TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

SRIJAN MANANDHAR

MQTT BASED COMMUNICATION IN IOT

Master of Science thesis

Examiner: Prof. Kari Systä
Examiner and topic approved by the
Faculty Council of the Faculty of
Department of Pervasive Systems
on 31st May 2017

ABSTRACT

SRIJAN MANANDHAR: MQTT based communication in IoT
Tampere University of Technology
Master of Science thesis, 48 pages, 0 Appendix pages
November 2017
Master's Degree Programme in Information Technology
Major: Pervasive Systems
Examiner: Prof. Kari Systä
Keywords: IoT, MQTT, HTTP, REST, API

There are many studies on simplifying the development and management of IoT applications. One of those research work is to use IoT devices as the RESTful resources or even IoT devices as application servers that can expose RESTful resources. However, REST style is not suitable for some network conditions. Hence, this document proposes to use MQTT in place of REST for accessing and managing the services provided by the IoT devices. As a result, IoT application development and management become possible in all network conditions. In this thesis, firstly REST-based implementation of IoT development and deployment platform is described. Then issues with this implementation are discussed, which is the motivation for using MQTT as the protocol for this platform. Then finally, MQTT based implementation is described with mapping of HTTP request-response used in REST-based implementation to MQTT style request-response.

PREFACE

I would like to thank my supervisor Kari Systä from the Tampere University of Technology for providing me the guidance in the writing of this theses. I am also thankful to my family for supporting and encouraging me throughout my studies in TUT.

Last but not the least, I would also like to express my sincere thanks to all of my friends.

Tampere, 22.11.2017

Srijan Manandhar

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND SYMBOLS

HTTP	HyperText Transfer Protocol
REST	Representational State Transfer
MQTT	Message Queue Telemetry Transport
API	Application Programming Interface
IoT	Internet of Things
TUT	Tampere University of Technology
URL	Uniform Resource Locator
URI	Uniform Resource Identifier
NAT	Network Address Translation
RR	Resource Registry
IDE	Integrated Development Environment

1. INTRODUCTION

With the improved technology in software and hardware, IoT devices are becoming more powerful. And with this speed of technological advancement, the future IoT devices will be programmable, will be able to host applications and even remote management of these applications will be possible. Remote management refers to remotely access the application programming interfaces of the device to install, update, delete and manipulate the initiation and termination of the applications in the device. Considering the heterogeneous nature of IoT devices, each kind of device may have its own proprietary communication protocol. Hence, for the possibility of remote management of every IoT devices, knowledge of all corresponding communication protocol is required. This way of communication makes it difficult to interact with all the IoT devices without having knowledge of proprietary systems. However, the web can be such a platform which can integrate the devices, providing the common interacting platform for users.

IoT devices can be integrated into the Web using RESTful resources to expose the functionality of the IoT devices [18]. REST uses URI as identifiers for resources on the Web. And in case of IoT systems, URI will be used to identify IoT devices as resources or resources provided by the IoT devices. RESTful APIs for every device will be accessible from the Web resources and HTTP request can be used to interact with the exposed resources of the device. For instance, HTTP request to get the temperature reading from a sensor using RESTful API. This paper [28] also discuss on using REST resources for sharing device readings. Similarly, there are a research work [5] which discuss on using REST in the context of future IoT devices which can host applications and behave as an application server by exposing RESTful resources. However, use of RESTful resources may not be ideal in every condition.

Most of the IoT devices are sensors, actuators, and embedded devices and are millions in number and their numbers are increasing day by day. Considering the millions of devices which go on and off for their functioning, it will be impossible

to allocate IP address for all [14]. IPv4 address will not suffice for the addressing of all IoT devices [7]. Therefore scheme such as Virtual Private Network (VPN) or Network Address Translations (NAT) must be used to incorporate all the available IoT devices to connect to the Internet. NAT appears to abstract the address of the devices that operate behind it. This may be a problem in REST, as it is based on HTTP request-response between client and server and they must have IP address to communicate with each other. Hence, there might be an issue when the server tries to communicate with the client which resides behind the NAT. Here, the server cannot connect to the client as it is unaware of the IP address of the client which is behind the NAT. The solution for communication between server and client in NAT case can be to use the protocol which does not depend on the IP address of the client-server endpoints but only focuses on the delivery of data between those endpoints.

MQTT is such a protocol which is lightweight, data-centric and specially designed for resource constrained devices like IoT devices [4]. MQTT uses publish-subscribe messaging for communication between IoT devices. When using this protocol, devices do not need to know the existence of other devices. All the devices publish and subscribe the messages to the central broker, and this broker will handle delivering messages to subscribers. Hence, using MQTT the communication between IoT devices should work on the local network as well as in the network which uses Network Address Translations (NAT).

This thesis is research on how to use MQTT as the protocol for IoT application deployment and remote management of those applications, such that it can work in all network conditions.

This thesis work takes reference from the REST based IoT application development and deployment platform known as 'LiquidIoT' [5]. In LiquidIoT, REST resources are used for remote management of IoT applications. Operations like install, update, start and stop performed using HTTP request-response mechanism on those REST resources. In this thesis, REST resources of LiquidIoT are mapped into the corresponding MQTT topics and HTTP request-response used to access REST resources are mapped to publish-subscribe in MQTT. Hence, this thesis is the research on using lightweight protocol MQTT in place of REST resources to control and monitor applications in IoT devices.

The remainder of the document is structured as follows. Chapter 2 discusses the the-

oretical aspects about the MQTT, REST, LiquidIoT platform, and the REST-based implementation of LiquidIoT Chapter 3 discusses in detail about the implementation details of MQTT topics based design of LiquidIoT. Chapter 4 discusses the results of comparison between the two designs, evaluation of this work with the similar other works on this topic and chapter 5 summarize the whole document with notes on limitations, recommendations, and future directions.

2. BACKGROUND

2.1 REST

REST is an architectural style for building the web and other network-based software [17]. This thesis considers RESTful interfaces in the context of services exposed and consumed by IoT devices. REST follows client-server architectural style which brings the separation of concern principle. According to this principle, the server will be concerned about providing the resources and client will be concerned about accessing those resources from the servers using HTTP [16]. Everything that REST makes available need to be available in terms of resources. Resources can be everything in REST that can be named and identified. For example, the temperature sensor is a resource which can provide information about the temperature of the surrounding where it is installed. The following sections will explain essential elements of RESTful architecture.

2.1.1 Resource Identifier

Resources are identified using resource identifier or URI. A resource can give single information or the collection of information. For example, "devices" is a resource which gives the information about the list of devices. Whereas, "device" is a single source of information.

The resource "devices" is identified by URI,

```
http://192.168.0.1/devices
```

where http:// is the protocol used, 192.168.0.1 is host IP address and /devices is the URL for the resource

And the resource "device" is identified by URI,

```
http://192.168.0.1/device/{device_id}
```

which gives the information about the device with id "device_id".

Similarly, applications installed in the device is also a resource and it is identified by URI,

```
http://192.168.0.1/device/{device_id}/apps
```

And individual application resource in the device is identified by URI,

```
http://192.168.0.1/device/{device_id}/{app_id}
```

Resource identifier contains information about the identification of the resources but it does not give information about the operation that will be performed on the resources.

2.1.2 Uniform Interface

Resources in the REST should have the uniform interface with the fixed set of HTTP methods and representation that it returns. REST uses HTTP methods GET, POST, PUT, DELETE for performing operations on resources. The GET method is used for information retrieval which does not change the resources. In contrast, POST, PUT and DELETE method make changes to the resources. POST method is non-idempotent which means every time this method is applied it creates a new resource. Whereas, GET, PUT, DELETE are idempotent methods, which means the result of applying these operations to the resource once or multiple times is always same. For example, if DELETE method is used to delete the device, identified by "http://192.168.0.1/device/123", then the device with id 123 will be deleted. And it will not matter if the DELETE method on that resource is again performed, as the resource will not exist after the first delete operation.

2.1.3 Representation

Representation is used to represent the state of the resource and to perform the action on that resource. It is defined with a media type which is used by the recipient to process the resource representation [17]. On the web, resource representation includes popular media type HTTP and HyperText Markup Language (HTML). The media type such as Javascript Object Notation (JSON) and Extensible Markup Language (XML) are used widely for web services.

2.1.4 Statelessness

The clear division of application state between the client and the server is the key characteristic of RESTful architecture. REST is stateless, which means the server does not keep records of the information about the request made by clients. Each request is a unique request for the server and client must contain all the information required. The server does not know anything about the previous request or the future request by the client. Hence, the state of the resources is maintained by the server. And the client maintains its application state independent of the server.

2.1.5 Hypermedia Driving Application State

The client must have the knowledge of resource identifier to access the resources provided by the server. They can access these resources either by the dedicated discovery formats or they can follow the links provided in the resources. These links allow clients to use the resource identifiers and media type defined in the resource representation to access the resources provided by the server. Hence, the client can navigate around the resources using these links.

HTTP is a client-server based architecture [16], where clients send the request to the server and receive data as a response. This request-response pattern makes REST well suited for controlling and monitoring IoT devices. The most important thing in REST is the resource and in the context of IoT, these resource constrained devices itself can be the resources. IoT devices acting as the resource are identified using specific URI, operations that can be performed on these devices are available as the set of well-defined methods and the data they provide are represented with the

appropriate media type. These resources can be found by querying for the list of available IoT devices, which will provide the links to the resources with the resource identifiers. Future IoT devices will be powerful enough to run the Web server which will enable them to expose their own REST resources. IoT devices will be able to run different applications and resources exposed by them will be used for application deployment and management [5].

2.2 MQTT

MQTT is an application layer protocol which works on top of transport layer protocol TCP/IP for transferring messages. It is a protocol suitable for resource constrained devices and it was invented by Dr. Andy Standford-Clark and Arlen Nipper in 1999. It is lightweight, open, simple and easy to use protocol, which makes it ideal for communication in resource constrained environments such as IoT [4]. The design goal of MQTT is to ensure reliable message delivery in constrained environments such as low network bandwidth and unreliable networks for resource constrained devices. And the message delivery is done using client-server publish-subscribe messaging protocol [4].

2.2.1 Publish/Subscribe Pattern

In the client-server model, there is a direct coupling between the communicating client and server [24]. In contrast, in publish-subscribe (pub/sub) pattern both publisher and subscriber act as the client. A publisher that is sending the message does not need to know about the existence of the subscriber. Hence, pub/sub pattern decouples the publisher client and the subscriber client [31]. The pub/sub decouples publisher and subscriber such that they do not need to know each other's existence, they do not need to run at the same time and they can publish and subscribe asynchronously. The publishing and subscribing of messages are done to the central agent called the broker. The broker performs filtering and delivering the published message to the interested subscribers or group of subscribers. The broker decides on which subscriber will get what message based on the filtering of the messages.

2.2.2 Message Filtering

The broker uses message filtering such that each subscriber gets the messages based on their subscription. The pub/sub systems have message filtering options like Subject-based filtering, Content-based filtering, and Type based filtering [31], [15].

- Subject-based filtering

  This is the filtering based on subject or topic. The topic is attached to each message. Subscriber interested in the message will subscribe to the topic related to the message.

- Content-based filtering

  This is filtering based on the content of the message [34]. For example, filtering of the message containing temperature data that is higher than 25-degree Celsius.

- Type based filtering

  This is filtering based on type information like temperature, carbon dioxide etc. This type information is attached to each message.

2.2.3 MQTT Subject-based filtering

MQTT uses subject-based filtering of messages [3]. A subject or a topic is text-based hierarchical structure, which consists of one or more topic levels. The forward slash separates each hierarchy in the topic. For example,

```
/tut/tietotalo/firstfloor/luminance
```

Here, "tut" is the first topic level, "tietotalo" is second topic level and so on, in the topic hierarchy. This topic is self-explanatory as it uses texts, and these texts show the intent of the topic. Any subscriber subscribing to this topic will get the published message about the luminance in first floor of Tietotalo in TUT.

The subscriber can subscribe to the single topic as above or it can subscribe to multiple topics at once using wildcards. For example,

```
/tut/tietotalo/+/luminance
```

Above topic shows the use of the single level wildcard. This topic can be used to subscribe to the luminance data from all the floors of the Tietotalo in TUT.

```
/tut/tietotalo/#
```

This topic shows the use of the multilevel wildcard. It can be used to subscribe to all the data that can be collected from all the available floors of Tietotalo in TUT.

The topic is attached to each message published or subscribed by the client. And broker uses these topics to decide if a subscribing client will receive the message or not. The following section will discuss the working of MQTT Client and Broker.

2.2.4 MQTT Client and MQTT Broker

MQTT client can publish and subscribe messages simultaneously with specific topics to the broker. It can be a resource constrained device like a sensor, a backend-server or the graphical client which can communicate using MQTT.

MQTT broker is the central part of any publish/subscribe protocol. The broker is responsible for receiving all published and subscribed messages, filtering of those messages and deciding on the suitable subscribers for sending the message. Everything that publisher sends and subscriber receives must go through the broker.

2.2.5 MQTT Connection

The MQTT connection is always established between the client and the broker, two clients are never connected directly with each other. The client establishes the connection with the broker with the CONNECT message. In response to this CONNECT message broker sends CONNACK message. The return code in CONNACK message determines if the connection was successful or not. Figure 2.1 shows the MQTT connection establishment between the client and the broker. After the client connects successfully with the broker, it can now publish or subscribe messages to the broker.

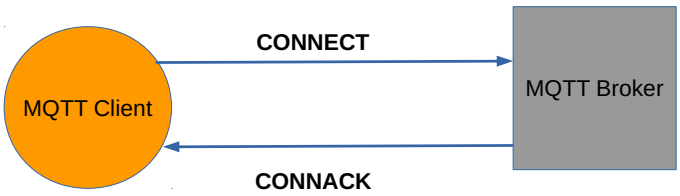


Figure 2.1 *MQTT connection*

PUBLISH	
topic	/devices/get
payload	"{'devices':'[list of devices]','response_id':'response_id'}"

Table 2.1 *PUBLISH message from RR*

2.2.6 MQTT PUBLISH and SUBSCRIBE

Each publish and subscribe message contains the topic which is used by broker for filtering of messages. The PUBLISH message contains the payload along with the topic. This payload is the actual content to be transmitted. It can contain binary data, textual data, and even JSON or XML data. Table 2.1 shows the content of the PUBLISH message.

The SUBSCRIBE message contains the list of topic names in which the subscriber is interested in. Hence, the subscriber can subscribe to one or more topics of interest. Table 2.3 shows the SUBSCRIBE message format.

Table 2.2 shows the simple representation of the PUBLISH message. It only shows the topic and payload among others as the content. Similarly, the table 2.3 shows

PUBLISH	
topic	/room1/temperature
payload	"temperature: 25-degrees"

Table 2.2 *PUBLISH message*

SUBSCRIBE	
topic1	/room1/temperature
topic2	/room2/temperature
topic3	/room3/temperature

Table 2.3 *SUBSCRIBE message*

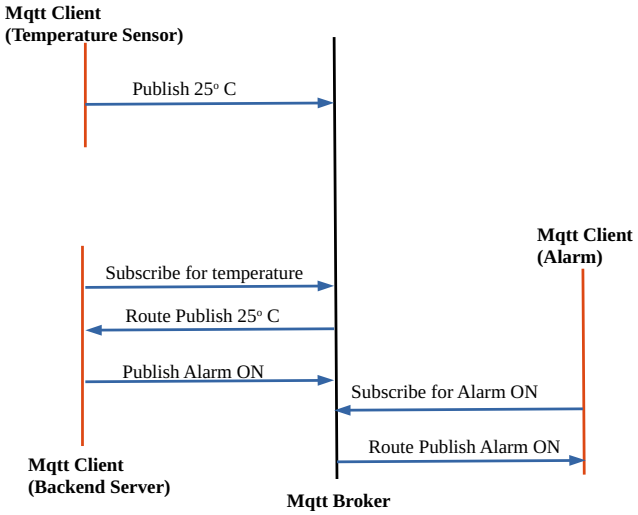


Figure 2.2 *MQTT clients and broker*

the simple representation of the SUBSCRIBE message. Actual PUBLISH and SUBSCRIBE message contains other contents as well, which are discussed in detail in [4], [2].

In MQTT publish/subscribe messaging system, the sender publishes data on specific topic and receiver subscribes to interested topics and gets the corresponding message through the broker. Figure 2.2 shows the working of MQTT clients and broker. It shows that the temperature sensor, backend server, and the alarm are all acting as MQTT client. Here, temperature sensor collects the data about the surrounding temperature and sends a PUBLISH message to the broker. This PUBLISH message consists of the temperature data and the topic attached to it. Then the broker will send this published data to the backend server which is subscribing to this topic

using SUBSCRIBE message. The backend server uses the data contained in the subscribed message and sends the PUBLISH message with data "Alarm on". The MQTT client alarm will be subscribing to this message, hence, the broker will send this published message received from backend server to Alarm, thereby turning the alarm on.

2.3 Comparison of REST and MQTT

This section compares the REST concepts to those of the MQTT and explores the relationship between them. Comparison of REST and MQTT is done similar to the one discussed in the paper [25], where REST and actor based computation model is compared with each other. REST is an architectural style, whereas MQTT is publish-subscribe based protocol, it seems that there is no similarity between them. However, the objective of the both REST and MQTT is same, that is the exchange of information between endpoints in the network. As per [17] REST consists of elements which are described in section 2.1. Everything in REST architecture works around resources. Hence, the resource is the main entity in REST.

Whereas, in MQTT, as per the paper [4] consists of elements which are described in section **??**, MQTT client can be service provider as well as the receiver. Hence, the client is the main entity in MQTT.

The following subsections discuss the comparison between REST elements and MQTT elements.

2.3.1 Client-Server/Publisher-Subscriber

In REST client refers to the requesting client and server refers to the provider of service corresponding to the request. Whereas, in MQTT both the publisher and subscriber are clients. In MQTT publisher and receiver are decoupled, they do not need to know each other's existence. In contrast, REST client must know the address of the server to send the request and get the response. In both REST and MQTT sender is providing services and receiver is accessing those services. The only difference in MQTT is the sending and receiving happens asynchronously with the help of central component called the broker.

REST	**M**QTT
Resources	MQTT client
URI	Topics
Client-Server	Publisher-Subscriber
Request-Response	Publish-Subscribe

Table 2.4 *Comparison of REST and MQTT elements*

2.3.2 URI/MQTT Topic

In REST, resources are identified by using URI, which is the location of the server and the resource available on the server. In MQTT, the topic is attached to the published message and subscription is made on the same topic to get that message. This topic is the hierarchical structure which enables broker to filter which subscriber receives the message from the publisher.

2.3.3 Request-response/Publish-subscribe

REST works on the principles of HTTP, and it uses request-response pattern. HTTP request-response is synchronous, unlike publish and subscribe in MQTT which is asynchronous. HTTP specifies set of methods and return values. MQTT does not have the concept of response and return value, but it can be implemented to send acknowledge topic which will be subscribed by the publisher of actual content.

2.4 IoT Application Development and Deployment Platform

2.4.1 LiquidIoT

Department of Pervasive Computing in the Tampere University of Technology has developed an IoT development platform called LiquidIoT [5]. LiquidtIoT can be used to create, deploy, run and manage IoT applications. This platform is inspired by principles of DevOps for the development and deployment of IoT applications The main objective of this development platform is to provide

- Integrated development, deployment, and management tool

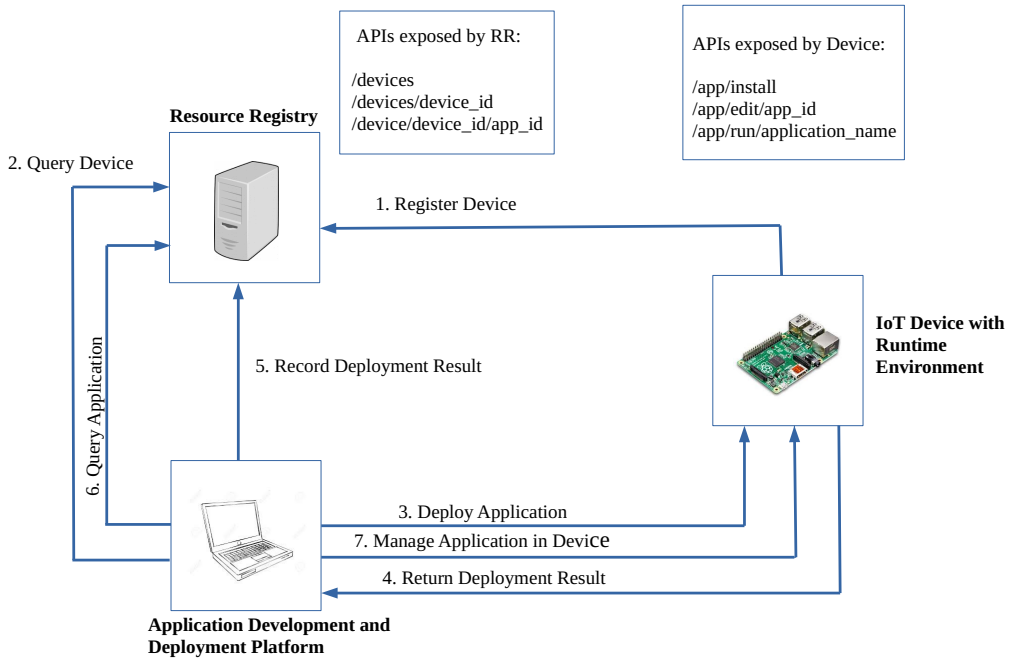


Figure 2.3 *LiquidIoT using REST API resources*

- Remote Management of IoT Devices and Applications

These objectives are fulfilled by the use of components which are the building blocks of the platform.

2.4.2 Components of LiquidIoT

The LiquidIoT platform has three main components, application development framework and web IDE, application runtime environment and resource registry [22].

- Application development framework and web IDE

IoT applications are developed and deployed using web-based IDE. Application development framework enables the creation of these IoT applications. These applications can function by sending the data collected by the IoT device and it can also provide services using RESTful APIs which can be accessed using HTTP protocol.

- Application runtime environment

  The runtime environment based on Node.js technology enables IoT devices to act as application servers. It allows IoT devices to execute applications and provide operations using RESTful APIs. Similarly, with the help of these APIs, it provides services for application deployment and management of applications.

- Resource Registry (RR)

  Resource Registry keeps tracks of all the available IoT devices and applications installed on these devices. It provides information about available devices, applications installed in the devices, list of operations that can be performed in the device as RESTful APIs. Similarly, it also keeps the record of application management such as, installation, updating, and deletion of application in the device.

The LiquidIoT platform uses HTTP as a communication protocol and RESTful APIs to provide services. The APIs provided by Resource Registry and IoT device both are RESTful APIs. The platform similar to LiquidIoT discussed in [5] was created with the set of RESTful APIs in both RR and IoT device.

2.4.3 LiquidIoT using REST

LiquidIoT platform described in [5], [22] uses API to specify how to interact with the Resource Registry and IoT device. A similar platform was created to make the comparison between REST and MQTT based implementation. This section discusses the REST based platform used in this thesis, and 3 will discuss the MQTT based implementation.

The programming language independent REST APIs was used to create resources as the interacting interface for clients. These interfaces are described using OpenAPI

specification [1] which can be used to create API documentation and code that implements the API logic. The OpenAPI specification defines the resource path and different operations that can be performed on these resources using corresponding HTTP methods. Operations are defined using input parameters and response with return types.

- Resource Registry provides RESTful APIs for discovering resources in IoT system. Resources can be devices, device capabilities, applications and application interfaces. RESTful APIs defined in Resource Registry are as follows

```
/devices
POST: Register the device
GET: Return all the registered device
```

This defines the resource with resource identifier "/devices" and method available on the resource is POST and GET. POST request to this resource will register IoT device with RR by sending a POST request to this API with device details as the payload. And, GET request to this resources will return the list of all the registered devices from the RR.

```
/devices/{device_id}/apps
GET: Returns installed applications in device given
the device_id
POST: Record the details of installed application in
the given device to RR
```

This defines the resource with two methods GET and POST. GET request to this resource will return the list of the application installed in the device. Whereas, POST request will create the record of the application installed in the IoT device. Similarly, the following resource is defined to record the details of updated or deleted application in the IoT device.

```
/devices/{device_id}/{app_id}
PUT: Record the details of updated application in the
```

```
given device, as per the device_id and app_id to RR
DELETE: Remove record of deleted application from the
given device to RR
```

- Runtime environment in IoT devices provides RESTful APIs for deployment and remote management of applications. The deployment of applications may provide additional APIs that provides an interface to interact with them. RESTful APIs defined in the IoT devices are,

```
/app/install
POST: Install application in the device
```

This defines the API with POST method. POST request will contain the application to be installed in the device as payload. POST request to this API will install the application on the device. Here, the device id is not included in the resource itself because the device is identified using IP address which is unique for every device.

```
/app/edit/{app_id}
PUT: Update given application in the device
DELETE: Delete given application in the device
```

This defines the API with PUT and DELETE method. PUT request to this resource will update the application with ID app_id. DELETE request to this resource will delete the given application with ID app_id in the device.

```
/app/run/getTemperature
GET: Return the result after execution of application
in the device
```

If application named getTemperature is deployed, then this resource will be available on the IoT device. GET request to this resource will run the application. The result of execution will be returned as per the operation available in the application.

2.5 Building Blocks of LiquidIoT using REST

This thesis is the work on the implementation of IoT development and deployment platform using REST API and MQTT topic, and comparison between each other. This section describes the building blocks of development platform which operate almost similar to the original implementation of LiquidIoT defined in [5]. The following explains the use of tools and operations which vary with the original implementation of LiquidIoT.

2.5.1 Development Tool

In the actual implementation of LiquidIoT, web IDE is the development tool. Web IDE is useful when development must be carried out in computers with low resources, saving time-related to configuration and task management such as running unit tests in parallel to the development. Hence, use of web IDE helps save time and resources, and it allows the developer to focus on the development rather than in the configuration and managing tasks.

In contrast, in this new implementation web-based IDE was substituted by the use of bash command-line terminal and Sublime Text IDE. For this thesis, RR and development tool existed in the computer used for development and Raspberry Pi was used as IoT device. Since all the tools and IDE are set up in the system and testing and deployment of the application were to be done in single IoT device, use of desktop IDE was similar to web-based IDE. Hence, desktop IDE was used as the development tool.

2.5.2 Deployment Tool

Docker [27] was used to deploy applications in the IoT device. Docker is not used in the original LiquidIoT platform, it was used for study purpose in this thesis work.

Docker image is a standalone package that is enough to run the software application [27]. This docker image is executable package and it is executed as docker containers. This concept of docker image was used to create the application image in this implementation. Docker image can be shared hence the applications that are to be installed are shared and will be accessible to all interested IoT devices. When

IoT devices download the image and execute it the application will run as docker container.

In this implementation, applications are run on IoT devices as docker containers. For every new and updated application, Docker image was built and corresponding containers were created. When delete operation was carried out on applications, corresponding docker image and its containers were deleted.

Building the docker image and running the Docker containers using simple Docker commands were easy. There was no problem in the performance while running docker containers in the local system. But considering the resource constrained nature of the IoT device, that is Raspberry Pi in this implementation, docker imposed some performance delays. Another issue with Docker was, whenever new application was installed new Docker container was created and it was required to be hosted on the new port. Hence, if there are two application installed in the device then request to two separate Docker containers must be made to retrieve the corresponding result.

2.5.3 Swagger framework

Swagger framework [30] was used to define the REST API in both RR and IoT device. This framework was easy to use and helpful to document the APIs. YAML syntax was not familiar at first to start, but later on, it was most helpful for the documentation of the API and creating API logic.

2.6 Reason for choosing MQTT

The HTTP request-response pattern is not suitable for communication with the device whose address is not known. For client-server request-response to work they must know each other's existence. Hence, in the network where NAT is used and IoT devices are behind NAT, REST resources exposed as IoT devices and resources exposed by IoT devices will not be accessible.

The REST-based implementation of LiquidIoT works well on the local network. But IoT devices are distributed across the wide area network. Wide area network connects multiple networks which make use of NAT (Network Address Translation).

NAT is used for security purpose and to translate private address of IoT devices to public address available on the Internet. IoT devices behind the NAT will have private local addresses which will not be known to other networks. Since IoT device will not be addressable, REST implementation of LiquidIoT will not be able to deploy the applications to that devices or make any request to the interface available in the device.

In MQTT there is no problem with clients behind a NAT because MQTT client communicates with the broker which can be accessed on the Internet. Also, MQTT client sends the first message to the broker as a CONNECT message, thereby allowing connection open from the broker to the client and vice-versa. Hence, this thesis is a work on using MQTT for IoT application development and deployment.

As stated earlier REST-based implementation of LiquidIoT will face issues in addressing the IoT devices that are behind NAT. Since the address of IoT device behind NAT is not known, the connection cannot be established for request-response. An alternative approach which works in all network situations was required for the LiquidIoT platform. MQTT is growing technology in the context of resource constrained devices like M2M and IoT [4]. As LiquidIoT is a platform for application development and deployment for IoT devices, MQTT is suitable in this case regarding the resource constraints. Moreover, in MQTT publisher and subscriber are decoupled in time, space and synchronization [2],[15]. This means publisher and subscriber do not need the address of each other, they do not need to run at the same time and execution of the operation in publisher will not disturb the operation in the subscriber. Hence, they are totally independent of each other. This feature favors MQTT to work in every network environment and this approach is suitable for the operation of the LiquidIoT platform.

The original LiquidIoT platform works on the principle of HTTP request-response mechanism. The client sends the request to the server for accessing the resources they provide. The status of every request is returned with HTTP response. For example, web IDE sends the request to IoT device to install the application on the device. IoT device responds with the result of application installation to the web IDE. In this case, web IDE is a client and IoT device is a server. MQTT does not have the concept of request-response, however, it can be managed to work like request-response. This can be done by publishing the additional response topic by the requester as a result of the successful subscription. The transition from REST

to MQTT must be done in such a way that it does not affect the functionality of the LiquidIoT platform. And, this transition should only focus on the communication between the entities in the LiquidIoT platform. Hence, implementation of LiquidIoT using MQTT was carried out in following steps,

- Using MQTT elements in place of REST for the IoT application development and deployment, and
- Mapping HTTP request-response mechanism using MQTT publish-subscribe pattern.

3. IMPLEMENTATION

This chapter describes the implementation of MQTT in place of REST by mapping REST resources to MQTT topics and HTTP request-response to publish-subscribe messages.

At first, mapping of REST resources to MQTT topics is defined. However, only mapping of REST resources to MQTT topics will not explain the request-response mechanism used in LiquidIoT. Hence, the second section shows the mapping of request-response to publish-subscribe. In request-response, messages are sent from one endpoint to other directly. But, publish-subscribe messaging is performed using the broker. Hence, unlike request-response MQTT client sends subscribe messages to all the interested topics to the broker. And when the broker receives the matching published message, it routes the message to the interested subscribing clients.

HTTP response includes status code to acknowledge the status of the message delivery. This acknowledgment sending is not built-in in publish-subscribe. Hence, the third section describes the implementation of the status code in the MQTT topic.

3.1 Implementation of MQTT

We can implement the MQTT in place of REST in following ways,

- First approach is to use the MQTT broker and MQTT client. The broker will be the central component between publisher and subscriber. And all other components in the platform, that is IoT device, RR and application development IDE will be MQTT clients. The broker will handle the management of messages that are published and subscribed by the MQTT clients. It will keep the record of all the subscribers and decide on which subscriber will get the published message. It will make the decision based on the topic-based filtering of the message. Each published and subscribed message will have topic

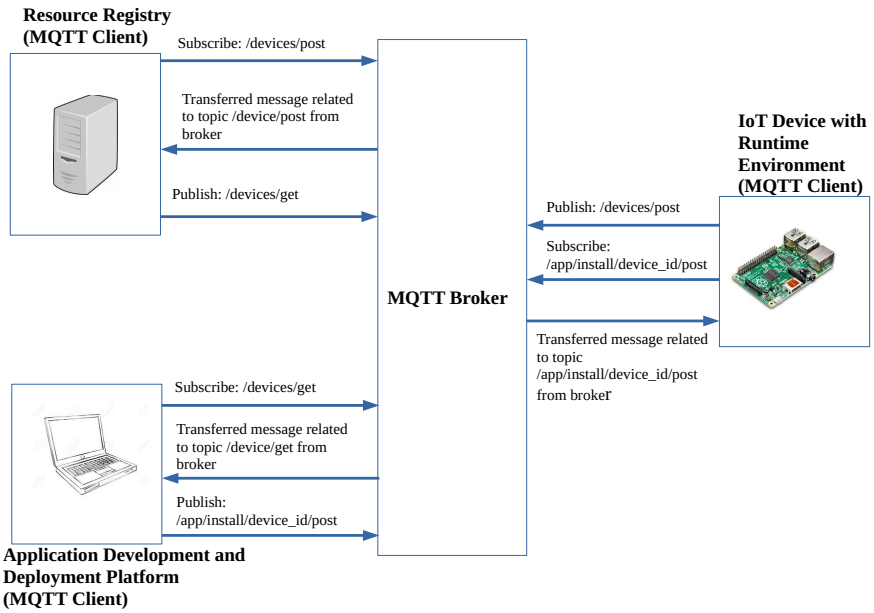


Figure 3.1 *LiquidIoT using MQTT*

attached to it and broker will make use of this topic to filter the interested subscriber. There can be many subscribers interested in the same topic, in this case, the broker will send the published message to all the subscribers whose topic of interest matches with the published topic [4].

- Second approach is to use the enhanced broker which translates REST resources into MQTT topics and vice-versa, like the QEST broker defined in paper [11]. In this approach, only the IoT devices will be acting as MQTT clients. Since the broker has built-in REST bridge which translates REST resources into MQTT topics, other components in the platform, that are IDE and RR can use HTTP requests. For example, the resource /tut/tietotalo/-firstfloor/temperature will be exposed as topic /tut/tietotalo/firstfloor/temperature. HTTP GET request to this resource will get the response that

consists of the published value issued with the topic. Similarly, HTTP PUT request to resource /tut/tietotalo/firstfloor/temperature will create a publish message containing the request body with the corresponding topic.

Both the approach works in the network with NAT, which causes the issue in the original LiquidIoT platform. The first approach is straightforward where no extension to the broker is required. It only requires the work of translation of communication model from HTTP request-response to MQTT publish-subscribe pattern. But in second approach broker must be equipped with handling both REST resources and MQTT topics. Also, the broker must be capable of translating from REST resources to MQTT topics and vice-versa. This may complicate the implementation of MQTT in LiquidIoT, as this approach focuses more on enhancing the broker rather than changing the model of communication from REST to MQTT.

LiquidIoT platform was implemented using MQTT, where no additional implementation in the broker was required. Other approach described above needed the enhancements in the broker, which enables it to handle both REST resources and MQTT topics. Since this latter approach makes ground for the different research topic in itself, in this thesis, we are focusing just on the implementation of the broker as the message filtering and delivery agent between the MQTT clients.

For the MQTT implementation, Mosca npm package [9] was used as a MQTT broker and for the client Mqtt.js npm package[10] was used. The implementation is shown in the figure 3.1. It is discussed below using an approach similar to the one used in [25]. An example related to REST resources is translated to corresponding MQTT topics. Then the following sections will discuss on translating HTTP request-response to publish-subscribe adding response codes to MQTT messages.

3.1.1 Mapping REST resources to MQTT topics

Mapping of REST to MQTT was first done by mapping the REST resources to corresponding MQTT topics. The following will show the example of REST resources and then map those resources as MQTT topics.

	/devices	**/devices/12345**
GET	List the registered devices, return 200	List details about device with ID 12345, return 200
POST	Register the device, return 201	Install application denoted in the payload of POST request, return 201
PUT	Method not defined, return 405	Update the details of the device with ID 12345, return 200
DELETE	Method not defined, return 405	Delete the details of device with ID 12345 from RR, return 200

Table 3.1 *REST resources provided by Resource Registry*

Example using REST resources

We now take an example of getting the list of IoT devices from RR and application management in IoT device, in the context of LiquidIoT using REST. For simplicity, we are not discussing the development and deployment methods in this example. Thus, the example comprises of services provided by RR and services provided by the device for create, read, update and delete operations on the application. These services provided by RR and device are RESTful resources in the context of REST, whereas they are topics in the MQTT context. In REST resources must have resource identifier, such that the resources can be identified. Similarly, in MQTT, topics must be defined by the publisher such that interested subscriber can get this published message from the broker.

First let's consider the device listing services provided by the RR the main resource is the /devices. The subset of this resource is another resource /devices/12345 which gives the information about the individual device with ID 12345.

In REST, resources are implemented as uniform interfaces and they are defined with the set of HTTP methods. Table 3.1 shows that resource /devices have uniform interface with support for GET and POST methods. Resources are identified in REST with the resource identifier or URI. For instance, http://192.168.0.10/devices/12345 is a URI with host address denoted by http://192.168.0.10 and resource name /devices/12345. The host address will give the address of the server and resource name will address the resource available on the server. However, in MQTT address of the server is not necessary as MQTT clients communicate with the help of the broker.

Hence, we will be mapping the resource name part of the URI in MQTT.

In the context of MQTT, these resources are mapped into topics. As MQTT uses topic-based filtering of messages, MQTT clients will publish and subscribe to broker on the specific topic. Unlike URI, the topic does not have the location or the address information of the publisher or subscriber. It denotes an intent to subscribe the published message and an intent to publish the message to the interested subscriber. Topic has specific hierarchical structure which resembles the hierarchy in the resource identifier. Both topic and URI uses the slash "/" character to separate the hierarchy. Hence, REST resources were mapped into the topic name that resembles the resource name.

The mapping of resource /devices can be done with the single level topic /devices. Since topics must be specific, and it was not clear from the topic /devices that if we are mapping the method GET or POST that are defined in the REST context. So, the method name was included in the topic. For example,

```
/devices
Methods available:
GET
```

This resource with GET method was mapped to,

```
/devices/get
```

Similarly, REST resource with both GET and POST methods are shown below,

```
/devices/device_id
Methods available:
GET
POST
```

This resource was mapped into two specific MQTT topic as,

```
/devices/device_id/get
/devices/device_id/post
```

Method	**R**EST Resources	**M**QTT Topics
GET	/devices	/devices/get
POST	/devices	/devices/post
GET	/devices/device_id	/devices/device_id/get
POST	/devices/device_id	/devices/device_id/post
PUT	/devices/device_id	/devices/device_id/put
DELETE	/devices/device_id	/devices/device_id/delete

Table 3.2 *Mapping RR resources to topics*

	/app/install	**/app/edit/app_id**
POST	Install application in device, return 201	Method not defined, return 405
PUT	Method not defined, return 405	Update the application with ID app_id in the device, return 200
DELETE	Method not defined, return 405	Delete the application with ID app_id in the device, return 200

Table 3.3 *REST resources provided by IoT device*

As HTTP methods defined in the REST resources, the topic specific to those HTTP methods is created. Hence, the topic was used to map the different HTTP methods defined in the REST resources. Table 3.2 shows the mapping of resources in RR to the MQTT topics.

Table 3.3 shows the REST resources exposed by IoT device and HTTP method setup for the resource. Table 3.4 shows the mapping of resources in IoT device to MQTT topics. Mapping is done such that the MQTT topics resembles the corresponding REST resource. In Table 3.4 we can see that MQTT topics contain the additional device_id in the topic hierarchy as compared to the resources. This was required since the topic must be specific to unique device denoted by its ID. It was not

Method	**R**EST Resources	**M**QTT Topics
POST	/app/install	/app/install/device_id/post
PUT	/app/edit/app_id	/app/edit/device_id/app_id/put
DELETE	/app/edit/app_id	/app/edit/device_id/app_id/delete
GET	/app/run/getTempApp	/app/run/getTempApp/device_id/get

Table 3.4 *Mapping IoT device's resources to topics*

required in the REST resources because the address of every device was known before request-response.

Mapping Request/Response to Publish/Subscribe

Above Table 3.2 and 3.4 show the resources being mapped to corresponding topics. It can be seen from the table that resources and topics resemble each other. But it's not clear from the topic only, which MQTT client is the sender and which one is the receiver. Hence, after mapping the resources to suitable topics, these topics must be published and subscribed to map the request-response pattern used in REST. When using publish-subscribe the messages are not directly sent to the corresponding MQTT clients, but it goes through the central broker. This broker handles the filtering of publish-subscribe messages and delivery of the published message to the interested client. Hence, in this implementation, MQTT clients send subscribe and publish the message in advance to the broker.

Let's consider the resource "/devices" which is mapped to topics "/devices/get" and "/devices/post" as shown the in Table 3.2. IDE sends GET request to resource "/devices" of RR. This resource is mapped to topic "/devices/get". This means IDE subscribes to this topic, which is similar to HTTP request, and RR will publish device information by using the same topic which resembles HTTP response. The payload of the PUBLISH message will be the list of registered devices. This publish/subscribe message is sent to the broker which handles the delivery of the published message to the subscribing client, which is IDE in this case. As broker is the first place to receive both the published and subscribed messages, MQTT client does not know when the message was published or subscribed. Hence, the RR publishes the devices lists to broker using topic "/devices/get" whenever new devices are registered.

Similarly, POST request from IoT device to resource "/devices" is mapped to topic "/devices/post". In this case, IoT devices send publish message as request to register. RR sends subscribe message and waits for the publish message from the interested device which desires to register. Hence, IoT device will PUBLISH the message with this topic. The payload of the PUBLISH message will contain the ID of the IoT device. RR subscribing to this publish message will register the device and publishes response message which resembles HTTP response for the requesting

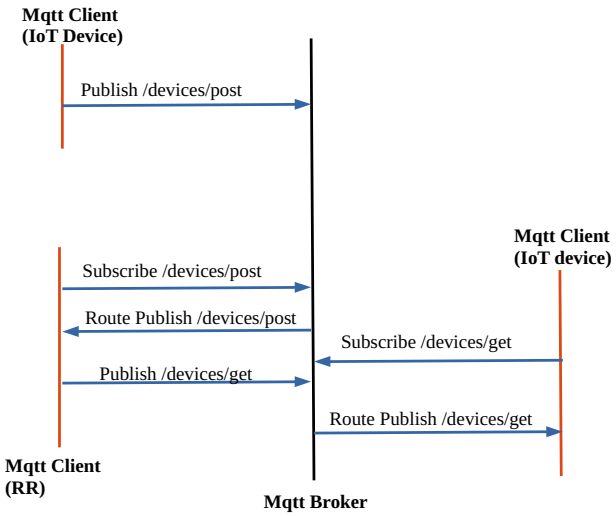


Figure 3.2 *Publish-Subscribe mapping of request-response on "/devices" resources*

IoT device. This publish-subscribe mechanism is shown in the figure 3.2.

We can see from the figure 3.2 and 3.3 that the MQTT clients publishes and subscribes messages to the broker. For instance, RR publishes the list of registered devices with topic /devices/get. Here, RR does not know how many clients subscribe to this topic, and it does not know if the message was delivered successfully to subscribing client or not. In MQTT there is no feature of acknowledging the publisher about the status of the message delivery to the subscriber. But here we are using MQTT in request-response style, hence the acknowledgement of message delivery must also be implemented in MQTT. The following section will discuss on mapping HTTP response codes in MQTT.

3.1.2 Mapping HTTP response codes in MQTT

REST resources have the uniform interface with the predefined set of methods and response with the return value. REST uses HTTP request-response pattern, where every HTTP request will have the response with corresponding HTTP status code. In MQTT, clients can publish and subscribe the message, but subscriber does not

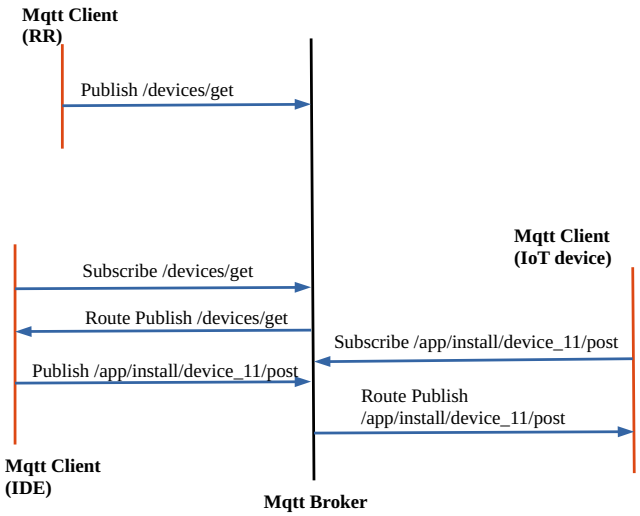


Figure 3.3 *Publish-Subscribe mapping for app installation in device*

send the response back to the publisher. Publisher does not need to know if the published message will be subscribed or not. It does not even have an idea about the subscriber who has subscribed to the published message. But to map the REST to MQTT in LiquidIoT, we must map the HTTP request-response. Hence, we can implement an additional response message which will be published by the recipient of the message. This response message will also contain the status code similar to the HTTP status code. We can map the HTTP request-response using following methods,

- First approach is to include the response code in the payload itself. The recipient of this message can find the status of the message by extracting the response code from the payload. In this approach, first, there will be publish-subscribe messaging to transfer the response ID to the sender. Then the sender will first subscribe to the response topic using the response ID that it got from the previous publish-subscribe, and then publish the topic with the content. The receiver who is interested in this content will first subscribe to the topic of interest to the broker. After getting the publish message, the broker will filter the topic and send the published message to interested subscriber or receiver.

In the response, the receiver will publish the response message using the topic containing response ID to the broker. This response topic is same as that subscribed by the sender which will get the response from the payload of the message.

- Second approach is to include the status code in the MQTT topic hierarchy. In this approach, the sender will first subscribe to the response topic containing status code, then publish the message. For example, if the sender publishes with topic /tut/tietotalo/firstfloor/temperature, before publishing it will subscribe to the message /tut/tietotalo/firstfloor/temperature/200. The receiver subscribing to topic /tut/tietotalo/firstfloor/temperature will get the message from the broker. As a response subscriber will publish the message to topic /tut/tietotalo/firstfloor/temperature/200. This topic is already subscribed by the sender, hence it will get the response message from the broker. Hence, by subscribing to the topic with response codes, the publisher can get the status of the message delivery to the subscriber. The main downfall of this approach is that publisher must subscribe to all the possible topics with different response code in advance before publishing the content message. This can use lots of resources in subscribing and unsubscribing the message. IoT devices are very large in number and they are growing every day. Hence, this approach may not suit the resource constrained nature of the IoT devices as there can be many subscribing and unsubscribing of the message to the broker for single request-response. And the available resource might all be consumed in subscribing and unsubscribing.

Let's consider the Table 3.1, it shows that for every method defined on the resource there is a numerical return code. These codes are HTTP status code defined in HTTP specification [16]. For instance, let's take the GET method defined in the /devices resource. The successful operation of GET request on this resource will return 200. Similarly, PUT method is not defined in this resource, so PUT request to this resource will return 405. The following sections will show the implementation of HTTP style response code in MQTT based publish-subscribe pattern.

PUBLISH	
topic	/devices/get
payload	"{'devices':'[list of devices]','response_id':'response_id'}"

Table 3.5 *PUBLISH message from RR*

IDE query RR for list of registered devices

The HTTP style response code was mapped to MQTT using two approaches as discussed in section 3.1.2. In the first approach, response code was included in the payload of the message. Let's consider there are one RR and one IDE for simplicity, such that we do not need to consider the case where there can be multiple IDEs as MQTT client. The following defines the mapping of resource "/devices" with GET method,

1. IDE sends SUBSCRIBE message to BROKER with topic /devices/get. This resembles the HTTP request to /devices resource.
2. RR sends PUBLISH message to BROKER with topic /devices/get which resembles the HTTP response from the RR. This PUBLISH message contains the payload with list of registered devices and the response_id as shown in Table 3.5
3. BROKER will transfer the PUBLISH message to IDE. IDE will extract the response_id from the payload.
4. RR sends SUBSCRIBE to topic /devices/get/{response_id} to get the acknowledge message from the IDE in successfully getting the list of registered devices.
5. IDE sends PUBLISH message with topic /devices/get/{response_id}. This PUBLISH message contains the payload with response code as shown in table 3.6
6. BROKER will filter the topic and send this publish message to RR which is expecting the response.

The response ID exchanged between IDE and RR will be unique, which will be generated by the RR in this scenario. To implement the response code in MQTT

PUBLISH	
topic	/devices/get/response_id
payload	"{status:200}"

Table 3.6 *PUBLISH message from IDE*

additional publish-subscribe messages were required and also dynamic topic creation using response ID was also required.

The mapping of POST method defined for resource "/devices" is similar, that will contain the 201 status code in the payload of PUBLISH message. And for two methods PUT and DELETE which are not defined for resource "/devices", RR will PUBLISH a message which will contain 405 status code in the payload. The following shows the pub/sub mapping of PUT method on resource "/devices",

1. RR sends SUBSCRIBE message to BROKER with topic /devices/put
2. IDE sends SUBSCRIBE message with topic /devices/put/response_id and then PUBLISH message to BROKER with topic /devices/put and the payload containing response_id.
3. BROKER will transfer message from IDE to RR
4. RR extracts the response_id and PUBLISH message to BROKER with topic /devices/put/response_id. This PUBLISH message contains the payload with response code 405.
5. BROKER will transfer message from RR to IDE which is expecting a response

Hence, we need extra PUBLISH and SUBSCRIBE messages to map the response sending behaviour of HTTP. In the case of sending 405 response code, there is no actual content transfer between publisher and subscriber. Although the number of publish-subscribe messages is same as in the case of GET or POST method, these publish-subscribe messages are used just for replying with 405 response code. Considering the resource constrained nature of IoT devices these additional publish-subscribe messages can consume valuable resources which can lead to performance problems.

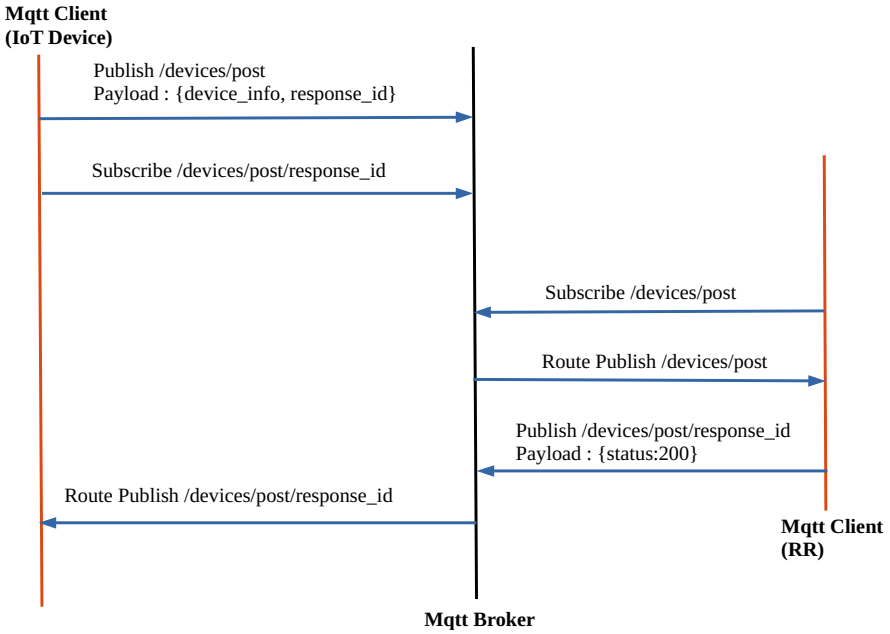


Figure 3.4 *IoT device registration with response code*

IoT device registration with RR

Figure 3.4 shows the device registration with RR and acknowledgement from RR. It shows the sequence of messages sent from MQTT client to broker and message routing from broker to corresponding MQTT client. Registration request from the IoT device is sent as publish the message to broker using topic /devices/post. RR will be subscribing to this topic and hence broker will route this publish message to RR. IoT device will be listening for the status of registration using subscribe message with response id. After successful registration RR will send status 200 using publish message which will be routed to IoT device by the broker.

Similarly, figure 3.5 shows the application installation using IDE to the IoT device. This figure also shows the request-response between the IDE and IoT device using publish-subscribe messaging. Additionally, it also shows the exchange of status code between IDE and IoT device to determine the installation status of the application in the device. After getting the success status published by IoT device, IDE publishes message which contains the app details and response code, which will be routed to

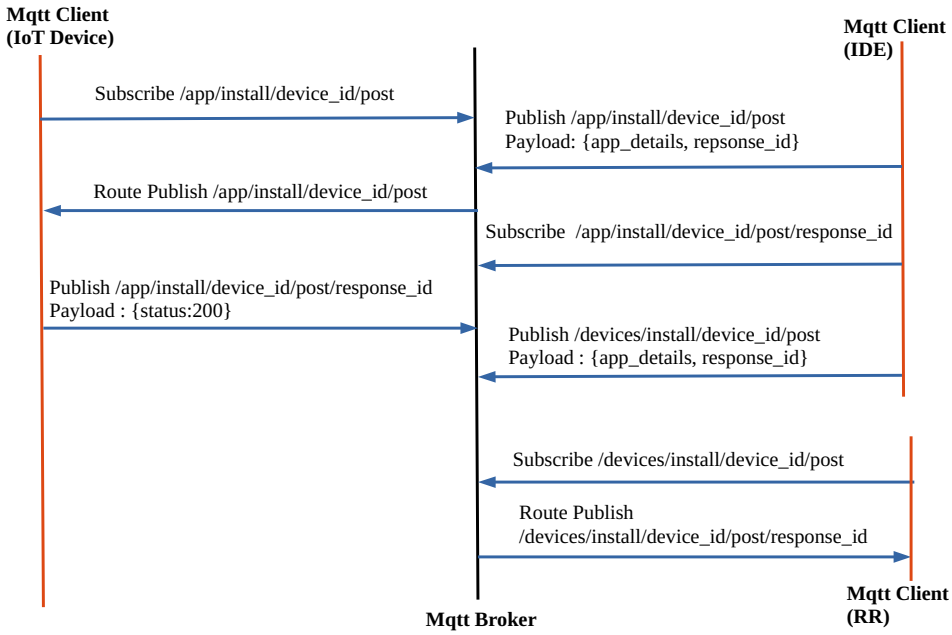


Figure 3.5 *Application installation in IoT device with response code*

subscribing MQTT client RR through the broker.

Using response codes in MQTT topic

The second approach for mapping response code was to use response code in the topic itself. The MQTT client which needs response will subscribe to the topic with response code in the topic hierarchy. This topic will be specific to corresponding status code, hence broker can filter and transfer response to the client with ease. The following shows the publish/subscribe mapping of GET method on resource "/devices" using this approach

1. IDE sends SUBSCRIBE message to BROKER with topic /devices/get, which resembles HTTP request
2. RR sends PUBLISH message to BROKER with topic /devices/get, which resembles HTTP response.

3. Simultaneously, RR sends SUBSCRIBE message to BROKER with topic /devices/get/200. This is the additional request from RR to get the response code from the IDE.

4. BROKER will transfer the PUBLISH message after filtering of topic to IDE

5. IDE sends PUBLISH message to BROKER with topic /devices/get/200 as the response to acknowledge that list of devices is received successfully.

6. BROKER will filter the topic and send this publish message to RR which is expecting the response.

This approach also needed extra PUBLISH and SUBSCRIBE messages like the first approach. In addition, in this approach, if we need to address available status code that is possible, then there will be the separate topic for each status code. As a result, this additional topic will add extra work of SUBSCRIBING and UNSUBSCRIBING for each status code, which will degrade the performance of the system. Hence, the first approach is feasible with respect to the second one.

4. EVALUATION

4.1 Results

The MQTT implementation of LiquidIoT worked similar to the REST-based implementation while considering the performance. The tests were done using RR, IDE and IoT devices as MQTT clients. During testing desktop-based IDE and one IoT device were used. Raspberry Pi was used as IoT device in which Docker-based runtime environment was installed. This runtime environment enabled the IoT device to the application server. RR, IDE and IoT devices were all MQTT clients which can publish and subscribe messages with specific topics. Along with these three components, there was central broker which handled all the published and subscribed messages.

The new implementation was used to test in both the local private network and the network with NAT environment. As MQTT clients communicate through the central broker, the network with NAT did not create any issue with the new implementation. IoT device first makes connect request to broker during subscribing or publishing, which enables the two-way communication between the broker and IoT device. Hence, NAT did not create issues that were seen in the REST-based implementation. As a result, the MQTT based implementation worked in NAT environment as expected.

Both the REST and MQTT implementation were tested for average time for message transmission. The average time was calculated using "process" module of Node.js in the IDE in case of both REST and MQTT. The following operations of the system were tested,

- Registering IoT device to RR

  Average time taken was recorded for registration of IoT device to RR. Device details were included in the payload of the published message which were

extracted by the RR to register the specific device. The payload size was in bytes as the device details was sent as JSON. In REST average time taken for registration process was 355 milliseconds and in the case of MQTT, it was 350 milliseconds.

- Getting list of devices from RR

  The list of devices was added to the payload of the published message by RR. This list contains the ID of all the devices that have been registered with the RR. In REST-based implementation, average time taken for getting the list of devices from RR was 75 milliseconds, whereas the average time was 73 milliseconds in case of MQTT.

- Application management in IoT device

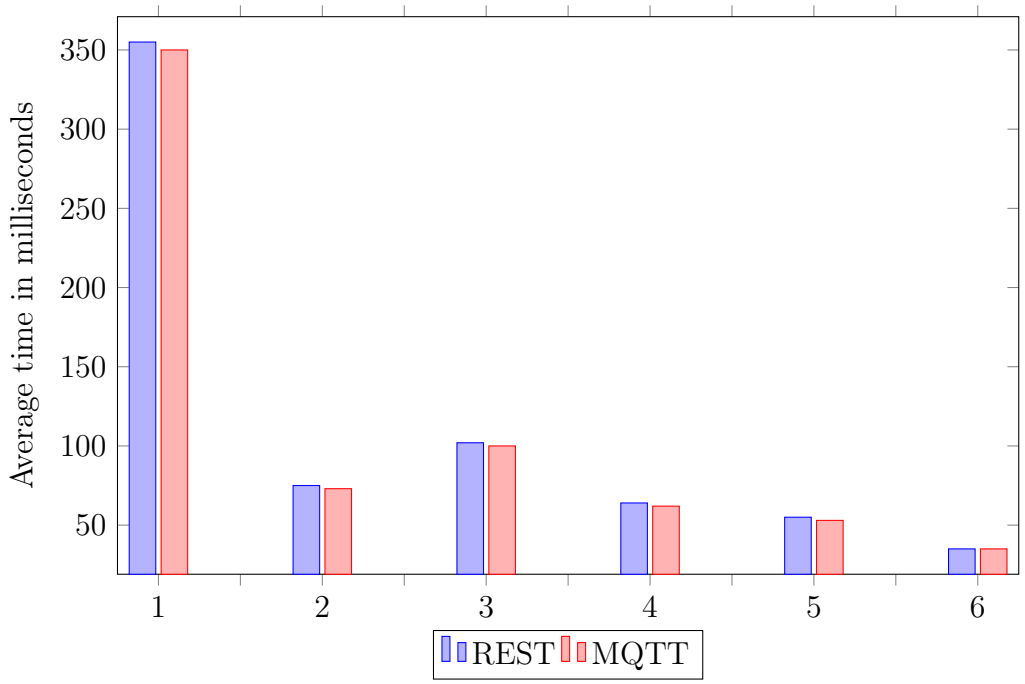
  In the case of installation of the application, information about the application and location to fetch the application was added as JSON in the payload of the published message by the IDE. Similarly, ID of the application which must be updated or deleted in the device was also added as JSON in the payload of the published message. In REST-based implementation installation of the application in device took the average time of 25 seconds, whereas average time for installation was 24 seconds in case of MQTT. This average time was measured in seconds due to the use of Docker as deployment agent. Docker took almost all of the time to execute and run the application on IoT device.

  Similarly, the average time for the update was 64 milliseconds for REST and 62 milliseconds for MQTT. And, the average time for delete operation was 55 milliseconds for REST and 53 milliseconds for MQTT.

- Execution of application in IoT device

  The average time of application execution was in average 35 milliseconds using REST and 35 milliseconds using MQTT. The result of execution was added as payload by the IoT device to the subscribing IDE.

The average time of execution for above-listed operations in both REST and MQTT based implementations are shown in the diagram 4.1. This diagram shows that the performance was almost similar in both the implementation. MQTT based implementation which was constructed to match the request-response pattern of REST still performed almost similar to the REST-based approach.



1. *Registering IoT device to RR*
2. *Getting list of devices from RR*
3. *Installation of application to device*
4. *Updating application in the device*
5. *Deleting application from the device*
6. *Executing application in the device*

Figure 4.1 *Comparison of Average time taken*

There was no performance issue while testing due to the payload size since the payload contained JSON whose size was not more than few kilobytes. Device ID, List of the registered device, Response codes and the result of application execution were all sent as payload to the subscribing client. In the case of application installation, the performance was little slow in both the implementation. However, this was not due to the MQTT or REST-based implementation. The issue was due to the use of Docker since it took all the time of installation for its internal functioning.

The testing environment did not contain a large number of IoT devices and IDE, and the network errors and network delays were not considered. Hence, tests may

vary for these conditions.

4.2 Related Work

There are many studies on using publish-subscribe messaging as means of communication for resource constrained devices. [36], [21], [34] discusses on the use of publish-subscribe as communication protocol for Wireless Sensor Networks (WSN). In WSN, sensors and actuators may change network address at any time, network links are likely to fail, and failed WSN nodes are replaced by new nodes. As publish-subscribe is asynchronous and it does not need to know about the existence of other endpoints in the network, it is best suited for WSNs. Also, it is the common and widely used variant of data-centric communication. These features make publish-subscribe suitable for asynchronous communication in the distributed network. However, these papers discuss the use of publish-subscribe in the general terms for WSNs and not the IoT as a whole.

Different protocols may be appropriate for different situations regarding the necessities of the IoT system. These papers [6], [33], [13] discusses on the comparison of different lightweight protocols regarding the data transmission time from endpoints and the bandwidth consumption in the IoT system. In this thesis, we are not comparing two different protocols, but we are comparing two different message passing mechanisms, one is MQTT publish-subscribe protocol and other is the REST architectural style. We are taking the performance evaluation into account, but most of the thesis compares between MQTT and the REST architectural style in the context of IoT application development and deployment platform. And, it compares the use of MQTT in the request-response style which is similar to request-response in REST.

This paper [29] analyzes both request-response and publish-subscribe as communication model in ubiquitous systems. The integration of request-response and publish-subscribe communication model is discussed to fulfill the need for the system that requires features of both. The simultaneous use of request-response and publish-subscribe is proposed for easy development of software solutions that required both synchronous and asynchronous communications. This paper brings the concept of using both request-response and publish-subscribe as a solution to implement benefits of both approaches, with higher abstraction level which is technology independent. This is a different way develop the IoT system. The devices that support

asynchronous communication will use publish-subscribe pattern and those devices that require client-server interaction will imply request-response pattern. Hence, the use of publish-subscribe will eventually make the system to work properly in NAT environment as well.

The request-response using publish-subscribe messaging system are discussed in [20], [12]. These papers describe the usefulness of replies in the publish-subscribe system. The PUB protocol described in [12] is similar to the approach described in this thesis, where the publisher of the original message subscribes to reply message in advance. And when the receiver publishes the reply message, the only original sender receives it. However, these papers discuss the implementation of publish-subscribe with the reply in general for systems that needed both data-push between endpoints and client-server interaction. And this thesis is based on implementing MQTT publish-subscribe protocol to map request-response for overcoming specific network problems in IoT.

The approach similar to this thesis is described in this paper [26]. This paper also describes the implementation of IoT-specific protocol MQTT for the LiquidIoT platform. And the motivation behind this paper is also same as for this thesis, that is to overcome the issue of NAT that is seen in REST-based implementation of LiquidIoT. According to this paper, MQTT is used in request-response style. In this approach unique topic is used for every request-response such that broker can handle filtering and delivering the message as per the request. The original sender of the message subscribes to a unique topic for getting the response and the receiver of the message sends the response message with payload which consists of the status code. The topic hierarchies are designed in such a way that topic for response message is derived automatically that matches the original sender's subscribe message. According to this paper, request ID is used in topic level for each request-response. This request ID is used for directing corresponding response message from the receiver to the designated sender. Hence, the use of request ID makes each request-response distinct and there is no mismatch in delivering response code. In this thesis, the status code is sent as the payload of the message. But topic creation for sending response uses the different approach. The sender of the message will add the response ID to the published message showing that it is interested in getting the response. Along, with that sender will subscribe to the topic which has response ID in its topic hierarchy. Then to send the response receiver will extract the response ID from the payload of the original message and publish the response

code to the topic containing response ID. This topic will match the topic subscribed by the sender.

A different approach for utilizing features of both REST and MQTT is described in [11] and [19]. These papers describe the use of bridge between REST and MQTT. [11] implements the new kind of broker called QEST broker which exposes MQTT topics to REST resources and vice-versa. This simplifies the communication between the entities in the network that can talk REST and MQTT both. Similarly, [19] describes the use of REST bridge which is used to translate the REST resources into MQTT topics. This paper discusses using the standard HTTP methods to interact with the MQTT devices for easy user interaction. Both the papers [11] and [19] focuses on extending the system by adding the feature which can translate REST resources to MQTT topics and vice-versa. However, the approach discussed in this thesis just requires the mapping of HTTP request-response into MQTT style request-response.

One reason for choosing MQTT for LiquidIoT platform is, MQTT has no issue related to NAT. These papers [32], [8], [35], [23] discuss about the benefit of using MQTT in the NAT environment. In case of protocols like HTTP which uses address client and server to make the connection, NAT becomes an issue. Different NAT traversal mechanisms, proxy servers are used to bypass the NAT. However, MQTT does not require any extra servers or functionality to be added. MQTT just works without any issue in the network with NAT.

5. CONCLUSION

In this document, MQTT was used as a protocol for the deployment and management of IoT application. IoT application development and deployment platform was built with reference to the LiquidIoT platform discussed in [5], [22]. Both REST and MQTT based platform was created and performance was measured for different functions it provides. REST-based platform was mapped into MQTT-based platform, where HTTP request-response was mapped to MQTT style request-response. MQTT itself does not have the concept of request-response, so publish-subscribe based messaging was used in a manner that replicated the request-response pattern. In addition, the HTTP response sending with the status code was also implemented using publish-subscribe. MQTT was used in place of REST because MQTT works without issue in the NAT environment. In REST based platform IoT devices behind NAT was not accessible for deployment or management of applications, as IDE was not aware of the address of the IoT device. This did not have any issue in MQTT as the MQTT clients do not need to know the existence of other clients in the network for sending or receiving the messages.

This document considers that the IoT devices are behind NAT and MQTT broker is connected to the Internet. Since, MQTT broker is the central component to which the IoT devices, IDE and RR publish-subscribe the messages, the existence of the broker must be known to all the MQTT clients. And, if the broker itself is behind a NAT, then the publish-subscribe can only be done by the clients in the network similar to that of the broker. Similarly, there will be an issue if we consider the IoT devices in different network conditions. MQTT is an application layer protocol which works on top of TCP/IP. If there are IoT devices that work on ZigBee [37] network or network with UDP as transport protocol then MQTT cannot communicate with these devices. In this scenario, the lightweight version of MQTT that is MQTT-S [36], [21] can be used which works with IoT devices in different networks.

MQTT protocol can be used for reliable delivery of messages. In this document, we

are using Quality of Service (QoS) 0 which is the lowest level of reliability offered by the protocol. However, MQTT protocol also provides QoS 1 and QoS 2 [4] which give more reliable delivery of the message, in addition to delay in message delivery. MQTT specification [4] also provides the feature of authentication, which we have not discussed in this document. For secure delivery of message, these authentication systems can be implemented in the future work.

REFERENCES

[1] "Open API Initiative," Available: https://openapis.org.

[2] "Mqtt essentials part 2 - Publish & Subscribe," 2015, Available: http://www.hivemq.com/blog/mqtt-essentials-part2-publish-subscribe.

[3] "Mqtt essentials part 5 - mqtt topics & best practices," 2015, Available: http://hivemq.com/mqtt-essentials-part-5-mqtt-topics-best-practices.

[4] "Mqtt version 3.1.1 plus errata 01. Edited by Andrew Banks and Rahul Gupta 10 December 2015," OASIS Standard Incorporating Approved Errata 01. http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/errata01/os/mqtt-v3.1.1-errata01-os-complete.html, Dec 2015, Latest Version: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html.

[5] F. Ahmadighohandizi and K. Systä, "Application development and deployment in IoT," *In: CLIoT 2016: The 4th Workshop on Cloud for IoT*, 2016.

[6] M. H. Amaran, N. A. M. Noh, M. S. Rohmad, and H. Hashim, "A comparison of lightweight communication protocols in robotic applications," *Procedia Computer Science*, vol. 76, pp. 400–405, 2015.

[7] L. Atzori, A. Iera, and G. Morabito, "The internet of things: A survey," *Computer networks*, vol. 54, no. 15, pp. 2787–2805, 2010.

[8] P. Bellavista and A. Zanni, "Towards better scalability for iot-cloud interactions via combined exploitation of mqtt and coap," in *Research and Technologies for Society and Industry Leveraging a better tomorrow (RTSI), 2016 IEEE 2nd International Forum on.* IEEE, 2016, pp. 1–6.

[9] M. Collina, "moscaâmqtt broker as a module," 2013, Available: https://www.npmjs.com/package/mosca.

[10] ——, "Use mqtt from the browser, based on mqtt. js and websocket-stream," 2013, Available: https://www.npmjs.com/package/mqtt.

[11] M. Collina, G. E. Corazza, and A. Vanelli-Coralli, "Introducing the qest broker: Scaling the iot by bridging mqtt and rest," in *Personal indoor and mobile radio*

communications (pimrc), 2012 ieee 23rd international symposium on. IEEE, 2012, pp. 36–41.

[12] G. Cugola, M. Migliavacca, and A. Monguzzi, "On adding replies to publish-subscribe," in *Proceedings of the 2007 inaugural international conference on Distributed event-based systems.* ACM, 2007, pp. 128–138.

[13] L. Durkop, B. Czybik, and J. Jasperneite, "Performance evaluation of m2m protocols over cellular networks in a lab environment," in *Intelligence in Next Generation Networks (ICIN), 2015 18th International Conference on.* IEEE, 2015, pp. 70–75.

[14] B. Emmerson, "M2m: the internet of 50 billion devices," *WinWin Magazine*, vol. 1, pp. 19–22, 2010.

[15] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM computing surveys (CSUR)*, vol. 35, no. 2, pp. 114–131, 2003.

[16] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Rfc 2616-http/1.1, the hypertext transfer protocol," 1999.

[17] R. T. Fielding and R. N. Taylor, "Principled design of the modern web architecture," *ACM Transactions on Internet Technology (TOIT)*, vol. 2, no. 2, pp. 115–150, 2002.

[18] D. Guinard, V. Trifa, and E. Wilde, "A resource oriented architecture for the web of things," in *Internet of Things (IOT), 2010.* IEEE, 2010, pp. 1–8.

[19] S. Herle, R. Becker, and J. Blankenbach, "Bridging GeoMQTT and REST," *In: Geospatial Sensor Webs Conference*, August 2016.

[20] J. C. Hill, J. C. Knight, A. M. Crickenberger, and R. Honhart, "Publish and subscribe with reply," VIRGINIA UNIV CHARLOTTESVILLE DEPT OF COMPUTER SCIENCE, Tech. Rep., 2002.

[21] U. Hunkeler, H. L. Truong, and A. Stanford-Clark, "Mqtt-sâa publish/subscribe protocol for wireless sensor networks," in *Communication systems software and middleware and workshops, 2008. comsware 2008. 3rd international conference on.* IEEE, 2008, pp. 791–798.

[22] O. Hylli, A. Ruokonen, N. Mäkitalo, and K. Systä, "Orchestrating the internet of things dynamically," *In: To appear in first International Workshop on Mashups of Things and APIs (MoTA) co-located with MIDDLEWARE 2016*, 2016.

[23] K. Kondratjevs, N. Kunicina, A. Patlins, A. Zabasta, and A. Galkina, "Vehicle weight detection sensor development for data collecting in sustainable city transport system," in *Power and Electrical Engineering of Riga Technical University (RTUCON), 2016 57th International Scientific Conference on*. IEEE, 2016, pp. 1–5.

[24] S. Kratky and C. Reichenberger, "Client/server development based on the apple event object model," *Atlanta*, 2013.

[25] J. Kuuskeri and T. Turto, "On actors and the rest." in *ICWE*. Springer, 2010, pp. 144–157.

[26] A. Luoto and K. Systä, "Iot application deployment using request-response pattern with MQTT," 2016.

[27] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, no. 239, p. 2, 2014.

[28] S. Nath, "Senseweb: An infrastructure for shared sensing," in *US-Korea Conference on Science, Technology, and Entrepreneurship (UKC)*.

[29] C. Rodríguez-Domínguez, K. Benghazi, M. Noguera, J. L. Garrido, M. L. Rodríguez, and T. Ruiz-López, "A communication model to integrate the request-response and the publish-subscribe paradigms into ubiquitous systems," *Sensors*, vol. 12, no. 6, pp. 7648–7668, 2012.

[30] Swagger, "A framework for api definition," Available: https://swagger.io.

[31] S. Tarkoma, *Publish/Subscribe Systems: Design and Principles*. John Wiley & Sons, 2012, ch. 7. Distributed Publish/Subscribe.

[32] J. Taylor, H. S. Hossain, M. A. U. Alam, M. A. A. H. Khan, N. Roy, E. Galik, and A. Gangopadhyay, "Sensebox: A low-cost smart home system," in *Pervasive Computing and Communications Workshops (PerCom Workshops), 2017 IEEE International Conference on*. IEEE, 2017, pp. 60–62.

[33] D. Thangavel, X. Ma, A. Valera, H.-X. Tan, and C. K.-Y. Tan, "Performance evaluation of mqtt and coap via a common middleware," in *Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), 2014 IEEE Ninth International Conference on.* IEEE, 2014, pp. 1–6.

[34] X. Tong and E. C. Ngai, "A ubiquitous publish/subscribe platform for wireless sensor networks with mobile mules," in *Distributed Computing in Sensor Systems (DCOSS), 2012 IEEE 8th International Conference on.* IEEE, 2012, pp. 99–108.

[35] M. Uehara, "A case study on developing cloud of things devices," in *Complex, Intelligent, and Software Intensive Systems (CISIS), 2015 Ninth International Conference on.* IEEE, 2015, pp. 44–49.

[36] B. Weiss, U. Hunkeler, A. Munari, W. Schott, and L. Truong, "A publish/subscribe messaging system for wireless sensor communication."

[37] A. Zigbee, "Zigbee specification," *ZigBee document 053474r13*, 2006.