                // Reset flag after settings are confirmed to be updated
                settingsUpdated = false;
            }
            String delay = sharedSettings.getOrDefault( key: "delay",  defaultValue: "1000");
            String qos = sharedSettings.getOrDefault( key: "qos",  defaultValue: "0");
            System.out.println("Ready to publish with delay: " + delay + ", qos: " + qos
            publishMessages(client, Integer.parseInt(delay), Integer.parseInt(qos), instanc
            Thread.sleep( millis: 1000);
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Figure 8: The use of lock

Fig 8 shows the use of lock to make sure that the publisher must publish messages after the analyzer updates the settings.

Like the Bonus Question 1 asks, there would be some problems like if the analyzer updates the parameters before the publishers send the messages. In theory, the publishers listen for the changes of control messages and send data according to the updated parameters. However, in practise, the transmission has delay, which might give rise to the situation that publisher started sending messages using the old parameters before the new parameters arrived.

With the aid of the lock, after a publisher completes transmission, if the parameters have not been updated, the threads will be blocked until the parameters are updated. Only then will these threads proceed with new publishing tasks. This

ensures that data does not become out of order during transmission due to the asynchronous nature of the threads.

2. **Functional Logic**
MQTT client initialization: First create an MqttClient and connect to the local MQTT broker. MemoryPersistence is used to ensure that state is maintained even when the disk is unavailable.

Connection options: Set MqttConnectOptions where setCleanSession(true) ensures that every connection is a clean session, which is beneficial for application scenarios where old messages don't need to be kept.

3. **Message Handling**
Handle incoming messages according to their different message topics.

1. In the case of the "request/instancecount" topic, a new instance is dynamically created based on the request and submitted to the thread pool for execution.
2. If other topics start with "request/", the sharing settings are updated and the blocked threads are notified when the expected number of updates is reached.

4. **Run Logic**

1. runPublisher method: This is the method that actually runs each publisher instance. Each instance subscribes to the "request/#" topic and waits until all settings to be updated to start sending messages on a regular basis.
2. Publish message logic: Send messages continuously based on delay and QoS in the HashMap storing settings, and after each instance runs for one minute, publish its final message count to another topic. By sending an extra final count to another topic, the analyzer can easily retrieve the expected message numbers for later data statistics.

2.2 Analyzer

1.**Member Variables**

1. client: An MqttClient object used for communication with the MQTT broker.

2. dataMap: A ‘ConcurrentHashMap‘ used to store data from various publishers, supporting high concurrent access. This can prevent data confusion caused by the concurrency of threads. Because at the same time, there may be multiple publishers publishing data, using a regular HashMap might lead to data confusion. However, using ConcurrentHashMap allows data to be stored sequentially and safely.

3. systemMetricsData: A HashMap used to store system metrics data, typically including performance and health information published from the MQTT system.

4. currentInstanceId, currentPublisherQoS, currentAnalyzerQoS, currentDelay: Variables that store the current instance ID, publisher QoS, analyzer QoS, and message delay, respectively.

```
// MQTT broker URL
private static final String BROKER_URL = "tcp://localhost:1883";
// MQTT client instance
private MqttClient client;
// Stores data received from publishers
private Map<String, InstanceData> dataMap = new ConcurrentHashMap<>();
// Stores system metrics received
private Map<String, List<String>> systemMetricsData = new HashMap<>();
private int currentInstanceId = 0;
private int currentPublisherQoS = 0;
private int currentAnalyzerQoS = 0;
private int currentDelay = 0;
```

Figure 9: Member Variables of Analyzer

2. **Data Structure**
In order to save the received data and analyze it later, I created two new classes
2.1 Message Class
It represents a single message with its value and the timestamp when it was received. This simple structure is crucial for analyzing the order and timing of MQTT messages. The code about it is shown in **Fig 10**.

```java
/**
 * Represents a single message with its value and the timestamp when it was received.
 * This simple structure is crucial for analyzing the order and timing of MQTT messages.
 */
static class Message {
    // The numerical value or identifier of the message.
    int value;
    // The timestamp when the message was received, used for timing analysis.
    long timestamp;

    /**
     * Constructs a new Message instance with specified value and timestamp.
     * @param value The numerical value of the message, often used as a sequence number.
     * @param timestamp The exact time the message was received, measured in milliseconds since epoch.
     */
    public Message(int value, long timestamp) {
        // Set the message value, typically used as an identifier or counter.
        this.value = value;
        // Set the timestamp of when the message was received.
        this.timestamp = timestamp;
    }
}
```

Figure 10: Structure of Message

2.2 InstanceData Class

This class represents the data structure for storing information about the messages received for each topic. As the **Fig 11** shows, the published count denotes the expected received message number and received messages are organised by the Quality of Service (QoS) level of the analyzer.

```java
static class InstanceData {
    // The expected number of messages
    int publishedCount;
    // Maps each QoS level to a list of received messages.
    Map<Integer, List<Message>> messagesByQos = new HashMap<>();

    /**
     * Constructor for the InstanceData class. Initializes the published count to zero
     * and prepares the map to store lists of messages for each QoS level from 0 to 2.
     */
    public InstanceData() {
        this.publishedCount = 0;
        for (int i = 0; i < 3; i++) {
            messagesByQos.put(i, new ArrayList<>());
        }
    }
}
```

Figure 11: Structure of InstanceData

3. **Key Methods and Functions**

(a) runAnalyser(int numInstances):

The input parameters denote the maximum number of publishers. The number of publisher instances will increase from 1 to the value of ‘numInstances‘ at end.

1. Initializes the MQTT client and connects to the broker.
2. Sets up message callbacks to handle connection loss, message arrival, and delivery completion.
3. Subscribes to relevant topics and loops through different configurations of instance counts, publisher QoS, analyzer QoS, and message delays.
4. Sends configuration requests to publishers to control their behavior, waits for message accumulation, and then proceeds with data writing and statistical analysis.
5. Within each publication, the publishers would publish for 1 minute. Consider the situation that the transmission needs time, the analyzer will wait for a longer time which is 70 seconds here to ensure that every messages arrive.

```
// Loop to handle configurations and message accumulation
for (int instanceId = 1; instanceId <= numInstances; instanceId++) {
    for (int publisherQos = 0; publisherQos < 3; publisherQos++) {
        for (int analyzerQos = 0; analyzerQos < 3; analyzerQos++) {
            currentInstanceId = instanceId;
            currentPublisherQoS = publisherQos;
            currentAnalyzerQoS = analyzerQos;

            updateSubscription(analyzerQos);
            for (int delay : new int[]{0, 1, 2, 4}) {
                currentDelay = delay;
                sendConfigurationRequests(instanceId, publisherQos, delay);
                Thread.sleep( millis: 70000);
                // Wait for messages to accumulate
                writeSysMetricsToFile();
                calculateStatistics(instanceId, publisherQos, analyzerQos);
            }
        }
    }
}
```

Figure 12: Key part of ‘runAnalyzer‘

(b)handleMessages(String topic, MqttMessage message):
In the callback function of client, since the analyzer subscribe 3 topics, it would dispatch messages to appropriate handler functions based on the message's topic.

1. counter messages. If this topic updates, it means that the analyzer receives messages sent by the publishers. The analyzer will add it to the 'dataMap', which is a hashmap whose key is the unique message topic path of string type and value is an instance of InstanceData.

2. published count messages. If this topic updates, it means that the analyzer receives messages about the expected number of received messages. The analyzer will store it into the attribute of 'InstanceData' class for later data processing.

3. system metrics. If this topic updates, it means that the analyzer receives messages about system information. We will discuss this how system information changes when sending messages.

The detailed code is shown in **Fig 13**.

```
private void handleMessages(String topic, MqttMessage message) throws IOException {
    // Convert the binary payload to a string for processing.
    String payload = new String(message.getPayload());
    // Split the topic string by '/' to analyze its components.
    String[] topicParts = topic.split( regex: "/");

    // Split the topic string by '/' to analyze its components.
    if (topic.startsWith("counter")) {
        // Handle counter messages, which typically involve updating or tracking message counts from publishers.
        handleCounterMessage(topicParts, payload, currentAnalyzerQoS);
    } else if (topic.startsWith("published_count")) {
        // Handle published count messages, which provide the total count of messages sent by a publisher.
        handlePublishedCountMessage(topicParts, payload);
    } else if (topic.startsWith("$SYS")) {
        // Handle system metrics messages, which provide diagnostic and health information from the MQTT broker or system.
        handleSysMetrics(topic, payload);
    }

}
```

Figure 13: Code part about 'handleMessages'

4. **Thread Safety and Concurrency Management**

1. Like mentioned before, ConcurrentHashMap instead of normal hashmap is used to avoid data confusion when storing data retrieved.

2. Synchronization blocks are used (Both when handling message data and doing calculations) to ensure thread safety in a multi-threaded environment. **Fig 14** shows one example where the synchronization block is used in my code.

```
synchronized (data) {
    data.messagesByQos.get(currentSubQos).add(new Message(Integer.parseInt(payload), System.currentTimeMillis()));
}
```

Figure 14: Synchronization Blocks are Used

3 Question 2

3.1 Wireshark

To wireshark the impact of the different QoS levels, I write a publisher and subscriber for screenshot. Both of them will subscribe to the topic ‘test/wireshark‘ and the publisher will send only one message with content ‘Hello MQTT‘ in it using different QoS levels. Just like what **Fig 15** shows.

```
public static void main(String[] args) {
    // MQTT broker URL and port
    String broker = "tcp://localhost:1883";
    // Client identifier used to communicate with the MQTT broker
    String clientId = "Hello MQTT";
    // MQTT topic to publish messages
    String topic = "test/wireshark";
    // Message content to send
    String content = "Hello MQTT";
```

Figure 15: Code about Testing Different QoS

3.1.1 Keep Alive

```
2864 253.239379    127.0.0.1          127.0.0.1          MQTT      46 Ping Request
2866 253.239456    127.0.0.1          127.0.0.1          MQTT      46 Ping Response
```

Figure 16: Screenshot of Ping

In MQTT protocol, the mechanism of keeping the connection is implemented by PING requests and responses like shown in **Fig 16**. When my subscriber starts up, it needs to maintain a persistent connection to the server.

1. If there is no data transfer for a certain amount of time, the server may assume that the client has been disconnected. PING requests and responses prevents such misjudgments.

2. By periodically sending PING requests, the client can confirm that the connection status with the server is normal. If the server does not respond to the PING request within the expected time, the client can reconnect or take other measures.

3.1.2 SYS information

The $SYS topic is typically used to publish system information and status of MQTT proxy servers. **Fig 17** shows the various system information when I started the subscriber and used the publisher to publish messages.

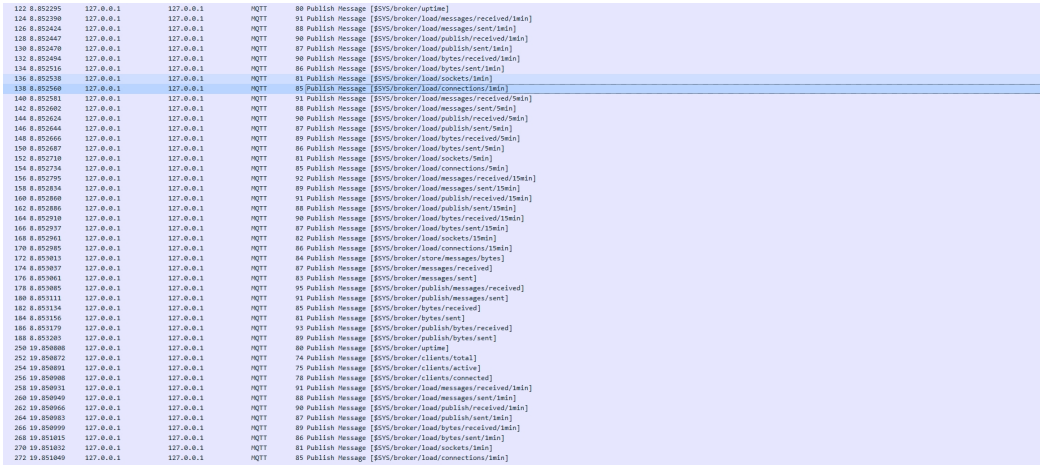

Figure 17: Screenshot of SYS

For example,

1. The information in the format of $SYS/broker/uptime means the time when the proxy server is running. The message content is typically the number of seconds since the proxy server started.

2. The information in the format of $SYS/broker/load/messages/received/x means the number of messages received in past x minutes.

3. The information in the format of $SYS/broker/load/messages/sent/x means the number of messages sent in past x minutes.

4. The information in the format of $SYS/broker/load/sockets/x means the number of socket connections in the past x minute.

5. The information in the format of $SYS/broker/load/connections/x means the number of connections in the past x minute.

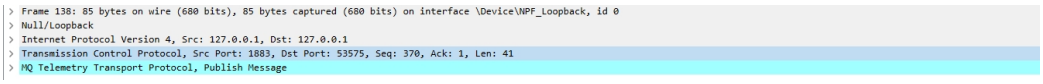

Figure 18: Detailed information about ‘SYS‘

Fig 18 shows the detailed information if we click on a system frame.

1. ACK 1: It means that it has received the frame numbered ‘0‘ and waiting for the frame numbered ‘1‘. This matches the fact that we published one message called ‘Hello MQTT‘ using publisher.

2. MQTT Publish Message: This line clearly indicates that this is an MQTT publish message. This means that the client has published a message to the agent.

3.1.3 QoS=0

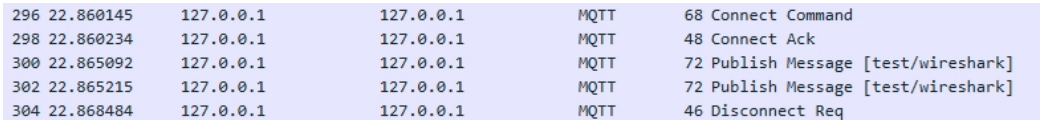

Figure 19: Screenshot of QoS 0

Fig 19 shows the screenshot of wireshark after the publisher sent the data with QoS 0.

1. Connect: The Connect command is the first message sent by the client to the server to establish a network connection to the server. When the MQTT client sends the Connect command to the server, the server responds with a Connect Ack (Confirm connection request) message indicating whether the connection request was accepted and the reason for the connection failure (if any).

2. DisConnect: The Disconnect command is sent by the client to notify the server that it is closing the connection. After sending the Disconnect command, the client should stop communication and close the network connection.

In this part, my code only published one message. Therefore, the screenshot correctly reflects the connection and disconnection part of publisher sending the first message. **Fig 20** shows the detailed information of two publish frames in **Fig 19**. It can be seen that both of the images reveals that the QoS level is at most once, which is corresponds to the fact. And the dst port of left one is 1883 and the src port of right one is 1883 as well. This means that both transfers from publisher to broker and transfers from broker to other places will be sent only once, regardless of receipt acknowledgement. It is corresponds to the part **1.1.1**.

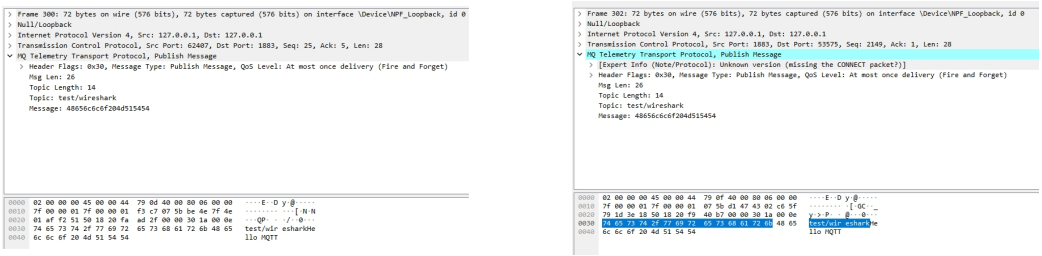

Figure 20: Detailed Information of QoS 0

3.1.4 QoS=1

139	4.933608	127.0.0.1	127.0.0.1	MQTT	68 Connect Command
141	4.933677	127.0.0.1	127.0.0.1	MQTT	48 Connect Ack
143	4.937259	127.0.0.1	127.0.0.1	MQTT	74 Publish Message (id=1) [test/wireshark]
145	4.937315	127.0.0.1	127.0.0.1	MQTT	72 Publish Message [test/wireshark]
147	4.937345	127.0.0.1	127.0.0.1	MQTT	48 Publish Ack (id=1)
149	4.939741	127.0.0.1	127.0.0.1	MQTT	46 Disconnect Req

Figure 21: Screenshot of QoS 1

Fig 21 shows the wireshark screenshot after the publisher sent the message with QoS 1. Compared to **Fig 19**, the only difference is that there is a 'PUBLISH ACK' in **Fig 21**. This is because, like I mentioned in part **1.1.2**. The broker will acknowledge the receipt of information to ensure that every message is sent to the broker at least once.

3.1.5 QoS=2

145	5.789386	127.0.0.1	127.0.0.1	MQTT	68 Connect Command
147	5.789445	127.0.0.1	127.0.0.1	MQTT	48 Connect Ack
149	5.792678	127.0.0.1	127.0.0.1	MQTT	74 Publish Message (id=1) [test/wireshark]
151	5.792726	127.0.0.1	127.0.0.1	MQTT	48 Publish Received (id=1)
153	5.792925	127.0.0.1	127.0.0.1	MQTT	48 Publish Release (id=1)
155	5.792959	127.0.0.1	127.0.0.1	MQTT	72 Publish Message [test/wireshark]
157	5.792984	127.0.0.1	127.0.0.1	MQTT	48 Publish Complete (id=1)
159	5.795386	127.0.0.1	127.0.0.1	MQTT	46 Disconnect Req

Figure 22: Screenshot of QoS 2

Fig 22 shows the wireshark screenshot after the publisher sent the message with QoS 2. It is exactly the same as what I mentioned in **Fig 3** in part **1.1.3**.To ensure the guarantee of "exactly once ", the MQTT protocol defines a handshake process consisting of four steps, in which each message transmission requires confirmation.

3.2 Data analyze

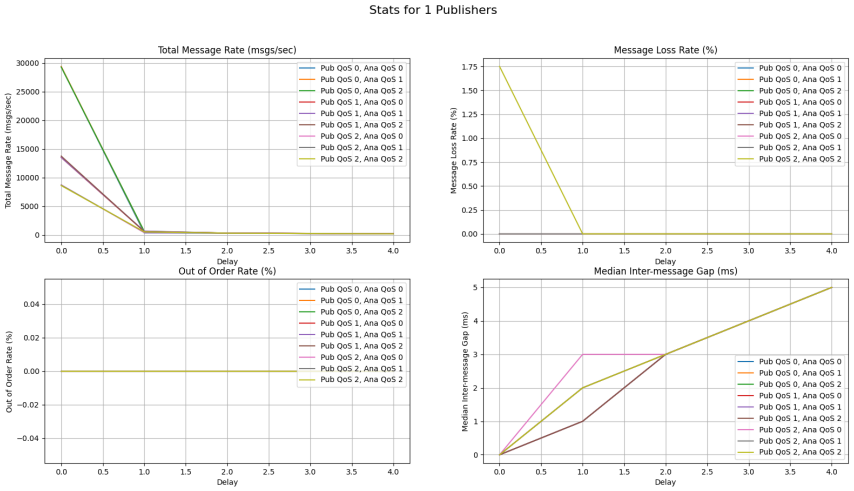

Figure 23: Stats for 1 publisher

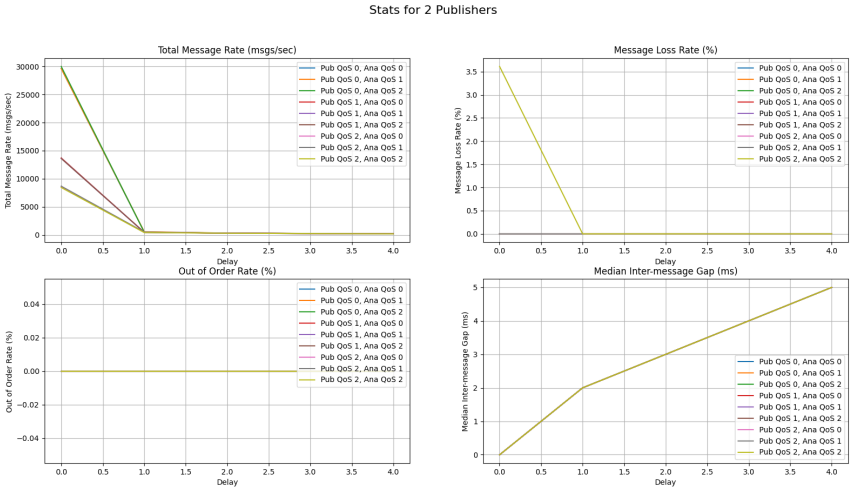

Figure 24: Stats for 2 publishers

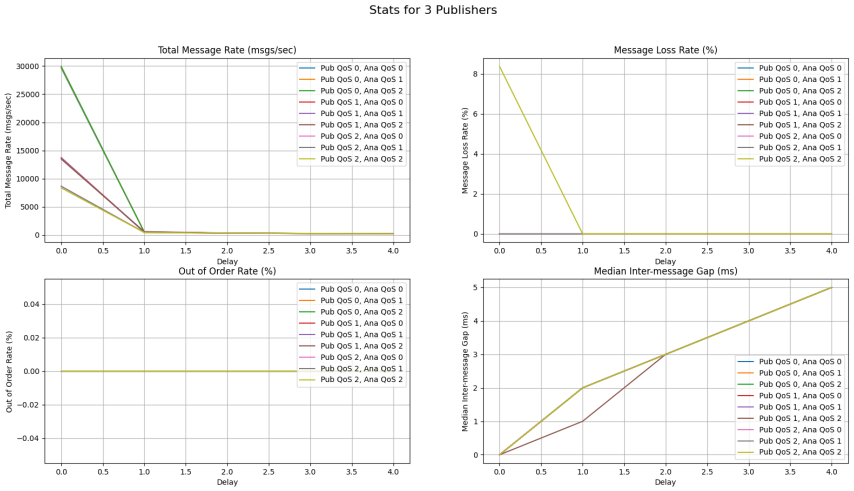

Figure 25: Stats for 3 publishers

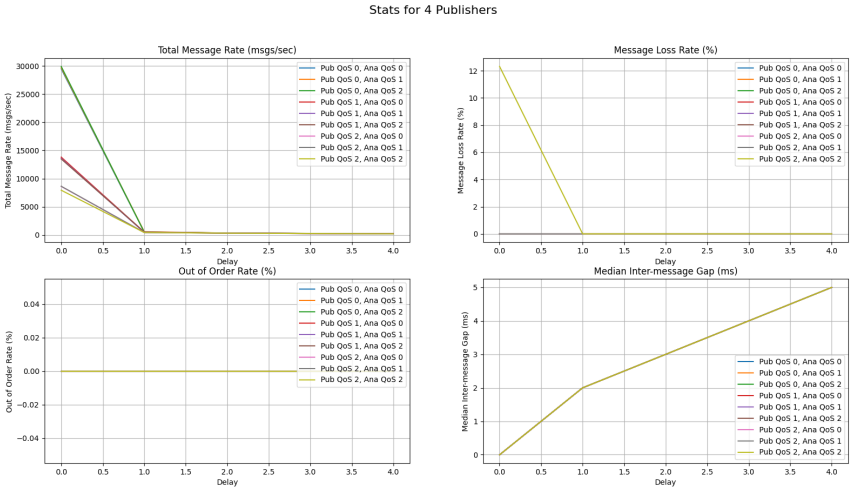

Figure 26: Stats for 4 publishers

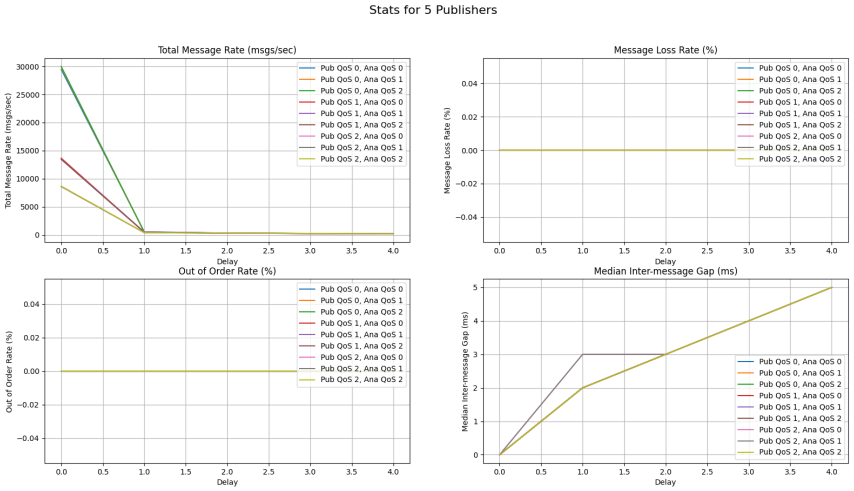

Figure 27: Stats for 5 publishers

From **Fig 23** to **Fig 27**, they shows how total message rate, message loss rate, out of order rate and median inter-message gap change as the delay and combination of QoS change within the 180 test samples.

Total Message Rate

1. From 1 publisher to 5 publishers, it can be seen that the total message rate decreases as the value of delays increases. This is normal because the publisher will sleep for the duration of the delay after sending a message, and then send another message.

2. Because my computer's CPU has 16 cores, and at most there are only 5 publishers. In Java, when we use multithreading, a thread task is assigned to a core for execution (not considering hyper-threading). Therefore, due to my computer's good configuration, the increase in the number of publishers does not significantly affect the total message rate.

Message Loss Rate

1. As the number of publishers increases from 1 to 4, with both the subscriber and publisher set to a QoS level of 2, the need for reliable transmission requires the transmission of many acknowledgment frames, which causes network congestion and increases the network's burden. Therefore, with a very low delay, the packet loss rate gradually increases. As the delay

increases, it indirectly alleviates network pressure, causing the packet loss rate to gradually decrease to zero.

2. However, as the number of publishers increases to 5 with both Qos levels set to 2, the message loss rate decrease to 0. It is abnormal, which has following reasons:
   When the broker or client realized that the message loss rate was too high, which had reached the value of 12% in **Fig 26** in the case of 4 publishers. The device or software might adjust the resource allocation strategy. For example, the computer might give this task a higher priority and the broker might increase buffer size to store the incoming messages.

Out of Order rate

The out of order rate of all my samples is 0, which is because of my good coding habits and proper use of data structures. I have carefully illustrated the details in **Part 2**, i.e. the Code Part.
In a multithreaded environment, if the sending and processing of messages are not properly synchronized, it is easy to generate race conditions and cause out-of-order messages.

1. In a multithreaded program, using the synchronized keyword or locks ensures that a publisher will only send messages after the control information has been updated thus avoiding data confusion. Additionally, when the analyzer receives data, it also uses locks to ensure that the data is written into the data structures in an orderly manner, preventing confusion in shared data due to race conditions. Moreover, the analyzer updates the control parameters only after it has processed all the messages received during the current period. From my perspective, it is likely why the out-of-order rate in my system is very low.

2. Appropriate data structure:
   As mentioned in Code part, a 'ConcurrentHashMap', which is thread-safe, is used to store data. It can prevent that when different messages arrives, the stored data gets messed.

Median-Inter Message Gap

1. Firstly, in these five graphs, their median-inter message gaps are all greater than or equal to the delay time, which is reasonable. Because each message has to wait for a time of delay before it can be sent.

2. There are some gaps that are greater than the delay. It might have following reasons:
   The level of QoS:
   It can be seen that for messages with higher QoS level, the message gap might become higher. Higher QoS levels need more steps about acknowledgements. In this case, it might result in additional network traffic and processing time. This might increase the gap betweem messages.
   Thread scheduling and synchronization overhead:
   The actual execution order and timing of threads may be affected by thread scheduling policies and synchronizations such as locks. As shown in **Fig 28**, the data arrived time was recorded as the system time when the code started to store it. It is possible that the data arrived on time, but because of the synchronization mechanism it was blocked outside the lock, its arrival time might be recorded a little late when it was saved.

```
private void handleCounterMessage(String[] topicParts, String payload, int currentSubQos) {
    // Check if the topic has the correct format and expected number of parts.
    // Typically, this would be in the format "counter/instanceId/qos/delay".
    if (topicParts.length != 4) {
        // If the topic does not have the expected segments, do nothing.
        return;
    }
    // Create a unique key from the topic parts to use for storing and retrieving data
    String key = String.join( delimiter: "/", Arrays.copyOf(topicParts,  newLength: 4));
    // Ensure there is an entry for this key in the data map. If not, initialize it.
    dataMap.putIfAbsent(key, new InstanceData());
    InstanceData data = dataMap.get(key);
    synchronized (data) {
        data.messagesByQos.get(currentSubQos).add(new Message(Integer.parseInt(payload), System.currentTimeMillis()));
    }
}
```

Figure 28: Possible Reason for Causing Message Gap higher

Different QoS Combinations

Based on the previous discussion, let's now summarize and discuss the impact of different QoS combinations on data.

1. Due to the acknowledgement mechanism of QoS 1 and QoS 2, as the QoS level goes higher, it can be clearly seen that the total message rate becomes lower.

2. Also, this mechanism will make the networks crowded to increase the message loss rate.

3. From my data, the out-of-order rate and median-inter message gap cannot reflect the change of the combination of QoS levels. I prefer to attribute the results and trends of these two data to my code, which is the software part.

4 SYS information

Because the $SYS topic was continuously updating during the process, so the data recorded about this topic was kind of chaotic. However, we can still extract some useful data to do discussion.

Received and Sent Data

```
System metric [$SYS/broker/bytes/received]: 47143925816
System metric [$SYS/broker/bytes/received]: 47143925816
System metric [$SYS/broker/bytes/received]: 47144817706
System metric [$SYS/broker/bytes/received]: 47152146347
System metric [$SYS/broker/bytes/received]: 47159729792
System metric [$SYS/broker/bytes/received]: 47167288710
System metric [$SYS/broker/bytes/received]: 47175141126
System metric [$SYS/broker/bytes/received]: 47183001920
System metric [$SYS/broker/bytes/received]: 47185065118
System metric [$SYS/broker/load/publish/sent/1min]: 344994.33
System metric [$SYS/broker/load/publish/sent/1min]: 344994.33
System metric [$SYS/broker/load/publish/sent/1min]: 362269.46
System metric [$SYS/broker/load/publish/sent/1min]: 888551.61
System metric [$SYS/broker/load/publish/sent/1min]: 1342234.83
System metric [$SYS/broker/load/publish/sent/1min]: 1716405.98
System metric [$SYS/broker/load/publish/sent/1min]: 2026790.74
System metric [$SYS/broker/load/publish/sent/1min]: 2285820.89
System metric [$SYS/broker/load/publish/sent/1min]: 2060027.53
Current number of publishers: 1
Current Publisher QoS: 0
Current Analyzer QoS: 0
Current Delay: 0
```

Figure 29: Screenshot of SYS When 1 publisher

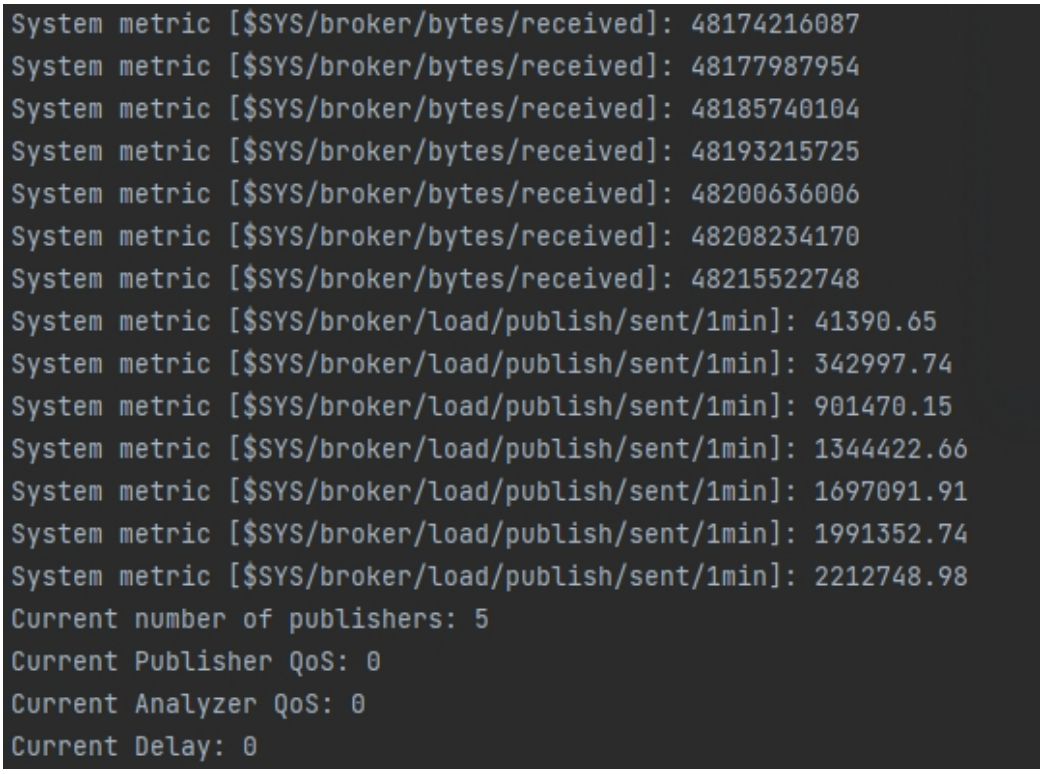

Figure 30: Screenshot of SYS When 5 publishers

The received data amount and the sent data amount are approximately the same. In my opinion, this is like one person continuously doing the job the whole time compared with 5 persons taking turns to do the job. The total received data amount would be approximately same. Since the received data amount is approximately same, the sent data of the broker would be the approximately same. The reason why the broker sent amount was lower at firstly in **Fig 30** might be due to initial blocking. Since MQTT is based on TCP, it might have employed the slow start algorithm, thereby sending less data initially.

Connection and Socket

Fig 31 shows the comparison about the the number of connections between 1 publisher and 5 publishers. **Fig 32** shows the comparison about the number of sockets between 1 publisher and 5 publishers.

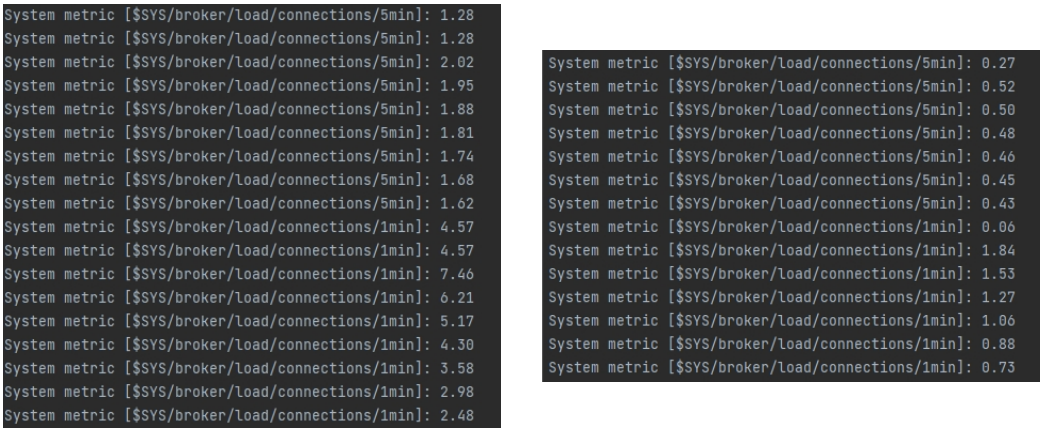

Figure 31: Connection Value Comparison

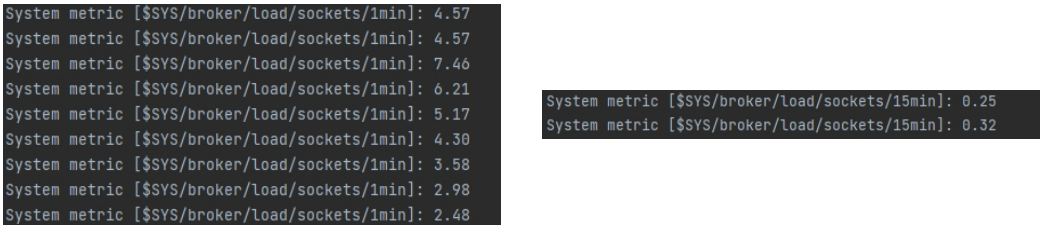

Figure 32: Sockets Value Comparison

Intuitively, it is expected that the number of connections and sockets to increase with the number of publishers. However, counter-intuitively, the data shows that as the number of publishers increase, these values decrease.
It might be due to these reasons:

1. With more publishers, the data traffic can be more balanced distributed across connections, potentially leading to a more efficient handling of connections and data.

2. With only one publisher, the connection might be more crowded. Therefore, the broker might need to start new connections or sockets to reduce stress. However, it is possible that the broker might be better at handling 5 less crowded connection. Therefore, the connections and sockets of higher number of publishers might be less.

3. Resource Reallocation strategy: with only one publisher, the broker might only allocate the buffer zone with smaller size. Therefore, the higher number of connections and sockets is needed. However, as the publisher number goes up, the resource reallocation strategy might be triggered to force the broker to allocate a larger buffer zone for this task.

5 Question 3

a) Performance Challenges:

1. Network congestion and delay:
   As the number of messages goes up, the data might meet the situation that message loss or delay. This is likely to happen in congested network nodes, such as congested routers or switches.

2. Resource constraints:
   The resource of sensors is not infinite. Since there are thousands of publishers, it means that the sensors need to publish data to different brokers. The configuration of the sensor limits its performance. For example, in this assignment, because my computer's CPU has 16 cores, it can easily handle the task of simultaneously launching five publishing tasks. However, in reality, the CPU of a sensor normally cannot be that good to fit the demand of sending data to many brokers at the same time. Moreover, the configuration of routers and switches will limit the performance as well.

3. Different QoS levels:
   Select the level of QoS is like a trade off between reliability and performance. For a faster speed and larger capacity, of course we want to set the QoS level to 0. However, different applications may have different requirements for real-time performance. For example, when using chatting software, reliability is more important than speed. In high-volume environments, meeting the quality of service (QoS) requirements of all subscribers may put additional pressure on system resources.

4. Extensibility problem:
   As the number of sensors and subscribers increases, MQTT systems need to scale efficiently to handle more messages and connections. To achieve this point, MQTT system needs to optimize data routing policies to find a

shorter path, or implementing better load balancing mechanisms to avoid congestion.

b) Different QoS levels:

1. QoS 0:
   It is aimed at 'At Most Once', which means it would not provide reliable service. In this case, it can reduce communication cost, which is suitable in scenarios where network conditions are good and message loss is tolerable. However, QoS 0 does not provide any recovery mechanism when it comes to message loss. In reality, when the network scale to a large amount, message loss becomes inevitable and intolerable. QoS 0 is no longer applicable.

2. QoS 1:
   It is aimed at 'At Least Once'. Therefore, there would be potential duplicate messages. It is suitable in scenarios where message arrival needs to be acknowledged and duplicate messages are tolerant. However, it will increase the stress of the receiver, because extra resources are needed to handle duplicate messages.
   Compared to QoS 0, it would be more reliable. And it would be more efficient than QoS 2. Therefore, in reality, common MQTT communication without specific requirements often set the QoS level to 1.

3. QoS 2:
   It aimed at 'Exactly Once'. Therefore, it would cost most among these 3 levels. The four-step handshake process ensures that each message is received only once. It ensures data integrity and avoids duplication. This can be applied to software with high reliability requirements, such as chatting software. However, MQTT is commonly used in IoT and famous for its simplicity and concision. Therefore, in my opinion, in reality, common sensors would not be so powerful to provide this expensive service and it is also unnecessary at most time.

6 Bonus Question

6.1

The principle of this is similar to the synchronization block introduced in **Part 2**. The code is shown in **Fig 33** and **Fig 34**.

The code part in **Fig 33** is in the callback function which will be called when the subscribed topics update. The use of ‘lock‘ ensures that the publisher will not do any publication work until two parameter, which are delay and QoS, are updated. ‘updateCount‘ is used to count the number of updated parameters. Only when the analyzer finishes the data processing work and updates two parameters, the ‘lock‘ can be unlocked. After it is unlocked, it will notify all other threads blocked outside the ‘lock‘ that they are allowed to enter the ‘publishMessages‘ function to do publication work like shown in **Fig 34**.

```
    } else if (topic.startsWith("request/")) {
        String key = topic.split( regex: "/")[1];
        sharedSettings.put(key, new String(message.getPayload()));
          System.out.println("Updated shared settings with " + key +

        synchronized (lock) {
            updateCount++;
            if (updateCount == expectedUpdates) {
                settingsUpdated = true;
                // Notify that all settings have been updated
                lock.notifyAll();
                // Reset the update count for next updates
                updateCount = 0;
            }
        }

    }
}
```

Figure 33: Callback Function in Publisher

```
while (true) {
    synchronized (lock) {
        while (!settingsUpdated) {
            // Use while loop to avoid spurious wakeup
            // Wait until all settings are updated
            lock.wait();
        }
        // Reset flag after settings are confirmed to be updated
        settingsUpdated = false;
    }
    String delay = sharedSettings.getOrDefault( key: "delay",  defaultValue: "1000");
    String qos = sharedSettings.getOrDefault( key: "qos",  defaultValue: "0");
      System.out.println("Ready to publish with delay: " + delay + ", qos: " + qos);
    publishMessages(client, Integer.parseInt(delay), Integer.parseInt(qos), instanceId);
```

Figure 34: Synchronization Block Used in Publisher

6.2

When trying to start 10 publishers at the same time, the computer becomes extraordinary slow. And after I launched the 'Analyzer' class, it showed the errors like in **Fig 35**.

```
已断开连接 (32109) - java.net.SocketException: Connection reset
    at org.eclipse.paho.client.mqttv3.internal.CommsReceiver.run(CommsReceiver.java:197)
    at java.base/java.lang.Thread.run(Thread.java:833)
Caused by: java.net.SocketException Create breakpoint : Connection reset
    at java.base/sun.nio.ch.NioSocketImpl.implRead(NioSocketImpl.java:323)
    at java.base/sun.nio.ch.NioSocketImpl.read(NioSocketImpl.java:350)
    at java.base/sun.nio.ch.NioSocketImpl$1.read(NioSocketImpl.java:803)
    at java.base/java.net.Socket$SocketInputStream.read(Socket.java:966)
    at java.base/java.net.Socket$SocketInputStream.read(Socket.java:961)
    at java.base/java.io.DataInputStream.readByte(DataInputStream.java:271)
    at org.eclipse.paho.client.mqttv3.internal.wire.MqttInputStream.readMqttWireMessage(MqttInputStream.java:92)
    at org.eclipse.paho.client.mqttv3.internal.CommsReceiver.run(CommsReceiver.java:137)
    ... 1 more
```

Figure 35: Screenshot of the errors

Reasons why this error happened are mainly two:

1. The network connection is not stable
   If the connection is not stable, it would probably be reset.

2. Concurrent connection problem:
   If too many clients attempt to connect to the server simultaneously, the server may reset connections because it cannot handle the load.

Regardless of the reason, it reflects that as the number of publishers increases, the network load also increases. The challenges and difficulties in properly handling data also become greater.

7 How to run my code

1. Make sure ‘Maven‘ is installed.
2. Open the project in IntelliJ.
3. Right click the ‘pom.xml‘.
4. Click ‘Download Sources and Documentation‘ and after that click ‘Reload project‘.
5. Then you can click the ‘run‘ button at top right corner to start my code.

The Publisher.java and Analyser.java in ‘java‘ file are codes for implementing the 180 tests in the question. You should first execute Publisher.java and then execute Analyser.java.

The codes in ‘Bonus‘ file are for attempting bonus questions and codes in ‘WireShark‘ file are for wireshark analysis. There is one single python file used for drawing data.

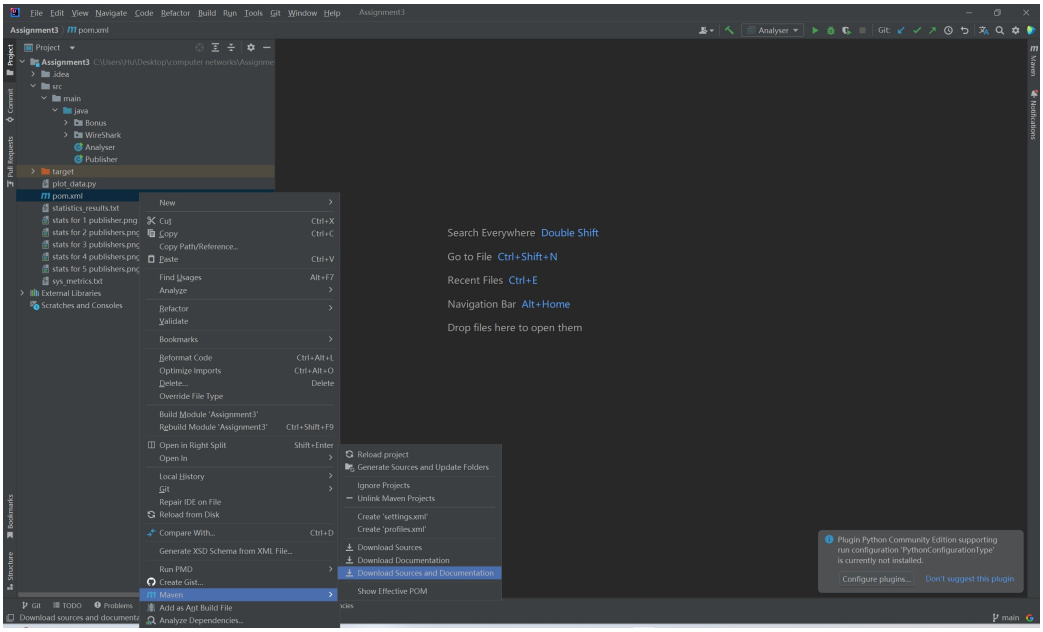

Figure 36: How to run the code