A Comparison of IoT Communication Frameworks: APIs and Performances

Jianbin Lai

A Thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of

Master of Computer Science (Computer Science) at

Concordia University

Montréal, Québec, Canada

February 2024

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Jianbin Lai**

Entitled: **A Comparison of IoT Communication Frameworks: APIs and Performances**

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science (Computer Science)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the Final Examining Committee:

_____ Chair
Dr. Joey Paquet

_____ Examiner
Dr. Essam Mansour

_____ Examiner
Dr. Tse-Hsun (Peter) Chen

_____ Supervisor
Dr. Yann-Gaël Guéhéneuc

_____ Co-supervisor
Dr. Sandra Céspedes

Approved by _____
Joey Paquet, Chair
Department of Computer Science and Software Engineering

_____ 2024

_____
Mourad Debbabi, Dean
Faculty of Engineering and Computer Science

Abstract

A Comparison of IoT Communication Frameworks: APIs and Performances

Jianbin Lai

The Internet of Things (IoT) and the number of deployed IoT devices are growing exponentially nowadays. These devices play a pivotal role in diverse domains, including smart homes and connected health, undertaking vital functions like monitoring indoor pollutants. Given their constrained processing and memory capacities, IoT devices engage in communication with each other, as well as with edge devices and cloud servers, through specialized protocols.

The two main protocols for communications among IoT devices are the Constrained Application Protocol (CoAP) and Message Queuing Telemetry Transport (MQTT). Many organisations have implemented these protocols in different programming languages, different libraries/frameworks, as closed or open source. By March 1, 2023, there were 35 public libraries of CoAP and 40 of MQTT. These libraries have different characteristics, like levels of completeness and runtime performances.

In an era marked by the rapid expansion of demands across diverse IoT domains, the need for IoT applications increases correspondingly. Because of diverse and complex user requirements in different domains, it is impractical to assume that all software applications uniformly adhere to the same protocol. Consequently, developers tasked with programming software for IoT devices encounter the challenge of selecting the suitable protocol, such as MQTT or CoAP, and determining its libraries (e.g., `Californium`, `java-coap`, `Paho MQTT`, or `HiveMQ MQTT Client`). To the best of our knowledge, comprehensive and clear comparisons of the API and performance of these existing protocols and their implementations are lacking, leaving developers without the necessary insights to make well-informed choices.

In this thesis, we implement multiple IoT scenarios by employing the CoAP and MQTT protocols along with various implementations, subsequently subjecting them to a comprehensive comparative analysis in terms of API and performance metrics encompassing both static metrics, packet size, and runtime metrics evaluation. Our findings reveal that among the four libraries, `HiveMQ MQTT Client` demonstrates superior performance in static metrics, while `java-coap` exhibits the smallest packet size. Drawing upon our examination of the two protocols and their respective implementations, we conclude that CoAP and its `java-coap` implementation is faster than MQTT.

We thus provide developers software for IoT devices with evidence of the CoAP and MQTT protocols and their libraries. Moving forward, our aspiration is to broaden the scope of this research to encompass additional communication protocols and IoT frameworks, while also exploring diverse software characteristics and metrics.

Acknowledgments

This research work is funded by the Natural Science and Engineering Research Council (NSERC) and Concordia University. My sincere thank you to all academic personnel who made this research possible.

I want to give a special thank you to my supervisors, Professor Yann-Gaël Guéhéneuc, Sandra Céspedes, Maxime Lamothe, Weiyi Shang for the guidance, support, and advice they have provided throughout my time as a student.

A warm thank you to those who helped with the surveys and interviews that validate this research.I would also like to thank my colleagues in the Ptidej lab for their friendship, guidance, and weekly discussion that sparkled new ideas and questions.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In 2023, Statista estimated that 15.14 billion Internet of Things (IoT) devices existed worldwide, to become 29.4 billion in 2030 [2]. These IoT devices find applications across many domains, performing a range of critical tasks such as patient monitoring, forest fire control, and search and rescue missions. As they have become prevalent in our daily lives, there is a growing demand for developers to create and implement software systems for these IoT devices. These IoT applications run on devices or cloud servers communicating via different protocols. The two prominent protocols for IoT communication are the Constrained Application Protocol (CoAP) [3] and Message Queuing Telemetry Transport (MQTT) [4], which are designed for IoT devices with limited capabilities [5], such as limited battery and processing resources.

CoAP and MQTT are two standards, RFC7252 [3] and ISO/IEC 20922[1], that have been implemented by different organisations in different programming languages, both as closed and open source. For example, the Eclipse Foundation[2] provides `Californium` and `Paho MQTT` open-source libraries in Java of CoAP and MQTT, respectively. The IBM Watson IoT Platform Message Gateway[3] provides an implementation of MQTT.

By Mar. 1st, 2023, there were 35 reported libraries of CoAP[4] and 40 of MQTT[5], with different

[1] https://www.iso.org/standard/69466.html

[2] https://newsroom.eclipse.org/news/announcements/eclipse-iot-working-group-celebrates-its-10th-anniversary

[3] https://www.ibm.com/docs/en/wip-mg/5.0.0.1

[4] https://en.wikipedia.org/wiki/Constrained_Application_Protocol#Implementations, as of March 1st, 2023

[5] https://en.wikipedia.org/wiki/Comparison_of_MQTT_implementations, as of March 1st,

characteristics, like levels of completeness and runtime performances. For example, some libraries are open-source/commercial libraries, while some are designed with/without security solutions. As a result, developers need to select the appropriate protocol and the library (e.g., `Californium`[6], `java-coap`[7], `Paho MQTT`[8], or `HiveMQ MQTT Client`[9]). To the best of our knowledge, they do not have clear and thorough comparisons of the API and performance of protocols and their libraries to facilitate decision-making. Although some previous works compare both protocols' performance under certain network conditions [5, 6, 7], to the best of our knowledge, no previous work compared several libraries of the two protocols in terms of their APIs and static metrics, packet size metrics and runtime metrics.

In this thesis, we evaluate the protocols and libraries by devising different scenarios. To be specific, we only consider open-source libraries in Java to ease replicability. We set up an experimental infrastructure using Raspberry Pis, which helps us collect both static code metrics and count packet size of libraries and runtime metrics when running in different scenarios. Please note that our implementations utilize MQTT Version 5.0 [8], representing an upgraded version of MQTT. Based on our experiment, the results show that `HiveMQ MQTT Client` has the best performance on static metrics among the four libraries. Besides, `java-coap` uses the least packet size by our program demonstrations and shows the best performance on speed. Moreover, CoAP is faster than MQTT. Conclusively, we draw comprehensive conclusions and recommend the adoption of `HiveMQ MQTT Client` due to its advantageous features in terms of simplified coding, testing, maintenance, and network configuration.

The remainder of this thesis is organised as follows: Chapter 2 provides background information. Chapter 3 summarises related work. Chapter 4 describes our approach. Chapter 5 presents our implementation and results. Chapter 6 discusses our results and threats to their validity. Chapter 7 concludes with future work.

2023

[6] https://github.com/eclipse-californium/californium

[7] https://github.com/PelionIoT/java-coap

[8] https://github.com/eclipse/paho.mqtt.java

[9] https://github.com/hivemq/hivemq-mqtt-client

Chapter 2

Background

In an IoT system, the IoT devices serve as pivotal endpoints. Notably, the broker (MQTT) we used in this experiment also plays a role as the endpoint. Furthermore, the communication within this network is facilitated through various IoT protocols, encompassing options such as AMQP (Advanced Message Queuing Protocol), Bluetooth, Bluetooth Low Energy (BLE), CoAP, MQTT, Zigbee, etc.

The Open Systems Interconnection model (OSI model) divides any communication channel into seven layers: physical, data link, network, transport, session, presentation layer, and application [9]. The different protocols are designed and can be classified into specific layers. In our experiment, we search IoT protocols which are based on the application layer, the top layer of the seven-layer model, so the protocols are closest to the developers. It does not require developers to have a lot of knowledge about the other six layers, so it is relatively easy for developers to learn and start programming. As Figure 2.1 is shown, when two endpoints exchange data with a protocol at the application layer, data would be passed down to the lower layer at the sender side, and the receiver side will get the data and be passed up layer by layer to the higher layer till the application layer. It also means that protocols at the application layer could be based on different protocols at the lower layer, such as the transport layer.

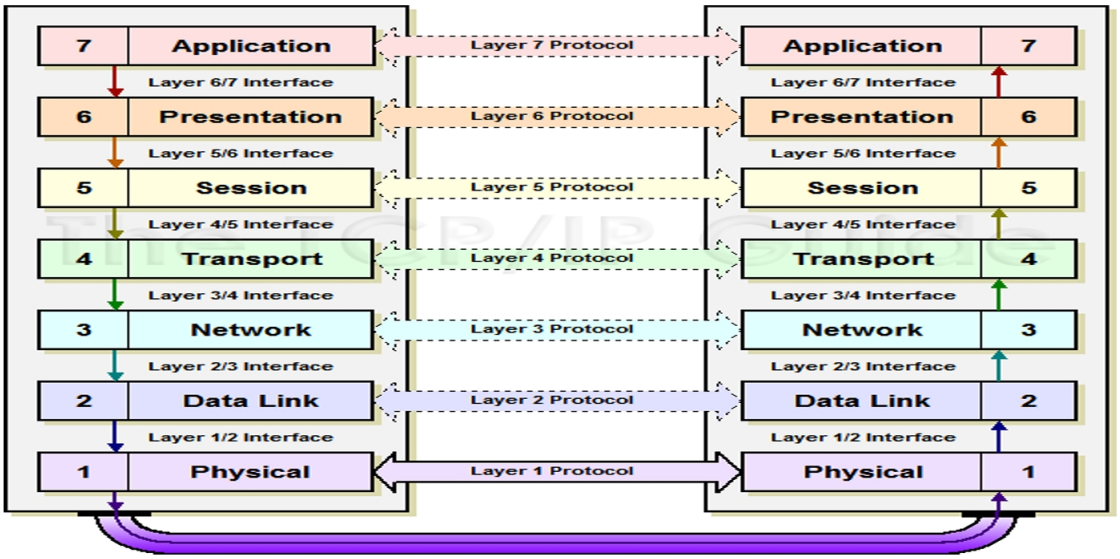

Figure 2.1: Diagram for seven layer model taken from [1]

Diverse institutions may allocate different protocols to distinct layers of the OSI model, we aim to identify and prioritize protocols consistently designated as application layer protocols. Then We made sets of the IoT protocols mentioned in [10], [11] and [12], and then we made an intersection of the sets. At the intersection, three protocols—AMQP, CoAP, and MQTT—are concurrently classified at the application layer.

However, previous works [5, 13] showed that while AMQP offers more aspects of security, it consumes more power than CoAP and MQTT, and it is not as lightweight as CoAP and MQTT, so AMQP cannot be used for battery-powered IoT devices. Previous works [7] also studied HTTP (they did not consider HTTPs) and reported that HTTP requires a large bandwidth because of the sizes of its messages and, therefore, more processing power and memory than CoAP and MQTT. Thus, we exclude AMQP and HTTP in this study and we focus on CoAP and MQTT.

For the sake of simplicity, we use the same terms of ‘sender’ and ‘receiver’ to show and emphasize the direction of the valuable data flow, even though in our experiment, for example, the client (receiver) can send the ACK packet to the server (sender), we still name the client as the receiver, and the server as senders. Specifically, when considering CoAP, the term ‘sender’ means the server, and the ‘receiver’ means the client. Besides, when considering MQTT, the term ‘sender’ means the publisher when the publisher sends a message to the broker, or it means the broker when the

broker sends a message to a subscriber. Conversely, the term 'receiver' means the broker when the publisher sends a message to the broker, or the subscriber when the broker sends a message to the subscriber. Though the client/broker/publisher can send the packet (ACK, PUBACK packet, etc.) to the server/publisher/broker, in this or analogous context, we will explicitly delineate their roles (server, client, publisher, broker, subscriber) instead of the sender or receiver.

2.1 CoAP

CoAP is mainly based on User Datagram Protocol (UDP) and designed for constrained devices and constrained networks [3]. UDP is generally considered a transport layer protocol, which is lower than the application layer. UDP offers the communication a minimum protocol mechanism when the program sends messages[14]. That explains the reason that CoAP based on UDP is friendly to constrained devices.

CoAP primarily supports one-to-one communication between a server and a client. It facilitates a request-response messaging pattern, enabling the client to initiate a request to the server, followed by the server providing a response to the client. Besides, it also provides developers with a feature based on an observer design pattern [15] so that the client does not need to send a GET request to the server to get the state of the server each time, which is similar to the publish-subscribe messaging pattern that works in MQTT. As shown in Figure 2.2, the client sends a GET request to the server for registration. After registration, it gets notifications from the server without sending a GET request at some intervals of time. CoAP is a RESTful protocol that supports GET, PUT, POST, and DELETE methods. CoAP provides two different levels of reliability. In terms of endpoint, a CoAP endpoint is the source or destination of messages [3]. The reliability modes of communication is described as follows:

- Non-confirmable Message (NON): Endpoint can send a Non-confirmable packet with the message to the destination without requiring the destination to send back an Acknowledgment (ACK) packet after the destination receives the message.
- Confirmable Message (CON): Endpoint can send a Confirmable packet with the message to

the destination, after the destination receives the message correctly, the destination sends back an Acknowledgment (ACK) packet.

More detail about the observer design pattern of CoAP implementation is shown in Figure 2.3 and Figure 2.4. The former figure shows the process when a client observes a non-confirmable type resource, and the latter figure shows the process when a client observes a confirmable type resource.

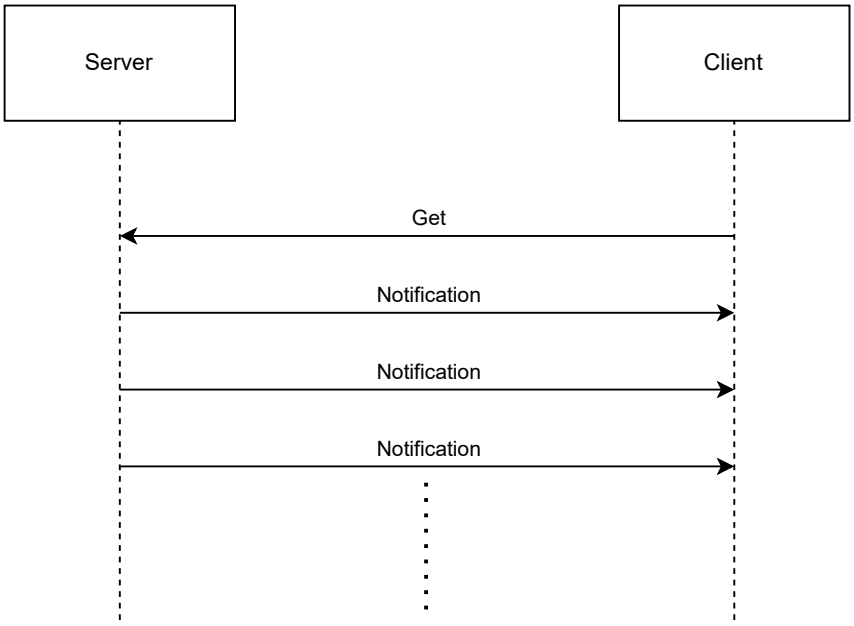

Figure 2.2: Diagram for CoAP Communication

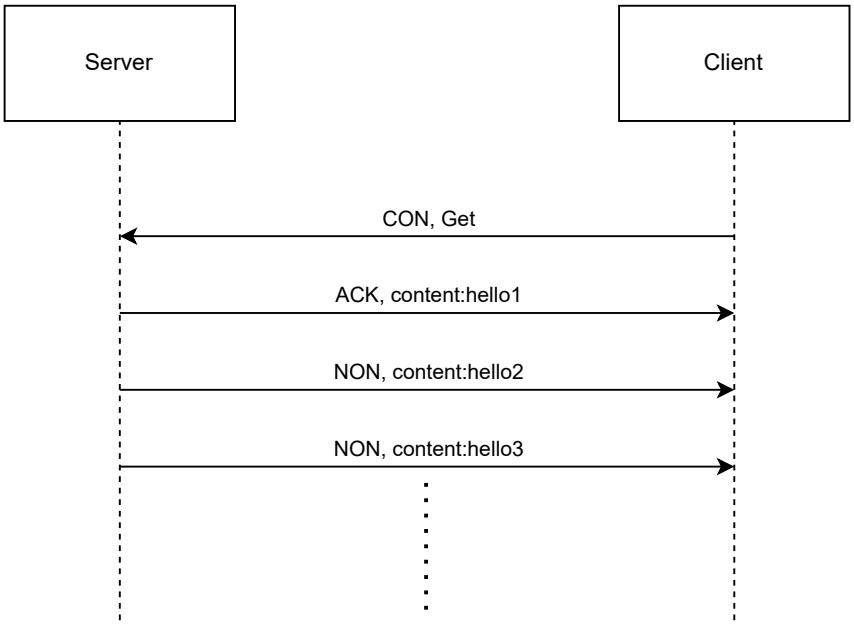

Figure 2.3: Diagram for CoAP Communication (NON)

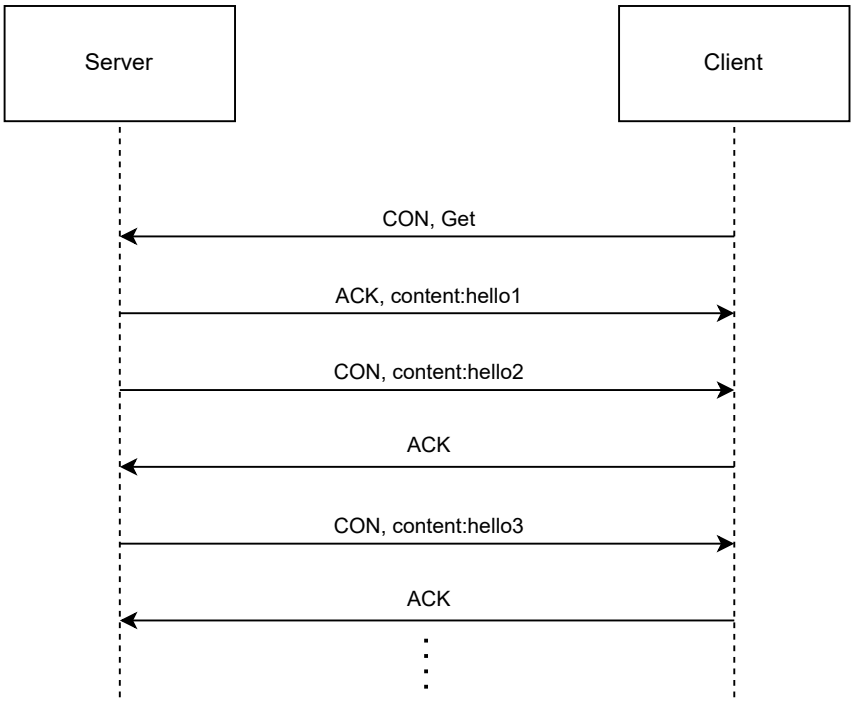

Figure 2.4: Diagram for CoAP Communication (CON)

2.2 MQTT

MQTT runs on top of TCP [8], but the MQTT-SN protocol [16] provides MQTT based on UDP. Transmission Control Protocol(TCP) is a transport layer protocol, the same as UDP. MQTT follows a publish-subscribe messaging pattern [8]. There are three actors in the communication: publisher, broker, and subscriber. The publisher publishes the topic with related content to the broker, and the subscriber can connect with the broker and subscribe to the topic.

Figure 2.5 shows how MQTT works. There are three publishers, one broker and three subscribers. **Subscriber1** and **Subscriber2** both subscribe to *TopicA*, and **Subscriber3** subscribes to *TopicB*, but no subscriber sends a SUBSCRIBE packet to the broker to subscribe to *TopicC*. Therefore no subscriber will receive updates for *TopicC*.

MQTT provides three levels of Quality of Service (QoS) :

- QoS0: When a publisher sends a message about a topic to the broker or the broker forwards that message to a receiver that does not acknowledge the server, the sender would not store the topic and do retransmission. Figure 2.6 shows the case of the communication from publisher to broker with QoS0.

- QoS1: The receiver would acknowledge the server with a PUBACK packet, and the topic would not be deleted until the sender receives the acknowledgement. The same message might be sent at least one time because if the sender does not receive the PUBACK packet, the sender will resend the message to the receiver. Figure 2.7 shows the case of the communication from publisher to broker with QoS1.

- QoS2: This ensures that the receiver only receives the message exactly once. The receiver sends back the PUBREC packet after receiving the message from the sender, then the sender sends the PUBREL packet to the receiver, and after that, the receiver sends the PUBCOMP packet to the sender so that this process can guarantee that the receiver gets each message one and only one time from the sender. Figure 2.8 shows the case of the communication from publisher to broker with QoS2.

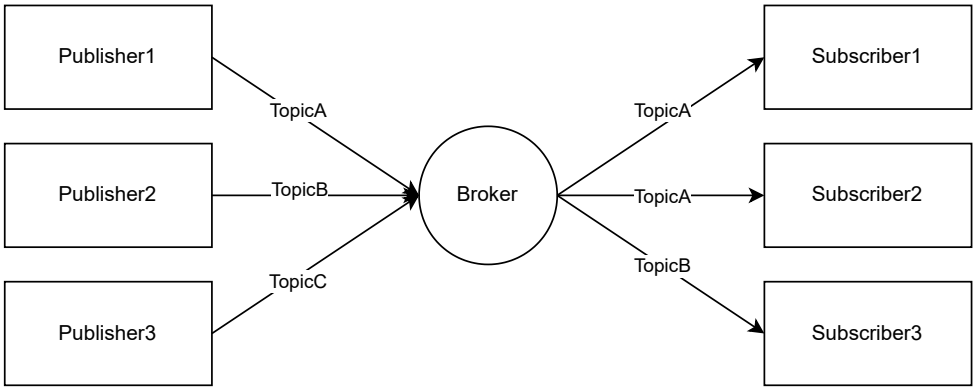

Figure 2.5: Class diagram for MQTT Communication

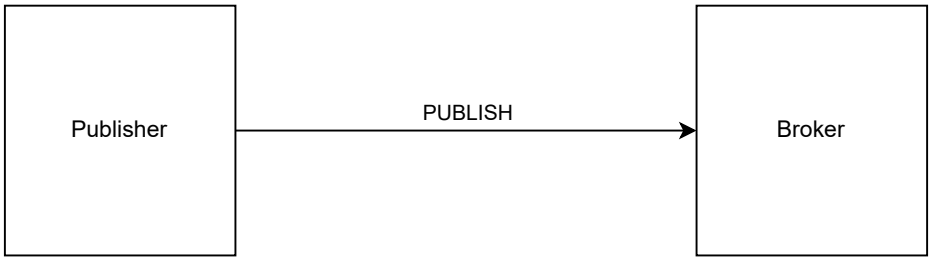

Figure 2.6: Class diagram for MQTT Communication with QoS0

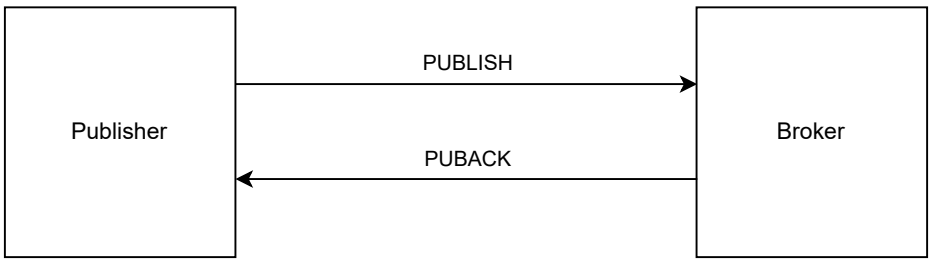

Figure 2.7: Class diagram for MQTT Communication with QoS1

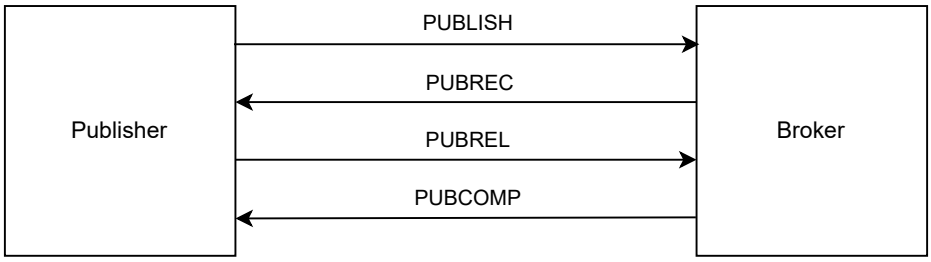

Figure 2.8: Class diagram for MQTT Communication with QoS2

2.3 Comparison and Discussions of CoAP and MQTT characteristics

After introducing CoAP and MQTT, the comparison of characteristics between them is shown in Table 2.1. It is necessary to understand the comparison before doing the tests because we need to choose the same or similar characteristics to do the test. There are some details of the table as follows:

- 'Underlying Protocol': According to the specification from [3] and [17], CoAP could be used over UDP or TCP. Even though MQTT relies on TCP [8]. Besides, MQTT-SN runs on UDP.
- 'Security': In our thesis, we chose to apply security solutions to our experiment, and we found several security solutions. In the security layer, Datagram Transport Layer Security (DTLS) provides security over UDP communication, and Transport Layer Security (TLS) offers security over TCP. In addition, there is another security solution called Object Security for Constrained RESTful Environments (OSCORE), which protects CoAP at the application layer [18].
- 'Messaging exchanging mode': Table 2.1 shows a comparison of them. The messaging exchange model in CoAP could be asynchronous and synchronous. However, MQTT and MQTT-SN are asynchronous because their publish-subscribe communication model contains the broker to receive the topic from publishers and to notify subscribers.

	CoAP	MQTT	MQTT-SN
Underlying Protocol	UDP, TCP	TCP	UDP
Reliability	NON,CON	QoS0, QoS1, QoS2	QOS-1, QoS0, QoS1, QoS2, QoS3
Security	DTLS(UDP), OSCORE, TLS(TCP)	TLS	DTLS
Messaging exchange Mode	Asynchronous, Synchronous	Asynchronous	Asynchronous
Messaging pattern	Request-response pattern Observer pattern	Publish-subscribe pattern	Publish-subscribe pattern

Table 2.1: Comparison among CoAP, MQTT and MQTT-SN

Chapter 3

Related Work

Several prior studies have addressed individual comparisons among APIs, CoAP, MQTT, and even the specific contrast between CoAP and MQTT. However, there is a dearth of existing research that comprehensively compares multiple libraries of various protocols (CoAP and MQTT) within the context of IoT with the API focus.

3.1 Comparisons of APIs

Brito et al. proposed an approach to compare two versions of the same API, called APIDiff [19]. With APIDiff, they assessed the stability of a given API across its different versions to notify API users of possible breaking and non-breaking changes. Different from [19], we do not use APIDiff to compare different versions of the same API. Instead, we assess the APIs in the four libraries' implementations of CoAP and MQTT with static metrics and runtime metrics and the consequence to the packet size during the transmission.

Jang et al. extracted dynamic birthmarks to detect software similarity [20]. Birthmarks are derived based on the frequency of API calls. Developers can use birthmarks by source code and binary files to statistically analyze the similarity between programs, but they chose the dynamic birthmarks scheme because it can include both programs' static and dynamic data. Besides, the scheme can provide more information data than the static birthmark scheme by using more techniques, such as logging of API calls. The results in [20] demonstrate that their proposed scheme can facilitate the

assessment of similarity when evaluating various FTP and SSH client programs. In this thesis, we do not use birthmarks because we do not combine source code and binary files together to compare APIs. However, we focus on comparing the code at the source code level, excluding binary files.

3.2 About CoAP and MQTT

In [21], authors mentioned that CoAP is the standard proposed by the Internet Engineering Task Force (IETF) at the application layer of IoT communication. Secure CoAP needs DTLS to protect the data that is sent. However, DTLS is designed with the assumption of being deployed on devices of comparable power. To decrease the consumption of power, the authors present an integration of DTLS and CoAP, which is called Lithe, which uses the DTLS compression solution. They evaluated their implementation on the Contiki operating system and showed improvements in packet size, energy consumption, processing time, and network-wide response times. In our thesis, we do not use the DTLS compression solution because we compared the cases with DTLS with other cases with TLS, so we keep the simple implementation without the compression solution.

Kovatsch et al. mentioned that it is expected that more IoT devices will be used [22]. As for application technology, the requirement of supporting the increasing number of devices becomes important. IoT devices would send data of their state and exchange control orders often. Cloud services need to be scalable so that they can support future requirements about a large number of connecting IoT devices. In [22], the authors present an architecture with three stages: network stage, protocol stage, and business logic stage. This architecture is aimed at constrained devices. They implemented the architecture and evaluated this architecture by increasing the concurrency of clients. To be specific, they stressed the server for a specific time, and then cooled down the machine to continue the next step to test. They found that throughput in their Californium CoAP framework is higher than high-performance HTTP Web servers. In our thesis, although we do not present a system architecture of CoAP and evaluate it, we are inspired by this work to put the throughput metrics in our experiment to evaluate the CoAP and MQTT runtime performance.

Yassein et al. mentioned that there is a large portion of IoT applications protocols using TCP or UDP to communicate [23]. Those different IoT communication protocols have different features

corresponding to different requirements. The work discussed the MQTT architecture, message format, QoS, etc. The authors considered that the scope of MQTT is designed to be applied in restricted memory and limited energy-consuming devices. From this work, we are inspired to use QoS in our scenarios. Moreover, we also compare the MQTT implementations at the coding level.

In [24], the authors mentioned that IoT offers the possibility to receive sensor data from IoT devices. In the smart city and medical domains, high-quality data is required for monitoring the devices in real-time conditions. They use MQTT as the target IoT protocol in the experiment. In this experiment, they used the DHT11 temperature and humidity sensor as a publisher to gain the information and to publish the status information to the PC Server acting as a broker, and then an Android Smartphone as a subscriber receives the message from the broker. On the subscriber side, they added a feature to save data in a MySQL database. The authors compared the HTTP and MQTT protocols and concluded that the speed of exchanging data in MQTT performs better than HTTP. In our thesis, we are inspired by the experiment, but without temperature, humidity sensor and database because we did not want the time to collect the data in the sensors or other participants to affect the experiment. We try to limit the factors as little as possible in our experiment. Besides we applied extra static metrics to our experiment.

3.3 Comparisons of CoAP and MQTT

Based on previous works, Al Enany et al. [5] collected secondary studies that compared MQTT and other protocols at the application layer. The comparative studies between MQTT and CoAP in three sets of different criteria: (1)latency and bandwidth with different sizes of messages, (2)delays and network traffic, (3)the privacy of users and the delays when sending notifications, they concluded that MQTT is highly reliable despite its large message size. Besides, MQTT has higher performance and less delay than CoAP, even in high-traffic networks, but MQTT consumes more power than CoAP. Furthermore, analysis between MQTT and AMQP were conducted across two scenarios: evaluating performance in the presence of unstable wireless networks and assessing performance under varying delay ratios. The paper concluded that MQTT consumes less power and incurs fewer delays than AMQP, but AMQP is better at security and is good at conditions where

there are high loss and high delay. In our thesis, we were inspired by one of the previous works mentioned in the paper about packet size. In addition, we made static metrics about the packet size of the implementations of CoAP and MQTT, while the previous work did not.

Stefanec and Kusek [13] studied that IoT devices have been widely deployed, and people have taken energy consumption into great consideration. In the work, they measured the energy consumption of HTTP, HTTP2, CoAP, MQTT, and AMQP protocols. They used two multimeters, one for a voltmeter and the other to measure the voltage. Besides, they recorded the data from multimeters and timestamps. The cases included the combination of protocol, packet size and different QoS. During the process, they also noted the data they needed and calculated the power usage. After that, they obtained the power consumption by multiplying the power usage with the used time. They concluded that when comparing the protocols in forms of power consumption, CoAP is the best, followed by MQTT, and the worst is AMQP. In our thesis, we did not compare the consumption between CoAP and MQTT, but this paper helped us filter the target protocols in our experiment.

Based on previous work, Naik [7] stated that understanding the advantages and disadvantages of IoT protocols is important when applying the protocols to projects. They introduced AMQP, CoAP, HTTP and MQTT, and compared different characteristics between them, such as architecture, header size, message size, standards, etc. They presented the analysis of the four protocols and concluded that CoAP has the lowest message size and overhead. HTTP uses the most power and resources, and it has the largest bandwidth and delay. MQTT provides the best performance of QoS with the lowest interoperability because the publish-subscribe pattern in MQTT could not cover all the scenarios. The support for security and provisioning in AMQP is highest among the protocols. Though the standardization of HTTP is higher, its usage is less frequent when developers apply it in IoT industry scenarios among the four protocols. In our thesis, we are inspired to take the different levels of reliability (NON, CON, QoS0, QoS1) into the scenarios of our comparison.

Thangavel et al. stated that CoAP and MQTT are suitable for resource-constrained devices in Wireless Sensor Networks (WSNs) that include sensor nodes and gateways [25]. The process of the dataflow is that the sensor nodes collect the data and send the data to a gateway, then the gateway

sends the data to the server(CoAP)/broker(MQTT), and finally, the data is sent to clients(CoAP)/-subscribers(MQTT). They use the Wide Area Network emulator (Wanem) to emulate a lossy network. In their experiment, they developed a common middleware offering CoAP and MQTT a programming interface that offers API calls to publish messages and to check if the process is successful and the message is accepted. The experiment assessed the performance metrics of delay and total data transferred in each message. They showed the experimental results in a scenario where low packet losses result in MQTT having a lower latency than CoAP. On the contrary, given high packet losses in the network, MQTT shows a higher latency than CoAP. When the message size is small, and the loss rate is low (equal to or less than 25%), the average bytes generated per second of CoAP is lower than MQTT. In their experiment, they used their middleware with common API features which could support different protocols, but in our thesis, we chose to simulate the simple and the classical architecture as illustrated in Figure 2.2 and Figure 2.5.

In [26], authors stated that, in constrained conditions, efficient communication between IoT devices is important. They performed an experiment about the communication of CoAP and MQTT, including the connection and sending messages parts. Besides, they compared the characteristics of CoAP and MQTT. They reported that the CoAP protocol supports one-to-one communication. For example, one client sends a request to one specified server, and the server directly responds to the client. In addition, CoAP also supports one-to-many or many-to-many multi-cast requirements. while in the communication of MQTT, messages are transferred in a many-to-many way because the communication requires a broker to transmit the message between many publishers and many subscribers. They suggested using CoAP when real-time performance and latency are not required and using MQTT when the case requires a lot of update messages. In our thesis, we did not consider the latency metric, but we were inspired to conduct our experiment by not only observing the sending data but also taking the connection into consideration.

Chapter 4

Approach

We now describe our approach to compare different libraries' implementations of CoAP and MQTT. We divide our approach into four steps described as follows:

(1) Survey and select different libraries of the protocols.

(2) Collect, devise, and implement scenarios using the chosen libraries.

(3) Set up an IoT infrastructure and run the scenarios.

(4) Collect static metrics on the implemented senders and receivers (Static Metrics).

(5) Analyse packet sizes theoretically and practically (Packets Metrics).

(6) Collect runtime performances when running the scenarios (Runtime Metrics).

(7) Compare the collected measurements and recommend a protocol/library.

4.1 Survey and select different libraries of the protocols: The first step of our approach consists of identifying existing libraries of the two chosen protocols, CoAP and MQTT, and selecting a set of libraries implementing these protocols. We followed these steps:

(1) Identify and search the libraries of the protocols.

(2) Filter out libraries not implemented in Java language.

(3) Filter out libraries not supporting receivers and senders.

(4) Filter out non-open-source libraries.

We selected four libraries: `Californium` and `java-coap` for CoAP, and `HiveMQ MQTT Client` and `Paho MQTT` for MQTT.

4.2 Set up an IoT infrastructure: To obtain the data necessary for our analyses (that follows), we need an infrastructure to run the programs on the scenarios and collect the data.

4.3 Collect, devise, and implement scenarios using the chosen libraries: Once we have selected the IoT infrastructure, we need programs to exercise the scenarios and collect the data. We implement these scenarios using the libraries.

4.4 Collect static metrics on the implemented senders and receivers: We use usual static code metrics to measure and compare the implementations in the different scenarios we devised in Section 5.3 of the four chosen libraries. The metrics are well-known and representative of code characteristics. We choose the following metrics for our experiment as previous work used in measuring and comparing the modularity related to Design Pattern (DP) [27] and to compare different versions of software [28]. We use SciTools Understand to collect static metrics on the code implementing our scenarios.

- Coupling Between Object (CBO): the authors in [29] mentioned that CBO measures the number of classes coupled with the target class. In our experiment, some of our demos on the sender or receiver sides might include more than one class, so we will sum the CBO of the classes. If this metric is lower, it means that it has good reusability, maintainability and changeability[10].
- Cyclomatic complexity(CC): McCabe in [30] proposed a measure called cyclomatic complexity, which refers to the complexity of the program. When the value is high, it means that it is more complex and harder to understand, test, and maintain. API ID 'CYLCOMATIC' is Cyclomatic complexity in Understand software from Sci Tools, as known as

[10] https://learn.microsoft.com/en-us/visualstudio/code-quality/code-metrics-class-coupling?view=vs-2022, as of March 1st, 2023

McCabe Cyclomatic. If this metric is lower, it means that the program is less complex, so developers can easily test and troubleshoot [11].

- Lack of Cohesion between methods (LCOM): Chidamber and Kemerer in [29] mentioned that the Lack of Cohesion in Methods(LCOM) value is affected by the number of disjoint pairs. If the LCOM value is lower, it means a high level of cohesion between the methods in a class. API ID 'Percent Lack Of Cohesion' is Lack of Cohesion in Methods(LCOM) in Understand software from Sci Tools.
- Lines of Code (LOC): This metric refers to lines of code in a computer program that is written in a programming language. It includes package import lines, but it excludes blank lines and comment lines. In our experiment, some of our demos on the sender or receiver sides might include more than one file, therefore we will sum the LOC of the files. If LOC number is too large, it may imply code defects and cost of maintenance.

4.5 Analyse packet sizes theoretically and practically: We also consider the size of packets in bytes transmitted during the communication, because different protocols and different libraries may have different sizes, which would impact the performance. Our research includes both a theoretical analysis and a practical experiment, and we also consider adjusted practical packet size for the two protocols, CoAP and MQTT.

4.6 Collect runtime performances when running the scenarios: We also use usual running performance metrics to measure and compare the implementations in the different scenarios we devised in Section 5.3 of the four chosen libraries. We choose goodput, throughput, and overhead and measure them when running the scenarios.

(a) Before computing goodput and throughput, we must measure ΔTime. We record the time when the receiver connects to the sender as $Time1$ and record the time $Time2$ when the receiver receives the last message in NON/QoS0 scenario, or when the receiver sends ACK packet of the last received message (excluding the transfer time of this ACK from receiver to the sender) in QoS1/CON scenario.

[11] https://learn.microsoft.com/en-us/visualstudio/code-quality/code-metrics-cyclomatic-complexity?view=vs-2022, as of March 1st, 2023

$$\Delta Time = Time2 - Time1 \tag{1}$$

(b) After getting the ΔTime, we can use it as the denominator in Equation 2 and Equation 3 to get the goodput and throughput. In [32] and [33], the goodput and throughput are defined as follows.

$$Goodput = TotalData\_bytes/\Delta Time \tag{2}$$

$$Throughput = (TotalBytesReceived)/\Delta Time \tag{3}$$

(c) In this step, we calculate the overhead metric. Note that this overhead measurement is not the overhead in each message packet. Instead, the overhead in this step means all the bytes minus the total valuable data in communication between sender and receiver.

$$Overhead = (ConnectBytes + InteractMessageBytes) - Data \tag{4}$$

The example of CoAP with CON scenario process is shown in Figure 4.1. The example of MQTT with QoS1 scenario process is shown in Figure 4.2. The formula to calculate the throughput and overhead is shown as follows. Specifically, in the example of MQTT, p5 and p7 are not considered because p5 data does not reach the subscriber in Equation 8.

$$CoAPCONThroughput = (p2 + p3)/\Delta Time \tag{5}$$

$$MQTTQoS1Throughput = (p10 + p14)/\Delta Time \tag{6}$$

$$CoAPCONOverhead = (p1 + p2 + p3 + p4) - Data \tag{7}$$

$$MQTTQoS1Overhead = (p1 + p2 + p3 + p4 + p6 + p8 + p9 + ... + p16) - Data \tag{8}$$

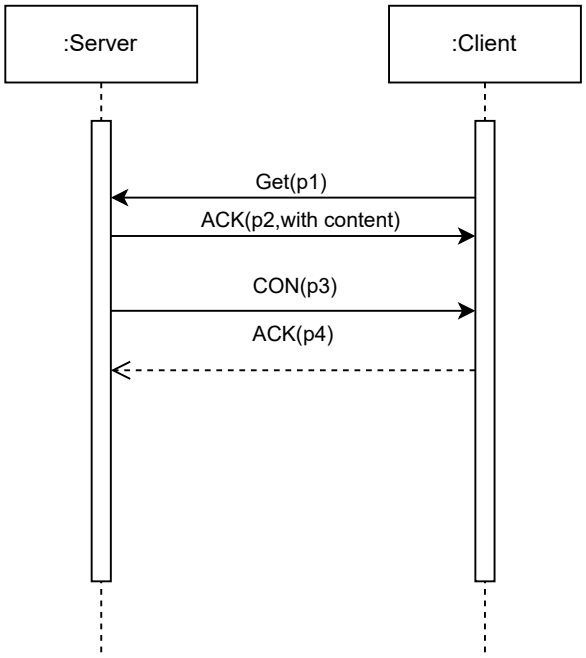

Figure 4.1: Diagram for CoAP Communication

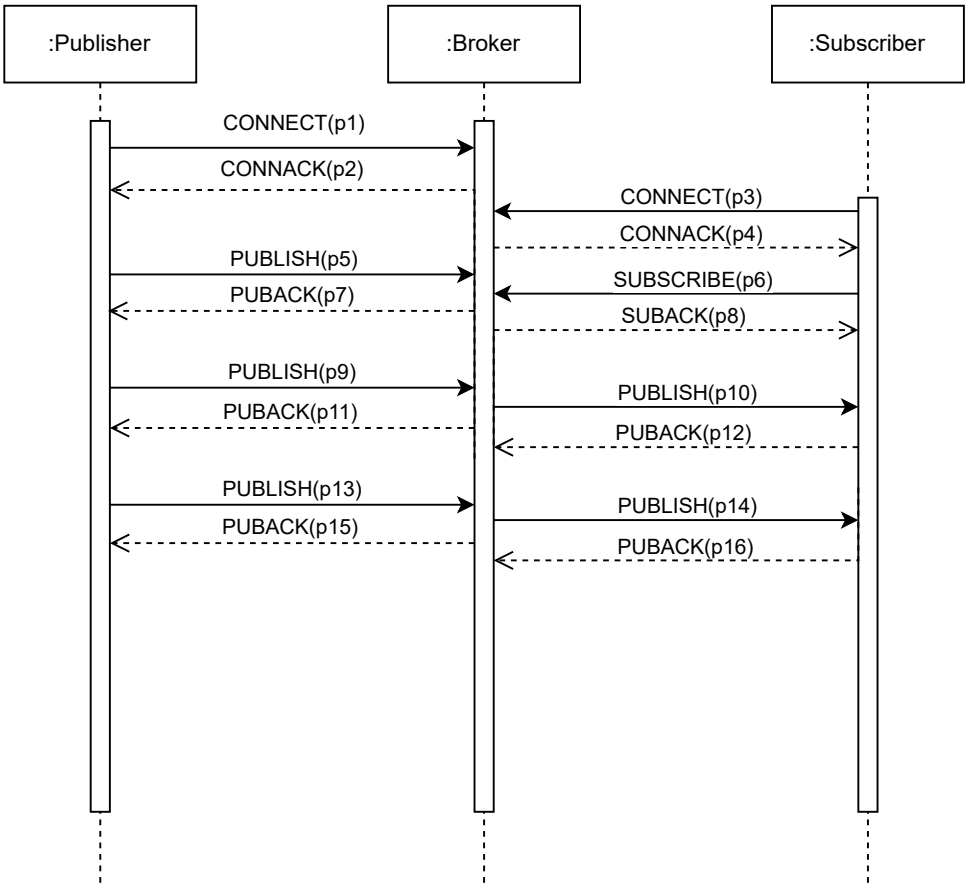

Figure 4.2: Diagram for MQTT Communication

4.7 Compare the collected measures and recommend a protocol/library: Finally, the last step of our approach is to compare protocols and libraries based on the collected measures. We will discuss the protocols and their libraries and make recommendations.

Chapter 5

Implementation, Application, and Result

5.1 Survey and select different libraries of the protocols

In this phase, we provide an account of the subsequent steps that detail the examination and filtration processes for selecting libraries in our experiment:

(1) Use Wikipedia to select the implementations of protocols. We found that Wikipedia is a reliable source of information regarding IoT protocols such as CoAP and MQTT. We peruse the Wikipedia pages about CoAP and MQTT implementations. Each of these two pages[4,5] contains a table showing a list of the libraries implementing each protocol. By March 1, 2023, there were 35 libraries of CoAP in the table and 40 libraries of MQTT.

(2) Filter Java language repository. We choose to consider only libraries written in the Java programming language for several reasons because Java is an object-oriented programming language so it is possible to compute well-known software metrics. Besides, it is a platform-independent language that provides execution flexibility. Therefore, applications can be deployed in versatile environments. Table 5.1 and Table 5.2 shows the result of this step. In these tables, the column 'Latest update date' is about the latest commit date because the libraries update frequently as up-to-date as possible. However, sometimes the latest commit date cannot be traced due to some reasons, such as it is commercial. In this case, this column shows the least released date or the data shown in Wikipedia. (Thingstream, HIVE MQ

BROKER, JoramMQ, PubSub+, OpenHAB MQTT binding)

(3) Search the libraries supported for both server and client for CoAP libraries and search MQTT libraries supported for both publisher and subscriber instead of broker. Generally, the message would be sent between the server and the client or the publisher and the subscriber. Consequently, CoAP libraries limited to client-side support are deemed insufficient for the purposes of this study. Instead, the CoAP libraries offering comprehensive support for both server and client functionalities are selected for observation and analysis. Similarly, with regard to MQTT libraries, due to the broker's role as an intermediary entity, the MQTT libraries related to the broker have been excluded.

(4) Search open-source libraries. Generally, developers tend to use open-source libraries because the source code in open-source libraries is publicly available to developers, so developers can easily pull the code and change it, and the developers are allowed to new the issue to help the development and maintenance of the library. As for the open-source libraries platform, GitHub is one of the most famous platforms. As of January 2023, there are over 372 million repositories in Github[12]. Therefore, open-source libraries in GitHub are considered in this study. We did not choose jcoap because it is from Google Code which was a platform to allow people to collaborate open-source projects, because the project hosting service of Google Code was closed in 2016[13].

[12] https://en.wikipedia.org/wiki/GitHub
[13] https://opensource.googleblog.com/2015/03/farewell-to-google-code.html

Name	Programming Language	Client/Server	License	Latest update date
Californium	Java	Client + Server	EPL+EDL	Feb 14, 2023
CoAP Shell	Java	Client	Apache License 2.0	Jun 1, 2021
java-coap	Java	Client + Server	Apache License 2.0	Jan 12, 2022
jcoap	Java	Client + Server	Apache License 2.0	June 12, 2012
nCoap	Java	Client + Server	BSD	Mar 20, 2018
Sensinode Java Device Library	Java SE	Client + Server	Commercial	Unkown
Sensinode NanoService Platform	Java SE	Client + Server	Commercial	Unkown

Table 5.1: Java language libraries of CoAP

Name	Programming Language	Type	License	Latest update date
PubSub+	C, C#/.Net, Java, JavaScript (NodeJs), Python, Go	Broker	Commercial license, free version	Jan 14, 2021
Paho MQTT	C, C++, C#, Go, Java, JavaScript, Python, Rust	Client	Eclipse Public License 1.0, Eclipse Distribution License 1.0 (BSD)	Aug 6, 2022
Thingstream	C, C++, Java, JavaScript, Python, Go	Client, Broker	Commercial licence version 2.0	Mar 14, 2019
HiveMQ MQTT Client	Java	Client	Apache License version 2.0	Feb 15, 2023
HiveMQ Community Edition	Java	Broker	Apache License 2.0	Mar 1, 2023
HiveMQ	Java	Broker	Commercial license	Feb 7, 2023
JoramMQ	Java	Broker	Commercial	June 7, 2022
moquette	Java	Broker	Apache License version 2.0	Feb 26, 2023
OpenRemote MQTT Broker	Java	Broker	AGPLv3	Feb 27, 2023
OpenHAB MQTT binding	Java	Client	Eclipse Public License	Apr 21 2020

Table 5.2: Java language libraries of MQTT

CoAP libraries' results are shown in Table 5.3. This table shows that the library of java-coap is more active than nCoap, so nCoap is not considered. Finally, Californium and java-coap library are selected as experiment subjects for CoAP library.

MQTT libraries' results are shown in Table 5.4, and the table shows that OpenHAB binding is one of the add-ons in OpenHAB, which includes Bluetooth Binding, Zigbee binding, ZWave Binding, etc. Therefore it is not considered. Finally, we selected Paho MQTT and HiveMQ MQTT Client library as experiment subjects for MQTT? library. Consequently, we obtain four libraries: `Californium`, `java-coap`, `HiveMQ MQTT Client` and `Paho MQTT`.

Name	Programming Language	Client/Server	License	Latest update date	Latest released date
Californium	Java	Client + Server	EPL+EDL	Feb 14, 2023	Feb 8, 2023
java-coap	Java	Client + Server	Apache License 2.0	Jan 12, 2022	Mar 28,2018
nCoap	Java	Client + Server	BSD	Mar 20, 2018	Oct 12,2014

Table 5.3: Java language libraries of CoAP

Name	Programming Language	Type	License	Latest update date
Paho MQTT	C, C++, C#, Go, Java, JavaScript, Python, Rust	Client	Eclipse Public License 1.0, Eclipse Distribution License 1.0 (BSD)	Aug 6, 2022
HiveMQ MQTT Client	Java	Client	Apache License version 2.0	Feb 15, 2023
OpenHAB MQTT binding	Java	Client	Eclipse Public License	Apr 21 2020

Table 5.4: Java language libraries of MQTT

5.2 Set up an IoT infrastructure

After choosing the protocols and libraries, there should be an environment to run the experiment. We deployed an infrastructure composed of three Raspberry Pi 4B, two with 8 GB of RAM and one with 4GB. Our implementations are available online for further studies and applicability[14,15]. In terms of CoAP, we used two Raspberry Pis with 8GB RAM for the server and the client respectively. In terms of MQTT, we use Eclipse Mosquitto in a Docker in the Raspberry Pi with 4GB RAM. To be specific, the two Raspberry Pis with 8 GB of RAM are installed with Wireshark to capture packet

[14] https://github.com/KeithLaiKB/java_learn_coap_comparison
[15] https://github.com/KeithLaiKB/java_learn_mqtt_comparison

flows. When collecting the runtime metrics, the router is not allowed to connect to the Internet to avoid the other network flow irrelevant to this experiment.

Those devices are set in the same local area network in this experiment. In CoAP, we assigned a static IP address for the server in my router configuration web page so that the client does not need to change the destination IP address in the program to access the server when the server's IP address changes. Similarly, in MQTT, we assigned a static IP address to the broker for the publisher and subscriber to access.

When the devices are set in different network areas, the server in CoAP and the broker in MQTT are recommended to be with the public static IP address. Because for example, a CoAP server needs some network configuration like port forwarding settings in the router or applying an IoT SIM card with a public static IP to connect to the network, which is time-consuming to configure and test the connectivity. However, it is easier to configure the broker than CoAP server to set a public static IP address, such as renting a server from a cloud computing platform anytime (Google Cloud, Amazon Web Service, Ali Cloud, etc.) and then doing the configurations immediately, which is a time-saving process and easy for developers to configure. If the devices are configured in different network areas, it is not fair to compare metrics in Step 4.6, because we want to set them in the same network condition and do the comparison, this is the reason that we choose to configure the experiment in the same local area network.

5.3 Collect, devise, and implement scenarios using the chosen libraries

After choosing the two protocols and the libraries, we need to identify and select scenarios illustrating them by exercising the APIs of their libraries, and then we implement the scenarios.

5.3.1 Collect libraries characteristics

Table 5.5 summarises the main characteristics of the four libraries. From the table, it shows that libraries have different characteristics, and in fact, the four libraries do not implement the protocols equally. Therefore, we need to choose some similar characteristics to make the comparison relatively equal. There are explanations of some details as follows:

- Regarding OSCORE in the 'Authentication' aspect, OSCORE provides `Californium` with authentication features. In terms of MQTT, the two libraries of MQTT already provide API for the publisher and subscriber to connect to the broker with usernames and passwords. We tried to compare the four libraries relatively equally in the 'Authentication' aspect, however, OSCORE includes the security solution, which means that it could not be compared with the cases that `HiveMQ MQTT Client` without security solution and `Paho MQTT` without security solution. In addition, we tried to compare the four libraries relatively equally in the 'Transport-layer Security' aspect, OSCORE is at the application layer, which could not be compared with the cases that `HiveMQ MQTT Client` and `Paho MQTT` with Transport Layer security solutions. As a result, we decided not to include the case of `Californium` with OSCORE into our experiment.

- Regarding *SSLContext* in the 'Transport-layer Security' aspect, it is a public class in a package called *javax.net.ssl* allows developers to create a specified instance which stands for secure socket protocol implementation[16]. Developer could init the instance with the sources of authentication keys (*KeyManager*), the sources of peer authentication trust decisions (*TrustManager*), the source of randomness for the generator (*SecureRandom*). Then, developers could use *SSLContext* to help start security communication between endpoints. Compared to configuring the security communication connection at the terminal, *SSLContext* makes this process easier. For example, it can help start a security conversation with TLS. In terms of use of *SSLContext*, in the code shown in Code 5.1, at the `Paho MQTT` publisher side, the publisher specifies version 1.3 of the Transport Layer Security protocol with specified *TrustManager* and *java.security.SecureRandom* instance but without *KeyManager*.

```
1 SSLContext context = SSLContext.getInstance("TLSv1.3");
2 context.init(null, tmf.getTrustManagers(), new java.security.SecureRandom());
```

Code 5.1: Part of SSLContext configuration code lines

After outlining the characteristics, we delved into their intricacies to ensure a more equitable

[16] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/javax/net/ssl/SSLContext.html

		CoAP		MQTT	
		Californium	**java-coap**	**HiveMQ**	**PahoMQTT**
Authentication	**Server (CoAP) / Broker (MQTT)**	OSCORE	N/A	N/A	N/A
	Client (CoAP) / Pub(MQTT) / Sub (MQTT)	OSCORE	N/A	usernames/passwords	
Transport-layer Security	**DTLS directly**	YES	NO	?	?
	DTLS with SSLContext	NO	NO	?	?
	TLS directly	?	NO	NO	NO
	TLS with SSLContext	?	Yes	YES	YES

Table 5.5: Characteristics of the libraries ('Servers' refers to servers and brokers. 'Clients' refers to clients, publishers, and subscribers. A question mark means that the implementation might exist, but we could not find a simple way to implement it).

basis for comparison. In order to compare the four libraries well, it is imperative to maintain control over the variable. There are some variables we considered:

- In terms of communication type, as introduced in Chapter 2, MQTT uses publish-subscribe messaging pattern. CoAP is not the same protocol as MQTT, so we employed the feature of observing the resource to keep a similar messaging pattern foundation before making the comparison between CoAP and MQTT.

- In terms of the 'Transport-layer Security' aspect, to be specific, JDK 8u261 includes an implementation of TLSv1.3[17]. To manipulate the experiment's parameters, the default setting of the `HiveMQ MQTT Client`, which is TLSv1.3. Consequently, in this comparison, `Paho MQTT` utilizes TLSv1.3 to maintain control over the variable. For `java-coap`, it does not support DTLS directly and DTLS with *SSLContext*, but supports TLS with *SSLContext*. Therefore `java-coap` with security scenario uses TLSv1.3 with *SSLContext*. As for `Californium`, using an *SSLContext* for a *DTLSConnector* is not supported, but developers can use DTLS directly solution, which supports DTLSv1.2 by default. These solutions are shown in Table 5.6.

Repository	Solution
Californium	DTLS v1.2 directly
java-coap	TLS v1.3 with SSLContext
PAHO MQTT	TLS v1.3 with SSLContext
HiveMQ MQTT Client	TLS v1.3 with SSLContext

Table 5.6: Security Solutions in Comparison.

[17] https://www.oracle.com/java/technologies/javase/8u261-relnotes.html

5.3.2 Devise scenarios

After summarizing the details of the four libraries, we went to devise scenarios. These libraries can be used in a wide variety of scenarios, but we want to make them into groups to compare. The scenarios are shown in Table 5.7, and the explanation is shown as follows:

- NON vs QoS0, with/out security. The reliability of NON is similar to the QoS0 scenario because the receiver receives the message without sending back an ACK packet or a PUBACK packet in return. For example, on a non-secure scenario basis, we applied the NON to the reliability of CoAP communication and applied QoS0 to the reliability of MQTT communication. Besides, we also put this comparison on a secure scenarios basis.

- CON vs QoS1, with/out security. The reliability of CON is similar to the QoS1 scenario because after the receiver receives the message, the receiver is required to send back an ACK packet or a PUBACK packet in return. For example, on a non-secure scenario basis, we applied the CON to the reliability of CoAP communication and applied QoS1 to the reliability of MQTT communication. Besides, we also put this comparison on a secure scenarios basis.

		CoAP	**MQTT**
No Security		NON	QoS0
		CON	QoS1
Security	**Californium DTLS v1.2 directly** **java-coap TLS v1.3 with SSLContext** **PAHO MQTT TLS v1.3 with SSLContext** **iveMQ MQTT Client TLS v1.3 with SSLContext**	NON	QoS0
		CON	QoS1

Table 5.7: Scenarios in comparison.

5.3.3 Implement scenarios

After devising the scenarios, we implemented them. In terms of security scenario, the security solution of the four libraries requires configuration in lots of details, we followed the repositories instructions on Github of `Californium`[6], `java-coap`[7], `Paho MQTT`[8] and `HiveMQ MQTT Client`[9]. Among the scenarios, distinct libraries exhibit different implementations. However, certain action parts are commonly shared across these implementations. We will delineate these parts in the following sections:

In CoAP, we added a resource to the server so that the client can observe this resource. In our experiment, we need to set the NON or CON scenario, and developers can set the reliability of this resource with the API called *setObserverType(Type type)* in the constructor in `Californium` and *setConNotifications(boolean conNotifications)* in `java-coap` as shown in Code 5.2.

```
1 this.setObserveType(Type.NON);          // Californium
2 this.setConNotifications(false);        // java-coap
```

Code 5.2: Set observing resource to send NON to the client

After implementing the resource in CoAP, the developer needs to add the resource to the server. For example, Code 5.3 shows we created an object of a class called *Cf_ObserverResource* here extends *CoapResource* which is a Java class in `Californium`, and then the developer can use this class to customize the resource like reliability and message format. We used the object of *CoapServer* which is the class of `Californium`, and added the resource to the server as shown in Code 5.4. Besides, 5683 is the port of the server side.

```
1 Cf_ObserverResource myobResc1 = new Cf_ObserverResource("Resource1");
```

Code 5.3: Create resource to server (`Californium`)

```
1 CoapServer server = new CoapServer(configuration,5683);
2 server.add(myobResc1);
```

Code 5.4: Add resource to server configuration (`Californium`)

In CoAP, the client can observe the resource with APIs called *observe(CoapHandler handler)* in `Californium` and *observe(ObservationListener observationListener)* in `java-coap`. In terms of the parameters of *CoapHandler* and *ObservationListener*, they help get the content from the server and count the number of received message. Code 5.5 shows the process when the client observes the resource of the server in `Californium`.

```
CoapHandler  myObserveHandler = new CoapHandler() {
    @Override
    public void onLoad(CoapResponse response) {
        System.out.println(response.getResponseText());
        numberOfMessages = numberOfMessages + 1;
    }
    @Override
    public void onError() {
    }
};
client.observe(myObserveHandler);
```

Code 5.5: Client observes resource (Californium)

In MQTT, we used Eclipse Mosquitto to be the broker. When applying the authentication features, we need to set a username and password pair in Mosquitto, and then the publisher or the subscriber can access this broker with the pair. Code 5.6 shows the process of authentication in Paho MQTT.

```
connOpts.setUserName("IamUser");
connOpts.setPassword("123456".getBytes());
```

Code 5.6: Auntentication configuration (Paho MQTT)

In MQTT, before starting to exchange data, the publisher and subscriber should connect to the broker. In our experiment, we set that the publisher or the subscriber has to wait until a successful connection is established before proceeding to the next step. In HiveMQ MQTT Client, the publisher or subscriber uses the API called *connect(@NotNull Mqtt5Connect connect)* to connect the broker. In this API, the 'connect' includes connection configurations such as authentication. Besides, the notation '@NotNull' in this API means that this parameter should not be null. In Paho MQTT, the publisher or subscriber uses the API called *connect(MqttConnectionOptions options)* to connect the broker. However, in Paho MQTT, this API will let the connection not timeout, while in HiveMQ MQTT Client is different, so we implemented a 'while' loop. This loop mimics the intended action, ensuring that the publisher or subscriber patiently awaits the successful establishment of the connection before proceeding to the next step. Code 5.7 shows those implementations

of HiveMQ MQTT Client and Paho MQTT.

```
//HiveMQ MQTT Client
client1.connect(connectMessage);
while(client1.getState().isConnected()==false) {
    //do nothing, just wait for connected
}

//Paho MQTT
client1.connect(connOpts);
```

Code 5.7: Connection configuration (HiveMQ MQTT Client and Paho MQTT)

After connecting to the broker, the developer needs to set the topic of the message and the QoS of the communication on the publisher side and then send the message. Code 5.8 and Code 5.9 shows the implementations of HiveMQ MQTT Client. In these codes, 'MqttQos.AT_MOST_ONCE' here means QoS0 and 'statusUpdate' is the content that changes each time. Our data length is fixed at 10 bytes.

```
c1 = publishBuilder1.topic("Resource1");     // topic setting
c1.qos(MqttQos.AT_MOST_ONCE);                // qos setting
```

Code 5.8: Set topic and communication reliability (HiveMQ MQTT Client)

```
c1.payload(("Hi!" + String.format("%07d", statusUpdate)).getBytes()); // payload
c1.send();                                                            // publish
```

Code 5.9: Construct content and send the message (HiveMQ MQTT Client)

In MQTT, the subscriber needs a method to get and output the message and then subscribe to the topic to get the message with specific QoS. Code 5.10 and Code 5.11 shows the implementations of Paho MQTT. In these codes, 'Resource1' and '0' mean the topic name and QoS level separately. Furthermore, The *messageArrived(String topic, MqttMessage message)* is overridden in a class *MyMqttCallback* that implements *MqttCallback* which is an interface in Paho MQTT.

```
client1.subscribe("Resource1",0);
```

Code 5.10: Subscribe the topic (`Paho MQTT`)

```
@Override
public void messageArrived(String topic, MqttMessage message) throws Exception {
    System.out.println(new String(message.getPayload()));
    numberOfMessages = numberOfMessages +1;
}
```

Code 5.11: Obtain and ouput the message (`Paho MQTT`)

5.4 Collect static metrics on the implemented senders and receivers

For static metrics, we collect values of CBO, CC, LCOM and LOC by following steps:

(1) Start the Understand software.

(2) Click the 'File' tab, and new a project.

(3) Select the directory of the project.

(4) Export the metrics into CSV file. In the options window, select the option about 'CountLineCode', 'CountClassCouple', 'Cyclomatic' and 'PercentLackOfCohesion'.

(5) Repeat the step above about the sender and receiver in each case.

From the tool, we can get the results into tables. To be specific, Table 5.8 shows that those metrics cover all the classes in our codes. However, in Table 5.9, LCOM is counted in each class because the result is a percentage format, which means that the results cannot be added together.

According to the description of the metrics mentioned in Step 4.4, the Table 5.8 shows the results about LOC, CBO and CC. To be specific, in CoAP, compared to `java-coap`, the three metrics (LOC, CBO, and CC) of `Californium` are better both in non-secure and secure scenarios, even taking advantage of server and client separately. Besides, the result of these metrics in 'Total' of MQTT, the metrics in `HiveMQ MQTT Client` are better both in non-secure and secure scenarios

in the combination of publisher and subscriber, except the CBO metric in the security scenario of `HiveMQ MQTT Client` and `Paho MQTT`, where they are equal.

In total, based on the previous result of LOC, CBO, and CC, we chose `Californium` from our CoAP group to compare with `HiveMQ MQTT Client` from our MQTT group in the combination between Server/Pub and Client/Sub scenario, Figure 5.1 shows that from those metrics, `HiveMQ MQTT Client` performs better in non-security and security scenarios because LOC in `HiveMQ MQTT Client` shows much less in the non-security scenario and CC in that show less in both non-security and security scenarios, but CBO in that is just slightly larger.

Specifically, the results show that in Server/Pub Side both in non-security and security scenarios, `HiveMQ MQTT Client` is better than `Californium` in all three metrics of LOC, CBO, and CC. Furthermore, In the Client/Sub Side, both in non-security and security scenarios, `Californium` emerges as the superior choice.

In addition, the results in Table 5.8a and Table 5.8b show that the security scenario increases the LOC and CBO, because security needs security classes and more code lines to configure the security settings. At the same time, in most of the results in these tables, the CC in security scenarios is slightly more than in non-security scenarios, which is expected because generally security scenarios needs more configuration.

Library	Metric	Sever	Client	Total
Californium	**LOC**	92	33	125
	CBO	17	10	27
	CC	16	6	22
java-coap	**LOC**	102	60	162
	CBO	20	19	39
	CC	18	10	28
HiveMQ MQTT Client	**LOC**	40	46	86
	CBO	13	15	28
	CC	4	7	11
Paho MQTT	**LOC**	35	70	105
	CBO	9	23	32
	CC	4	12	16

(a) Non-Security Scenario

Library	Metric	Sever	Client	Total
Californium	**LOC**	135	68	203
	CBO	33	24	57
	CC	19	8	27
java-coap	**LOC**	282	122	404
	CBO	62	34	96
	CC	42	14	56
HiveMQ MQTT Client	**LOC**	107	113	220
	CBO	31	31	62
	CC	8	11	19
Paho MQTT	**LOC**	99	134	233
	CBO	24	38	62
	CC	8	16	24

(b) Security Scenario

Table 5.8: LOC, CBO, and CC metrics in non-security and security scenarios

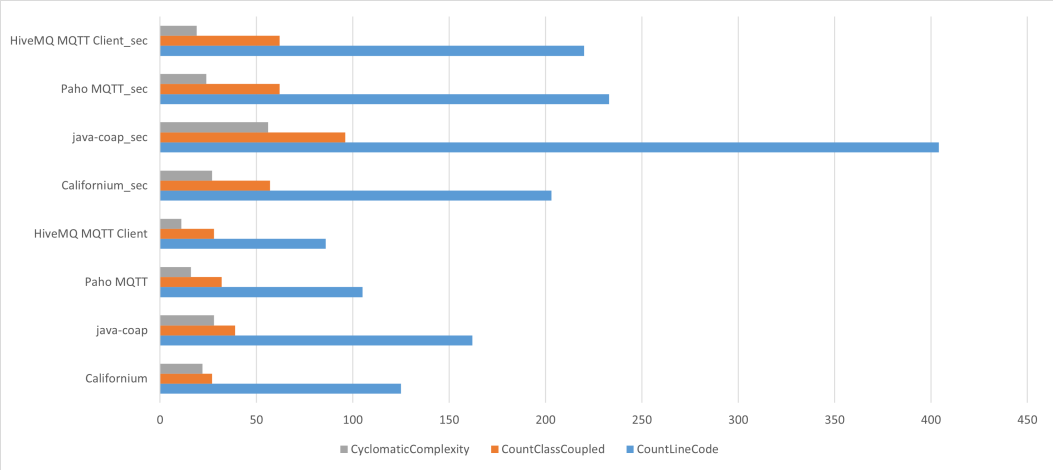

Figure 5.1: Combination between Server/pub and client/sub in non-security and security bar chart

Before discussing the result of Table 5.9, there is some explanation of the token in this table. To be specific, 'Launcher' in Table 5.9 is mainly to create and start the Server/Pub or Client/Sub. The 'Resource' is used in CoAP to deal with the message, and 'UpdateTask' is a timer to send messages periodically and to avoid the condition that when there are two resources required to run together, the second resource does not keep sending messages until the first resource is finished if there are not two threads. However, those two MQTT libraries do not need that. More details with explanation in those four libraries are described as follow:

- In `Californium`, Launcher is related to the class *TestMain_JavaCoap_Obs_Server* on the server side, and the Resource is related to the class *Cf_ObserverResource*, but Launcher on the client side is related to the class *TestMain_Cf_Obs_Client*.

- In `java-coap`, Launcher is related to the class *TestMain_JavaCoap_Obs_Server*, and the Resource is related to the class *JavaCoap_ObserverResource*, but Launcher on the client side is related to the class *TestMain_JavaCoap_Obs_Client*. In security scenario, ExtraTransportCfg is a class called *SingleConnectionSocketServerTransport* from `java-coap` library, which is an existing class used to test the security case [18].

[18] https://github.com/PelionIoT/java-coap/blob/master/coap-core/src/test/java/com/mbed/coap/transport/javassl/SingleConnectionSocketServerTransport.java

- In `HiveMQ MQTT Client`, Launcher is related to the class *TestMain_Hivemqmqttclient-_Publisher* on the publisher side, but Launcher on the subscriber side is related to the class *TestMain_Hivemqmqttclient_Subscriber*.

- In `Paho MQTT`, Launcher is related to the class *TestMain_Pahomqtt_Publisher* on the publsher side, but Launcher on the subscriber side is related to the class *TestMain_Pahomqtt_Subscriber*. At the same time, on the subscriber side, the MsgCallback is related to the class *MyMqttCallback*, and in this class, one of the settings is to print out the value with the specified format.

In this Table, we can get the result that in CoAP, the LCOM in `Californium` is equal to `java-coap` when they are comparing the Launcher, Resource and UpdateTask, `HiveMQ MQTT Client` is equal to `Paho MQTT` when they are comparing the Launcher in both non-security and security scenarios. However, the LCOM metric is not compared between CoAP and MQTT in our experiment because some important features are divided into different classes.

Library	Side	Class	LCOM
Californium	**Server**	Launcher	0
		Resource	35
		UpdateTask	50
	Client	Launcher	50
java-coap	**Server**	Launcher	0
		Resource	35
		UpdateTask	50
	Client	Launcher	50
HiveMQ MQTT Client	**Server**	Launcher	0
	Client	Launcher	66
Paho MQTT	**Server**	Launcher	0
	Client	Launcher	66
		MsgCallBack	0

(a) Non-Security Scenario

Library	Side	Class	LCOM
Californium	**Server**	Launcher	0
		Resource	35
		UpdateTask	50
	Client	Launcher	50
java-coap	**Server**	Launcher	0
		Resource	35
		UpdateTask	50
		ExtraTransportCfg	62
	Client	Launcher	50
HiveMQ MQTT Client	**Server**	Launcher	0
	Client	Launcher	66
Paho MQTT	**Server**	Launcher	0
	Client	Launcher	66
		MsgCallBack	0

(b) Security Scenario

Table 5.9: LCOM in non-security and security scenarios

5.5 Analyse packet sizes theoretically and practically

Firstly, we got the theoretical packet size from the specifications. Secondly, the practical metric is derived from our practical test from the four libraries. Thirdly, we adjusted the data according to the theoretical packet size. The steps of this analysis are shown as follows:

(1) Get the theoretical packet size from the specifications. In this step, we computed the theoretical packet size from the specification in Shelby, Zach and Hartke, Klaus and Bormann, Carsten [3] and in OASIS Message Queuing Telemetry Transport (MQTT) TC [8]. From those specifications, the packet size is about the non-secure scenarios.

(2) Summarised the packet size from the four libraries, `Californium`, `java-coap`, `HiveMQ MQTT Client` and `Paho MQTT`. The result is based on successful communication without error. For example, in `java-coap`, the client sends a GET request to the server without the resource name when observing a resource, the server will send back an ACK packet with a response code called 4.04 whose description is 'NOT FOUND', in this case, we do not consider this into our metric.

(3) Adjust the practical packet size according to the theoretical packet size to get the adjusted practical packet size. The result refers to the practical data but is adjusted according to theoretical data and systematically and logically.

After collecting the data on the packet size, we get the result in tables. Before introducing the result in tables, there are some token need to be introduced firstly:

- As for the connection part in CoAP, there are two types of packets: one is CON packet about GET request from the client to the server, which we called it GET_CON packet, and the other one is the ACK packet from the server to the client. MQTT has four types: CONNECT packet, CONNACK packet, SUBSCRIBE packet and SUBACK packet.

- 'With Content': The condition 'With content' in Table 5.10 and in Table 5.12 means that this packet must contain the content, at least one byte. Those libraries of CoAP have the attribute of the payload marker, but the two of MQTT are not. In CoAP, if there is a payload, there will be a payload (y) position, followed by a payload marker using one byte. This is the reason why y is specified in the cell in Table Table 5.10, but it is unnecessary to highlight the y in Table 5.11. Besides, in terms of GET_CON packets, it means CON packet, but it is used to send the GET request from the client to the server.

- 'Content-Format': it shows the representation format of the message. In `java-coap`, the server can send the ACK back to the client without setting Content-Format. However, in `Californium`, the API called *respond(String payload)*, inside automatically sets the 'Content-Format'. Even though the API itself implicity include the Content-Format, we still use the API called *respond(ResponseCode code, String payload, int contentFormat)* to explicitly specify the Content-Format. Therefore, `java-coap` sets 'Content-Format' in ACK packet from the server to the client so that it could also be well compared by `Californium` in Step 4.4 and Step 4.6. However, this step shows the bytes used in `java-coap` with or without setting 'Content-Format', providing readers with insights into the potential packet size.

- 'GET_CON': In `java-coap` library, GET_CON packet has to include the resource name for the client to access the server, but in `Californium`, it does not have to. Besides, the 'Content-Format' in ACK packet from the server to the client could be set manually in `java-coap` library, and it would affect the size of the packet, but in Californium, the content format is set by default in the libraries.

- '1st Msg' and '2nd Msg': In CoAP observer pattern condition, ACK packet could be the first message from the sender to the receiver, and the NON or CON packet would be the second message from the sender to the receiver, but MQTT uses the PUBLISH packet to send the first message and the rest. This is why there is a condition about the first message and second message about the PUBLISH packet in Table 5.10. In those two libraries in MQTT, the packet size of the first message is sometimes different from the second message, like the second message might lack a topic name or own a subscription identifier.

From the result in Table 5.10, the practical metric shows that in CoAP, packet size in `java-coap` is smaller than `Californium` because in our demonstration program, `java-coap` uses a token with two bytes, but `Californium` uses eight bytes by default. In our observer pattern demonstration program, GET_CON, ACK (server sends the ACK packets for replying to the GET request sent from the client), NON and CON packets are all with tokens. In this case, the total bytes used by packets in `Californium` would be much more than in `java-coap`.

From the result in Table 5.11, the practical metric shows that in MQTT, packet size in `Paho MQTT` is smaller than `HiveMQ MQTT Client`. On the subscriber side, `HiveMQ MQTT Client` would use the Subscription Identifier attribute with the value when receiving the PUBLISH packet with Qos0/QoS1 from the broker or sending the SUBSCRIBE packet to the broker, but `Paho MQTT` does not.

From the result in Table 5.14, the adjusted practical packet size metric shows that the total packets' size of the connection part in MQTT is larger than in CoAP. Besides, the metric in Table 5.12 and Table 5.13 shows that in the observer pattern, through the whole network, the size of total packets used in the part of sending and receiving valuable messages (including ACK/PUBACK, excluding the connection part) in CoAP is less than in MQTT (no-security, because security will encrypt the data) because there are two processes in MQTT, messages from publisher to broker and message from broker to subscriber in this experiment.

Library		**Californium**	**java-coap**
CON	**Content is ""**	$15 + m$	$10 + m$
	With content	$16 + y + m$	$11 + y + m$
GET_CON	**Without resource name**	$13 + m$	NONE (Because it must contain resource name)
	With resource name	$14 + rsn + m$	$8 + rsn + m$
NON	**Content is ""**	$15 + m$	$10 + m$
	With content	$16 + y + m$	$11 + y + m$
ACK	**Content is "" (observe)**	$15 + m$	no Content-Format: $7 + m$
			set Content-Format: $9 + m$
	With content (observe)	$16 + y + m$	no Content-Format: $8 + y + m$
			set Content-Format: $10 + y + m$
	Client to server	$4 + z$ (default 4)	$4 + z$ (default 4)
Explanation	$y \in [1, i)$ means payload i means the number of bytes, the length is limited by the protocol $rsn \in [1, 255]$ means the resource name m means the number of bytes of other unnecessary attributes and unnecessary value z is undefined number, starting at 0		

Table 5.10: Comparison between Californium and java-coap

Library			**Paho MQTT**	**HiveMQ MQTT Client**
CONNECT			$19 + msgl + p_1 + m$	Same
CONNACK			$3 + msgl + p_1 + m$	Same
PUBLISH_QoS0	**Pub**	**1st Msg**	$6 + msgl + p_1 + tpn + m$	Same
		2nd Msg	$6 + msgl + p_1 + m$	Same
	Sub	**1st Msg**	$3 + msgl + p_1 + tpn + m$	$3 + msgl + p_1 + tpn + m + (1 + sbid)$
		2nd Msg	$3 + msgl + p_1 + tpn + m$	$3 + msgl + p_1 + tpn + m + (1 + sbid)$
PUBLISH_QoS1	**Pub**	**1st Msg**	$8 + msgl + p_1 + tpn + m$	Same
		2nd Msg	$8 + msgl + p_1 + m$	Same
	Sub	**1st Msg**	$5 + msgl + p_1 + tpn + m$	$5 + msgl + p_1 + tpn + m + (1 + sbid)$
		2nd Msg	$5 + msgl + p_1 + tpn + m$	$5 + msgl + p_1 + tpn + m + (1 + sbid)$
PUBACK			$3 + msgl + p_2 + m$	Same
SUBSCRIBE			$6 + msgl + p_1 + tpn + m$	$6 + msgl + p_1 + tpn + m + (1 + sbid)$
SUBACK			$4 + msgl + p_1 + m$	Same
Explanation	$msgl \in [1, 4]$ means message length $p_1 \in [1, 4]$, $p_2 \in [0, 4]$, p_1 and p_2 both mean property length $tpn \in [1, a)$ means the topic name $sbid \in [1, 4]$ means the value of Subscription Identifier a means the maximum length of topic(topic name) is limited by the protocol m means the number of bytes of other unnecessary attributes and unnecessary value			

Table 5.11: Comparison between HiveMQ MQTT Client and PahoMQTT

<table>
<tr><th colspan="2"></th><th>Theoretical</th><th>Adjusted practical</th></tr>
<tr><td>CON</td><td>Content is ""</td><td>$4+e$
$+(1+(opn_1*op_1+opn_2*op_2+opn_3*op_3))*opgn$
$+y$</td><td>$7+e+m$</td></tr>
<tr><td>NON</td><td>Content is ""</td><td>$4+e$
$+(1+(opn_1*op_1+opn_2*op_2+opn_3*op_3))*opgn$
$+y$</td><td>$7+e+m$</td></tr>
<tr><td rowspan="2">ACK</td><td>Content is ""
(observe)</td><td rowspan="2">$4+e$
$+(1+(opn_1*op_1+opn_2*op_2+opn_3*op_3))*opgn$
$+y$</td><td>$7+e+m$</td></tr>
<tr><td>Client to server</td><td>$4+e+z$</td></tr>
<tr><td>Explanation</td><td colspan="3">$e \in [0,8]$ means token
$opn_1 \in [0,1]$ means the number of OptionDelta (Extended)
$op_1 \in [0,2]$ means OptionDelta (Extended)
$opn_2 \in [0,1]$ means the number of OptionLength (Extended)
$op_2 \in [0,2]$ means OptionLength (Extended)
$opn_3 \in [0,1]$ means the number of OptionValue
$op_3 \in [0,a)$ means OptionValue
a means the maximum length of OptionValue is limited by the protocol
$opgn$ means the number of group of the Option, it is undefined number limited by the protocol, starting at 0.
$y \in [0,i)$ means payload
i means maximum length is limited by the protocol
m means the number of bytes of other unnecessary attributes and unnecessary value
z is undefined number, starting at 0</td></tr>
</table>

Table 5.12: Partial of theoretical and adjusted practical packet size metrics of CoAP

	Theoretical	Adjusted practical
PUBLISH_QoS0	$3 + msgl + tpn + p_1$ $+n_1 * (1 + 4) + n_2 * (1 + 2) + n_3 * (1 + 1)$ $+n_4 * (upgn * (1 + (2 + k_1) + (2 + k_2)))$ $+n_5 * (1 + (2 + k_1)) + n_6 * (1 + sbid)$ $+y$	$4 + msgl + p_1 + m$
PUBLISH_QoS1	$5 + msgl + tpn + p_1$ $+n_1 * (1 + 4) + n_2 * (1 + 2) + n_3 * (1 + 1)$ $+n_4 * (upgn * (1 + (2 + k1) + (2 + k_2)))$ $+n_5 * (1 + (2 + k_1)) + n_6 * (1 + sbid)$ $+y$	$4 + msgl + p_1 + m$
PUBACK	$1 + msgl + 2$ $+n_7 * 1 + p_2$ $+n_8 * (1 + (2 + k_1))$ $+n_4 * (upgn * (1 + (2 + k_1) + (2 + k_2)))$	$3 + msgl + p_1 + m$
Explanation	$k1 \in [0, 65535]$, $k2 \in [0, 65535]$, k1 and k2 means the content of the specific attribute $y \in [0, i)$ means payload i means maximum length is limited by the protocol $upgn$ means the number of group of the User Properties, it is limited by the protocol $msgl \in [1, 4]$ means message length $p_1 \in [1, 4]$, $p_2 \in [0, 4]$, p_1 and p_2 both mean property length $tpn \in [0, a)$ means the topic name $sbid \in [1, 4]$ means the value of Subscription Identifier $n_1 \in [0, 1]$ means Message Expiry Interval $n_2 \in [0, 1]$ means Topic Alias $n_3 \in [0, 1]$ means Payload Format Indicator $n_4 \in [0, 1]$ and $n_5 \in [0, 3]$ both mean different groups of attributes with same bytes $n_6 \in [0, 1]$ means Subscription Identifier $n_7 \in [0, 1]$, means reason code $n_8 \in [0, 1]$, means reason string a means the maximum length of topic(topic name) is limited by the protocol m means the number of bytes of other unnecessary attributes and unnecessary value	

Table 5.13: Partial of theoretical and adjusted practical packet size metrics of MQTT

In Table 5.14, in MQTT, the packet size of the SUBSCRIBE packet assumes that the topic name is one byte. Besides, the publisher and subscriber here does not use the Anonymous, which means that the username and password are required. Besides, the client identifier(ClientID) is required when they connect to the broker.

Library	**CoAP**		**MQTT**	
	Item	**Value**	**Item**	**Value**
Server/Publisher	NONE	NONE	Send CONNECT	$19 + msgl + p_1 + m$
			Receive CONNACK	$3 + msgl + p_1 + m$
Client/Subscriber	Get(observed)	$7 + e + m$	Send CONNECT	$19 + msgl + p_1 + m$
			Receive CONNACK	$3 + msgl + p_1 + m$
			Send SUBSCRIBE	$7 + msgl + p_1 + m$
			Receive SUBACK	$4 + msgl + p_1 + m$
Explanation	$e \in [0, 8]$ means token in CoAP $msgl \in [1, 4]$ means message length in MQTT p_1 means property length m means the number of other unnecessary message In this table, assume that the resource name of CoAP and topic name of MQTT use one byte			

Table 5.14: Comparison between CoAP and MQTT Connection adjusted practical size

5.6 Collect runtime performances when running the scenarios

We ran our program demonstrations in our IoT infrastructure and used the Wireshark software to capture the log about the packet sending and receiving. There are the steps as follows:

(1) Start the Wireshark to record the packets. If the test is related to MQTT, the broker should be started before because the publisher and subscriber need to connect to an available broker.

(2) Start the sender to send 200 messages. Each data contains only 10 bytes, such as 'Hi!0000001'.

(3) Start the receiver to send 100 messages. It is expected that the receiver will receive Data with 1,000 bytes in total, because some messages could not be received. For example, we opened the broker, then we opened the publisher, and then we started the subscriber. In this scenario, the subscriber is the last one to start. During the period that the subscriber connected to the broker, the publisher has already sent the messages to the broker, however, the subscriber does not subscribe to the topics, which means that the subscribers could not receive the messages sent by the subscriber before. In this case, the number of received messages might be less than 200, so we set the target number received by the subscriber to be 100.

(4) After the communication is finished, stop the Wireshark.

(5) Repeat the experiment 10 rounds for each case. We opened the Wireshark log and recorded the $Time1$ and $Time2$, which have been described in Step 4.5.

(6) Count the overhead. We opened the Wireshark log and recorded the size of the packets and sum together, which has been described in Step 4.5.

For example, Figure 5.2 and Figure 5.3 show parts of log at `Californium` client side captured by Wireshark. In Figure 5.2, the fourth packet captured by the tool is a GET request send by the client, and we set the time of this packet as $Time1$ in Equation 1, and in this figure, the fifth packet shows that this is the first CoAP protocol packet received by the receiver with data 'Hi!0000009'. In Figure 5.3, the No.220 packet captured by the tool is an ACK request send by the client, and we set the time of this packet as $Time2$ in Equation 1, and in this figure, the No.219 packet shows that this is the one-hundredth data received by the receiver with data 'Hi!0000108'. In log files by Wireshark, we can get the packet size of each packet, which can be used to calculate the metrics by the formulas mentioned in Step 4.6.

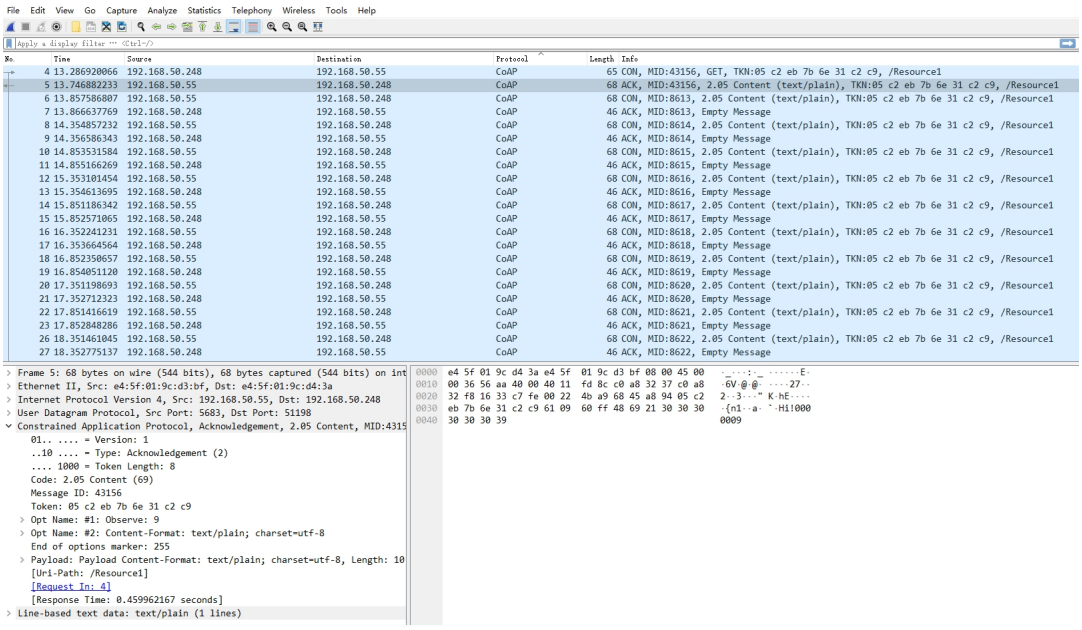

Figure 5.2: The beginning part of the log for `Californium` Communication in wireshark

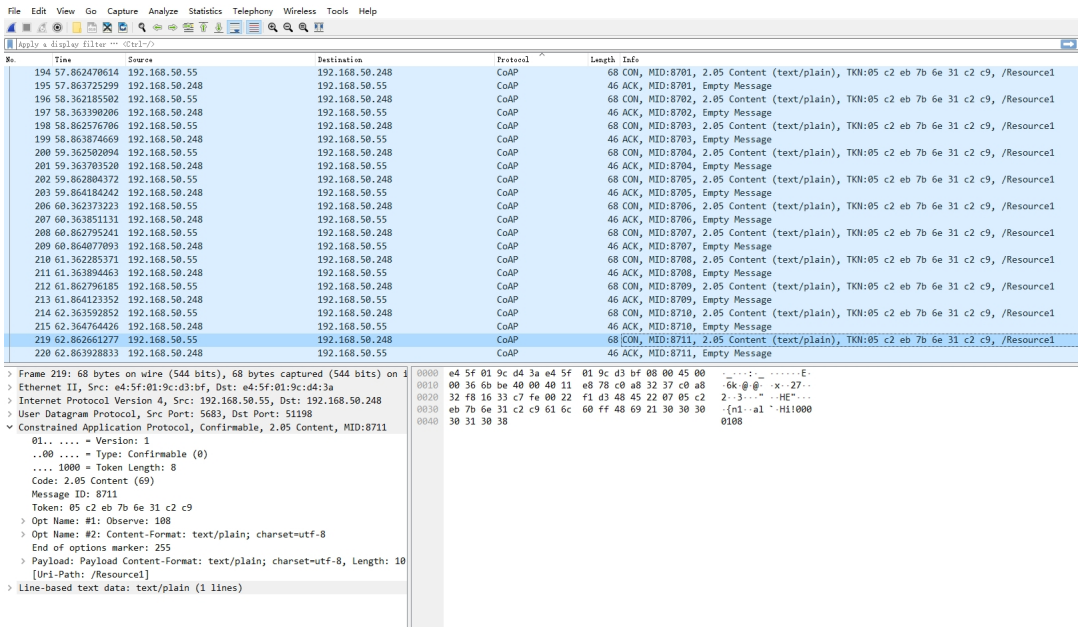

Figure 5.3: The end part of the log for `Californium` Communication in wireshark

After collecting the time and packet size used for each packet from the log, we got the results. Before discussing the result shown in Figure 5.4, Figure 5.5, and Figure 5.6 show the comparison of the four scenarios (16 cases) about goodput, throughput, and overhead, respectively, some details need to be introduced:

- 'Negligible': We used this adjective to express that object A is slightly better than B in group one, but B is slightly better than A in group two.
- Our experiment is based on non-continuous traffic shown in Figure 5.7. Under the assumption that packets can be completed within a fixed time interval without being fragmented into multiple subpackets for separate transmission, according to Equation 3, the last message will increase the 'communication used time' to an extent, but the packets make good use of this time so that high throughput condition could happen.
- 'Overhead' metric in Figure 5.6 refers to all the bytes through the communication between the sender and receiver, excluding the total valuable data such as 'Hi!0000001'. It is important to clarify that the term 'overhead' in this figure is not the overhead portion in each packet.

In terms of goodput in CoAP, `java-coap` with CON with no security is the best (CON/NON

with no security, CON/NON with security). Among CON cases, goodput in `java-coap` is better than `Californium`, but the difference is negligible among NON cases. In terms of throughput in CoAP, among secure cases, `java-coap` is higher, but for non-secure cases, `Californium` is higher. Because in secure cases, through the communication, `java-coap` uses more overhead in each packet than `Californium`, but in non-secure cases, `Californium` uses more overhead in each packet. In terms of overhead in CoAP, in the secure scenario, `Californium` is smaller than `java-coap`. However, in the non-secure scenario, `java-coap` is slightly smaller than `Californium`.

In terms of goodput in MQTT, `Paho MQTT` with QoS0 with the no security scenario is the best (Qos0/QoS1 with no security, Qos0/QoS1 with security). But in other scenarios, `HiveMQ MQTT Client` is better. In terms of throughput in MQTT, `HiveMQ MQTT Client` is higher. In terms of overhead in MQTT, in the secure scenario, `HiveMQ MQTT Client` is slightly smaller than `Paho MQTT`. However, in the non-secure scenario, `Paho MQTT` is slightly smaller than `HiveMQ MQTT Client`.

Based on our experiment, in goodput, CoAP is better than in MQTT. In throughput, throughput in MQTT is higher in the no-secure scenario. However, in most cases of security scenarios, their throughput is equivalent. In most cases, overhead in CoAP is smaller than in MQTT. As mentioned above, CoAP is better than MQTT.

Comparing the NON/QoS0 to the CON/QoS1 scenario, in most cases of goodput, they are equivalent. Except that the `Paho MQTT` with QoS0 in the non-secure case is visibly better than with QoS1 in the non-secure case. Because in our program demonstrations, the experiment is about non-continuous traffic, Figure 5.8 shows that sending back the ACK packet from client to server would not spend a lot of time because the ACK packet size here is really small. In throughput metric, in CoAP, the size of NON packet with data is similar to the size of CON with data, so the throughput does not make too much difference, while in MQTT, the size of QoS1 packet with data is larger than the the size of QoS0 with data, so it makes throughput in Qos1 scenario is higher than in QoS0 scenario according to Figure 5.8. In the overhead metric, that in QoS0/NON cases is smaller. As a result, QoS0/NON scenario is better.

Comparing the non-security scenario to the security scenario, In goodput, non-security cases

show better. In throughput, security cases (Group2) are obviously higher. Figure 5.7 shows that in the non-continuous scenario, when the total size of each packet could be sent in a fixed period if the overhead in each packet is more, the throughput could be higher. Because the size of the packet makes good use of this period of time. In overhead, non-security cases are smaller. Consequently, employing a security solution and opting for a solution without security measures both have their distinct advantages. The suitability of either approach for integration into developers' projects is contingent upon the specific requirements of the developers.

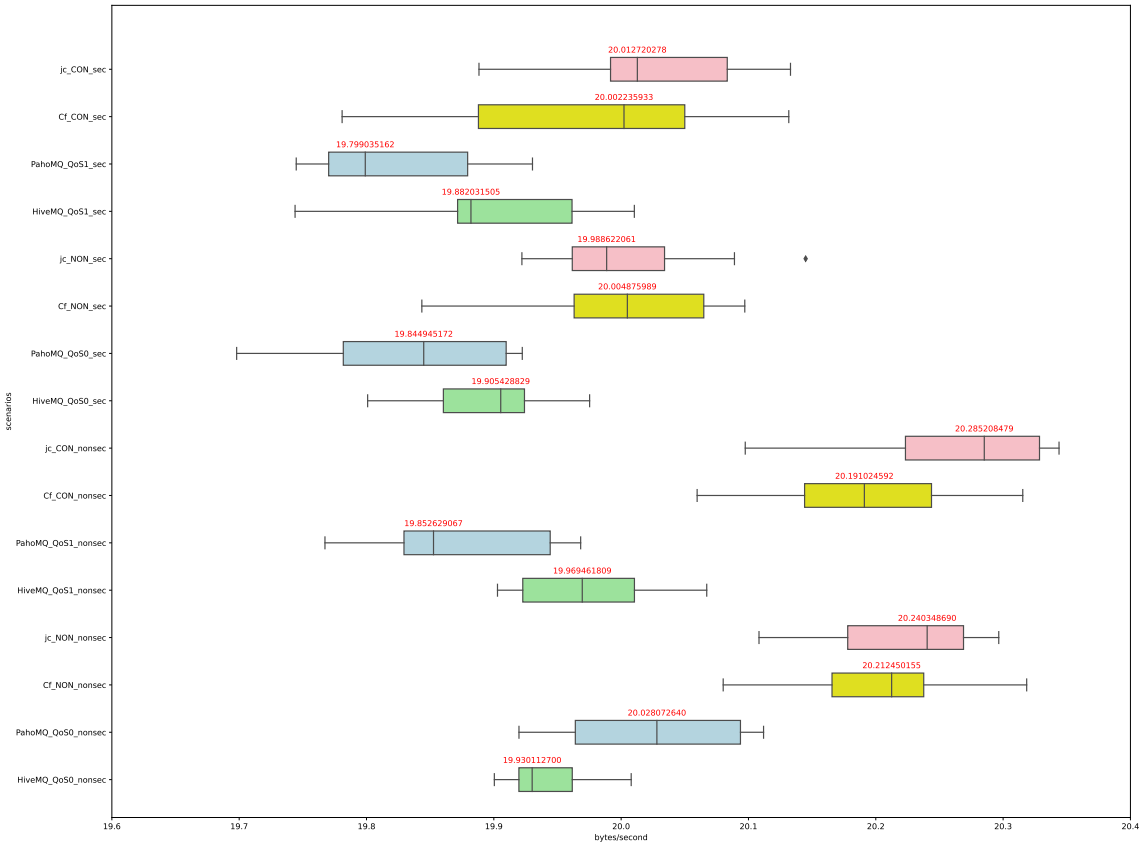

Figure 5.4: Goodput comparison of CoAP and MQTT

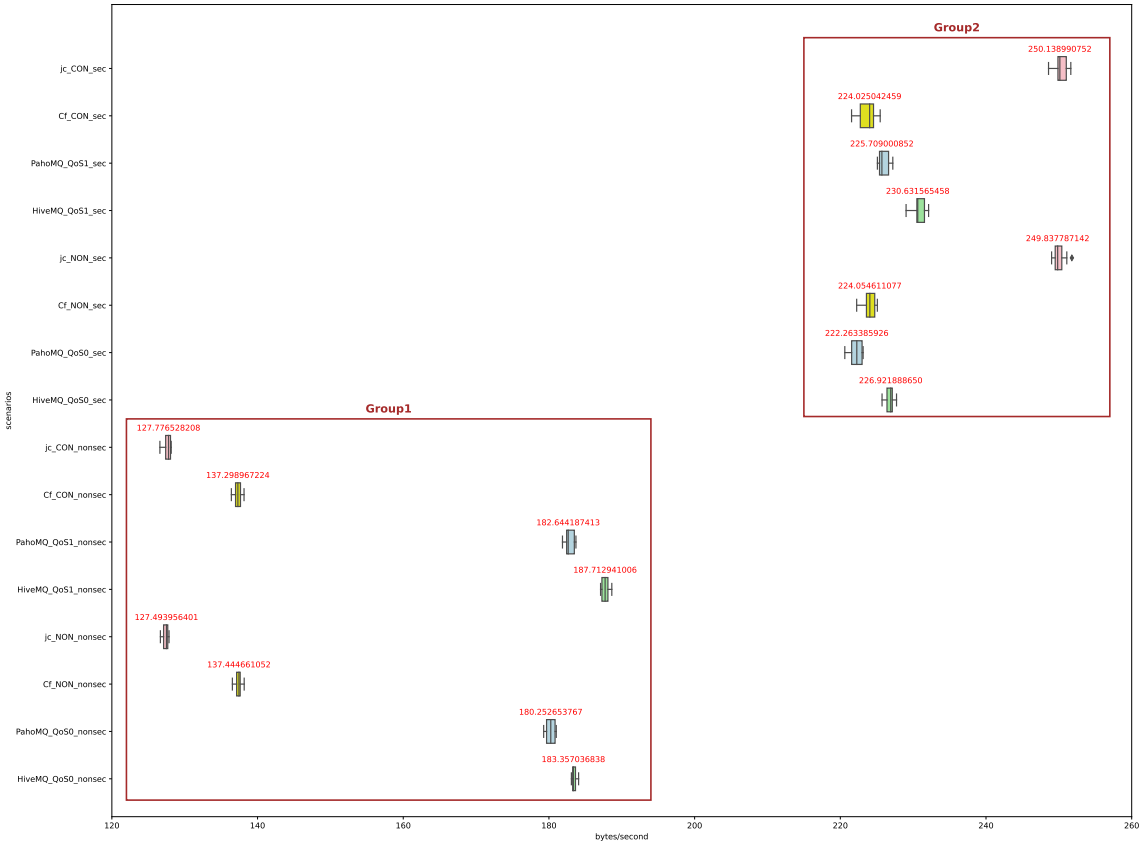

Figure 5.5: Throughput comparison of CoAP and MQTT

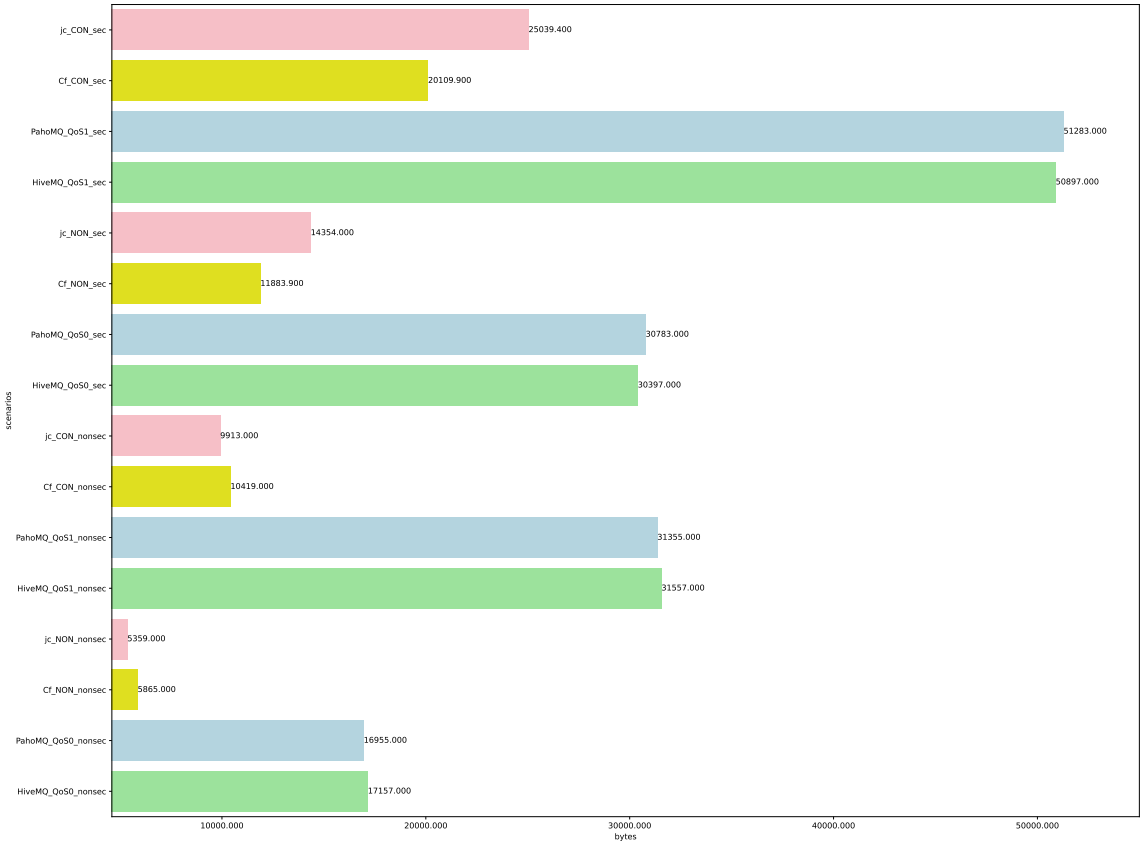

Figure 5.6: Overhead comparison of CoAP and MQTT

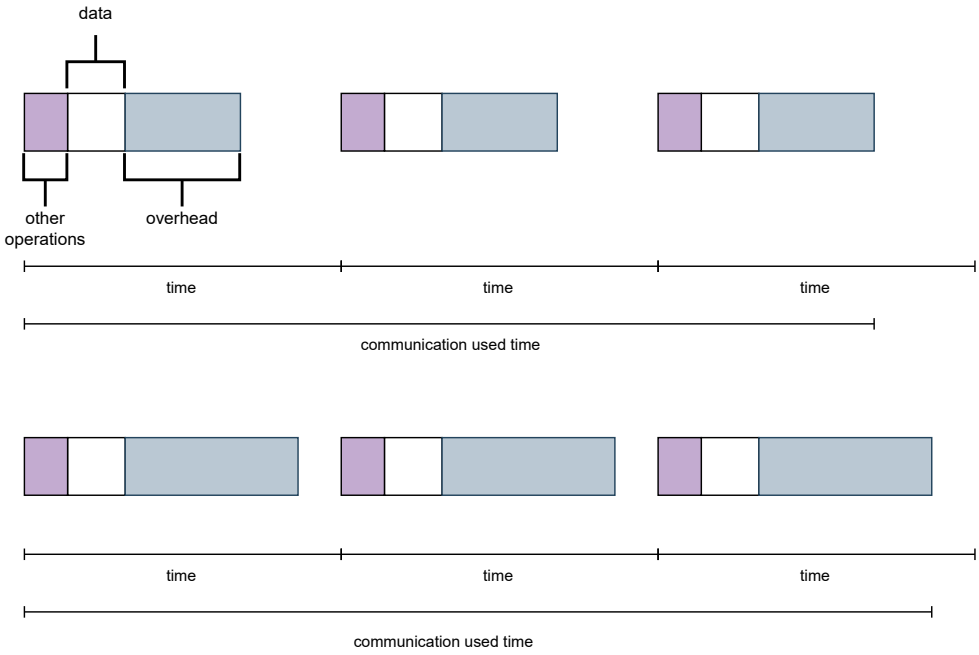

Figure 5.7: Comparison about different overhead in non-continuous traffic

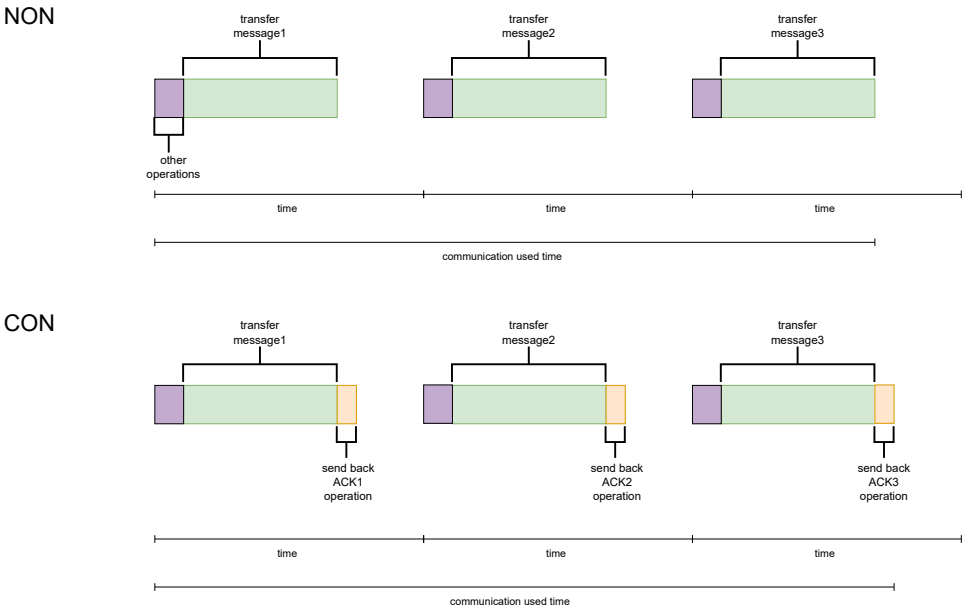

Figure 5.8: Communication packet in non-continuous traffic

5.7 Compare the collected measures and recommend a protocol/library

Based on the analysis of static metrics of the 'Total' in Table 5.8 from Section 5.4, our findings lead to the conclusion as follows:

- In terms of CoAP, `Californium` is recommended. Because it outperforms `java-coap` with less LOC, CC, and CBO, so `Californium` program uses fewer code lines and has good reusability, maintainability and changeability.
- In terms of MQTT, `HiveMQ MQTT Client` is recommended. Because it outperforms `Paho MQTT`, with less LOC and CC, which means that `HiveMQ MQTT Client` program is easier to code, reuse and maintain.
- Overall, `HiveMQ MQTT Client` performs the best among the four libraries. Because comparing LOC metric in `Californium` of CoAP, that in `HiveMQ MQTT Client` of MQTT shows much less in the non-security scenario. Furthermore, in both non-security and security scenarios, CC in `HiveMQ MQTT Client` shows less, even though CBO in that is just slightly larger.

Based on the analysis of packet size metrics in Section 5.5, in the non-security scenario, our findings lead to the conclusion as follows:

- The two libraries of MQTT would use higher packed size through the whole network than the two of CoAP because MQTT has two processes (publisher to broker, and broker to subscriber), and the accumulated packet size from both processes is larger than that of CoAP.
- From the tables in Section 5.5, `java-coap` is the best choice among the four libraries using the least packet size, followed by `Californium`.
- As a result, we recommended using `Californium`, even though `java-coap` uses two tokens of CoAP communication by default, which is less than eight tokens used by default in `Californium`. Because `Californium` is an active library in GitHub, so it is possible that it would be easily configured for setting tokens in the future version so that `Californium`

might use the least packet size in the future version. The difference would then be negligible when they have the same token length.

Based on the analysis of runtime metrics in Section 5.6, our findings lead to the conclusion that CoAP is faster than MQTT, and it shows better in the overhead metric. To be specific, among the four libraries in this experiment, in most scenarios, `java-coap` is the fastest, followed by `Californium`.

In all, based on our experiment, in pursuit of fast communication, we highly recommend using `Californium`. While in pursuit of the advantage at the coding level refers to the static metrics, we highly recommend using `HiveMQ MQTT Client`.

Chapter 6

Discussions

6.1 Discussions

From the experiment, the comparison between NON/QoS0 and CON/QoS1 scenario does not manifest any significant advantages at goodput based on the non-continuous traffic. In the majority of IoT applications in daily life, people usually access IoT devices in non-continuous traffic. For instance, in residential settings, occupants typically manage IoT devices by issuing specific commands as needed or receiving scheduled notifications, as opposed to exerting continuous control over the devices. When developers apply similar user requirements, they can choose CON/QoS1 scenario to improve reliability. However, if the IoT devices are deployed in a place where a lot of devices are communicating over the same network, it would lead to network congestion because our experiment result shows that the overhead metric in CON/QoS1 scenario is more than in CON/QoS0 scenario, which means there would be a lot of extra packets of ACK or PUBACK transmitted through the network.

In addition, our programs in the non-secure scenario run faster than the secure scenario because creating the connection securely will take time, and the secure cases need more overhead in each packet. However, the decision to employ a secure mode depends on the specific user requirements.

Generally, IoT devices are set in different area networks. For example, the mobile phone receives the temperature from a smart thermometer at home and uses the WiFi in school. In this case, the smart thermometer applying CoAP is not easy to configure as mentioned in Section 5.2, but

if applying MQTT protocols, developers can rent a could server to be the broker anytime, which is convenient for developers. Therefore, we highly recommend developers to use `HiveMQ MQTT Client`.

6.2 Threats to Validity

In this section, we discuss the threats to the validity. In our experiment, the code style will affect LOC. Even though, in this experiment, we make our program demonstration code as simple as possible with maintainable logic. Besides, achieving an extremely simple program with minimum code lines is inevitably challenging.

In this experiment, we used Understand Software to collect the static metrics because it is a famous and useful software. However, there is no guarantee that it has no bug, and at the same time, the packet size is computed by hand, which might cause some bytes to be ignored to record to affect the outcome, so it has an influence on validity,

In comparison between QoS0/NON scenario and QoS1/CON scenario, theoretically, QoS0/NON case should be faster than QoS1/CON case However, the result does not show an obvious difference. It might be affected by Raspberry Pi and the router's temperature, which affects the validity.

In this thesis, there are two libraries from CoAP protocol, and two libraries from MQTT, it could not make a very general conclusion to compare the CoAP and MQTT, which affects validity.

Chapter 7

Conclusion

In contemporary times, IoT is experiencing exponential growth, reflected in the increasing deployment of IoT devices so it is important to find the appropriate protocol and its libraries to apply to different scenarios.

We initially chose CoAP and MQTT as the target protocols. Then in Section 5.1, we surveyed and selected different libraries of the protocols and finally chose `Californium`, `java-coap`, `Paho MQTT`, `HiveMQ MQTT Client` as the target libraries. After setting up the IoT infrastructure according to Section 5.2, we collected and devised the scenario about the NON/QoS0, CON/QoS1, security and no security in Section 5.3. Besides we used the libraries to implement the scenarios.

In Section 5.4, we collected static metrics (CBO, CC, LCOM and LOC) on the implemented senders and receivers, and we concluded that `HiveMQ MQTT Client` demonstrates commendable reusability, maintainability, and changeability among these four repositories. In Section 5.5, based on our experiment, CoAP use less packet size than MQTT, and `Californium` is recommended. In Section 5.6, we collected runtime performances (goodput, throughput, overhead) when running the scenarios, we concluded that `java-coap` is the fastest, and the overhead metric shows that `java-coap` use the least bytes to communicate in the whole communication process.

Consequently, we recommend using `HiveMQ MQTT Client`, because programs with the libraries are easy to reuse and maintain, and it is easy for developers to deploy, even though it is not the fastest.

Bibliography

[1] Kumar, Sumit and Dalal, Sumit and Dixit, Vivek. The OSI model: Overview on the seven layers of computer networks. *International Journal of Computer Science and Information Technology Research*, 2(3):461–466, 2014.

[2] Statista. Number of internet of things (IoT) connected devices worldwide from 2019 to 2030(in billions), 2022. URL https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/.

[3] Shelby, Zach and Hartke, Klaus and Bormann, Carsten. The constrained application protocol (CoAP), 2014. URL https://datatracker.ietf.org/doc/html/rfc7252.

[4] MQTT.org. MQTT: The Standard for IoT Messaging, 2022. URL https://mqtt.org/.

[5] Al Enany, Marwa O. and Harb, Hany M. and Attiya, Gamal. A Comparative analysis of MQTT and IoT application protocols. In *2021 International Conference on Electronic Engineering (ICEEM)*, pages 1–6, New York, NY, USA, 2021. IEEE CS Press. doi: 10.1109/ICEEM52022.2021.9480384.

[6] Laaroussi, Zakaria and Novo, Oscar. A Performance Analysis of the Security Communication in CoAP and MQTT. In *2021 IEEE 18th Annual Consumer Communications & Networking Conference (CCNC)*, pages 1–6, New York, 2021. IEEE CS Press. doi: 10.1109/CCNC49032.2021.9369565.

[7] Naik, Nitin. Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP. In *2017 IEEE International Systems Engineering Symposium (ISSE)*, pages 1–7, New York, 2017. IEEE CS Press. doi: 10.1109/SysEng.2017.8088251.

[8] OASIS Message Queuing Telemetry Transport (MQTT) TC. MQTT Version 5.0, 2019. URL https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html.

[9] International Organization for Standardization. ISO/IEC 7498-1: 1994 information technology–open systems interconnection–basic reference model: The basic model. *International Standard ISOIEC*, 74981:59, 1996.

[10] IBM. Connecting all the things in the Internet of Things. URL https://developer.ibm.com/articles/iot-lp101-connectivity-network-protocols/.

[11] Microsoft. IoT technologies and protocols. URL https://azure.microsoft.com/en-ca/solutions/iot/iot-technology-protocols/.

[12] CISCO. IoT and Security Standards and Best Practices. URL https://www.ciscopress.com/articles/article.asp?p=2923211&seqNum=6.

[13] Stefanec, Tomislav and Kusek, Mario. Comparing energy consumption of application layer protocols on IoT devices. In *2021 16th International Conference on Telecommunications (ConTEL)*, pages 23–28, 2021. doi: 10.23919/ConTEL52528.2021.9495993.

[14] Jon Postel. Rfc0768: User datagram protocol, 1980.

[15] Klaus Hartke. Rfc 7641: Observing resources in the constrained application protocol (coap), 2015.

[16] Stanford-Clark, Andy and Truong, Hong Linh. Mqtt for sensor networks (mqtt-sn) protocol specification. *International business machines (IBM) Corporation version*, 1(2):1–28, 2013.

[17] Bormann, Carsten and Lemay, Simon and Tschofenig, Hannes and Hartke, Klaus and Silverajan, Bilhanan and Raymor, B. CoAP (Constrained Application Protocol) over TCP, TLS, and WebSockets, 2018. URL https://datatracker.ietf.org/doc/html/rfc8323.

[18] Selander, Göran and Mattsson, John and Palombini, Francesca and Seitz, Ludwig. Object security for constrained restful environments (oscore), 2019. URL https://datatracker.ietf.org/doc/html/rfc8613.

[19] Aline Brito, Laerte Xavier, Andre Hora, and Marco Tulio Valente. APIDiff: Detecting API breaking changes. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 507–511, 2018. doi: 10.1109/SANER.2018.8330249.

[20] Jang, Minwoo and Kook, Joongjin and Ryu, Samin and Lee, Kahyun and Shin, Sung and Kim, Ahreum and Park, Youngsu and Cho, Eig Hyun. An efficient similarity comparison based on core API calls. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 1634–1638, 2013.

[21] Raza, Shahid and Shafagh, Hossein and Hewage, Kasun and Hummen, René and Voigt, Thiemo. Lithe: Lightweight Secure CoAP for the Internet of Things. *IEEE Sensors Journal*, 13(10):3711–3720, 2013. doi: 10.1109/JSEN.2013.2277656.

[22] Kovatsch, Matthias and Lanter, Martin and Shelby, Zach. Californium: Scalable cloud services for the Internet of Things with CoAP. In *2014 International Conference on the Internet of Things (IOT)*, pages 1–6, 2014. doi: 10.1109/IOT.2014.7030106.

[23] Yassein, Muneer Bani and Shatnawi, Mohammed Q. and Aljwarneh, Shadi and Al-Hatmi, Razan. Internet of Things: Survey and open issues of MQTT protocol. In *2017 International Conference on Engineering & MIS (ICEMIS)*, pages 1–6, 2017. doi: 10.1109/ICEMIS.2017.8273112.

[24] Atmoko, Rachmad Andri and Riantini, Rona and Hasin, Muhammad Khoirul. IoT real time data acquisition using MQTT protocol. *Journal of Physics: Conference Series, abstract = The Internet of Things (IoT) provides ease to monitor and to gain sensor data through the Internet [1]. The need of high quality data is increasing to the extent that data monitoring and acquisition system in real time is required, such as smart city or telediagnostic in medical areas [2]. Therefore, an appropriate communication protocol is required to resolve these problems. Lately, researchers have developed a lot of communication protocols for IoT, of which each has advantages and disadvantages. This study proposes the utilization of MQTT as a communication protocol, which is one of data communication protocols for IoT. This study used*

temperature and humidity sensors because the physical parameters are often needed as parameters of environment condition [3]. Data acquisition was done in real-time and stored in MySQL database. This study is also completed by interface web-based and mobile for online monitoring. This result of this study is the enhancement of data quality and reliability using MQTT protocol., 853(1):012003, may 2017. doi: 10.1088/1742-6596/853/1/012003. URL https://dx.doi.org/10.1088/1742-6596/853/1/012003.

[25] Thangavel, Dinesh and Ma, Xiaoping and Valera, Alvin and Tan, Hwee-Xian and Tan, Colin Keng-Yan. Performance evaluation of mqtt and coap via a common middleware. In *2014 IEEE Ninth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP)*, pages 1–6, 2014. doi: 10.1109/ISSNIP.2014.6827678.

[26] Heđi, Ivan and Špeh, I. and Šarabok, Antonio. IoT network protocols comparison for the purpose of IoT constrained networks. In *2017 40th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 501–505, 2017. doi: 10.23919/MIPRO.2017.7973477.

[27] Mawal Mohammed, Mahmoud Elish, and Abdallah Qusef. Empirical insight into the context of design patterns: Modularity analysis. In *2016 7th International Conference on Computer Science and Information Technology (CSIT)*, pages 1–6. IEEE, 2016.

[28] Dennis Kafura and Geereddy R. Reddy. The use of software complexity metrics in software maintenance. *IEEE Transactions on Software Engineering*, SE-13(3):335–343, 1987. doi: 10.1109/TSE.1987.233164.

[29] Chidamber, Shyam R. and Kemerer, Chris F. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994. doi: 10.1109/32.295895.

[30] McCabe, Thomas J. A Complexity Measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976. doi: 10.1109/TSE.1976.233837.

[31] Sci Tools. What metrics does understand have?, 2023. URL https://support.scitools.com/support/solutions/articles/70000582223-what-metrics-does-understand-have-2.

[32] Tanenbaum, Andrew S. and WETHERALL, DAVID J. Computer networks fifth edition, 2011.

[33] Peterson, Larry L. and Davie, Bruce S. 1 - foundation. In Peterson, Larry L. and Davie, Bruce S., editor, *Computer Networks (Fifth Edition)*, The Morgan Kaufmann Series in Networking, pages 1–69. Morgan Kaufmann, Boston, fifth edition edition, 2012. ISBN 978-0-12-385059-1. doi: https://doi.org/10.1016/B978-0-12-385059-1.00001-6. URL https://www.sciencedirect.com/science/article/pii/B9780123850591000016.