Comparison of IoT Application Layer Protocols

by

Pinchen Cui

A thesis submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Auburn, Alabama
April 2017

Keywords: IoT, Protocols, Comparison

Approved by

Anthony Skjellum, Chair, Professor of Computer Science and Software
David Umphress, Professor of Computer Science and Software
Richard Chapman, Associate Professor of Computer Science and Software

Abstract

It has been almost 20 years since the concept of the Internet of Things was first proposed. Now, IoT has become a leading technology that is becoming pervasive in our lives. However, behind the wide deployment of IoT services, there is still no consensus on common of IoT protocols and architectures. In this thesis, we study four application layer protocols: MQTT, CoAP, AMQP, DDS, which are representative of current practice. We introduce and compare these four protocols on the conceptual level. This study considers the message model, messaging functionality, and security. Moreover, we use several Raspberry Pi's in common IoT protocol test platform to perform performance tests. A series of experiments based on message quantity, message size and transmission frequency have been performed. The test results show the four protocols' messaging capability in term of latency, CPU utilization, and memory usage.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the past decade, the number of devices connected to the Internet has grown at an unprecedented rate. The concept of the Internet of Things (IoT) makes not only the computers, smart phones, but also cars, dishwashers, televisions and other common household appliances connect to the Internet. This is the time of Internet, Gartner Research states: "4.9 billion connected things in use in 2015. . . and will reach 20.8 billion by 2020" [9] Eventually, the number of devices will exceed the earth's total human population.

IoT's roots can be traced back to 1999. A Massachusetts Institute of Technology (MIT) group that worked on networked radio frequency identification introduced the concept of IoT [10]. As with many new concepts, IoT today is describing the network connectivity between so-called things or devices and enables these objects to collect and exchange data [11]. IoT applications have been widely deployed in our daily life. IoT has already made significant contributions to provide a better quality of life. (Fig. 1.1) illustrates the overall concept of IoT, Applications bring high-quality smart service to education, transportation, industry, health-care and business, among others.

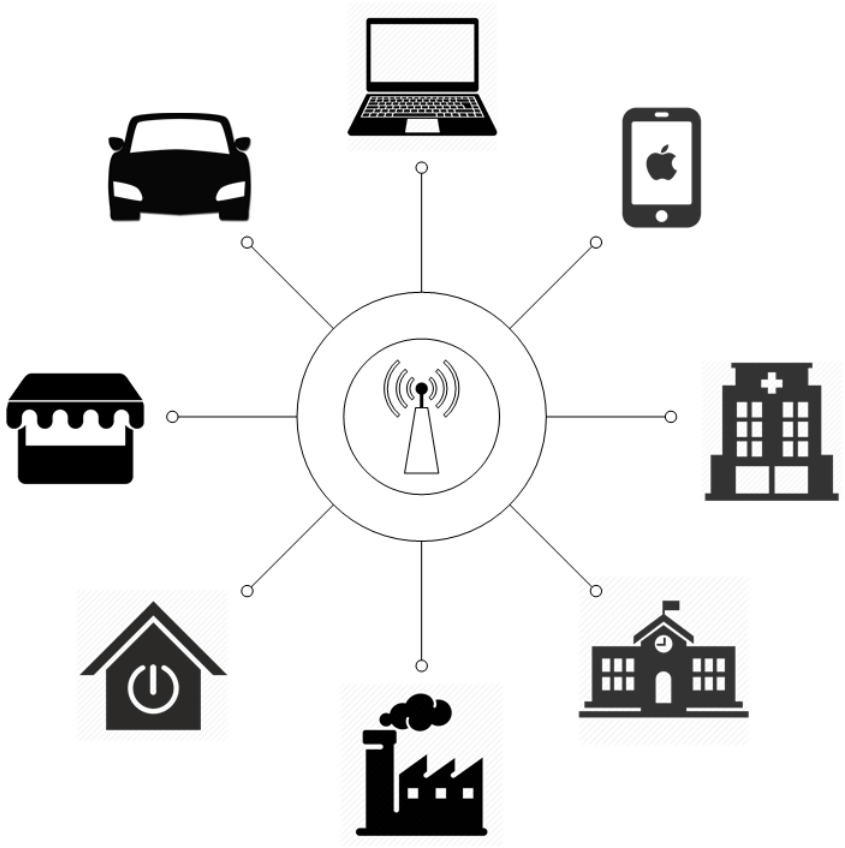

FIGURE 1.1: Overall picture of IoT

IoT networks almost always contains different types of devices including constrained-resource devices such as wireless sensors that cannot afford the overhead of using common protocols designed for resource-rich devices. As the consequence, new protocols were designed or streamlined to address resource constraints. To make Internet protocols more applicable to low-power, low-resource devices, a series of evolutionary steps have come to the protocol space. Change is taking place from the physical layer to the application layer. However, this thesis focuses only on application layer protocols (layer 4+ in the ISO/OSI Reference model). Because data exchange is the crucial in IoT, and the application layer provides the messaging functionality needed to make IoT services work, this is a good place to start. Other research will consider the changes occurring elsewhere in the communication stack.

A number of application layer protocols have been created to address a variety of requirements. Each such protocol focuses on different aspects of performance, functionality, overhead, and flexibility. To a greater or lesser extent, each addresses quality of service (QoS) and security. Therefore it could be a problem for the network or application designer, system architect, or

network manager who needs to concern himself/herself about many factors in choosing (respectively, managing) an application protocol or protocols[1] for a given IoT environment. On the other hand, it is also time assuming to become familiar with the features, implementations, performance, and security of all the application protocols. Providing insights on four such protocols is therefore a useful contribution of this thesis that can help such diverse group of designers, adopters, and managers.

Currently, many methods/protocols can be used to implement messaging capabilities needed in an IoT application, including WebSocket, Message Queueing Telemetry Transport (MQTT) [2], Extensible Messaging and Presence Protocol (XMPP) [12], Constrained Application Protocol (CoAP) [3], Advanced Message Queuing Protocol (AMQP) [4], Data Distribution Service (DDS)[7], HTTP, etc. However, this thesis will not cover all of the above methods. WebSocket [13], HTTP [14], and XMPP are all well-known standards and were not designed for IoT applications. Although they can be adapted for the IoT environment, the key limitation for these methods is that none supports a Publish/Subscribe architecture nor provides Quality of Service (QoS) options by default. Therefore, this thesis will focus on and compare MQTT, CoAP, AMQP and DDS, which are IoT-specific and/or provide publish/subscribe and QoS options.

There are several published survey papers regard IoT application layer protocols [1, 15–17], but most of them cover either the overall introduction or comparison of two (or three) protocols. However, this thesis go deeper and broader. The following chapters present the basic architecture, message format and security of these four protocols, and also including a comprehensive comparison regarding implementation, performance and security. The comparisons that result are based on a test platform for IoT application protocols, which is described in the experimental chapter (Chapter 3).

The remainder of this thesis is organized as follows: Chapter 2 offers a Conceptual Review and Comparison, including the basic background information of the four chosen IoT protocols, as well a short summary of conceptual comparisons that can found in Section 2.3. Chapter 3 covers Experimental Design including hardware and software selection, depicts of the experimental environment and offers a description of the experimental steps followed. Next, Chapter 4 provides Results and Discussion, while Chapter 5 offers Conclusions on this work.

[1] In many environments, multiple protocols will be needed for the foreseeable future because IoT systems work with different protocols and many sources of commodity-off-the-shelf parts will be used to build large IoT environments.

Chapter 2

Conceptual Overview and Comparison

This chapter offers a Conceptual Overview and Comparson of IoT protocols considered in this thesis.

2.1 IoT Architecture

First, we consider IoT Architecture overall, in a simplified form that is sufficient to explain what we undertake in our experimental comparisons that follow.

To help explain some of the terminology that follows, a list of definitions is helpful:

- edge device (or node) - devices such as sensors or actuators at the edge of the IoT system that sense or activate through connection to the physical world
- gateway - a networked computer that speaks directly with one or more edge devices under its control, aggregates data, potentially pre-processing it or doing analytics, and sharing it upstream with data consumers.
- broker - synonymous with gateway, a gateway device usually called broker in IoT environment.
- client-server [18] - client is the initiator of the connection and sends request to server; server is the service provider and responses to the client's request.
- publisher-subscriber [19] - publisher and subscriber are two standalone applications which both associate with a topic or list. Publisher is the the application publishes (sends) data to this topic or list; subscriber is the application subscribe to this topic or list and receives the corresponding publication from publisher.
- quality of service (QoS) [20] - the overall quality level of a network service performance, including delay, packet loss rate, error control and congestion control, etc.
- Transport Layer Security (TLS) [21] - An IETF standard aims to provide security for TCP/IP networks.

- Datagram Transport Layer Security (DTLS) [22]- An IETF standard aims to provide security for UDP/IP networks.
- Message authentication code (MAC) [23] - A piece of code generate from the message by using some particular algorithm, this code can be used to provide integrity and authentication.
- Hash-based message authentication code (HMAC) [24] - HMAC is a special type of MAC. It is based on hash function and a secret cryptographic key.
- Certification Authority (CA) [25] - A trusted third party issues digital certificates to the users. A certificate indicates the ownership of a corresponding public key.
- Kerberos [26] - An authentication protocol aims to provide strong authentication for client-server based network by using secret-key cryptography.
- Generic Security Services Application Program Interface (GSSAPI) [27] - An IETF standard API used to provide access to security service like Kerberos.
- Windows Challenge/Response (NTLM) [28] - A Microsoft security protocol aims to provide authentication, integrity, and confidentiality for the networks that include systems running the Windows operating system.
- Simple Authentication Layer Security (SASL) [29] - A framework aims to provide authentication and data security, it can be attached with other mechanism like kerberos or GSSAPI.

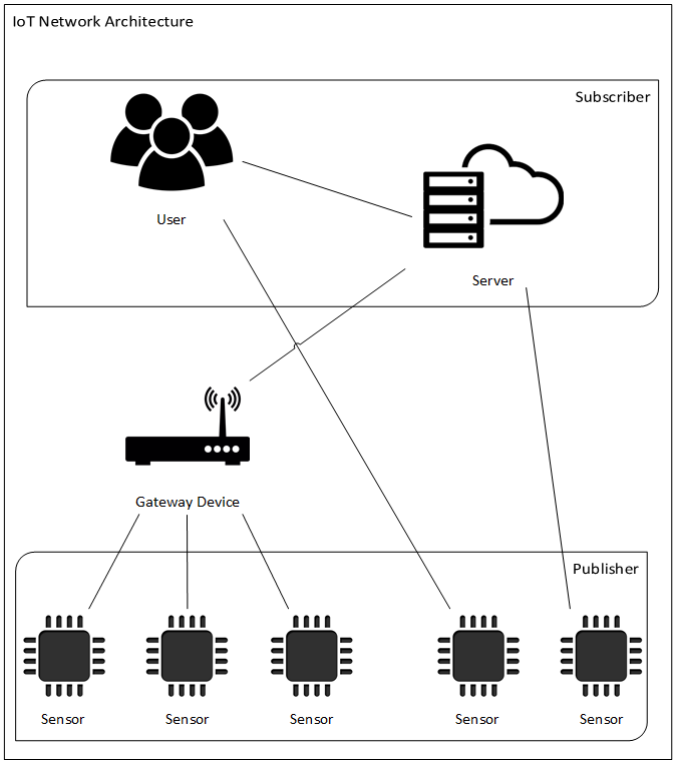

FIGURE 2.1: Architecture of IoT Network

An IoT network can be defined into two models: the Client-server model, and, the Publish-Subscribe model; however there always are two main components need to be implemented regardless of this particular choice [5]: 1) Data Consumer. A device performing as a subscriber/request sender that receives data from a data producer. 2) Data Producer. A device performing as a publisher/service provider that send data to the data consumer.

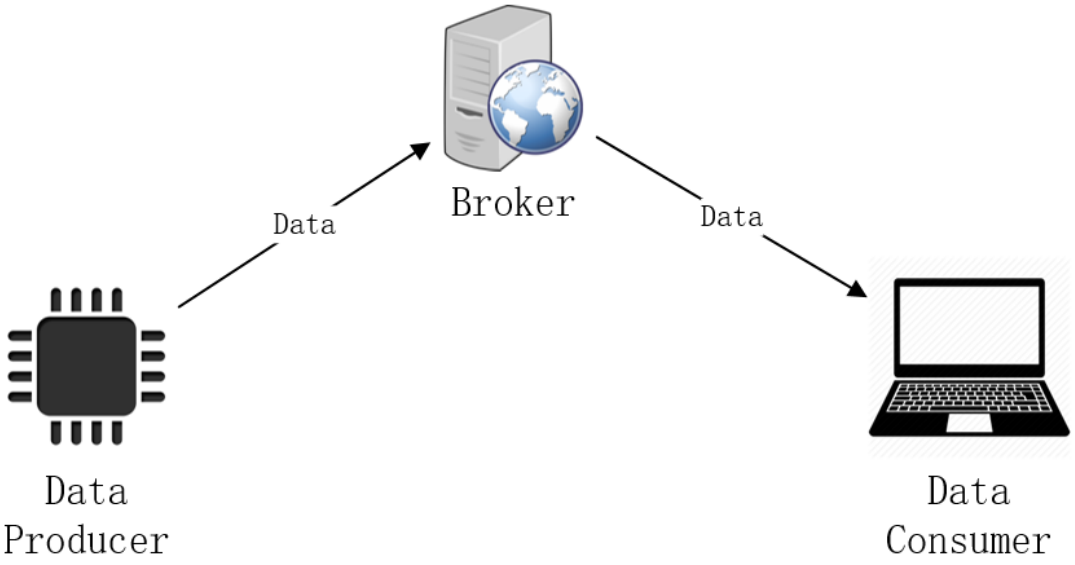

FIGURE 2.2: Broker Architecture of IoT

Normally, edge devices such as sensors act as as data producers, while the user and the remote server act as data consumers. A gateway device, namely the broker in the Publish/Subscribe model, may participate in the data transaction procedure (see Fig. 2.2), and in a brokerless architecture [30] (as shown in Fig. 2.3). As in CoAP and DDS, the gateway device will not participate in the data transmission [3, 7]; details will be introduced in the following sections. Normally, the gateway device can provide greater data processing power than the node devices and implement the processing or the pre-processing of the raw data acquired from the nodes. According to a given user's need, the communication pattern can sequence as a) User- Data consumer- Gateway- Data producer, b) User- Data consumer- Data producer, or c) User- Data producer (see Fig. 2.1).

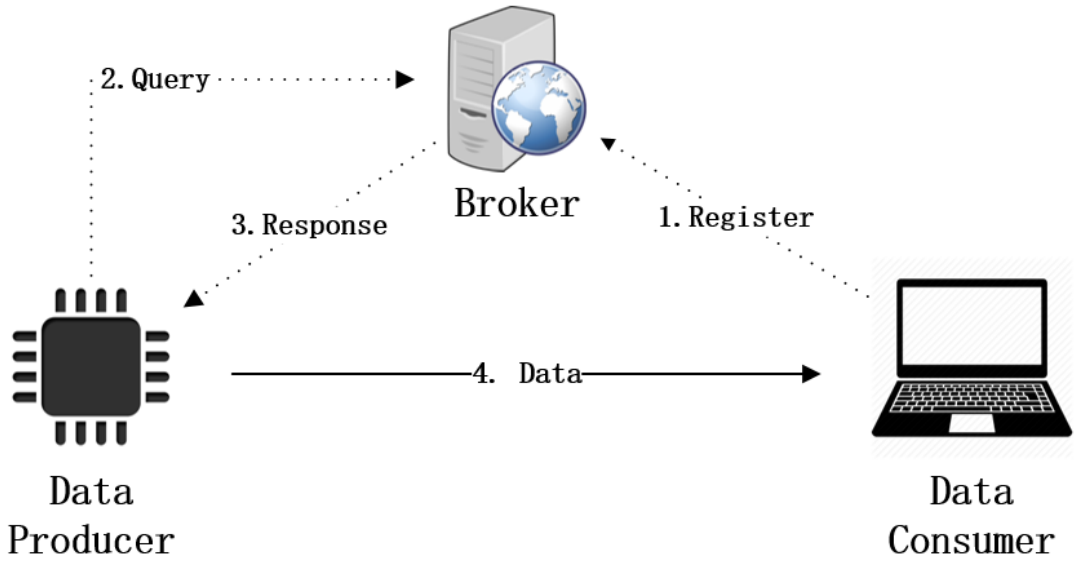

FIGURE 2.3: Brokerless Architecture of IoT

2.2 Application-Layer Protocols

The application layer [1] provides services for users. For instance, it can provide temperature data to the user who asks for it regarding a particular place in an infrastructure (like a room). The importance of this layer is it has the ability to provide high-quality services to meet users' needs. The application layer protocols define the messaging capability and effect the performance of the service.

The Four selected application protocols are as follows: Message Queueing Telemetry Transport (MQTT), Constrained Application Protocol (CoAP), Advanced Message Queuing Protocol (AMQP), Data Distribution Service (DDS) will be introduced and conceptually compared in this chapter. As mentioned previously, they are particularly relevant to IoT, as compared to many other protocols used in the pre-IoT era.

2.2.1 Message Queueing Telemetry Transport (MQTT)

MQTT is a lightweight publish-subscribe messaging protocol running on the top of TCP/IP. MQTT aims to provide ordered, lossless, and bidirectional connections for embedded devices and networks [2]. The first version of MQTT was created by Andy Stanford-Clark and Arlen Nipper in 1999, it was a part of IBM's MQ Series message queuing product line. In 2013, IBM submitted MQTT v3.1 to the OASIS specification body and, since then, MQTT has been a standardized protocol.

The publishe-subscribe model of MQTT is based on the Client-Server model, but the server running MQTT is more like a broker or gateway [2]. The MQTT client can publish a message on a topic or subscribe to a topic, and the MQTT server/broker will handle the messages that clients publish and forward them to the corresponding subscriber clients (see Fig. 2.4). The broker will store the list of topics and process the subscribe/unsubscribe requests from clients. Each topic will have a topic name as its label, and clients will send the subscription with a topic filter that include one or more topics.

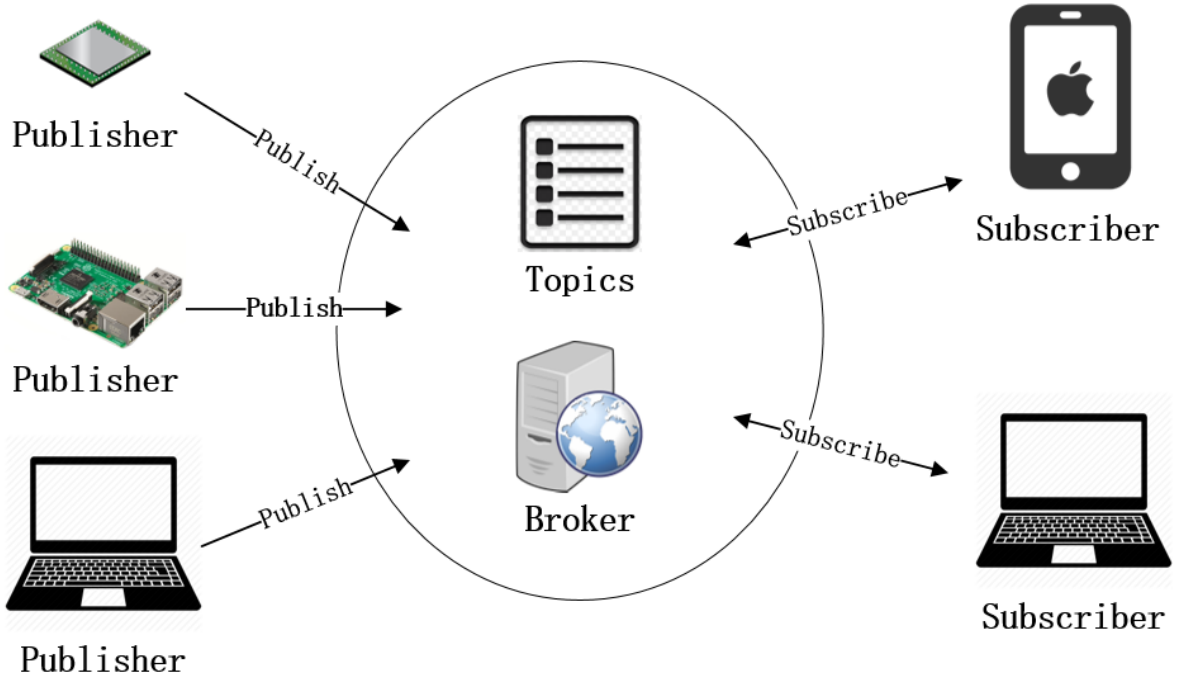

FIGURE 2.4: Architecture of MQTT (Adapted from [1])

Finally, the publish/subscribe procedure and QoS options all rely on MQTT control packets, which will be introduced in the following section.

2.2.1.1 Message Format and Message Model

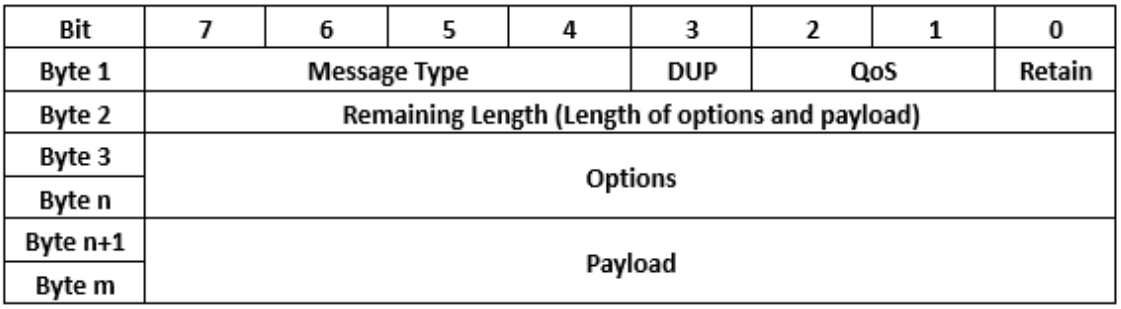

FIGURE 2.5: Packet Format of MQTT (adapted from [2])

Fig. 2.5 illustrates the message format of MQTT. First field is retain field. The second field is used for QoS options. Since MQTT is running on the top of TCP, MQTT can deliver message with three QoS levels [2]:

QoS 0: "At most once," where messages are not assured to arrive, lost may occur;

QoS 1: "At least once," where messages are assured to arrive but duplicates can occur; and,

QoS 2: "Exactly once," where message are assured to arrive exactly once with no duplication or loss. The QoS messaging examples can be found in Fig. 2.6, 2.7, and 2.8.

And, the third field, the DUP field, is used to indicate the duplication. Every resend message needs to set DUP field to 1.

The last field in the header is the type field used to declare the message type. And MQTT message has fourteen types, by using all these types MQTT can perform the whole messaging capability [2]:

Field Value 1, CONNECT: Client to Server, request to connect to server.

Field Value 2, CONNACK: Server to Client, connect acknowledgment.

Field Value 3, PUBLISH: Bidirectional, publish a message.

Field Value 4, PUBACK: Bidirectional, publish acknowledgement.

Field Value 5, PUBREC: Bidirectional, publish received.

Field Value 6, PUBREL: Bidirectional, publish released.

Field Value 7, PUBCOMP: Bidirectional, publish complete.

Field Value 8, SUBSCRIBE: Client to Server, request for subscription.

Field Value 9, SUBACK: Server to Client, acknowledgement of subscription.

Field Value 10, UNSUBSCRIBE: Client to Server, request for unsubscription.

Field Value 11, UNSUBACK: Server to Client, acknowledgement of unsubscription.

Field Value 12, PINGREQ: Client to Server, ping request.

Field Value 13, PINGRESP: Server to Client, ping response.

Field Value 14, DISCONNECT: Client to Server, request to disconnect.

Using most types are straightforward and easy to understand. Here, we will only introduce the use of PUBREC, PUBREL, PUBCOMP that work with the QoS options.

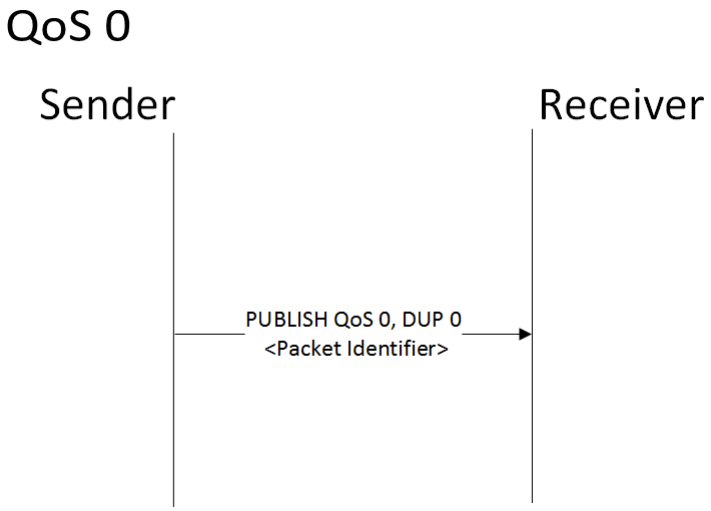

FIGURE 2.6: Messaging Example of QoS 0 (adapted from [2])

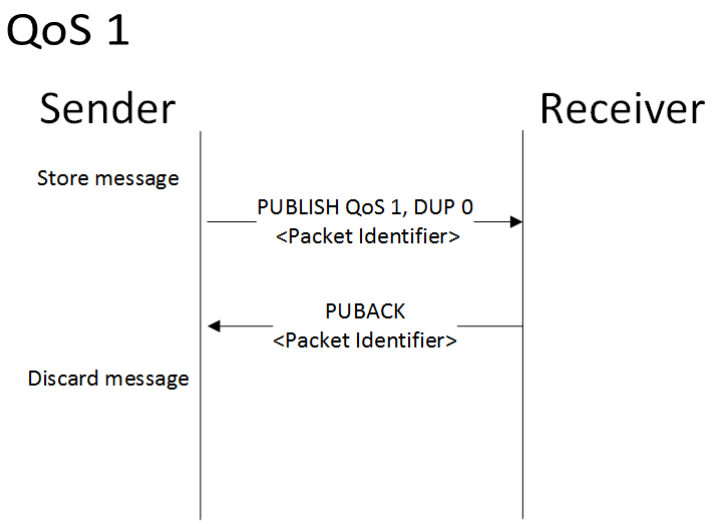

FIGURE 2.7: Messaging Example of QoS 1 (adapted from [2])

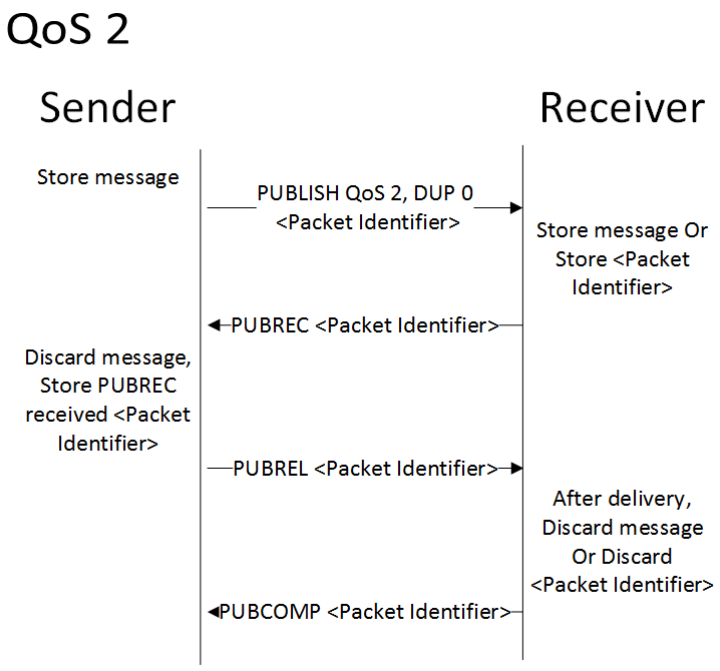

FIGURE 2.8: Messaging Example of QoS 1 (adapted from [2])

2.2.1.2 Security

MQTT is based on TCP/IP and its security relies on Transport Layer Security (TLS) [2]. TLS [21] is the successor of Secure Sockets Layer (SSL); it aims to provide security and integrity over computer networks. Authentication is based on certificates and privacy is based on an asymmetrical key crypto algorithm. A secure connection is started by a client: client sends the connection set up request to the server with a list of supported encryption options. The server picks one of the options and notifies client. The server also sends back its digital signature including the server name, CA and server public key. The client verifies the certificate and generates the session key by using the server's public key; then, the client sends it to server and the secure connection has been established.

Each cipher suite adopted by TLS [21] should contain four components: 1) Key Exchange Algorithm, 2) Authentication Algorithm, 3) Bulk Encryption Algorithm, 4) Message Authentication Code. For instance, TLS-ECDHE-ECDSA-WITH-AES-256-CBC-SHA384, ECDHE is the key exchange algorithm; ECDSA is the authentication algorithm; AES-256-CBC is the bulk encryption algorithm; and SHA384 is the MAC algorithm. If any of the components has been proven not to be secure, then only this algorithm need to be changed in future server-side code (in terms of what can be accepted), and in future client-side code (in terms of what is offered).

A series of cipher suites can be used in TLS but there is no mandatory security requirement in the MQTT specification; thus, the security of each implementation is done according to its design. However, by default the CONNECT packet may include username and password fields that can be used for client authentication and authorization.The PUBLISH packet may include a hash value to provide for integrity [2].

2.2.2 Constrained Application Protocol (CoAP)

In order to understand CoAP, the concept of Representational state transfer (REST) and RESTful is best introduced first. REST [31] is a kind of web architecture defined by Roy Fielding in 2000, which is an HTTP-based *stateless* Client-Server communication model. Resources are stored in the server, and each resource has an Uniform Resource Identifier (URI) as its tag and address. The client that is interested in the resource needs to visit the web address such as: `http://example.com/resources/URI`, and the representation displayed in the browser is the capture of the current state of the referenced `URI` resource [32]. All operations on resources are based on four methods in HTTP, which are GET, PUT, POST and DELETE [14].

CoAP was created by the IETF Constrained RESTful Environments (CoRE) working group [3]. Unlike REST and MQTT, CoAP works on top of UDP, which makes it more suitable for

constrained devices. Meanwhile, CoAP is also a RESTful protocol, which reduces the complexity of the implementation and the communication overhead as compared to HTTP [33]. Since UDP does not provide transmission reliability, CoAP adds an exponential back-off feature to provide a measure of reliability. Therefore CoAP can be divided into two layers; while the message layer add the UDP and asynchronous interactions, the request/response layer is dealing with the RESTful communications [3].

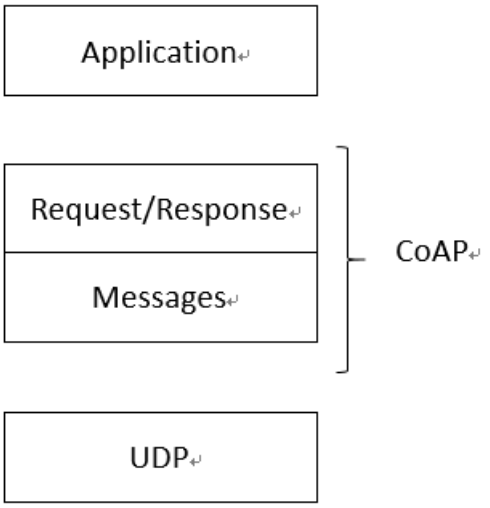

FIGURE 2.9: CoAP Layering (adapted from [3])

2.2.2.1 Message Model

The CoAP messaging model consists of four types of messages [3]: Confirmable (CON), Non-Confirmable (NON), Acknowledgement (ACK) and Reset (RST). The Confirmable message provides reliability; the message will keep being retransmitted until the sender receive a corresponding Acknowledgement message. If the message does not need reliability, the Non-Confirmable message can be used. Both CON and NON message provide duplication detection by adding a message ID in the header; when the recipient is unable to process the message, it can reply with a Reset message.

The request with four basic methods (GET, PUT, POST, DELETE) can be carried by a CON or a NON message [3]. CoAP provides three different types of responses. If the request is in a CON message and the resource is immediately available, then a Piggybacked response will be used, which means the response is carried by the ACK message. But when the server need a longer time to obtain the representation of the resource, and in order to avoid the situation that client keep sending the same request, the server may use the Separate response. The server can reply with the ACK as a empty message to inform the client that the request is received.

When the response is available, response will be sent back to the client in a CON message. Furthermore, if the request is in NON, response will be a Non-Confirmable response, a NON message will carry the response back to the client.

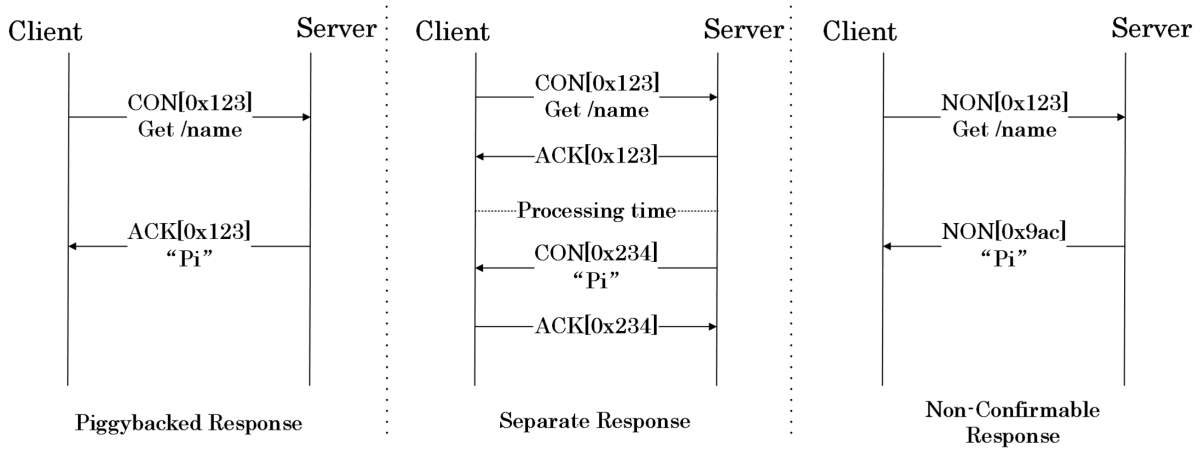

FIGURE 2.10: CoAP request and response (adapted from [3])

Caching for requests and responses may be enabled by the CoAP endpoints. In some cases, the request/response pair can be reused by using the prior request/response payload (supposing the client is checking the state of the same resource and its state has not changed); this feature increases performance by reducing response time and reducing network bandwidth consumption.

CoAP also provides an Observer mode (see Fig. 2.11) that is similar to the subscribe-publishe model. If a client is consistently interested in a resource and its state changes, then the client can register as a observer of said resource. The server maintains a list of observers for each resource and when the state of the resource has been changed, the server will notify these observers [33].

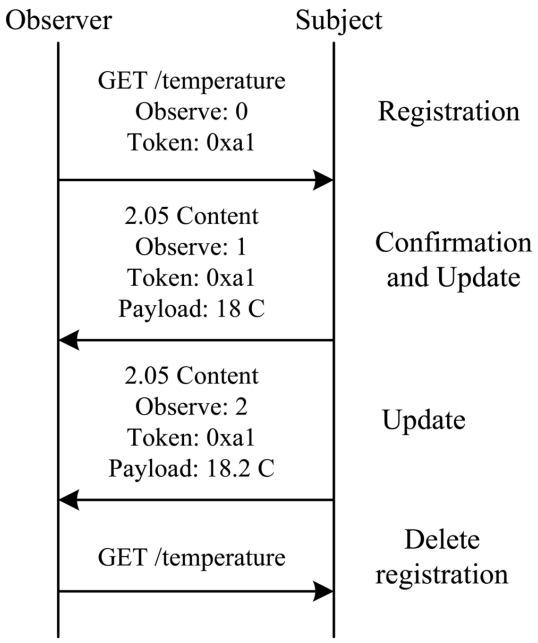

FIGURE 2.11: CoAP Observer Mode (Adopted from [3])

However, all above considers regular size payloads. If, however, the application needs to transfer a larger payload such as a high-resolution picture, then the payload apparently should be fragmented. Since TCP/IP does the work of segmenting and resequencing, but UDP/IP only provides the fragmentation. Therefore CoAP has the Block option to provide segmentation and resequencing [34]. The block option contains four pieces of information: request or response, block size, whether there are blocks following this block, and the relative sequence in all of the blocks. This option enables CoAP to support larger size payloads and still maintain stateless communication.

2.2.2.2 Message Format

Byte	0								1								2								3							
Bit	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
Field	Ver		T		TKL				Code								Message ID															
	Token (if any, TKL bytes)																															
	Options																															
	1	1	1	1	1	1	1	1	Payload																							

FIGURE 2.12: CoAP message format (Adapted from [3])

The header is four bytes long with five fields. The first field is a two bit Version field, indicating the version of CoAP. The second field is a 2 bit Type field indicating the type of message: (0,

CON), (1, NON), (2, ACK), (3, RESET). The third field is a 4 bit Token Length field indicating the length of the Token field. The fourth field is the 8 bits in length and is the Code field; this indicates the request method or response code. The last field is the Message ID field which is 16 bits long, used mainly to detect duplication. Following the header, there may be a Token field used to correlate request and response, and after the Token field, the optional Options field and Payload may be attached.

2.2.2.3 Link Format and Resource Discovery

As mentioned above, CoAP is a RESTful protocol. Thus all the resources can be considered as links, and the CoAP link format extends the HTTP link format header [35]. In the IoT world, users always need the devices to communicate with each other smoothly without manually configuration. But how is the client to know what resources are exist and what are the links of these resources? To address this need, CoAP provides a "well-known" interface accessed by querying the URI `/.well-known/core|`, namely an inquirer sends a GET request to `/.well-known/core`, then the server will respond with the list of all available resources known to it. And, in the meantime, if the endpoint wants to be discoverable, it can send POST requests to `/.well-known/core` as a registration of itself [36]. This feature makes CoAP usable in a brokerless model.

2.2.2.4 Security

CoAP runs on the top of UDP, so the security is relies on the Datagram Transport Layer Security (DTLS) [22] (the work of using IPsec on CoAP has been abandoned [37]). DTLS is security protocol works for datagram-based communications. It is based on TLS and can provide similar security guarantees.

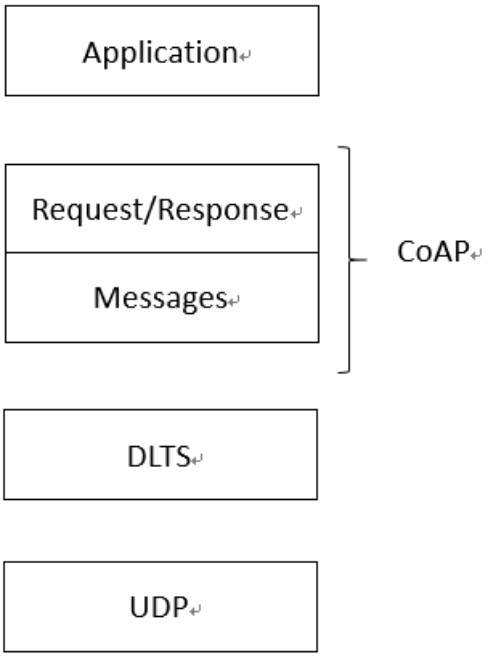

FIGURE 2.13: DTLS Secured CoAP (adapted from [3])

CoAP provides four security modes: NoSec, PreSharedKey, RawPublicKey, and Certificate. These are as follows:

NoSec: the DTLS is not enabled;

PreSharedKey: the connection will be encrypted by a pair of pre-shared key, and all the implementation of CoAP must support cipher suite TLS-PSK-WITH-AES-128-CCM-8.

RawPublicKey: the connection will be encrypted by a pair of asymmetric keys, and all the implementation of CoAP must support cipher suite TLS-ECDHE-ECDSA-WITH-AES-128-CCM-8.

Certificate: the connection will be encrypted by a pair of asymmetric keys with an X.509 Certificate. The certificate is binding to the subject and signed by a trust root.

However, two IETF drafts [38, 39] are still working on CoAP security, and these will be introduced in the comparison section.

2.2.3 Advanced Message Queuing Protocol (AMQP)

AMQP is an open standard application protocol for message-oriented middleware. AMQP can provide reliable and secure communication for both the peer-to-peer model and the publish-subscribe model.

AMQP defines a type system to standardize AMQP message that makes all AMQP implementations interoperable even they are written in different languages [4] . In simple terms, the message will always come with a constructor to declare the message type, for instance, URLs in UTF-8.

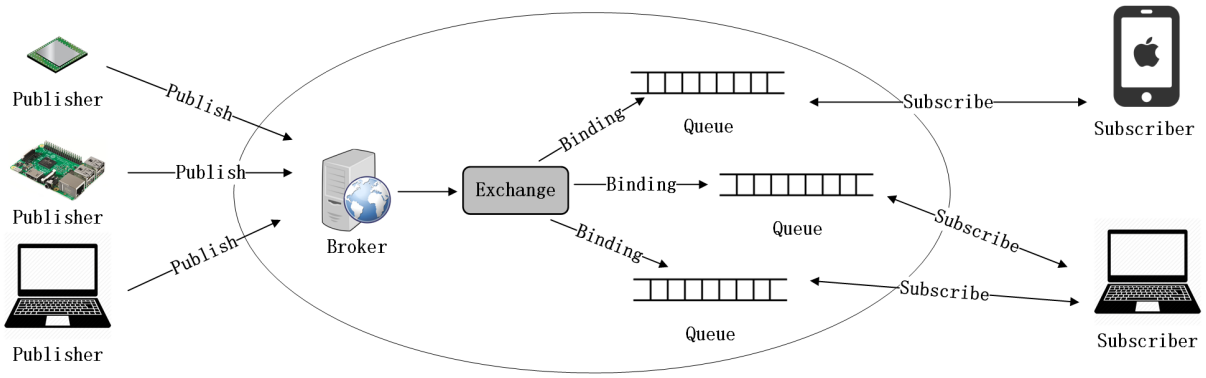

FIGURE 2.14: Architecure of AMQP (adapted from [1])

At the functional level, an AMQP broker has three components: Exchange, Queue, and Binding. These three can be used for routing and queueing. Exchange receives and forwards messages sent from the Publisher; Queue stores then forwards messages to the Subscriber; Binding describes the relationship between Exchange and Queue, which connect Exchange to appropriate Queue. More details will be introduced later in subsection 2.2.3.3 below.

At the architectural level, AMQP is made up of two layers [5]. The lower layer is the transport layer, which handles channel multiplexing, framing, content encoding, heart-beat, data representation, and error handling. Above it, the function layer which provides messaging capabilities.

2.2.3.1 Transport Layer

In this layer, AMQP defines its own link protocol, and the communication in this layer is frame-oriented [4]. Briefly, an AMQP connection can be divided into several channels, and the communication in these channels is called a session; each session can be attached by several links. Only the link will transfer messages; both session and connection are dealing with frames.

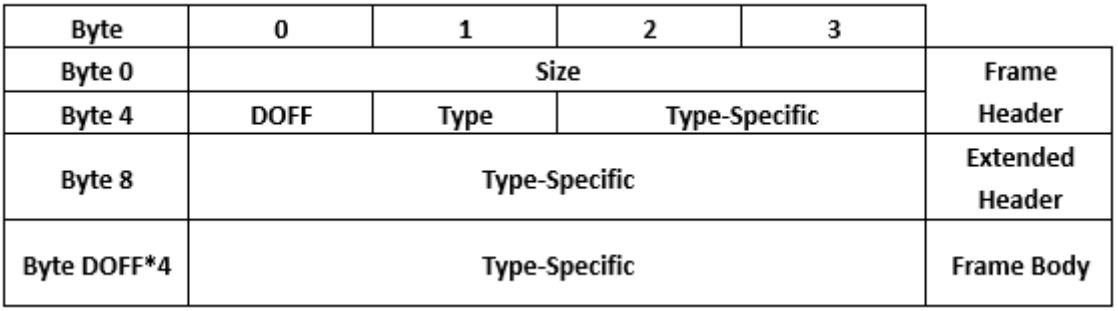

FIGURE 2.15: AMQP frame (adapted from [4])

An AMQP frame can be divided into three parts: frame header, extended header, and frame body. Only the frame header has a fixed width which is eight bytes and the header includes three fields: SIZE, DOFF and TYPE. The SIZE field contains the total size of the frame; the DOFF field contains data offset which gives the position of the frame body within the frame; and, the TYPE field contains the value to indicate the type of this frame, and there are nine kinds of frame types: OPEN, BEGIN, ATTACH, FLOW, TRANSFER, DISPOSITION, DETACH, END, CLOSE.

The nine kinds of frames will be used in sequence: OPEN the connection, BEGIN a session on the channel, ATTACH a link to this session, use FLOW to update the link state, then use TRANSFER to send a message. DISPOSITION will be used to inform remote peers of delivery state changes. And DETACH will be used to detach a link from a session, similarly END is used for ending a session while CLOSE will close the connection.

2.2.3.2 Function Layer

AMQP defines two terms to describe its messages [4]: Bare message and Annotated message. Bare messages can be considered as the message payload sent by the sender while the Annotated message is the message as seen at the receiver. A bare message includes three sections: standard properties, application-properties, and application-data. An annotated message also includes three sections: head annotation, bare message, and tail annotation.

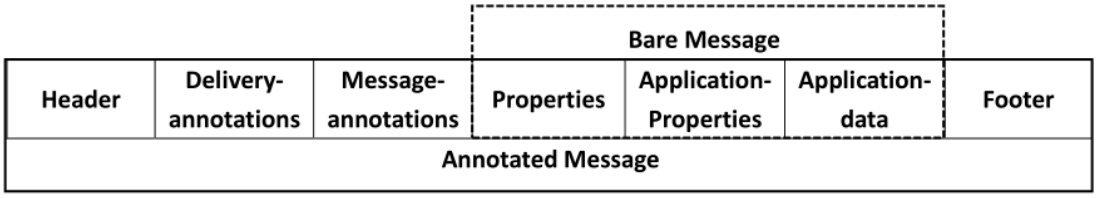

FIGURE 2.16: AMQP Message Format (adapted from [4])

Except for the bare message section, all the other sections can be omitted, and the omitted sections will be considered as filled with default values or empty. The Header section provides Priority/Time to Live/Delivery count of this message. The Delivery/Message Annotation section works as the extended section of the Properties section. The annotation section conveys delivery/message information not included in the Properties section. The Properties section of the bare message provides message ID/Source/Target/Subject/content type/expiration time/creation time of this message. The Application Properties section is used for structured application data, and intermediaries can use the data within this structure for the purposes of filtering or routing. The Footer section provides the message hashes, HMACs, signatures and encryption details.

2.2.3.3 Message-oriented Features

AMQP describes itself as a message-oriented protocol, and it provides some message-oriented features. Generally, AMQP provides two modes of message operation: the first one is Browse Mode in which the client can browse the messages stored in the queue without consuming these messages; the second one is the Consume Mode in which every message has been browsed by client would be consumed and deleted from the queue. Furthermore, AMQP also provides a Durable message feature. This feature can provide message durability when either the client or server dies or quits, and the messages can still be stored in the queue.

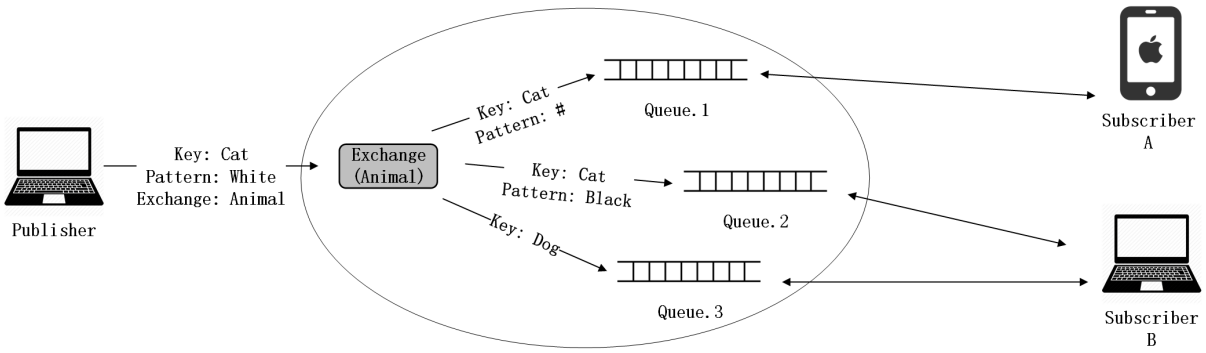

FIGURE 2.17: AMQP Messaging Features (adapted from [5, 6])

However, the most important part is the Exchange-Binding-Queue architecture, which provides much more flexible message distribution. According to the type of Exchange [6], several different levels of message distribution can be performed. And, no matter what type the Exchange should be, it must be bound with at least a Queue.

Each Binding contains a Binding Key, as in the example Fig. 2.17. The Exchange name is *Animal*, and the Binding key could be *Cat* or *Dog*. And the Binding Key can also associate with a Binding Pattern: *White* or *Black*. The specific message sent from the Publisher would contain a Routing Key: *Cat*, a Routing Pattern: *White*, and the Exchange name *Animal*.

Direct Exchange: This Exchange will only compare the Routing Key with the Binding Key, if these two keys match then the Exchange will route the message to Queue. If Exchange *Animal* is a Direct Exchange, then the message will be routed to Queue 1 and Queue 2.

Topic Exchange: This Exchange will compare both Key and Patterns, if Routing Key matches the Binding Key, and the Routing Pattern matches Binding Pattern, then the Exchange will route the message to Queue. If Exchange *Animal* is a Topic Exchange, then the message will be routed to Queue 1. The Pattern # means matching all the Patterns.

FanOut Exchange: This Exchange ignores the Routing Key, Binding Key and the Routing Pattern. It always routes all the messages to every Queue binds to this Exchange. This is the simplest distribution type, providing the same function like broadcast. If Exchange *Animal* is a FanOut Exchange, then the message will be routed to Queue 1, 2 ,3.

Header Exchange: This is a special Exchange type, which will ignore the Routing Key, Routing Pattern, and Binding Pattern. But the Binding Key is still active. An example is shown in Fig. 2.18 for a message with optional Header value `Format: .png, Size: Small` has been sent by Publisher. The `x-match=Any` means if there is any match to Format or Size then the message will be forwarded. The Exchange Figure receives the message and will compare the

Format and Size with the Binding Key, since the match option is Any, thus the message will be routed to Queue 1 and Queue 2.

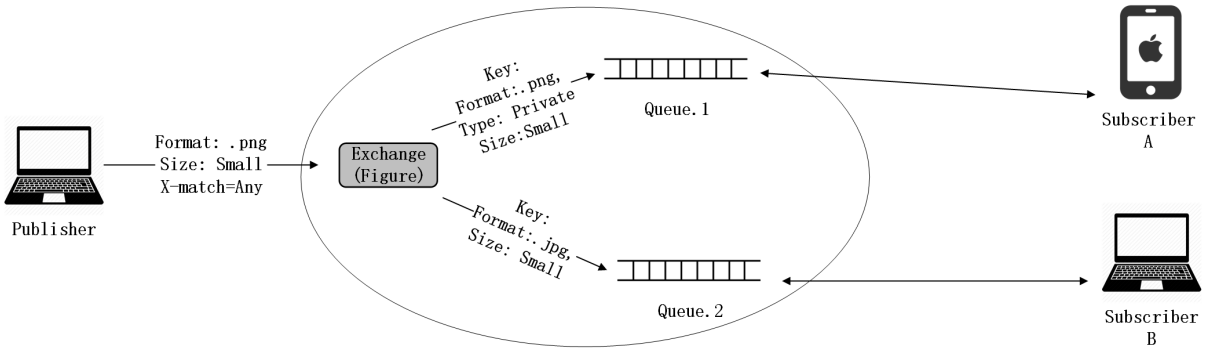

FIGURE 2.18: AMQP Messaging Features (adapted from [5, 6])

2.2.3.4 Security

There are two security choices in AMQP; the first is TLS, and another one is Simple Authentication and Security Layer (SASL). Both TLS and SASL provides authentication, data signing and encryption. An obvious difference between TLS and SASL is that SASL allows the user to do the authentication based on many mechanisms like GSSAPI [27], Kerberos [26], NTLM [28], and others. In contrast, TLS perform the authentication based on certificates. Generally, AMQP uses SASL to provide authentication and uses TLS to secure the tunnel. Note that there is no mandatory security requirement in the AMQP specification [4]; thus, the security provided depends on the implementation.

2.2.4 Data Distribution Service (DDS)

DDS is a soft real-time, publish-subscribe communication architecture [7] standardized by the Object Management Group (OMG). In contrast to MQTT and AMQP, DDS has a brokerless architecture and uses multicast to provide reliable and efficient communication.

2.2.4.1 Message Model

DDS can be divided into two layers [7]: Data-Centric, Publish-Subscribe (DCPS) and Data-Local, Reconstruction Layer (DLRL). DCPS is the layer providing the messaging capability; DLRL is the optional layer above DCPS, which handles the integration with applications and provides certain advanced features. DCPS is the core of DDS; DLRL can be considered as an add-on; we only consider DCPS.

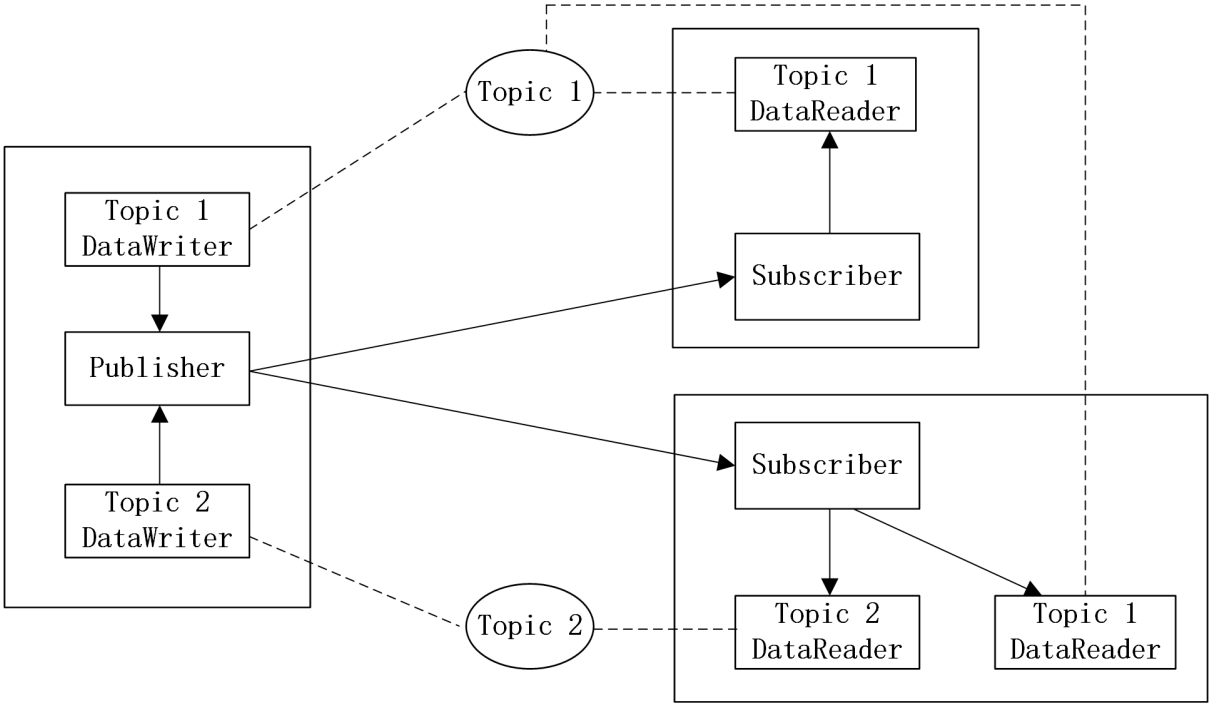

FIGURE 2.19: DDS architecture (adapted from [7])

The DDS communication model [7] consists of five components (see Fig. 2.19): Publisher and DataWriter on the sending side; Subscriber and DataReader on the receiving side; and, both side related to the Topic. Moreover, DDS defines the communication network space as a Domain' each container that contains a Publisher/Subscriber is called a DomainParticipant.

The Publisher is used for data distribution, which may publish different types of data whereas each DataWriter attached to Publisher is binding to one type of data. The Subscriber is used for receiving different types of data and make it available for the receiving application; Each DataReader attached to a Subscriber is binding to one type of data. Therefore, the context of publication/subscription is provided by Publisher/Subscriber, and described by DataWriter/-DataReader. The DataWriter/DataReader associates with the Topic, each Topic associates with a topic name, a data type and a QoS level. DDS provide 23 different QoS levels to deal with security, urgency, priority, durability, and reliability. DDS also include a built-in discovery service that allows Publisher/Subscriber to discover the existence of each other dynamically without contact to any name server.

2.2.4.2 Message Format

DDS does not define its own wire-protocol but use the Real-Time Publish Subscribe (RTPS) protocol as its wire protocol [8]. The RTPS protocol is designed to run over multicast and

connectionless best-effort transports such as UDP/IP, however RTPS can also runs over TCP in specific implementation [40].

Byte	0	1	2	3
Byte 0	Header			
Byte 16	SubMessage			
--- ---	--- ---			
Byte n	SubMessage (If any)			

FIGURE 2.20: RTPS Message Format (adapted from [8])

The RTPS message is composed of a leading Header followed by a variable number of Submessages [8] (see Fig. 2.20). The header of the RTPS message is 16 bytes long with five main sections (see Fig. 2.21). First is the Protocol field that is 4 bytes long; the value needed to be set to "RTPS"; the second field is the Version field that is two bytes long and is used to indicate the version of RTPS; The third field is the VenderId field, which is 2 bytes long. Each implementation of RTPS has to obtain a vendorId from OMG; The last field is GuidPrefix, which is 8 btyes long; each entity (Publisher/Subscriber/DataWriter/DataReader) has its own Globally Unique Identifier (GUID) and this field will contains this Guid to indicate source/target as appropriate.

Byte	0	1	2	3
Byte 0	R	T	P	S
Byte 4	Version		Vendor ID	
Byte 8	GuidPrefix			
Byte 12				

FIGURE 2.21: RTPS Message Header Format (adapted from [8])

The submessage contains two main parts [8]: submessage header and payload (see Fig. 2.22). The submessage header includes three fields: SubmessageId field used to indicate the type of the submessage. There are total 12 types of submessage; The flag field can be used by different types of submessage to provide some functions; and, the octecsToNextHeader field indicates the payload length.

Byte	0	1	2	3
Byte 0	Submessage ID	flags	octecsToNextHeader	
Byte 4	SubMessage Payload			

FIGURE 2.22: RTPS Submessage Format (Adapted from [8])

There are 12 types of submessages, as follows:

1) Data message contains information regarding the value of an application Data object.

2) DataFrag message allows the data to be transmitted as multiple fragments to overcome any oversize problem;

3) HeartBeat message sent by the Writer to verify the availability of Reader;

4) HeartBeatFrag message sent by the Writer to indicate what fragments are available;

5) GAP message sent by the Writer to indicate that information is no longer relevant to the Reader;

6) AckNack message sent by the Reader to indicate the delivery state;

7) NackFrag message used by the Reader to indicate what fragment is missing;

8,9) Info-SRC/Info-DST message used to modify the logical source/destination of the submessages following this submessage;

10) Info-Reply message used to indicate where to send the reply to this submessages and the submessages following this submessage;

11) InfoTimeStamp message provides a source timestamp; and,

12) PAD message is used for padding.

2.2.4.3 Security

DDS defines its security model as a plugin-based model [41], this model can work with five Service Plugin Interfaces (SPI) to provide information assurance. The five SPIs are as follows: Authentication, AccessControl, Cryptographic, Logging, and DataTagging. Users can either build their own SPIs or use the DDS built-in SPIs (however, DataTagging has no built-in version). Details follow.

Definition of Authentication Plugin [41]: Verifies the identity of the application and user that invokes operations on DDS. Includes functions to perform authentication between participants and establish a shared secret.

DDS-built-in Authentication Plugin, DDS:Auth:PKI-DH [41]. Implements authentication by using a trusted Certificate Authority (CA). It performs mutual authentication by using the RSA or ECDSA Digital Signature Algorithms and establishes a shared secret by using Diffie-Hellman (DH) or Elliptic Curve Diffie-Hellman (ECDH) Key Agreement Methods.

Definition of AccessControl Plugin: Enforces policy decisions on what DDS related operations that an authenticated user can perform. For instance, which Domains it can join, which Topics it can publish or subscribe to, etc.

DDS-built-in AccessControl Plugin, DDS:Access:Permissions [41]. Implements the AccessControl by using a permissions document signed by a shared CA. Permission document is a XML format file including Domain rules and Topic rules. For example, whether to allow unauthenticated participants to join this domain and which Reader/Writer has access to what Topic.

Definition of Cryptographic Plugin: Implements all cryptographic operations including encryption, decryption, hashing, digital signatures, etc.

DDS-built-in Cryptographic Plugin, DDS:Crypto:AES-GCM-GMAC [41]. Provides authenticated encryption using Advanced Encryption Standard (AES) in Galois Counter Mode (AES-GCM). It supports two AES key sizes: 128 bits and 256 bits. It may also provide additional reader-specific message authentication codes (MACs) using a Galois MAC (AES-GMAC).

Definition of Logging Plugin: Records all the DDS security-relevant events.

DDS-built-in Logging Plugin, DDS:Logging:DDS-LogTopic [41]. Implements logging by publishing information to a DDS Topic BuiltinLoggingTopic. All the participants need to have access to this topic, and it is predefined in permission document.

Definition of Data Tagging Plugin: Adds tags to data samples.

2.3 Comparison of Protocols

While he four protocols have been introduced in the above sections, this section will focus on their conceptual comparison. This section will not cover the detail performance data or security factors, but only compare the protocols on the concept level. A more detailed comparison will be covered in Chapter 3 based on our experimental results.

IoT is a relatively new, immature technology and not all the protocols that can be presently used in IoT were custom designed for IoT. Table 2.1 illustrates the short history of these four protocols. We note that only MQTT and CoAP were designed for IoT environment.

Table 2.1: Protocol History

Protocol	Proposed	Standardized	Develop Motivations
MQTT	1999	2013	A publish-subscribe based protocol that aims to provide messaging functionality for constrained devices and high latency networks.
CoAP	2013	Core- 2013 Features are still drafts	A RESTful application layer protocol designed to provide low-overhead Machine to Machine (M2M) communication.
AMQP	2003	2011	A message-oriented protocol designed to provide routing, queueing, reliability and security over a publish-subscribe model.
DDS	2001	2003	A publish-subscribe, soft real-time service designed to provide high performance communication over variable transport protocols and platforms.

2.3.1 Implementation and Functionality

The original motivation does not affect much; the implementation and deployment are also factors that can determine whether a protocol is good or not. And, as we know, some features of CoAP are still in drafts; thus the deployment of CoAP obviously cannot be as wide as other protocols. Table 2.2 illustrates that all these four protocols can be implemented in multiple programming languages, and except CoAP, all the remaining protocols have successful products in the market.

TABLE 2.2: Protocol Implementations

Protocol	Implementation	Size	Successful Product in the Market[42][43][44][45]
MQTT	Implementation in C, Java, Pyhthon, Erlang, etc. Well developed and open source applications can be found.	128KB+	1) Amazon, AWS IoT service using MQTT broker. 2) Facebook, Facebook Messenger App is using MQTT.
CoAP	Implementation in C, Java, Pyhthon, Go, etc. However, most implementations stop at the library level, users need to build their own application.	241KB+	Not Yet
AMQP	Implementation in C, Java, Python, etc. Well developed and open source applications can be found.	4000KB+	1) NASA, control plane of the Nebula Cloud Computing. 2) Red Hat, control plane of Red Hat's Cloud services use AMQP to control its internal operations. 3) RabbitMQ is widely used at AT&T Interactive, a local search provider.
DDS	Implementation in C, Java, Python, etc. Well developed and open source applications can be found.	1.0GB+	DDS solution has been widely used by technologically advanced companies, for instance, Google Cloud, IBM, and Microsoft.

It is worth mentioning that the size of an implementation may vary a lot within different vendors. The size data shown in Table 2.2 refers to the four selected implementations described in Chapter 3. The data in the table is the full functional implementation size after compilation, including dependencies, Broker, Publisher and Subscriber. However, the size of the implementation can be reduced based on a given user's requirement. For instance, in [46], the implementation of CoAP's main functionality only consumes 8.5KB of ROM.

CoAP has great potential because of to its low overhead and RESTful feature. However, at this time, the other three protocols are more mature and more widely utilized. Although there aren't many practical CoAP products yet, this is not to say that CoAP is deficient. The performance of CoAP will be shown in Chapter 4.

Table 2.3: Overall Functional Comparison (adapted from [1])

Protocol	RESTful	Transport	Architecture		Security	QoS	Header Length
			Request-Response	Publish- Subscribe			
MQTT	No	TCP	No	Yes	TLS	Yes	2 Bytes
CoAP	Yes	UDP	Yes	Yes	DTLS	Yes	4 Bytes
AMQP	No	TCP	No	Yes	TLS/SASL	Yes	8 Bytes
DDS	No	TCP/UDP	Yes	Yes	TLS	Yes	16 Bytes

De-emphasizing practical applications for the moment, and focusing instead on the specifications of these four protocols, Table. 2.3 offers a general functional comparison of these four protocols. CoAP is the only RESTful protocol, DDS is only one that can run above both UDP and TCP, and only AMQP can provide security rather than rely strictly on TLS. MQTT is the only protocol that does not provide interoperability, which means each data node must use the same vendor's implementation; that is a significant weakness. Another important difference is that DDS and CoAP can be brokerless while MQTT and AMQP are broker-based. That is because MQTT and AMQP do not provide automatic discovery; thus the message distribution in MQTT and AMQP must through topics/queues and a broker. If the broker is moved to a different server, then the clients must be reconfigured. This impairs fault tolerance. Moreover, all of these four protocols provide QoS but the QoS levels are different. Table. 2.4 provides a QoS comparison; in particular it shows that DDS provides an extremely rich set of QoS choices.

Table 2.4: QoS Comparison

Protocol	QoS Level	QoS functionality
MQTT	3	At most once; Exactly once; At least once
CoAP	2	Confirmable; Non-Confirmable
AMQP	3	At most once; Exactly once; At least once
DDS	23	23 Levels to provide: control reliability, volatility, liveliness, resource utilization, filtering and delivery, ownership, redundancy, timing deadlines and latency of the data.

Overall, MQTT is lightweight and simple but powerful enough to build an IoT application. However, if the application need to have durable message features and needs message queueing, then AMQP would be a better choice. DDS can be used to build a large-scale network with better service management and security performance. And, CoAP is a good choice for the state-transfer model; the node device works as the server and the user can read the resource in the server at any time instead of waiting for future publication.

2.3.2 Security of IoT

Security is always a significant factor. Since all these protocols utilize TLS [2–4, 7] to provide secure connection, so obviously the common attacks can be performed to TLS can also be performed against these four protocols' applications. Here, will only focus on three specific types of threats impacting the IoT environment .

- Unauthorized subscription/publication [41] : An Attacker may subscribe to a Publisher in order to steal the data, An attacker may publish fake message(s) or spam/junk message to all the subscribers. The system needs to authorize users before they become publishers and/or subscribers; this feature and concept is not provided by TLS, TLS only provides a secure tunnel for communication. However, by default, MQTT and AMQP have a Username/Password authorization option (AMQP provide it by using SASL), and DDS has a set of rules for authorization.

- Spoofing [41]: Spoofing is a common attack method that allows the attacker to masquerade as a valid client, and then redirect all the traffic of the client to the attacker's machine. If the attacker does not do anything with the traffic and then forwards it to client, this is called Eavesdropping, If the attacker receives the traffic but does not forward it to client, this is a Denial of Service attack. If the attacker receives the traffic and modifies the payload, this is called a Man-in-the-Middle attack. However, if the digital signature is enabled, this kind of attack will be denied. However this solution is insufficient in IoT environment. To generate and keep using digital signatures costs too much for node devices like sensors. DDS provides another solution that each publication to different subscriber come with a HMAC [24] with different key.

- Unauthorized access to data [41] : An attacker may access the data stored in a broker, gateway, or edge device. By hacking the broker, gateway, or edge device, an attacker may delete or modify a topic, read or clear all the messages stored in a queue, or modify the data in the edge device before it has been published. If only the edge device has been hacked, the problem is just a data point failure. But if the broker or gateway has been hacked, then the whole system (respectively a subsystem) is under attacker's manipulation.

All these four protocols can provide the secure tunnel based on TLS but beyond the tunnel, TLS cannot really help with certain attacks focus on the architecture of IoT. Generally, DDS provides the most comprehensive security features by default. However, it would not be hard to add some mechanisms to solve all the three attack types described earlier. This would just take additional effort to develop; we leave this as potential future work.

2.4 Summary

In this Chapter, the four selected protocols: MQTT, AMQP, CoAP, and DDS have been introduced. The discussion covered messaging model, message format and security. MQTT is a lightweight protocol running on top of TCP/IP that aims to provide reliable communication for constrained devices and high latency networks. AMQP is a message-oriented protocol which also runs on the top of TCP/IP; it provides message routing and queuing features. CoAP is a RESTful protocol that runs on the top of UDP/IP; it provides low-overhead communication in a HTTP-like way. Finally, DDS is a real-time, high performance solution for data distribution which can run either over TCP or UDP.

We note that, with the exception CoAP, all the other protocols are widely used in well-known enterprises. This Chapter also compared the four protocols at the conceptual level. For instance, MQTT has less functionality but the message header length is the shortest. DDS has the most comprehensive functionality but has the longest header length. Moreover, a few of security problems have been discussed, and overall DDS provides best security.

Chapter 3

Experimental Design

This Chapter covers the design and detailed steps of the experiments. The goal of the experiments are to test the four selected protocols by setting up a test platform and measuring performance and memory usage. As a secondary goal, this platform should be easy to deploy, low cost, and offer good extensibility, so that we can apply it in extended and expanded settings in the future as well.

The test platform assembled has six main components: 1) The Common Gateway, which can communicate in four selected protocols; 2-5) The four selected protocols gateways, each of which can perform as a particular protocol client to the common gateway or work as a particular protocol gateway for the nodes devices attached to it. 6) The HTTP-Server, which provides access to this platform and provides data analysis functionality.

3.1 Hardware and Software Selection

In these experiments, the Raspberry-Pi 3 Model B [47] was selected as the Gateway device. It comes with a 1.2 GHz, 64-bit quad-core ARMv8 CPU and 1GB RAM; it also has a built-in 802.11n Wireless component and support to add an extra network adapter via USB. The Pi is a great choice for a gateway since it is inexpensive yet powerful and supports multiple network adapters. Furthermore, the Web-Server can be run on any desktop or laptop.

Table 3.1: Software Selection Table

Component	Language	Implementation Name
MQTT Gateway	Python	Mosquitto[48]
CoAP Gateway	C	Libcoap[49]
AMQP Gateway	JS, C++	Apache QPID[50]
DDS Gateway	C++	OpenDDS[51]
Web-Server	Html, PHP	XAMPP[52]
Background Programs	Java, Scripts	TestDistributor

Table. 4.1 shows the open source code that has been used here. There is no specific standard for selecting implementations; however, Mosquitto [48] is one of the most widely used MQTT implementations, Libcoap [49] is the first and best known CoAP implmentation, QPID [50] is provided by Apache and fully supports AMQP 1.0 (not all implementations support the standard at level 1.0), and OpenDDS [51] is an implementation that support DDS running on TCP (not all the implementations support TCP).

3.2 Experimental Environment

FIGURE 3.1: Experimental Environment

Fig. 3.1 depicts the testbed. The five gateways use an add-on 8GB SD card with Raspbian Jessie [53] operating system and connect to the network via WiFi. Fig. 3.2 illustrates the network topology.

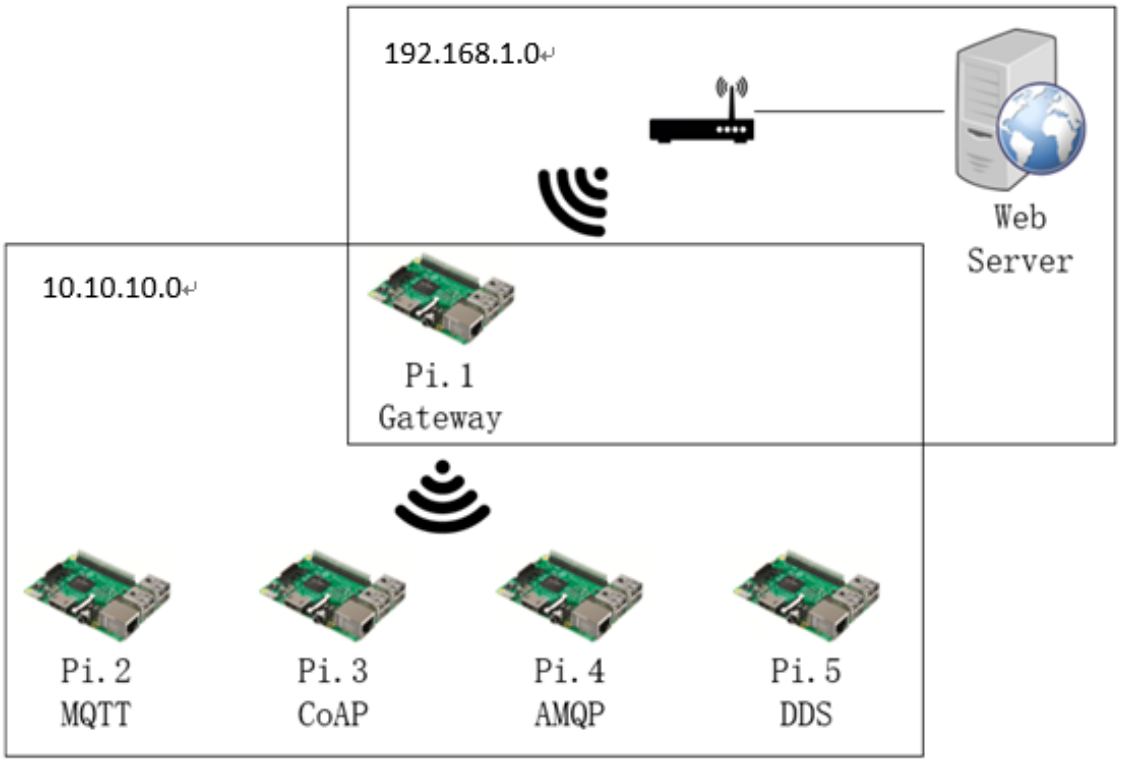

FIGURE 3.2: Network Topology

Pi.1 is the common gateway device; it's built-in network adapter connects to network 192.168.1.0. Thus, it can communicate with the Web Server. Pi.1 has an external network adapter, which was set up as the WLAN access point with network address 10.10.10.0. Pi.1 works as the gateway of this network. Pi.2 through Pi.5 are the four protocols gateways; they only connect to the 10.10.10.0 network via WiFi.

3.3 Experimental Steps

This platform seeks to provide an automatic protocol test function, which allows user using the web service prototyped here to set up a series of tests based on the following parameters: Protocol, Payload size, Send frequency and Quantity of Messages. Test results should provide message latency, as well as both CPU and memory usage.

FIGURE 3.3: Web Server Index Page

Fig. 3.3 is the index page of the Web Server. The user can select one of the four protocols and input the parameters of a given test Fig. 3.4.

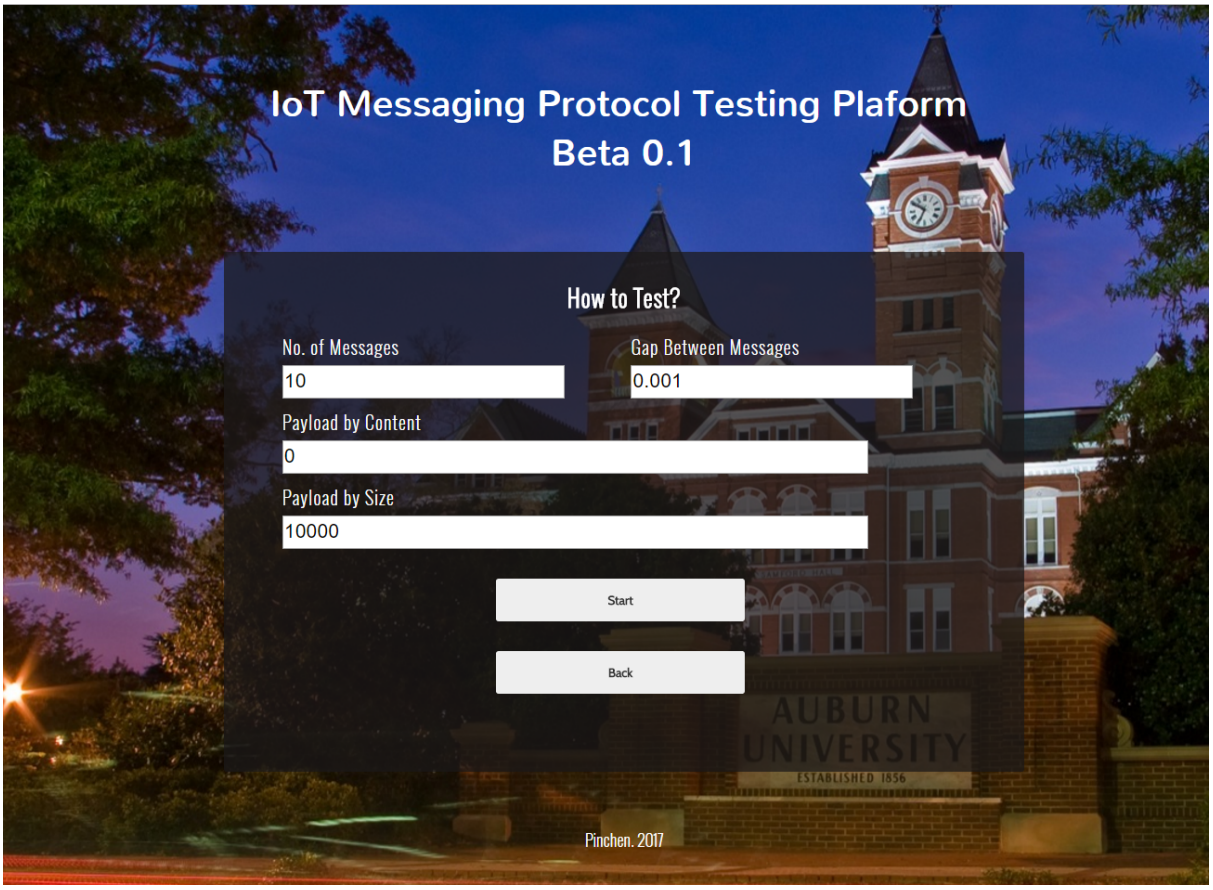

FIGURE 3.4: Web Server Test Setting Page

Then, the user can check the test status on the status page (see Fig. 3.5). Each test has four possible statuses: Submitted, Processing, Done, or Failed. Submitted means that the test has been submitted to the database but has not been sent to common gateway. Processing means that the test is recorded in the database and has been received by common gateway. Done indicates that the test has finished properly. Finally, Failed indicates the test fails because of to run time error or a time out. When a test fails, the system will automatically re-run the test until it returns a valid result.

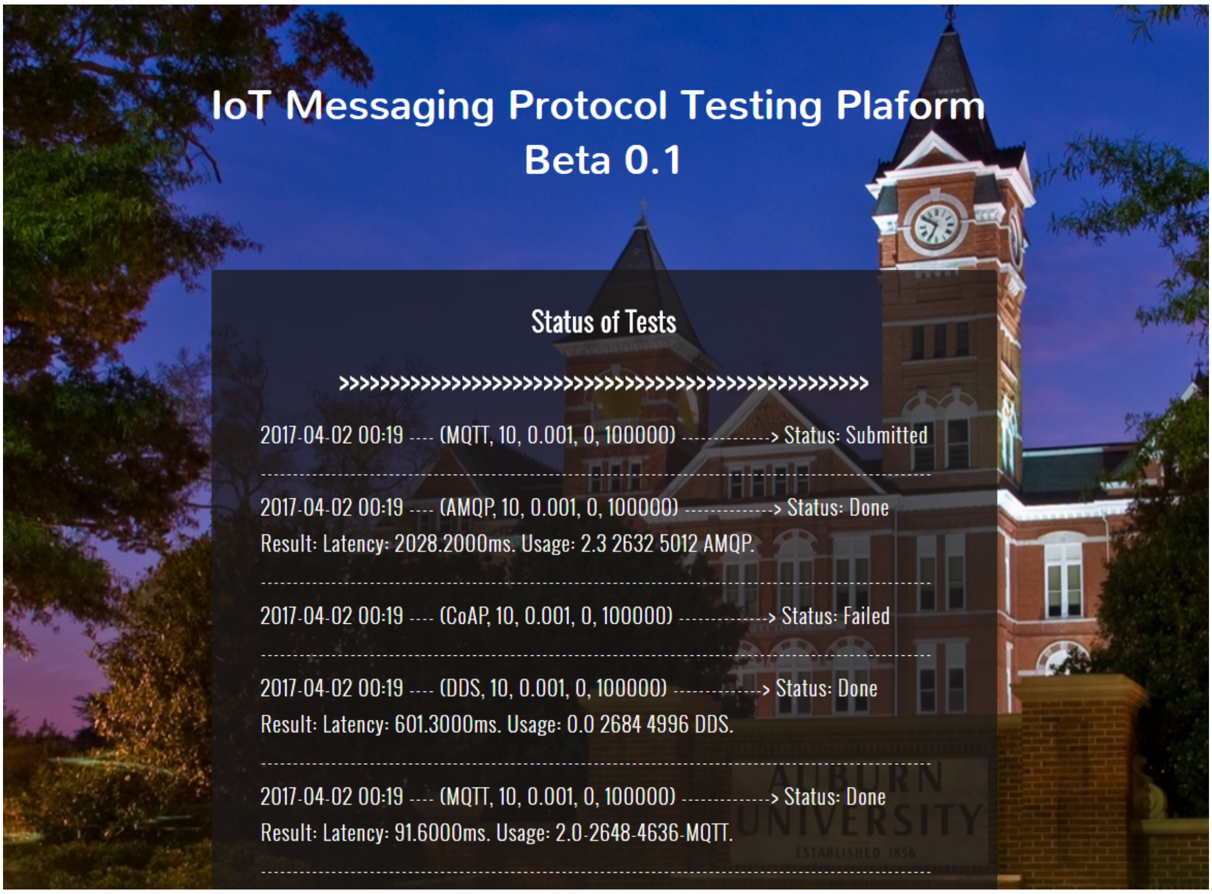

FIGURE 3.5: Web Server Test Status Page

All of the above procedures are implemented with PHP-based web pages and a background socket program named TestDistributor (see Fig. 3.6). TestDistributor will keep monitoring the status of the test, and take care of the test distribution and status updates.

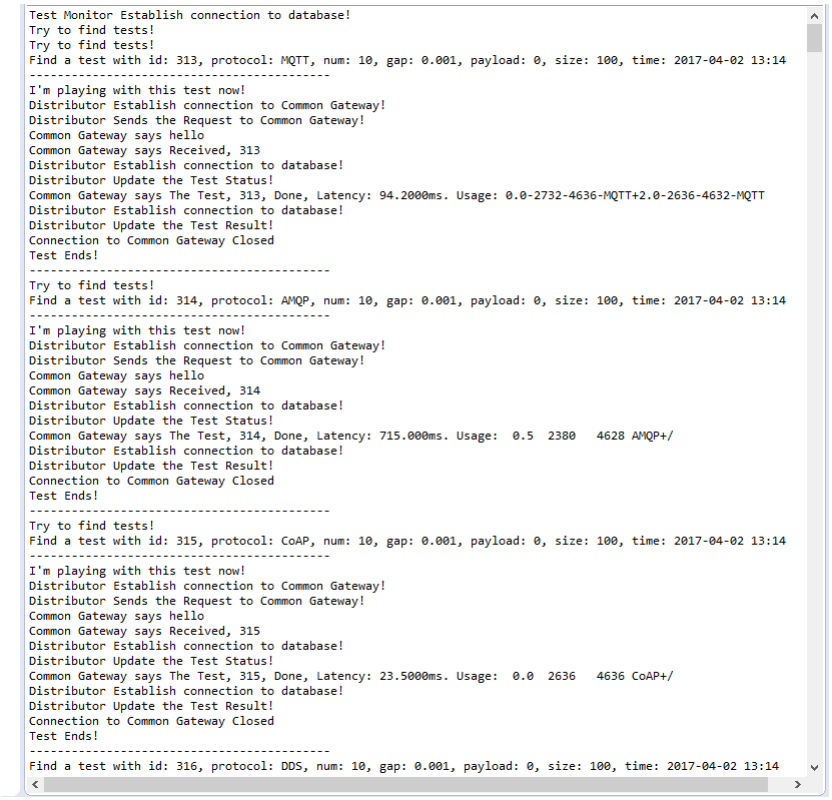

FIGURE 3.6: Screenshot of TestDistributor

On the other end, the common gateway uses another socket program to receive test requests from the TestDistributor. Then, the common gateway will deploy jobs to appropriate protocol gateways, and the protocol gateways will start the test. Test results will be returned to the common gateway, then forwarded to TestDistributor. All of the above test calls in the common gateway and in protocol gateways rely on a set of shell scripts.

3.4 Summary

In this chapter, a test platform has been described that we designed and established; it worked successfully to provide performance and memory-consumption data for this thesis. Five Raspberry Pi have been used to set up the testbed and all the works are using open source code. Only the MQTT implementation Mosquitto is well developed application package; all the other protocols implementation are open source libraries. All the Publish/Subscribe/Broker code that has been used in these experiments are written based on the example codes and tutorial provided by the particular implementation vendor.

The software approach devised here for experiments can be extended in the future.

Chapter 4

Results and Discussion

In the previous chapter, a test platform has been proposed and established. This platform provides the capability to test the performance of four selected protocols: MQTT, AMQP, CoAP and DDS. The tests can be based on the quantity of message, message payload, payload size and transmission frequency. In this Chapter, a formal set of tests have been designed and run. The results of these tests demonstrate the performance of the four protocols on latency, memory usage and cpu usage.

4.1 Test Design

By using this platform, a set of specific tests have been run. In this series of tests, all four protocols send confirmable messages, which makes sure the receiver will always receives the message. The broker is always located at the common gateway. For MQTT and AMQP, the protocol gateway will act as both of the Publisher and the Subscriber. Messages are sent to the common gateway and forwarded back to protocol gateway. And for DDS and CoAP, the Publisher and Subscriber will separately be located on common gateway and protocol gateway. Since in the brokerless architecture, the broker does not participate in data transmission. Thus, if Publisher and Subscriber should be located on the same device, then the test will be a meaningless localhost ping test.

The tests were divides into two groups:

1. Messages With Small Payload, the payload size is 10-100-250-500-750-1000 Bytes.
2. Messages With Large Payload, the payload size is 2000-5000-10000-20000-50000-100000 Bytes.

Furthermore, each group can be divided to two subgroup with different amount of messages, as follows:

1. 10 Messages, and

2. 1,000 Messages.

The 10-Message group can be used for the "Fire and Die" situation, which means each time the node data point will suddenly send the quantity of messages then goes back to sleep. And, the 1,000-Message group corresponds to the situation of continuously sending messages.

4.2 Test Results and Discussion

Before running all the tests, the initialization time of each protocol should be calculated. Initialization time is related to the protocol architecture and the complexity of implementation. Table 4.1 illustrates the initialization time. This time measurement is acquired by sending a message with payload size zero. Evidently, the initialization time will increase with the payload size. However, this kind of increase is almost linear, therefore, here will only shows the baseline.

TABLE 4.1: Initialization Time table (ms)

Size (Bytes)	MQTT	CoAP	AMQP	DDS
0	266ms	86ms	660ms	2750ms
100000	1581ms	99ms	1130ms	2855ms

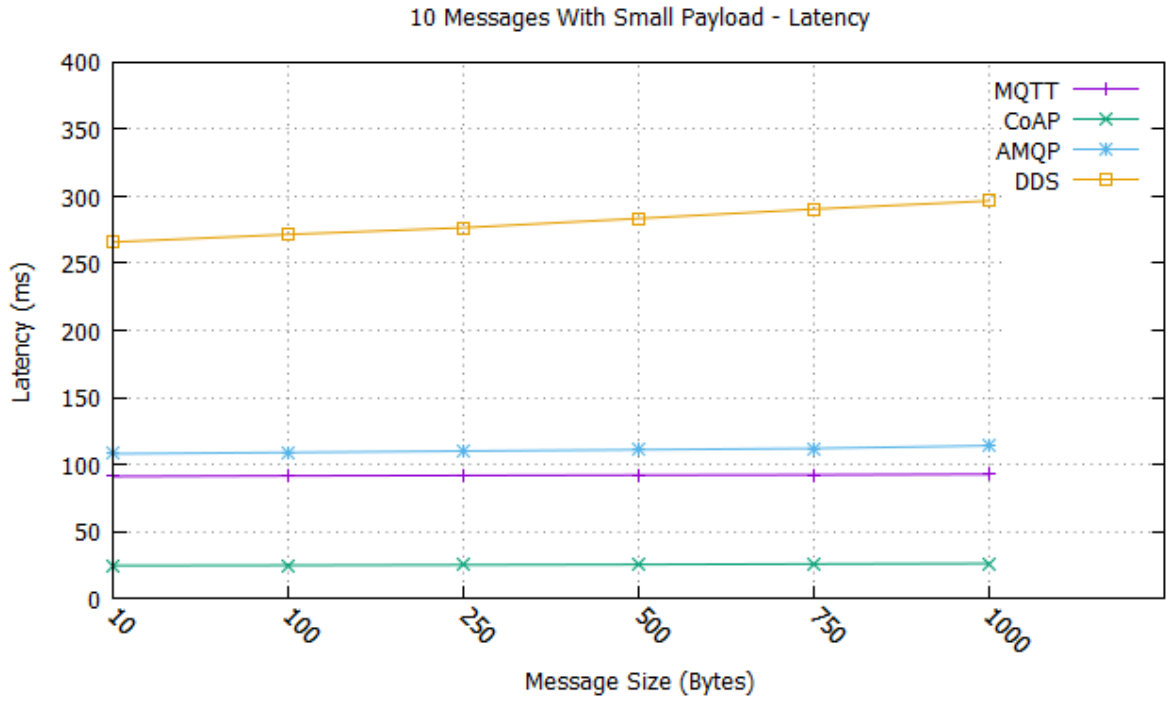

FIGURE 4.1: 10 Messages with Small Payload - Latency

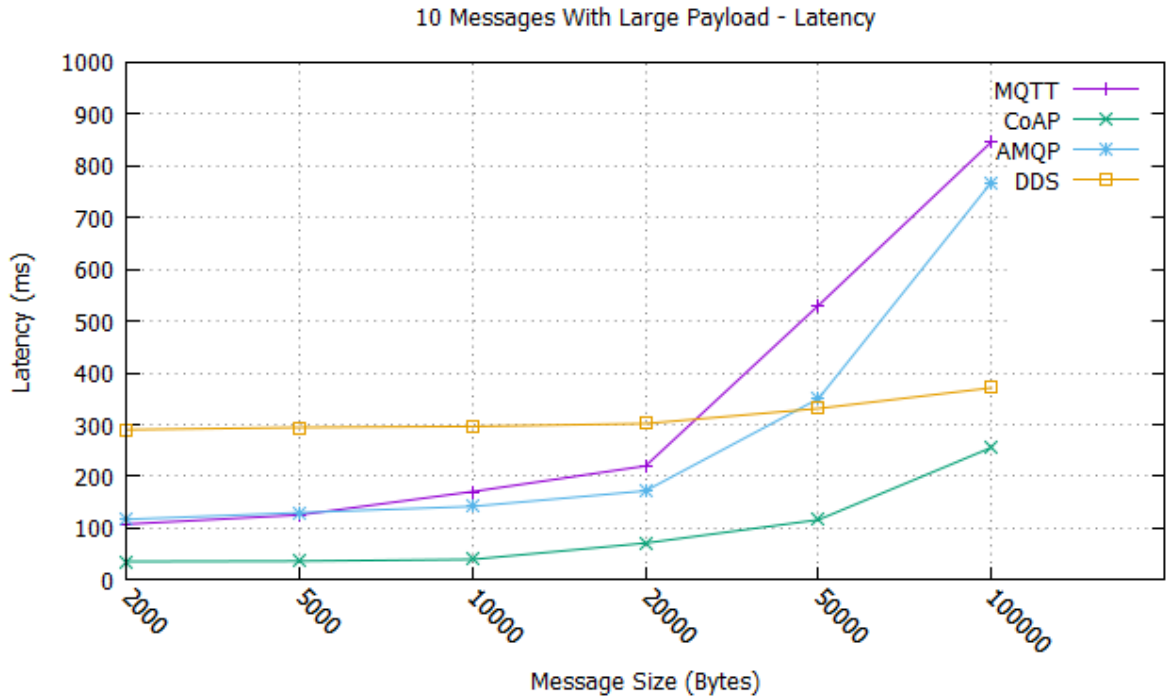

FIGURE 4.2: 10 Messages with Large Payload - Latency

For the 10-message group, latency is affected by the initialization time. When the message size is smaller than 5,000 Bytes, the four protocols order by latency time is as follows: DDS>AMQP>MQTT>CoAP. However, when the message size reaches 5,000 Bytes, MQTT message latency exceed AMQP message latency. When the message size is over 25,000 Bytes, MQTT message latency is higher than DDS message latency. If the message size is larger than 50,000 Bytes, the order will be: MQTT>AMQP>DDS>CoAP.

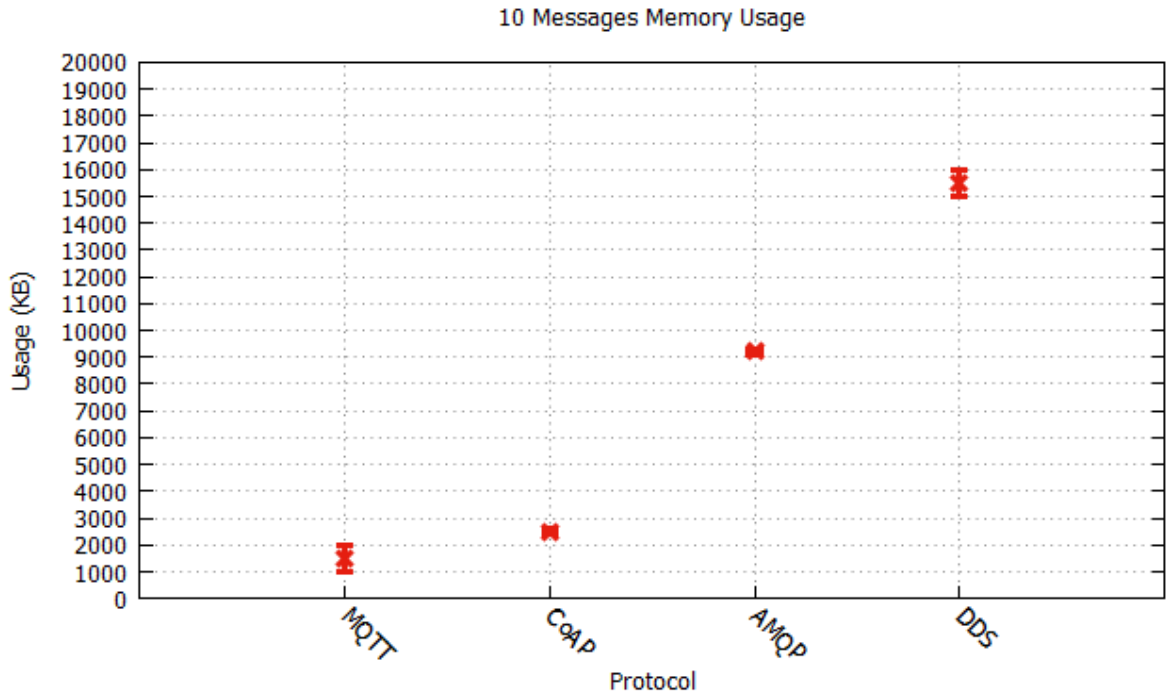

FIGURE 4.3: 10 Messages Usage - Memory

Memory usage should be affected by transmission frequency and message size. In Fig. 4.3, the transmission frequency is unchanged, and, with the message size increasing, MQTT memory usage changes from 900KB to 2000KB. CoAP does not change a lot, only varying from 2,500KB to 2,700KB. AMQP changes from 9,100KB to 9,400KB. DDS need larger memory space and the range is 15,000KB to 16,000KB.

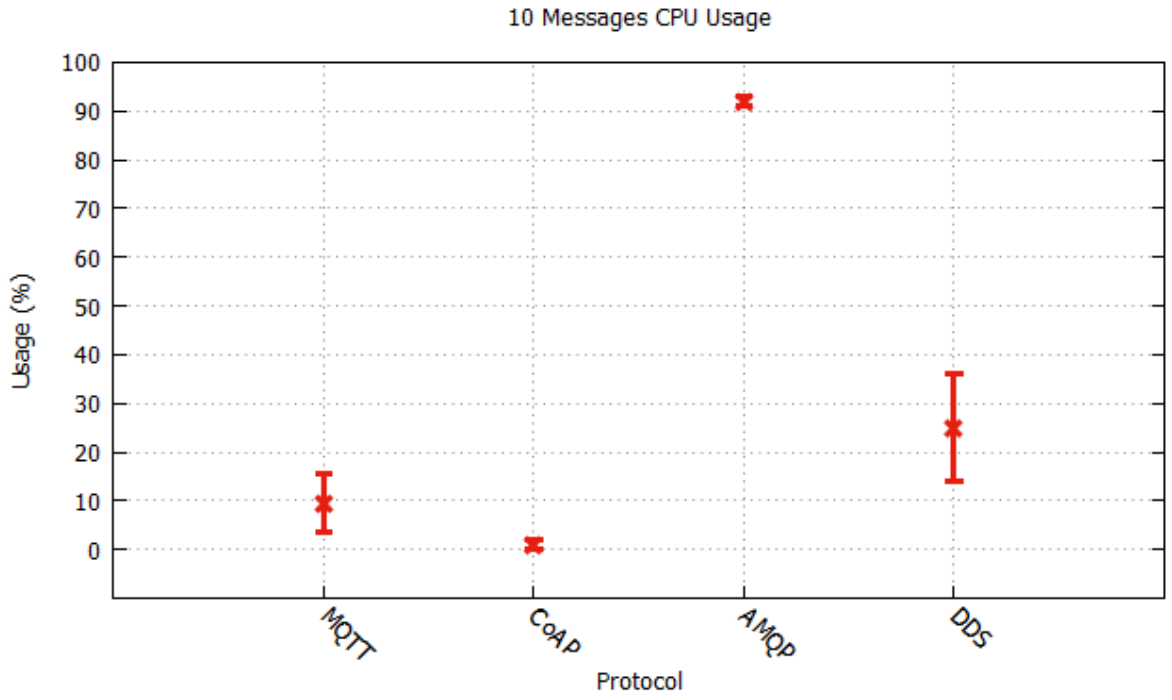

FIGURE 4.4: 10 Messages Usage - CPU

For the CPU usage aspect, MQTT need about 10% CPU power[1] to run the 10 Message test. CoAP need little more than 0%; AMQP need about 90%. DDS CPU usage increasing a lot with the message size increasing, which is from 16% to 36%. However, this result is also affected by each protocol implementation's initialization and internal properties.

[1] The CPU utilization in this experiment is talking about one core.

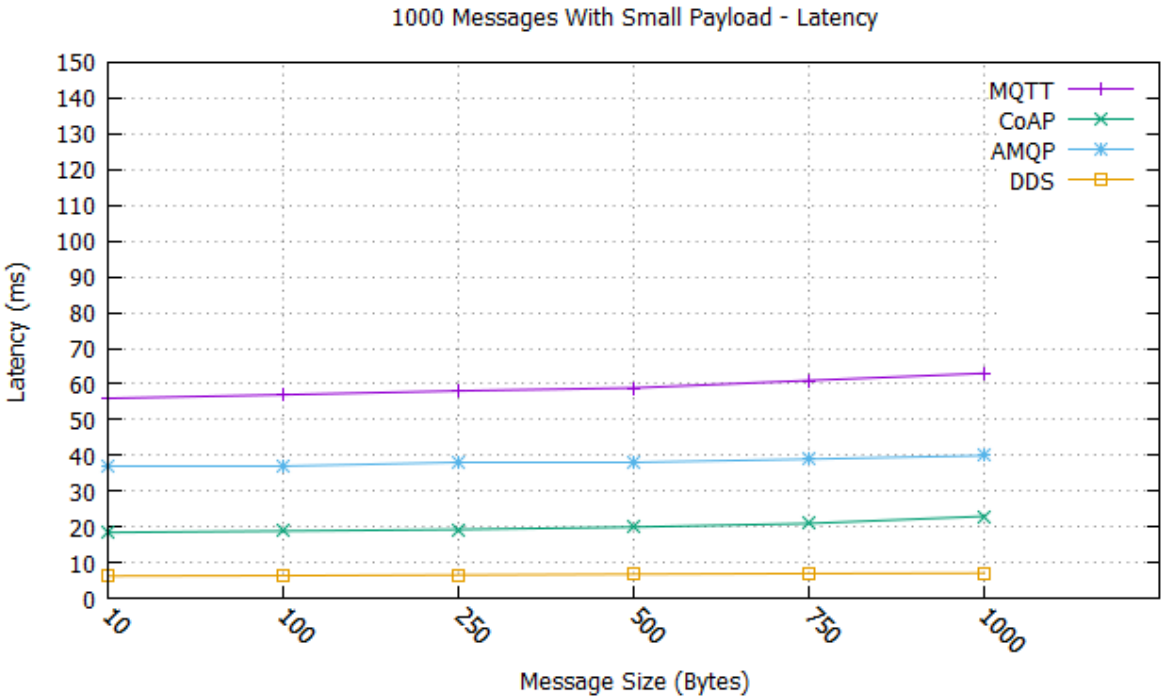

FIGURE 4.5: 1000 Messages with Small Payload - Latency

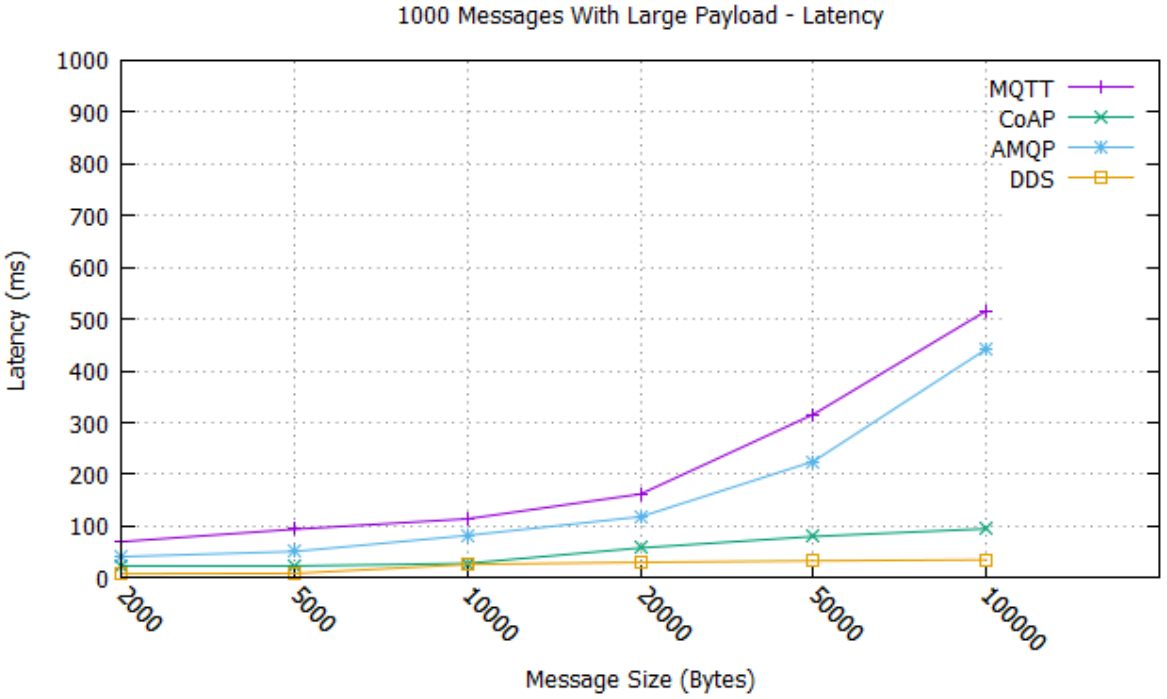

FIGURE 4.6: 1000 Messages with Large Payload - Latency

The 1000-Messages group indicate the real performance for consistent transmission . DDS as

a soft real-time data distribution solution should offer the latency at microsecond level [54]. Although a Pi is not a extremely powerful device and the network adapter also has limited performance. DDS still sent message at low latency, even when the message size is 100,000 Bytes, the latency is still under 50ms. CoAP provides relatively stable transmission results. But, when the message size increasing to 100,000 Bytes, the latency changes from 20ms to 100ms. Latency of AMQP and MQTT were heavily impacted by the message size, MQTT changes from 55ms to 520ms, and AMQP changes from 37ms to 440ms. However, in the continuously sending scenario, the latency order is as follows: MQTT>AMQP>CoAP>DDS.

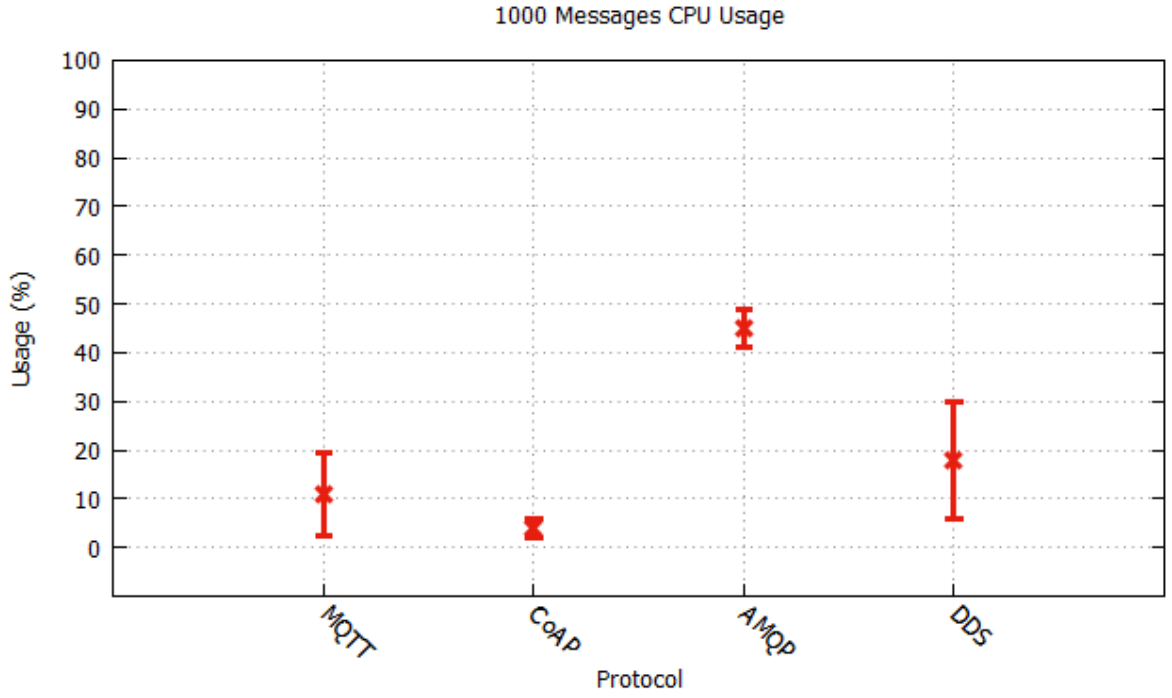

FIGURE 4.7: 1000 Messages Usage - CPU

Since the transmission volume increased to 1,000 messages, the influence of initialization should be much smaller than in the former tests. Furthermore, keeping the transmission at the same size for 1000 times, the CPU or memory usage should probably stay at a particular point for a while. Thus, the usage should be more stable and realistic than 10 message group. Results in Fig. 4.8 illustrate that AMQP still has highest CPU usage, which is around 45%. DDS uses 5% to 30% CPU power to send different size of messages. The usage range of MQTT is from 3% to 20%. And remain the same, CoAP show its stability, the CPU usage is around 2% to 5%.

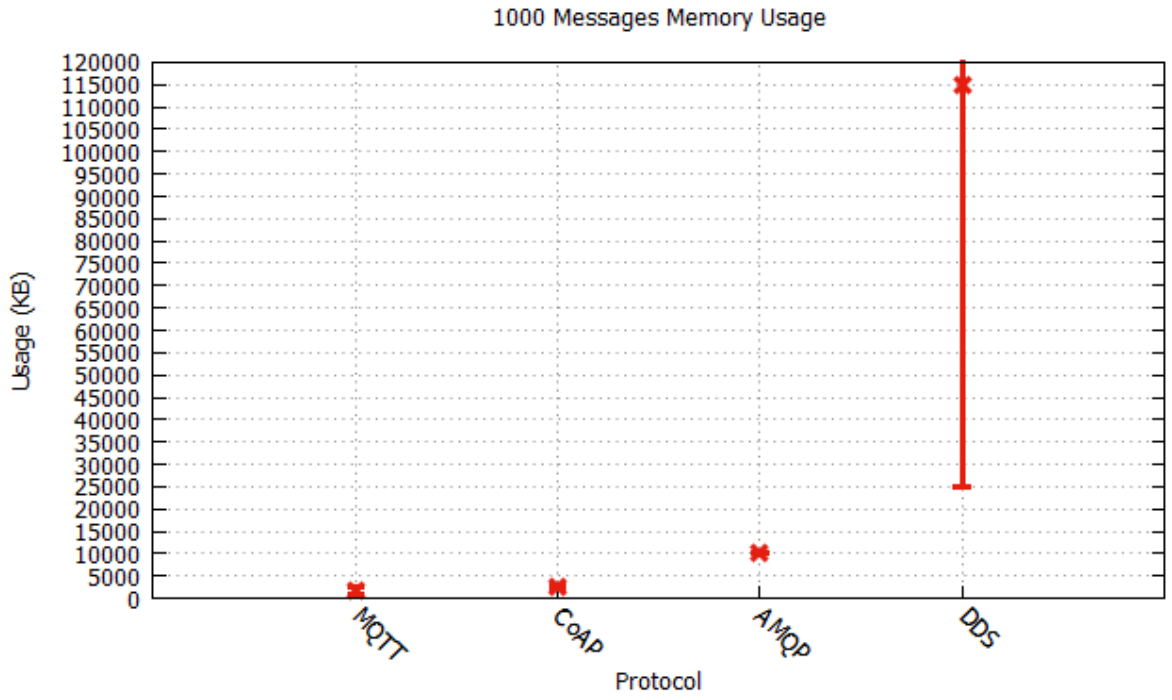

FIGURE 4.8: 1000 Messages Usage - Memory

According to Fig.4.8, DDS uses huge quantities of memory space when the message size is 100,000 Bytes. DDS need 25,000KB to 200MB (200,000KB) of memory to send a message size from 10 Bytes to 100,000 Bytes. MQTT and CoAP only uses almost uncountable memory space, MQTT: 2,000KB to 3,000KB and CoAP: 2,500KB to 3,000KB. AMQP is still performing with usage higher than MQTT and CoAP, which is around 100,00KB.

4.3 Summary

In this Chapter, a series of tests have been run and the results show that on the latency aspect, MQTT provides communication with high latency. AMQP is slightly better than MQTT. However, CoAP has much lower latency and the performance is more stable when the payload size increases. Moreover, DDS as a soft real-time solution; it has extremely low latency no matter the size of the payload.

By way of contrast, resource utilization is another story. Although the latency is high, MQTT does not use much CPU power or memory space. CoAP uses slightly more resources than MQTT but it still maintains a relatively low level overall. AMQP has much higher CPU and memory usage than MQTT and CoAP but at least the usage is stable. However, DDS uses huge quantities of memory space when the payload size and number of messages increases, and

the maximum usage could even reach to 200MB when sending 1,000 messages with 100,000KB (100MB) payloads.

Chapter 5

Conclusion And Future Work

In the previous Chapters, the four selected protocols have been introduced, compared, contrasted and tested for performance and memory footprint. A test platform has also been established by using Raspberry Pi's and open source resources. This Chapter summarize all the work performance thesis, and offer suggestions for future work.

5.1 Conclusion

In this thesis, the four selected IoT application layer protocols: MQTT, AMQP, CoAP and DDS. The four protocols have disparate emphasized functionality. We did this to elucidate how they are similar, how they are different, and offer insight into what an IoT architect, software designer and/or systems administrator might have to cope with in terms of function, overhead, security when creating IoT environments.

The high-level properties of the protocols can be summarized as follows:

1. MQTT aims to provide a data-centric communication with latency tolerance over the Publish-Subscribe model.
2. AMQP is also based on the Publish-Subscribe architecture, whereas it focuses on message-oriented features. Using Exchange-Binding-Queue components allows AMQP to queue and route messages in various ways.
3. CoAP is a RESTful protocol built upon UDP/IP and designed for resource-constrained networks. It not only works in the Client-Server model but also provides Observer Modes, which provide equivalent functionality to the Publish-Subscribe model.
4. DDS is a system proposed to provide high-performance data distribution; it provides real-time communication with a set of predefined QoS policies. The interoperability of DDS makes it work cross programming language and operating system.

A testing platform has been created and a series of tests were run on the four protocols performance. The results show that DDS has the lowest latency in the continuously working scenario. However, the memory and CPU usage of DDS is much larger than the other three protocol. Overall, CoAP always provides stable messaging at relatively low latency with acceptable system resources usage. Interestingly, CoAP is the newest and least widely used protocol in commercial products as of now.

This thesis also involved the successful implementation and deployment of these four protocols, and mentions security issues as well. Except for CoAP, all the other protocols are widely used and accepted. CoAP is still early in its development and acceptance, but has a bright future. For the security aspect, DDS provides the most comprehensive security features, but constrained environments may not be able to afford the overhead of using its approaches to authentication or encryption.

5.2 Future Work

There is some other work on these four protocols, such as [46], which addresses CoAP energy consumption testing, [55], which considers the packet loss of CoAP and MQTT, [56], which addresses performance test of RESTful services and AMQP. These bear further review and consideration in order to determine combined performance that would be observed by real applications while considering the constraints of energy consumption. Of course, different implementations of each protocol will vary in these regards, and we only considered one implementation of each protocol here. Further experimentation with multiple implementations of the same protocol may be worthwhile as well.

Regarding the performance concern, the experiments in this thesis are, as jusst mentioned, limited to one implementation per protocol and also the hardware was limited to only one kind of device. The experimental environment contains only one Publisher and Subscriber. Adding more implementations, interconnecting more hardware platforms, testing more performance factors and associating with larger-scale of data points will be a next logical step of this work. In particular, understanding how these protocols behave as the system is scaled up in nodes, gateways, brokers, and applications is all open to future consideration.

For the security concern, the main problem is not which protocol is more secure but rather how to adapt current security techniques into constrained networks. For instance, [57] works on DTLS-based, two-way authentication over CoAP. The results show that DTLS could be a feasible solution with only 20KB RAM needed to perform a full authentication. In the future, to advance the work of this project further, the main focus should focus on adding tests with security parameters into the experimental testing platform, adapting high overhead

authentication and encryption methods into IoT in ways that don't ruin performance. And, consider ways that don't cripple performance at "acceptable security levels." The latter idea has to be quantified and work from outside IoT needs to be studied and adapted where possible.

Finally, creating protocol inter-converters for gateways and brokers will be important, because an IoT environment created from commodity-off-the-shelf IoT products (COTS) will likely use multiple of the protocols. Evidently, the capabilities of QoS, messaging services, and interactions with gateways and applications are not all identical and this type of effort will introduce new opportunities and challenges. The disparate dependency on UDP/IP and TCP/IP will further complicate such interoperability and multi-protocol integration, but appears important to study.

5.3 Brief Summary

It is not hard to understand the concepts of a protocol and set up a service that using this protocol, but the problem is why should the user use this protocol instead of another one. The motivation of this thesis is to provide a basic guideline for choosing IoT application layer protocol, this guideline should not only cover the abstract concepts like message format and messaging model. It should also cover the content about the deployment and performance. Since the concepts cannot directly indicates the difference in practical, thus a test platform has been proposed and established. This test platform is still in beta version however it provides the basic capability to analyze the performance of protocols. And this is the most significant achievement of this thesis.

Bibliography

[1] Internet of things: A survey on enabling technologies, protocols, and applications. *IEEE Communications Surveys & Tutorials*, 17, 2015.

[2] Rahul Gupta Andrew Banks. Mqtt version 3.1.1. URL `http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/cos02/mqtt-v3.1.1-cos02.html`.

[3] RFC 7252. Constrained application protocol (coap).

[4] OASIS. Advanced message queuing protocol (amqp) version 1.0. URL `http://docs.oasis-open.org/amqp/core/v1.0/os/amqp-core-complete-v1.0-os.pdf`.

[5] Advanced message queuing protocol (amqp) version 0.91. . URL `https://www.rabbitmq.com/resources/specs/amqp0-9-1.pdf`.

[6] Amqp exchanges and bindings, . URL `https://www.cloudamqp.com/blog/2015-09-03-part4-rabbitmq-for-beginners-exchanges-routing-keys-bindings.html`.

[7] OBJECT MANAGEMENT GROUP. Data distribution service (dds) specification version 1.4. . URL `http://www.omg.org/spec/DDS/1.4`.

[8] OBJECT MANAGEMENT GROUP. The real-time publish-subscribe protocol (rtps) dds interoperability wire protocol specification version 2.2. . URL `http://www.omg.org/spec/DDSI-RTPS/2.2`.

[9] Gartner Research. Nov 2015. URL `http://www.gartner.com/newsroom/id/3165317`.

[10] Dave Evans. The internet of things: How the next evolution of the internet is changing everything. *Cisco IBSG*, April 2011. URL `http://www.cisco.com/c/dam/en_us/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf`.

[11] Wikipedia. Internet of things. URL `https://en.wikipedia.org/wiki/Internet_of_things`.

[12] XMPP Standards Foundation Peter Saint-Andre. Extensible messaging and presence protocol (xmpp). URL `https://xmpp.org/extensions/xep-0001.pdf`.

[13] I. Fette. Rfc6455, the websocket protocol. URL https://tools.ietf.org/html/rfc6455.

[14] IETF R. Fielding. Rfc2616, hypertext transfer protocol – http/1.1. URL https://tools.ietf.org/html/rfc2616.

[15] Francisco Vazquez-Gallego Jesus Alonso-Zarate Vasileios Karagiannis, Periklis Chatzimisios. A survey on application layer protocols for the internet of things. *Transaction on IoT and Cloud Computing 2015*.

[16] Othonas Soultatos-Ioannis Papaefstathiou Charalampos Manifavas Vasilios Katos Konstantinos Fysarakis, Ioannis Askoxylakis. Which iot protocol? comparing standardized approaches over a common m2m application. *2016 IEEE Global Communications Conference (GLOBECOM)*.

[17] Alvin Valera-Hwee-Xian Tan Colin Keng-Yan TAN Dinesh Thangavel, Xiaoping Ma. Performance evaluation of mqtt and coap via a common middleware. *2014 IEEE Ninth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP)*.

[18] Client-server model. URL https://en.wikipedia.org/wiki/Client%E2%80%93server_model.

[19] Publish-subscribe model. URL https://en.wikipedia.org/wiki/Publish%E2%80%93subscribe_pattern.

[20] ITU-T Recommendation. Terms and definitions related to quality of service and network performance including dependability. 1994. URL http://www.itu.int/rec/T-REC-E.800/en.

[21] Transport layer security version 1.2. URL https://www.ietf.org/rfc/rfc5246.txt.

[22] Rfc 6347, datagram transport layer security version 1.2. URL https://tools.ietf.org/html/rfc6347.

[23] Message authentication code(mac). URL https://en.wikipedia.org/wiki/Message_authentication_code#cite_note-8.

[24] Rfc 2104, keyed-hashing for message authentication (hmac). URL https://www.ietf.org/rfc/rfc2104.txt.

[25] Rfc 3280, internet x.509 public key infrastructure certificate and certificate revocation list (crl) profile. URL https://www.ietf.org/rfc/rfc3280.txt.

[26] T. Ts'o B.C. Neuman. Kerberos: an authentication service for computer networks. *IEEE Communications Magazine*, 1994. URL http://ieeexplore.ieee.org/abstract/document/312841/.

[27] Rfc 2743, generic security service application program interface version 2, update 1. URL https://tools.ietf.org/html/rfc2743.

[28] Microsoft. Microsoft ntlm. URL https://msdn.microsoft.com/en-us/library/windows/desktop/aa378749(v=vs.85).aspx.

[29] Rfc 4422, simple authentication and security layer (sasl). URL https://tools.ietf.org/html/rfc4422.

[30] Andrew Foster. Messaging technologies for the industrial internet and the internet of things. URL http://www.prismtech.com/sites/default/files/documents/MessagingComparsionNov2013USROW_vfinal.pdf.

[31] Representational state transfer. URL https://en.wikipedia.org/wiki/Representational_state_transfer.

[32] Roy Fielding. Representational state transfer (rest). URL http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm.

[33] RFC 7641. Observing resources in the constrained application protocol (coap).

[34] RFC 7959. Block-wise transfers in the constrained application protocol (coap).

[35] RFC 6690. Constrained restful environments (core) link format.

[36] draft-ietf-core-resource-directory 09. Core resource directory.

[37] C. Bormann. Using coap with ipsec. URL https://datatracker.ietf.org/doc/draft-bormann-core-ipsec-for-coap/.

[38] G. Selander. Object security of coap (oscoap). . URL https://datatracker.ietf.org/doc/draft-ietf-core-object-security/?include_text=1.

[39] G. Selander. Requirements for coap end-to-end security. . URL https://datatracker.ietf.org/doc/draft-hartke-core-e2e-security-reqs/?include_text=1.

[40] RTI Connext DDS Professional. URL https://www.rti.com/products/dds/routing-service.

[41] OBJECT MANAGEMENT GROUP. Dds security specification version 1.0. . URL http://www.omg.org/spec/DDS-SECURITY/1.0.

[42] AMQP Product. URL https://www.amqp.org/about/examples.

[43] MQTT in Facebook. URL https://www.ibm.com/developerworks/community/blogs/mobileblog/entry/why_facebook_is_using_mqtt_on_mobile?lang=en.

[44] MQTT in Amazon. URL http://docs.aws.amazon.com/iot/latest/developerguide/protocols.html.

[45] Dds product. URL https://objectcomputing.com/about/our-partners/.

[46] Matthias Kovatsch. A low-power coap for contiki. *IEEE Eighth International Conference on Mobile Ad-Hoc and Sensor Systems*, 2011.

[47] Raspberry pi 3 model b. URL https://www.raspberrypi.org/products/raspberry-pi-3-model-b/.

[48] Mosquitto. URL https://mosquitto.org/.

[49] Libcoap. URL https://libcoap.net/.

[50] Apache qpid. URL https://qpid.apache.org/.

[51] Opendds. URL http://opendds.org/.

[52] Xampp. URL https://www.apachefriends.org/index.html.

[53] Raspbian jessie with pixel. URL https://www.raspberrypi.org/downloads/raspbian/.

[54] OpenDDS Latency Test. URL http://opendds.org/perf/lab100125/latency_results.html.

[55] Dinesh Thangavel. Performance evaluation of mqtt and coap via a common middleware. *IEEE Ninth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP)*, 2014.

[56] Joel J. P. C. Rodrigues Joel L. Fernandes, Ivo C. Lopes. Performance evaluation of restful web services and amqp protocol. *Fifth International Conference on Ubiquitous and Future Networks (ICUFN)*, 2013.

[57] Wen Hu Michael Brnig Georg Carle Thomas Kothmayr, Corinna Schmitt. Dtls based security and two-way authentication for the internet of things. *Ad Hoc Networks*, 2013.