國立交通大學

資訊科學與工程研究所

碩 士 論 文

將 MQTT 協定資源整合至符合 ETSI M2M 標準之物聯網平台

Converging MQTT resources in ETSI standards based M2M platform

研　究　生：陳祥文

指導教授：林甫俊　教授

中 華 民 國　一 百 零 三 年 七 月

將 MQTT 協定資源整合至符合 ETSI M2M 標準之物聯網平台

Converging MQTT resources in ETSI standards based M2M platform

研 究 生：陳祥文　　　　Student：Hsiang Wen Chen

指導教授：林甫俊　　　　Advisor：Dr. Fuchun Joseph Lin

國 立 交 通 大 學

資 訊 科 學 與 工 程 研 究 所

碩 士 論 文

A Thesis

Submitted to Institute of Computer Science and Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer Science

July 2014

Hsinchu, Taiwan

中華民國一百零三年七月

將 MQTT 協定整合至符合 ETSI M2M 標準之物聯網平台

學生：陳祥文　　　　　　　　　　　　**指導教授：林甫俊**

國立交通大學資訊科學與工程研究所

摘要

如何整合多樣通訊技術與方法是物聯網相關領域的一大挑戰。在 2013 年的科技部深耕計畫中，我們在從德國 Fraunhofer FOKUS 引進的符合 ETSI M2M 標準的物聯網平臺 OpenMTC 上開發了不同的 M2M 應用，並示範使用這類型物聯網平臺的好處。我們使用了 HTTP 協定去整合非 ETSI M2M 相容的裝置，但還是有一些問題尚待解決。論文中我們提出了一個方法透過新設計的 handler application “MQTT Proxy” 將 Message Queuing Telemetry Transport (MQTT)協定整合進 ETSI M2M 的架構中。MQTT Proxy 一方面對 MQTT clients 而言是個 MQTT broker，另一方面也是個與 OpenMTC 溝通的 Gateway Application。我們比較了 MQTT Proxy 和 HTTP Proxy 不同的地方，並且介紹如何使用 MQTT Proxy 在 OpenMTC 上開發 M2M 的應用。

關鍵字：MQTT; ETSI M2M; OpenMTC

Converging MQTT resources in ETSI standards based M2M platform

Student: Hsiang-Wen Chen   Advisor: Dr. Fuchun Joseph Lin

Department of Computer and Information Science

National Chiao Tung University

Abstract

One of the key challenges of the Internet of Things (IoT) is the integration of heterogeneous technologies and communications solutions. In our 2013 MOST Deep Plowing Project, we developed different applications on OpenMTC, which is an ETSI standards based M2M platform developed by Fraunhofer FOKUS, to demonstrate the advantage of using a common service platform. We created handler applications using HTTP protocol to bridge non ETSI compliant devices into the OpenMTC, but still some issues remained to be solved. This thesis paper proposes a method to integrate Message Queuing Telemetry Transport (MQTT) protocols with the ETSI M2M architecture via new handler applications called "MQTT Proxy". The MQTT Proxy, on the one side, acts as an MQTT broker to the MQTT clients. While on the other side, it serves as a Gateway Application (GA) for interfacing with the OpenMTC. We compare the difference between the MQTT Proxy and the HTTP Proxy, and introduce how to develop M2M applications with the MQTT Proxy.

Keywords: MQTT; ETSI M2M; OpenMTC

誌謝

這篇論文能夠完成，首先感謝林甫俊教授的教導與指引。在一次又一次教授為學生指導論文的過程中，除了提供學生寶貴的教導與建議，更從教授身上學習到做研究所須的態度以及方法。特別感謝科技部深耕計畫和德國 Fokus 公司，提供經費以及 OpenMTC 平台讓我能夠完成我的研究。

感謝網聯世界技術實驗室的所有成員，在這兩年的研究所生涯中，不論是生活、課業、或研究上給予我許多的支持與建議，讓我能順利完成我的研究，並擁有兩年精彩的碩士生活。

最後感謝我的家人和朋友，感謝你們長期對我的陪伴以及照顧，讓我能專注於課業與研究上，祥文在此衷心的感謝。

Contents

List of Figures

List of Tables

Chapter 1 Introduction

In the past two decades, we have witnessed how Information and communication technologies (ICT) changed the way we live. The Internet of the Things (IoT), which includes many different fields of knowledge, is going to be the next landmark for ICT. Though there are only 1% of things in our world that are connected [1], this condition will be changed in the near future, and everything networked will become the norm. Ericsson predicts there will be 50 billion devices connected to the Internet by 2020 [2]. How to connect, communicate and utilize those 50 billion devices efficiently imposes a big challenge.

MQTT [3] was invented by IBM and Eurotech in 1999 as an extremely lightweight messaging transport protocol based on TCP/IP. Google trend analytics shown in Figure 1 indicates that people start to be interested in MQTT after IoT got significant attention. The MQTT protocol uses Publish/Subscribe models which is different from REST based Request/Respond models like HTTP, and has lots of advantages over HTTP in mobile application development [4]. It is currently being standardized by OASIS [5].

The ETSI (European Telecommunications Standards Institute) M2M architecture is one of the IoT technologies developed by European Telecommunication Standard Institute (ETSI) as a global M2M standard to converge diverse vertical applications onto a common platform. ETSI defines an M2M functional architecture and the interfaces required to support end-to-end services. A set of functionalities in the M2M core forms the service capabilities layer (SCL), and can be utilized by M2M applications through REST interfaces. We used OpenMTC [6], an ETSI-compliant M2M platform developed by Fraunhofer FOKUS for our M2M common service platform testbed. The OpenMTC platform provides the M2M software middleware, enabling the M2M functionalities, and providing convenient API for developers to implement M2M mechanisms more easily.

In the 2013 MOST Deep Plowing Project, our group developed different kinds of applications, such as Building Energy Saving, Bus Tracking, Smart Lighting, and Facility View, on the OpenMTC to demonstrate the advantage of using a common service platform. As most of the devices do not support ETSI M2M in the current stage these devices should communicate with the platform through the interworking proxy capability in the ETSI M2M architecture. However, the detailed specifications of the interworking proxy capabilities for existing sensor protocols are not released yet. Hence, we implemented the functionality of interworking proxy capability using “protocol adapter” and “handler application”, which allows to different protocol devices to communicate with each other.

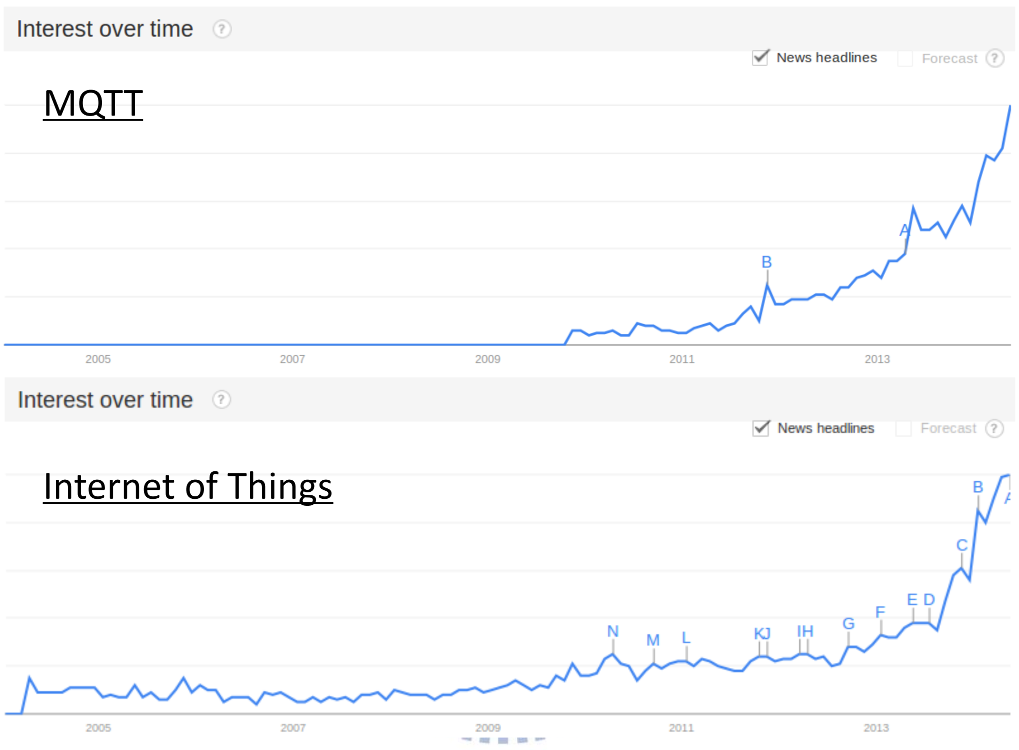


Figure 1: Google Trends Analytics of “MQTT” and “Internet of Things”

Even though our previous implementations worked pretty well, the way of implementing M2M applications on top of OpenMTC could still be improved. In this thesis, we proposed to converge MQTT resources in the ETSI standards based M2M platform. We developed a method to bridge MQTT protocols with the ETSI M2M architecture and implemented it as MQTT Proxy running on the M2M gateway. Moreover, we also leveraged the open source MQTT client and open source hardware platforms, such as Raspberry PIs, Arduino to verify our design and demonstrate how to develop M2M applications on the OpenMTC with the MQTT Proxy.

The rest of the thesis is structured as follows. Chapter 2 presents the background, the related work and explains the difference between these efforts and ours. Chapter 3 describes the problem statements, and Chapter 4 describes how to enable MQTT on OpenMTC. Chapter 5 compares between the MQTT Proxy and the HTTP proxy, and Chapter 6 introduces how to develop M2M application on OpenMTC using the MQTT Proxy. Finally, we made a conclusion and discussed potential future works in Chapter 7.

Chapter 2 Background

This chapter provides an overview of the MQTT protocol, as well as the ETSI M2M architecture and its interworking proxy capability. It also discusses the related works.

2.1 MQTT protocol

MQTT was invented by IBM and Eurotech in 1999, and is currently being standardized by OASIS. It is an M2M connectivity protocol designed as an extremely lightweight publish/subscribe messaging transport. MQTT is based on a client/server model, where every MQTT client should first connect to a central server, so called MQTT broker. After the connection is established, MQTT clients can publish/subscribe messages based on the notion of "topic" through the broker. An example of how MQTT works is shown in Figure 2: In the stage 1 all of the MQTT clients have to connect to the MQTT broker. In the stage 2, Client B and Client C subscribed to the topic "tp_A", so in the stage 3 they will receive messages from the MQTT broker after the broker received message with the topic "tp_A" which is published by the Client A. Detailed MQTT V3.1 protocol specifications could be downloaded online [7].

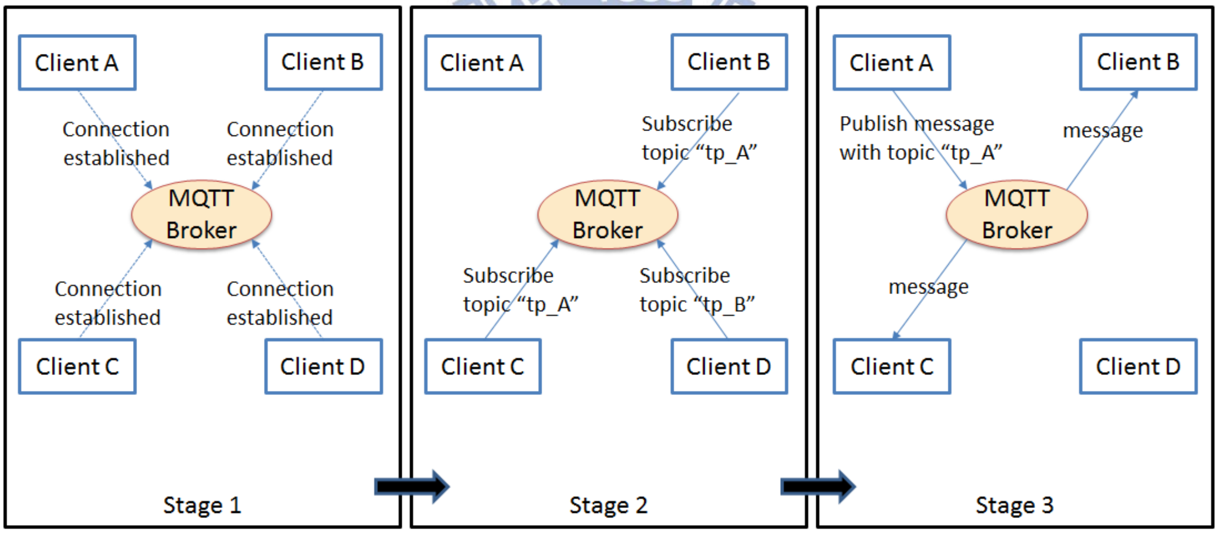


Figure 2: An MQTT example

There are also many useful built-in support features in MQTT but not in HTTP. Those features are summarized in Table 1, for example three-level quality of service, keep-alive messages, retained messages, clean sessions, last wills, and durable connections.

Table 1: Support features in MQTT not in HTTP

Feature Name	Explanation
Quality of service	MQTT defines three quality of service (QoS) levels for message delivery, namely “at most once”, “at least once”, and “exactly once” from low to high QoS. Higher QoS levels (0~2) ensure more reliable message delivery but consume more network bandwidth and cause higher latency.
Retained messages	The broker will keep the message even after sending it to subscribers. If a new subscription is submitted for the same topic, the MQTT broker will send retained messages to the new subscribing client.
Clean sessions and durable connections	The client and the broker will maintain session state information to ensure “at least once” and “exactly once” delivery. When a client connects to the broker, the broker will check whether the session information has been saved from a previous connection. If the session information exists and the clean session flag is set to true, the previous session information is removed. If the clean session flag is set to false, the previous session is resumed. If no session information exists, a new session is started.
Last will and testament	When a client connects to the broker, it can inform the broker that it has a will .If the client disconnects with the broker unexpectedly, the broker will send a “last will and testament” publication to subscribers who must know immediately when the connection is broken.

2.1 ETSI M2M Architecture

Due to the widespread deployment of 3G/4G mobile networking technologies and the proliferation of devices and sensors, the requirements for an M2M common service platform emerge from new business drivers. The ETSI TC (Technical Committee) M2M, established in Jan. 2009, developed an end-to-end high level architecture for M2M from telecommunication perspective. They identified the requirements of the M2M common platform, as shown in Figure 3, by use case studies such as connected consumer, automotive applications and smart metering, and provided the specifications for an M2M functional architecture in M2M Release

1 specifications [8]. These specifications are completed at the end of 2011. It then follows by Release 2 specifications that are completed at the beginning of 2013.

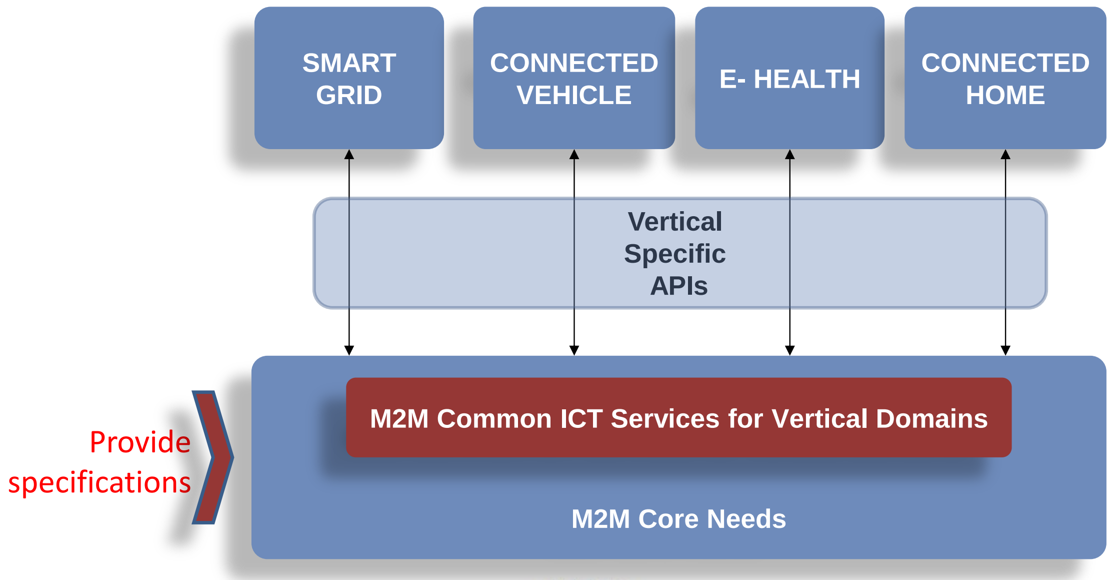


Figure 3: Common horizontal service layer

The ETSI M2M architecture consists of three domains: Device Domain, Network Domain and Application Domain as shown in Figure 4. NSCL, GSCL, and DSCL in Figure 4 are service capabilities layers for ETSI standard M2M management functions. They simplify and optimize M2M application development and deployment by exposing network functions through open interfaces. NSCL is the network service capabilities layer in the network and application domain. It provides M2M functions for communicating with GSCL or DSCL, and is shared by M2M applications in the Application Domain. GSCL and DSCL are the gateway service capabilities layer and the device service capabilities layer in the network and device domain respectively. They provide M2M functions for communicating with NSCL. Devices with DSCL can directly connect to NSCL, but M2M devices without DSCL have to connect to NSCL through the GSCL.

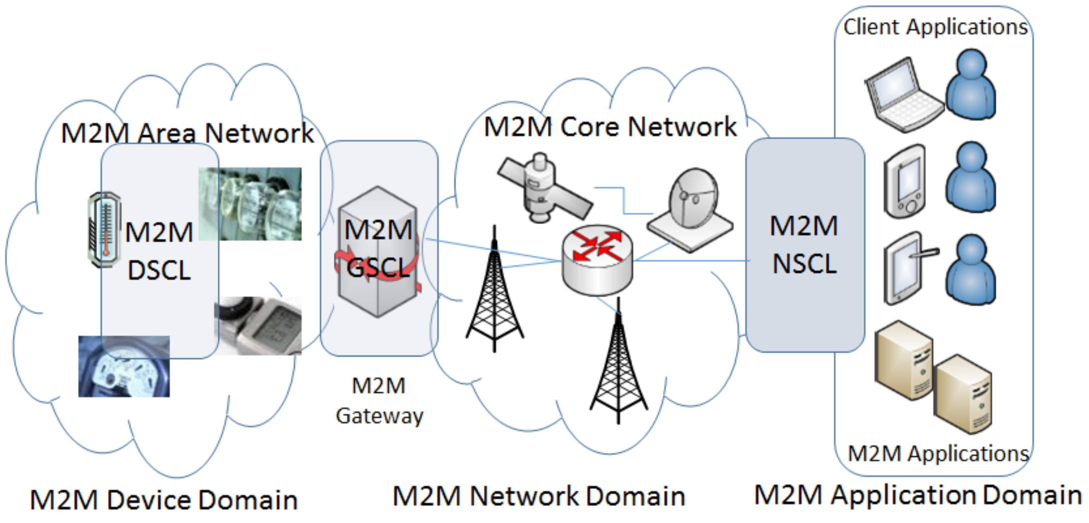


Figure 4: ETSI M2M architecture

The SCLs expose M2M functionalities through a set of service capabilities and open interfaces that can be shared by different applications. These service capabilities and open interfaces in GSCL and NSCL are shown in detail in Figure 5. The service capabilities include (underlying "x" could be replaced by N for network, and G for gateway): Application enablement (xAE), Generic communication (xGC), Reachability, addressing, and repository (xRAR), Communication seletion (xCS), Remote entity management (xREM), SECurity (xSEC), History and data retention (xHDR), Transaction management (xTM), Compensation broker (xCB), Telco operator exposure (xTOE), Interworking proxy (xIP). The open interfaces include mIa, dIa, and mId. The mIa allows M2M applications access M2M service capabilities in the NSCL. The dIa allows an application in an M2M device or M2M gateway to access M2M service capabilities in the DSCL or GSCL. The mId allows a DSCL or GSCL to communicate with NSCL and vice versa. Detail specifications of these interfaces are defined in TS 102 921 [9].

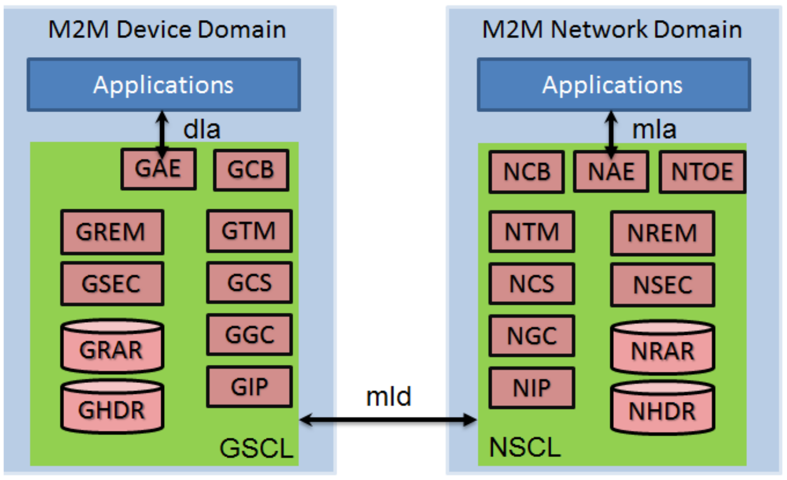


Figure 5: M2M capabilities and interfaces in GSCL and NSCL

2.3 Interworking proxy capability

ETSI M2M TS102 690 specifies an interworking proxy capability that provides interworking between non ETSI compliant devices and the SCL. However, no detailed specifications are provided in the ETSI specifications. The first related technical report TR 102 966 [10], interworking between the M2M Architecture and M2M Area Network technologies, is released in Feb. 2014. However, this report focuses on the resource representation of ZigBee, UPnP, and KNX protocols, and the mapping principles with the ETSI M2M resource tree.

There are some newly added functionalities for GIP/DIP in the Release 2 of TS102 690 specification. These newly added functionalities provide interworking between some M2M Area Networks and the GSCL/DSCL. GIP/DIP can be designed either as an internal capability of GSCL/DSCL or an application communicating via reference point dIa with GSCL/DSCL. These explanations correspond well to our previous experience of designing an interworking proxy capability in the SCL to service various end device. It is feasible to design an application that interfaces with the M2M Area Network while communicates via reference point dIa with GSCL/DSCL. We had done such handler applications we did in our previous work [11].

2.4 Related Works

Interworking between MQTT and REST has been proposed in the past. The following are two examples:

- **QEST Broker**

QEST Broker [12] is a new design of MQTT broker that can bridge two totally different protocols, MQTT and REST. It modifies the broker semantic to retain and syndicate the last payload seen on the MQTT topic, and exposes MQTT payload as a REST resource. The QEST broker architecture can expose MQTT topics as REST resources or REST resources as MQTT topics.

- **OSIOT**

OSIOT [13] developed and promoted open source standards for emerging IoT. The core components proposed in their horizontal platform are IoT Toolkit and Smart Object API, based on event driven programming paradigm and REST API resource structures. MQTT has been adopted as one potential event-driven communication method used to propagate events between REST endpoints.

The previous work such as QEST Broker and OSIOT enhances interworking of MQTT through REST API. However, such capability hasn't been incorporated in a standard platform. In the thesis, we aim to improve our way of developing M2M applications on OpenMTC using MQTT protocol. Hence, we focus on enabling MQTT in the ETSI M2M standard architecture. In this work, an "MQTT Proxy" based on the javascript version of MQTT broker implementation "mosca" [14] has been developed and integrated seamlessly with OpenMTC.

Chapter 3 Problem Statement

It is predicable that there will be a huge amount of devices connected around us, and one of the critical needs for IoT/M2M development now is a standardized common platform. ETSI M2M specifications are the one spearheading such an effort with International collaboration, and OpenMTC is the first prototype of ETSI-compliant M2M platform developed by Fraunhofer FOKUS. In our 2013 MOST Deep Plowing Project, we developed different kinds of applications, such as Building Energy Saving, Bus Tracking, Smart Lighting, and Facility View on OpenMTC to demonstrate the advantage of using a common service platform.

In the current stage, most devices do not support ETSI M2M service directly. Thus, these devices are required to communicate with the platform through the interworking proxy capability in the ETSI M2M architecture. However, no detailed specifications of the interworking proxy capabilities for existing sensor area network are available yet. We realized the functionality of interworking proxy capability using “protocol adapter” and “handler application”. The “protocol adapter” translates the protocol from sensor area network, and the “handler application” receives incoming data via HTTP and interfaces with the ETSI M2M architecture. The whole system diagram of our previous work is shown in Figure 6.

Even through the whole system can work, there are still some issues remained to be solved:

- **Reusability of the handler applications**

Handler applications received incoming data via HTTP and interfaced with the ETSI M2M via dIa. Unfortunately, each application defined its individual criteria about the formats of http requests and how to interface with the ETSI M2M. If there is a new requirement (e.x. supporting new type of devices) coming to our system, we have to redesign our handler applications to satisfy the new requirement. This kind of approach is obviously not suitable for highly flexible M2M application deployment. On the other hand, programming the handler application needs the knowledge of ETSI M2M. If the developers are not familiar with ETSI M2M, developing new M2M applications will be even more difficult because they have to build a particular handler application first.

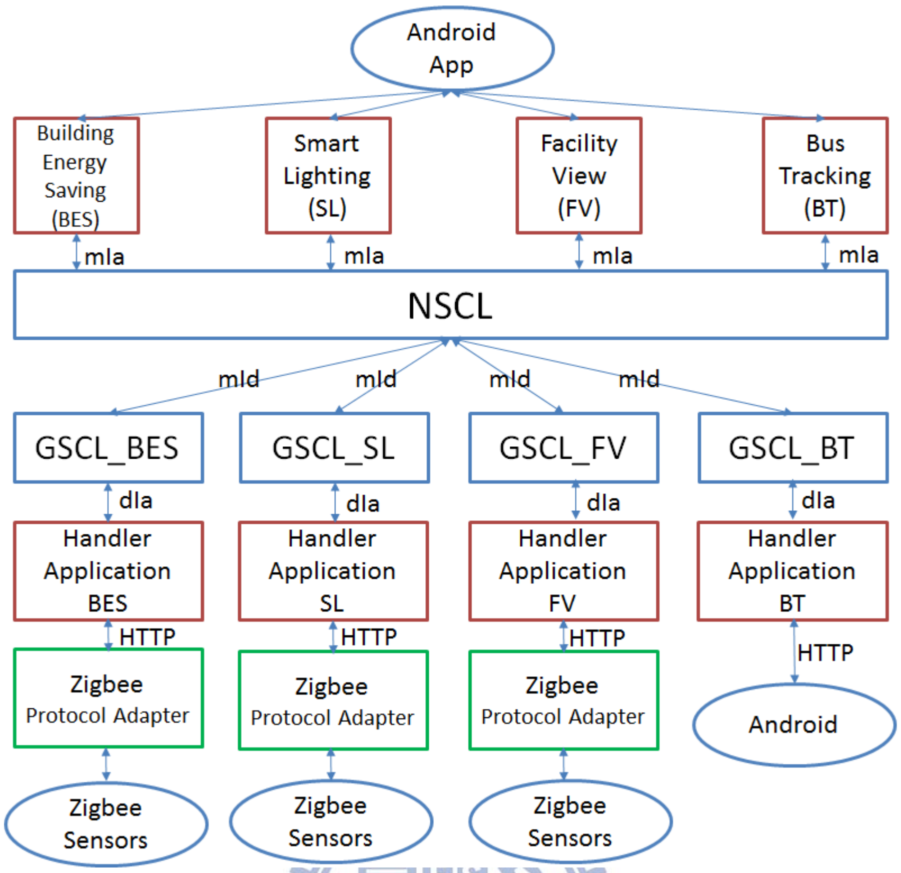


Figure 6: Whole system diagram in our previous work

- **Pushing data to clients**

In most of the M2M applications, clients need to frequently receive data or events from the M2M platform. In our previous approach, the way of receiving data from handler applications is using HTTP polling, which means the clients send the http request to handler applications in a regular interval to get the latest data. However, the suitable time interval is hard to determine. Moreover, it is not suitable for frequent data changing applications as it often consumes a lot of power in sending frequent requests, and adds the burden of http server.

To solve these problems, two solution strategies are possible:

First, instead of using many handler applications to serve different types of http requests, we should design a handler application with the general criteria which could be followed by most of the applications. For example, we can indicate which "container ID" and "resource tree of SCL" the incoming data belongs to, so the handler application knows where to store the data in the ETSI based M2M architecture.

Second, to solve the pushing data problem we faced in our previous approach, we should keep the connection between clients and the handler application. If the connection is kept, the handler application can send data to clients immediately without waiting next connection from clients.

After surveying IoT related technologies, we proposed a solution for these problems by converging MQTT resources in OpenMTC. MQTT can be integrated to any platform but integration with standards is still yet to be defined. This is an ongoing effort in oneM2M [15]. By integrating MQTT with OpenMTC (a pre oneM2M implementation based on ETSI M2M), we are able to gain more insight on how to integrate MQTT into a standard compliant platform.

In Chapter 4, we show how to converge MQTT resources in ETSI standards based M2M platform. Then in Chapter 5, we compared our approach to the previous approach. Finally in Chapter 6, we show how to leverage our approach for the implementation of real M2M applications.

Chapter 4 Enabling MQTT on OpenMTC

In this chapter, we explain the resource tree operations in ETSI M2M and the four scenarios of handler applications first. Then, we show the details of how to implement our handler application which enables MQTT on OpenMTC.

4.1 Resource Tree Operations

In ETSI M2M architecture, the RAR provides a storage capability for the states associated with applications, devices, gateways and servers. It keeps track of device connection information such as routable addresses, reachability status, subscriptions and notifications management, registration information, and application data. All the above information is stored in a tree structure called "Resource Tree", as shown in Figure 7. Each service capability layer has its resource tree, and applications who registered on a service capability layer could easily control M2M devices through manipulating the resource tree using RESTful operations, such as CREATE, RETRIEVE, UPDATE and DELETE.

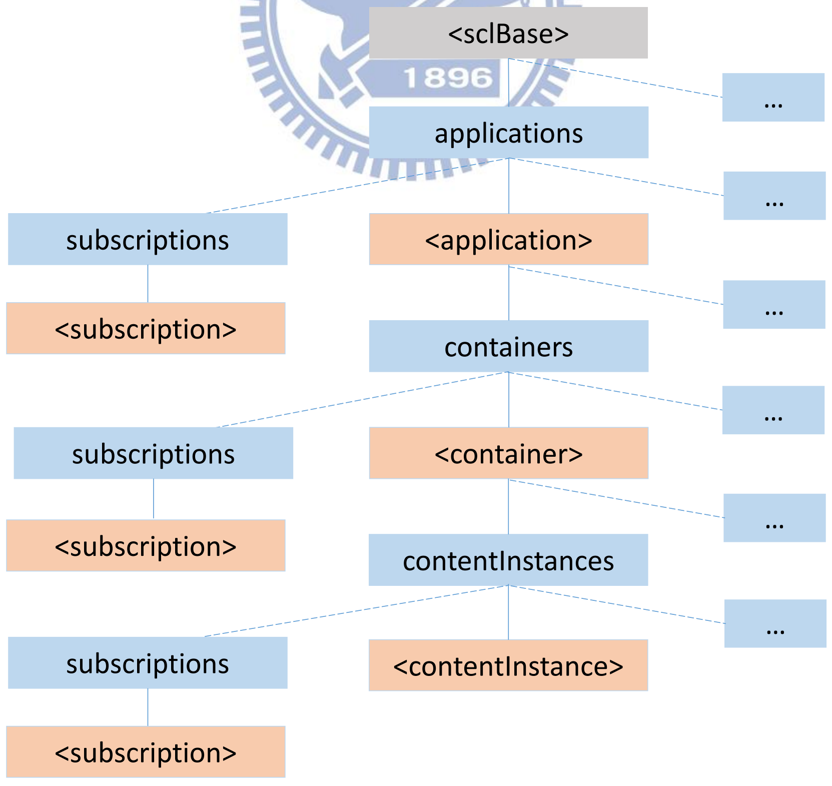


Figure 7: ETSI M2M resource tree

The following are the important instances of the resource tree for data exchange in the ETSI M2M architecture:

1. <sclBase> indicates the root of the resource tree.
2. The “applications” contain the resources of <application> that represent each application registered with the GSCL.
3. The “containers” contain the resources of <container> that are used to exchange data between applications.
4. The “contentInstances” contain the resources of <contentInstance> that are the actual data to be exchanged.
5. The “subscriptions” contain the resources of <subscription> that keep the URI where the subscriber wants to receive its notifications.

These important instances of the resource tree are utilized by the subscribe/notify mechanism for data exchange in the ETSI M2M architecture. The data exchanging mechanism in ETSI M2M can be illustrated by Figure 8. In Step 1, an applications could subscribe to a certain target data by creating the subscription resources under the corresponding container resources of the resource tree. In Step 2, when there is an incoming new data pushed into the target container, the new contentInstance is created in the target SCL. After that, in Step 3 SCL would send the notification with the information of that newly created contentInstance to the subscribed applications.

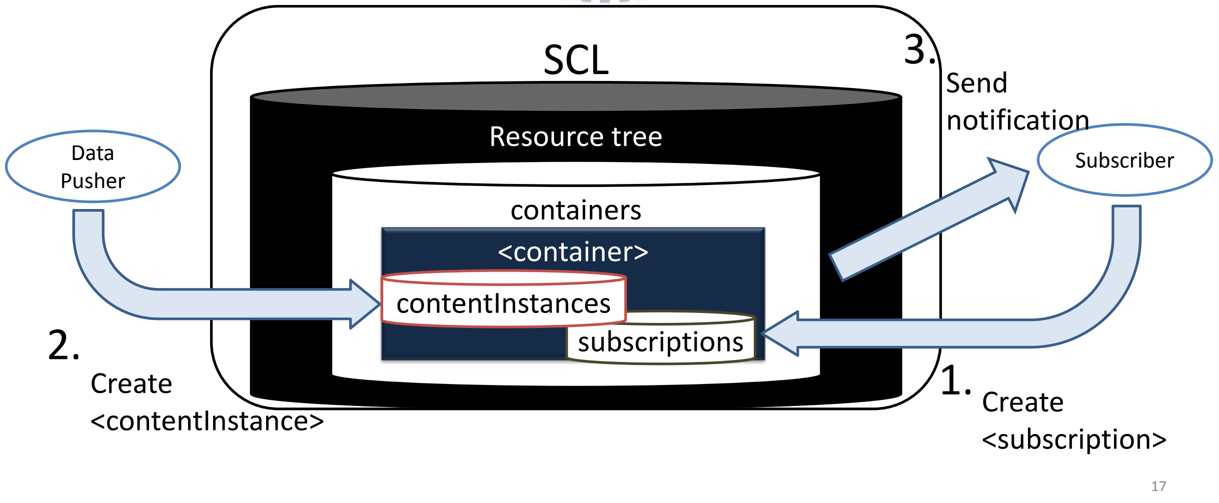


Figure 8: ETSI M2M data exchanging mechanism

4.2 Four scenarios of handler applications

The non ETSI compliant devices do not have capability to push data into SCL or subscribe the data. Hence, in our previous work we created handler applications to communicate with SCL in order to interfacing seamlessly with the ETSI M2M architecture. The handler application acts as a broker to help those non ETSI compliant devices interface with the ETSI M2M platform by manipulating the resource tree of SCLs.

According to the direction of data flow, we can split the handler application into two fundamental functionalities:

- **Collecting data to OpenMTC**

To collect data to OpenMTC from the M2M device domain, the handler application should open a service port for incoming data traffic. After that, the handler application would update the resource tree of NSCL. The <container> is used to exchange data between applications, so we have to collect MQTT packets into the <container> of the SCL resource tree.

- **Delivering data from OpenMTC to the clients**

The idea is same as collecting data to OpenMTC but in another direction of data transmission. The application handler should pre-subscribe to the containers for delivering data from OpenMTC to the clients first. After a new <contentInstance> is created in those containers, the application handler would get notification because of the pre-created subscription resource. The application handler could parse the notification and then deliver data to the clients.

On the other hand, according to the target resource tree we manipulated, there are two types of implementation:

- **Manipulating the GSCL resource tree**

Manipulating the GSCL resource tree means that the M2M data are contained in the GSCL resource tree. Other M2M applications can create the subscription resource in the GSCL, and whenever there is an incoming data, the handler application stores the M2M information in the GSCL resource tree, and GSCL will notify the M2M applications that subscribed this data through NSCL.

- **Manipulating the NSCL resource tree**

Manipulating the NSCL resource tree means that the M2M data are contained in the remote SCL resource tree. Other M2M applications can create the subscription resource in the NSCL, and whenever there is an incoming data, the handler application would retarget the request toward the remote NSCL, and M2M information will be kept in the NSCL resource tree. At the end, NSCL will notify the M2M applications that subscribed this data.

Table 2: Comparisons of two choices to collect data to OpenMTC

	Data contained in the GSCL	Data contained in the NSCL
Advantages	1. Data traffic would not go through the M2M Network Domain. It saves the bandwidth of the M2M Network Domain. 2. Lower latency for local applications.	1. M2M servers can be implemented in cloud environment with better fault-tolerance. 2. Lower latency for applications on NSCL.
Disadvantages	1. If the GSCL breaks without back up, data will lose. 2. Higher latency for applications on NSCL.	1. Data traffic would always go through the M2M Network Domain. 2. Higher latency for local applications.
Suitable Application	Application data are seldom shared by other applications on NSCL. The application which does not need historic data. Ex. Home automation.	Application data always upload to NSCL. Big data analysis. Ex. Customers habits analysis.

According to the direction of data flow and the target resource tree we manipulated, there are four kinds of scenarios for the implementation of the handler application. To enable our handler application for different cases, we have to include all these four scenarios. We name each scenario and list them in Table 3.

Table 3: Four Scenarios in Handler Application

	Collecting data to OpenMTC	Delivering data from OpenMTC to the clients
Data contained in the GSCL	Scenario I	Scenario III
Data contained in the NSCL	Scenario II	Scenario IV

4.3 Enable MQTT on OpenMTC

To enable MQTT in those four scenarios on OpenMTC, we built the new handler application, called "MQTT Proxy" which can be split into two functions including "MQTT Proxy_G" and "MQTT Proxy_N". The MQTT Proxy, on the one side, acts as an MQTT broker to the MQTT clients. While on the other side, it serves as an interworking proxy for interfacing with OpenMTC. The "MQTT Proxy_G" function covers the case of scenario I, III where the data is contained in the GSCL, and the "MQTT Proxy_N" function covers the case of scenario II, IV where the data is contained in the NSCL. The system architecture is depicted in Figure 9. Note that the OpenMTC platform was installed on two separate machines, one for GSCL and another for NSCL.

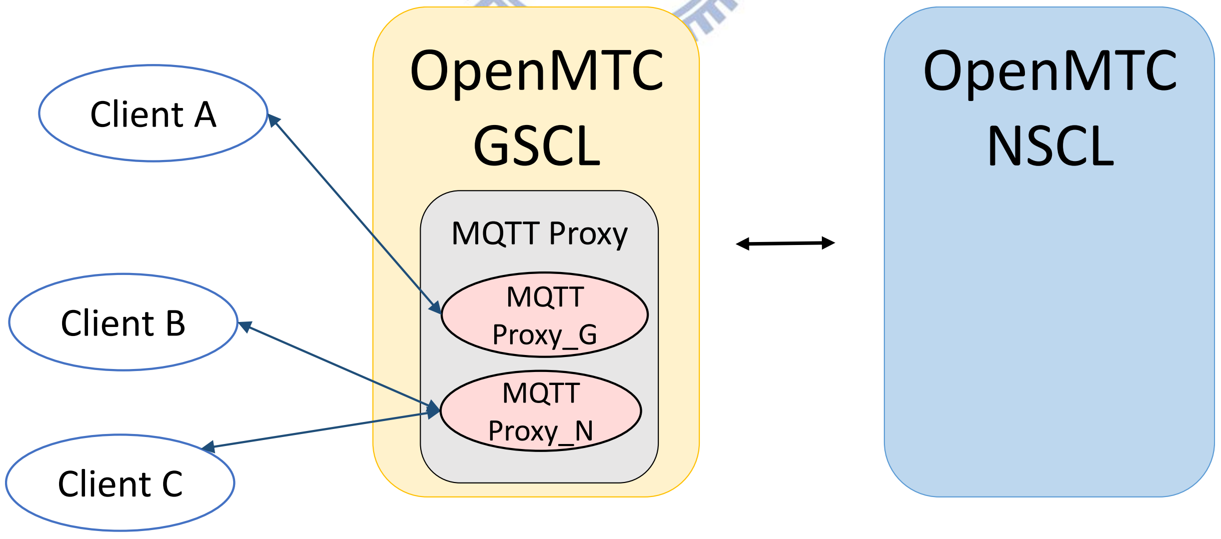


Figure 9: System Architecture

In both kinds of MQTT Proxy functions, there are three main operations: (1) bootstrapping and application initialization, (2) collecting MQTT packets from MQTT clients to OpenMTC, and (3) delivering data from OpenMTC to MQTT clients. In the following we will describe each in detail, and show the result of resource tree after all three operations.

- **Bootstrapping and application initialization**

Bootstrapping is the initial step for the whole system. After bootstrapping, reliable M2M communications can be established between the NSCL and the GSCL. In order to incorporate the MQTT Proxy in OpenMTC, in the initialization process a specific container subscribed by the MQTT Proxy for delivering data from OpenMTC to MQTT clients needs to be created under the <application> resource. Figure 10 depicts such a message flow. Note that specifically "MQTT Proxy" is used as the name of application and "uniqueContainer" as the specific container ID.

The MQTT Proxy_G function for bootstrapping and application initialization (scenario I, III) includes:

1) The MQTT Proxy asks the GSCL to create an <application> with the ID "MQTT Proxy" on the GSCL.

2) The MQTT Proxy asks the GSCL to create a <container> with the ID "uniqueContainer" on the GSCL.

3) The MQTT Proxy asks the GSCL to create a <subscription> to subscribe on contentInstances of the container "uniqueContainer" on the GSCL with the MQTT Proxy as the contact point of notification.

4) The MQTT Proxy is now ready to start collecting MQTT packets.

The MQTT Proxy_N function for bootstrapping and application initialization (scenario II, IV) includes:

1) The MQTT Proxy asks the GSCL to create an <application> with the ID "MQTT Proxy" on the NSCL.

2) The MQTT Proxy asks the GSCL to create a <container> with the ID "uniqueContainer" on the NSCL.

3) The MQTT Proxy asks the GSCL to create a <subscription> to subscribe on contentInstances of the container "uniqueContainer" on the NSCL with the MQTT Proxy as the contact point of notification.

4) The MQTT Proxy is now ready to start collecting MQTT packets.

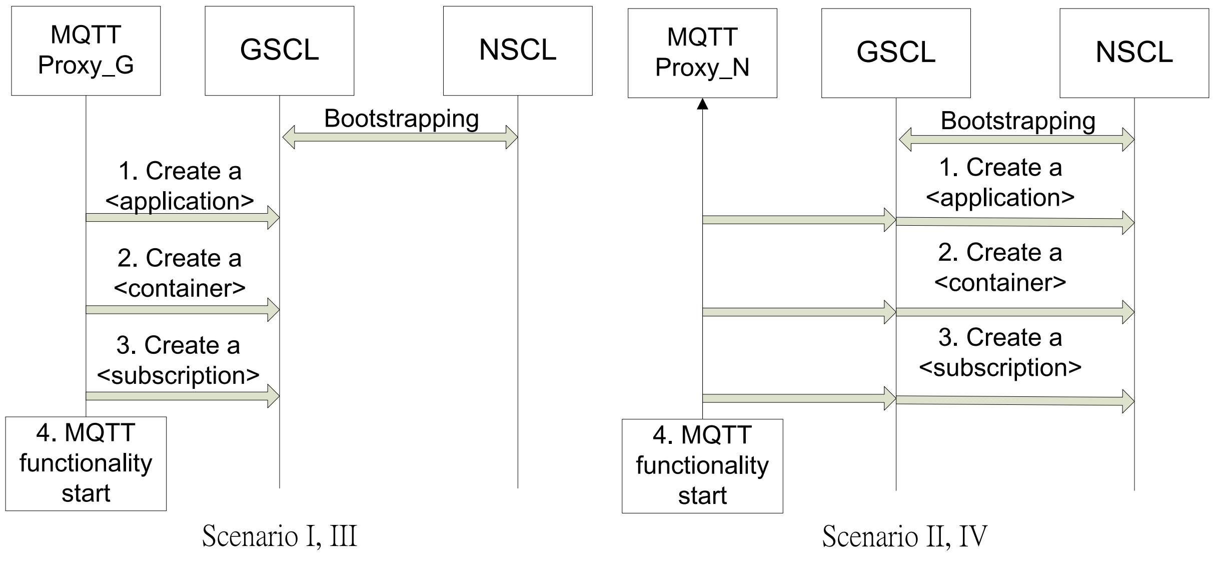

MQTT Proxy_G
GSCL
NSCL
Bootstrapping
1. Create a <application>
2. Create a <container>
3. Create a <subscription>
4. MQTT functionality start
Scenario I, III
MQTT Proxy_N
GSCL
NSCL
Bootstrapping
1. Create a <application>
2. Create a <container>
3. Create a <subscription>
4. MQTT functionality start
Scenario II, IV

Figure 10: Message flow of bootstrapping and application initialization

- **Collecting MQTT packets into OpenMTC**

In our design, each MQTT Proxy function keeps a list of the MQTT topics for those MQTT packets that have arrived before. If the arriving packets indicate a new topic, a new <container> with the topic as the ID will be created and all the incoming packets will be saved in the newly created <contentInstance> under that container. Figure 11 depicts such a message flow. Assume that there is an MQTT packet published with the topic “ABCD” from one of the MQTT clients.

The MQTT Proxy_G function for collecting MQTT packets into OpenMTC is shown in Scenario I of Figure 11 with the following steps:

1) When the MQTT Proxy gets the packet, it will deliver the packet to MQTT subscribers if there are any. It will also check whether the topic has been recorded before. If yes, go to (3), otherwise, go to the next.

2) The MQTT Proxy asks the GSCL to create a <container> with the ID “ABCD” on the GSCL.

3) The MQTT Proxy also asks the GSCL to create a <contentInstance> under the container “ABCD” on the GSCL and saves the data there.

MQTT Proxy_N function for collecting MQTT packets into OpenMTC is shown in Scenario II of Figure 11 with the following steps:

1) When the MQTT Proxy gets the packet, it will deliver the packet to MQTT subscribers if there are any. It will also check whether the topic has been recorded before. If yes, go to (3), otherwise, go to the next.

2) The MQTT Proxy asks the GSCL to create a <container> with the ID “ABCD” on the NSCL.

3) The MQTT Proxy also asks the GSCL to create a<contentInstance> under the container “ABCD” on the NSCL and saves the data there.

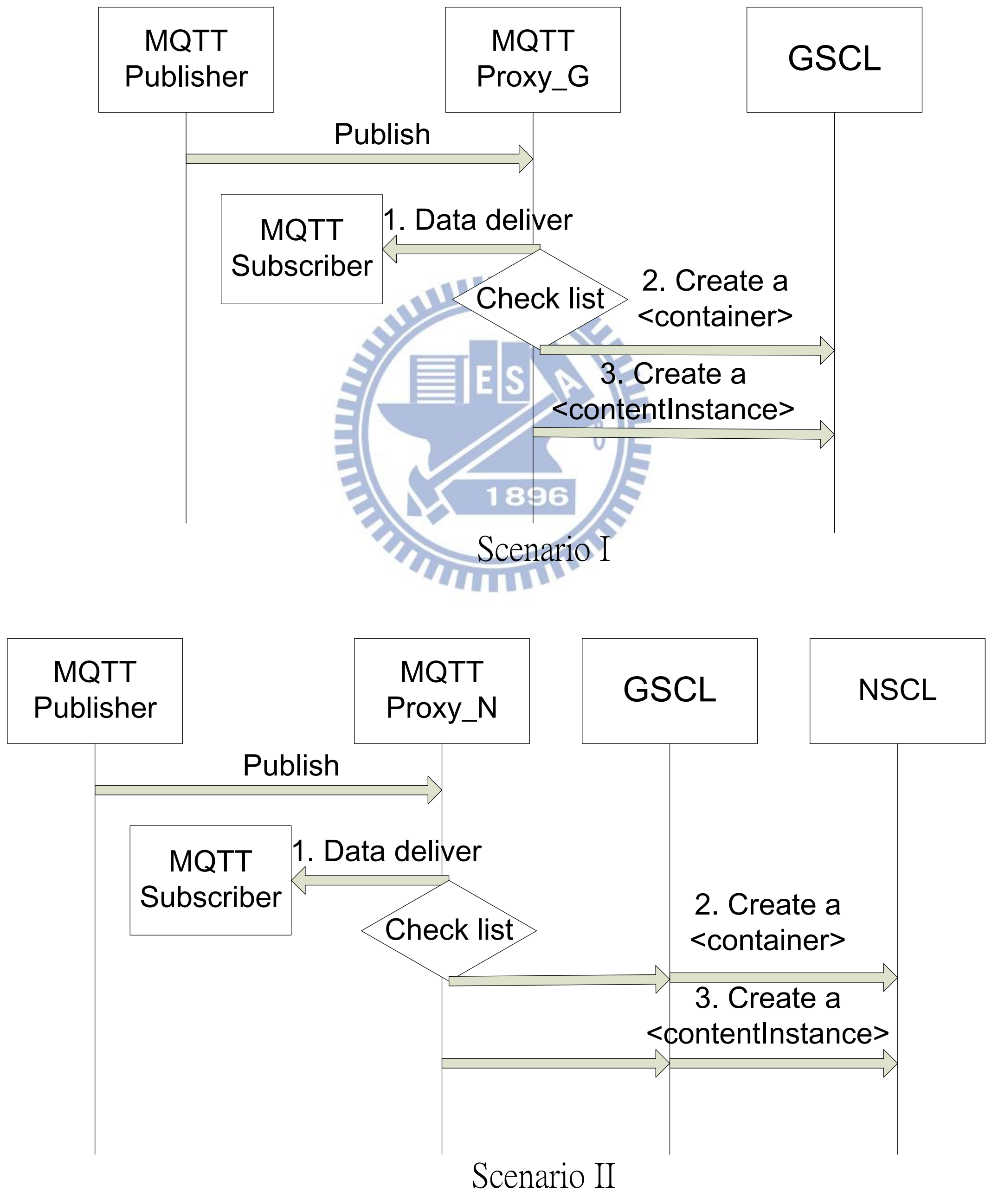


Figure 11: Message flow of collecting MQTT packets into OpenMTC

- **Delivering data from OpenMTC to MQTT clients**

Next we illustrate the message flow of delivering data from OpenMTC to MQTT clients in Figure 12. Assume that the application "Application_WXYZ" on OpenMTC intends to deliver data to MQTT clients who subscribe on the topic "WXYZ".

MQTT Proxy_G function for delivering data from OpenMTC to MQTT clients is shown in Scenario III of Figure 12:

1) The"Application_WXYZ" asks the GSCL to create a <contentInstance> under the container "uniqueContainer" in the GSCL resource tree and saves the information of the MQTT data and topic there.

2) By the <subscription> which was already created in the initialization, the GSCL will notify the MQTT Proxy directly using the contact point information of the MQTT Proxy.

3) When the MQTT Proxy gets the information of MQTT data and topic, it will send the packet with the data to the subscribers. Moreover, to make data also available to other applications on the NSCL the MQTT Proxy will create a new container in the GSCL if such a container does not exist before.

4) Assume the container does not exist before, the MQTT Proxy asks the GSCL to create an <container> with the ID "WXYZ" on the GSCL.

5) The MQTT Proxy asks the GSCL to create a <contentInstance> under the container "WXYZ" on the GSCL and saves the data there.

MQTT Proxy_N function for delivering data from OpenMTC to MQTT clients is shown in Scenario IV of Figure 12:

1) The application "Application_WXYZ" asks the NSCL to create a <contentInstance> under the container "uniqueContainer" in the NSCL resource tree and saves the information of the MQTT data and topic there.

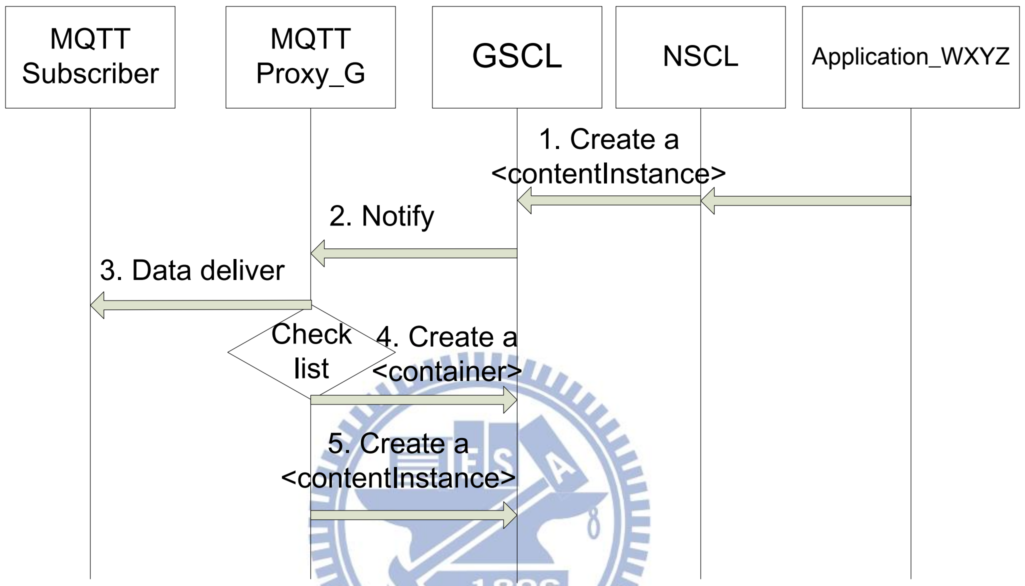
2) By the <subscription> which was already created in the initialization, the NSCL will notify the MQTT Proxy through GSCL using the contact point information of the MQTT Proxy.

3) When the MQTT Proxy gets the information of MQTT data and topic, it will send the packet with the data to the subscribers. Moreover, to make data also available to other

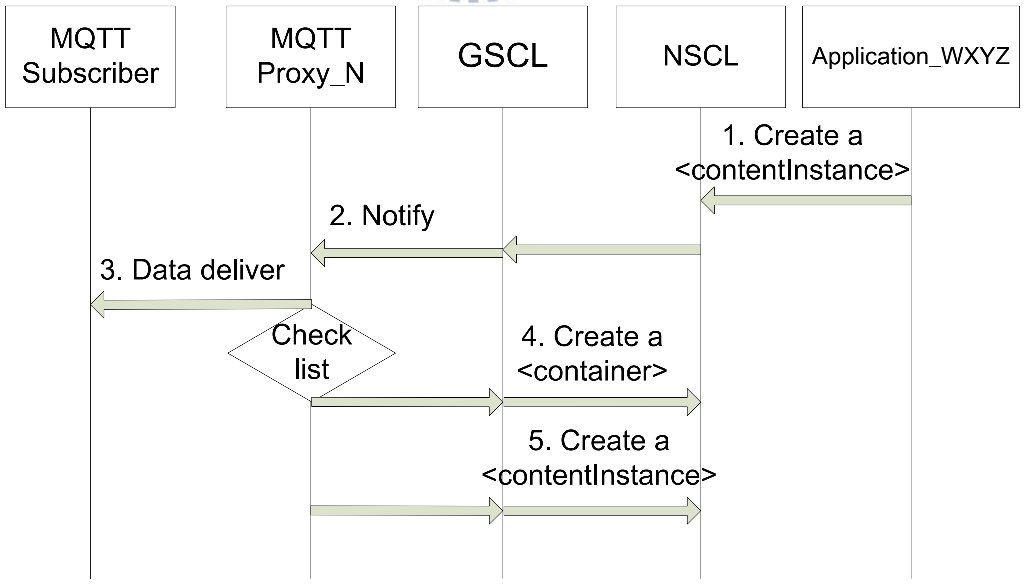
applications on the NSCL the MQTT Proxy will create a new container in the NSCL if such a container does not exist before.

4) Assume the container does not exist before, the MQTT Proxy asks the GSCL to create an <container> with the ID "WXYZ" on the NSCL.

5) The MQTT Proxy asks the GSCL to create a <contentInstance> under the container "WXYZ" on the NSCL and saves the data there.



Scenario III



Scenario IV

Figure 12: Message flow of delivering data from OpenMTC to MQTT clients

Figure 13 shows the resource tree of the GSCL in case of MQTT Proxy_G and that of NSCL in case of MQTT Proxy_N after all operations. Note that Container "uniqueContainer" enables OpenMTC applications deliver data to MQTT clients via the MQTT Proxy; Container "XYZW" allows data delivered to the MQTT proxy also available to other OpenMTC applications; and Container "ABCD" enables MQTT clients to deliver data to OpenMTC applications.

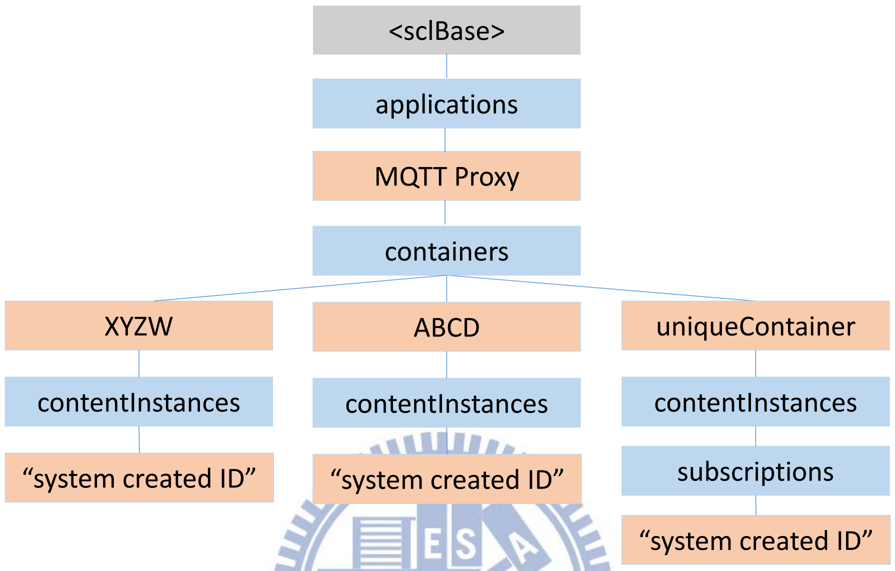


Figure 13: Resource tree of GSCL (MQTT Proxy_G)/NSCL (MQTT Proxy_N)

Chapter 5 Comparison and Experiment

We compare the handler application using MQTT with that using HTTP in this chapter. In Section 5.1, we reported the data delivering performance of both approaches in our experiment. The performance is measured by the latency between pushing the data to GSCL and receiving the data from NSCL. In Section 5.2 the two approaches then are compared based on power consumption and support features.

5.1 Latency Experiment

To do the latency experiment, we set up our testbed as depicted in Figure 14. The OpenMTC platform was installed on two separate machines, one with Intel Xeon CPU E3-1230 V2 CPU, 8GB memory, 1 network interfaces for GSCL, and another with Intel Core i5-3470 CPU, 8GB memory, 1 network interfaces for NSCL. Each machine was installed with Ubuntu 12.04 LTS, which is the operating system suggested by Fraunhofer FOKUS. Before the installation of OpenMTC platform, we also installed the required software for the platform such as nodejs [16] and mongodb [17].

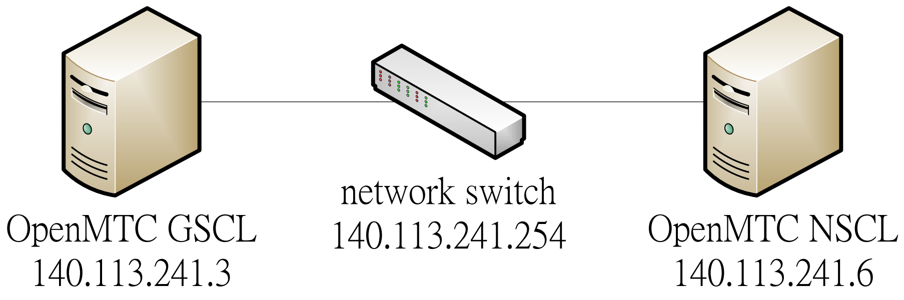
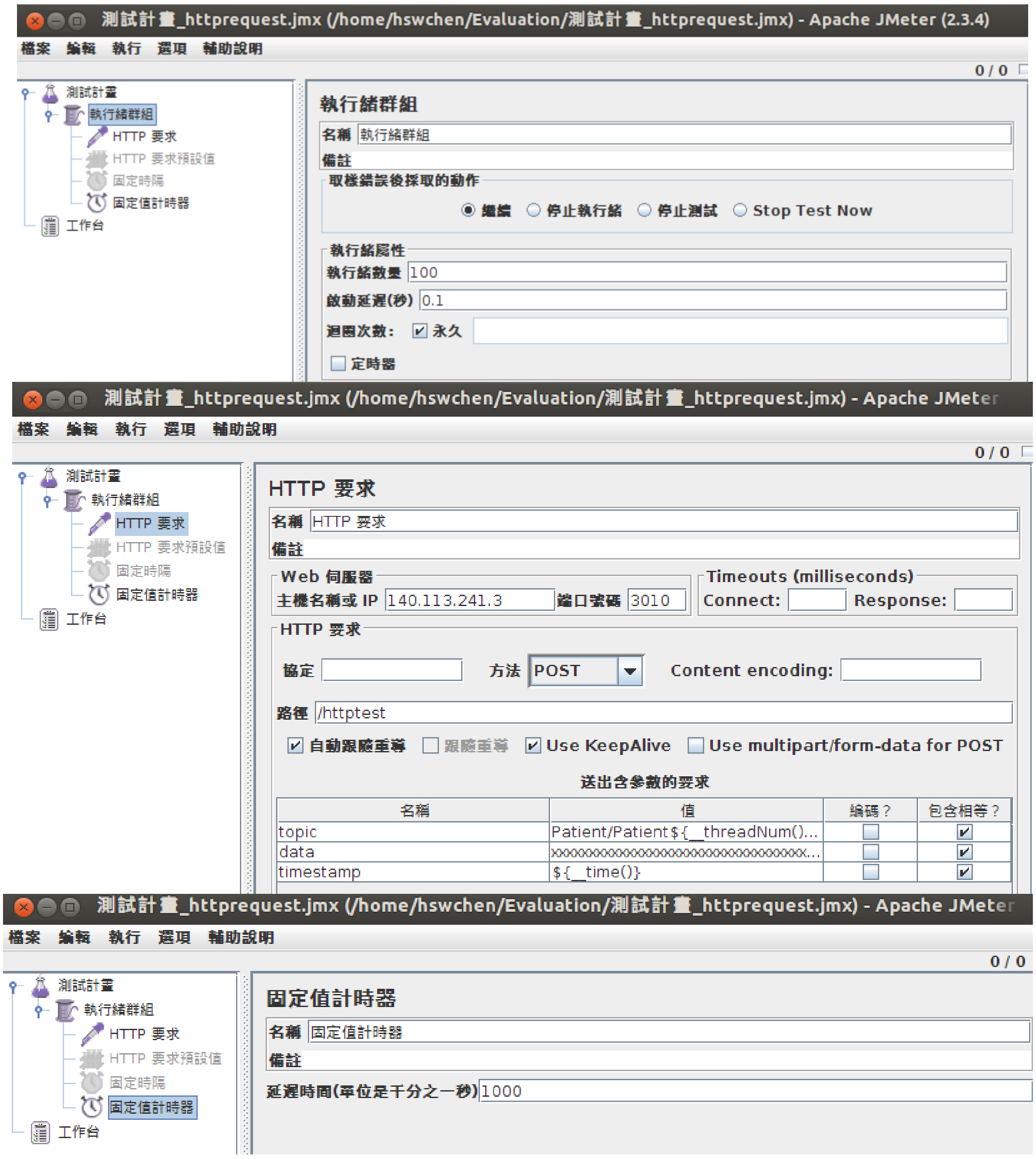


Figure 14: Experiment testbed

In our previous approach, the pushing data to clients problem is not well addressed. By using MQTT, we have developed a better approach than using HTTP for the handler application. Hence in the following, we only focus on the Scenarios I and II of handler applications in Table 3 of Section 4.2 that collects data to the OpenMTC. The HTTP Proxy in the experiment determines where the data contained in the resource tree by parsing the key "topic" of the http post payload. The MQTT Proxy is the same as what we introduced in Chap. 4. To measure latency between pushing the data to GSCL and receiving the data from NSCL, we created data generators to simulate network traffic with timestamps to our handler applications on GSCL. GSCL will then transfer data to NSCL. The latency is calculated by using current time of receiving data from NSCL minus the previous generated timestamp.

The HTTP data generator was created by Apache JMeter [18], which is an open source java application designed for testing web applications. What we need to do is to set up the configuration for our test. Figure 15 is the snapshot of detailed Jmeter configuration. We created a Test Plan as the root directory, and under this test plan we created a Thread Group. In the Thread Group, we can set up thread properties including the number of threads and the ramp-up period based on our experiment. Under the Thread Group, we added an HTTP Request Sampler and a Constant Timer. In the HTTP Request Sampler, we can set up the IP and the port number of the http server, the http method we used, and the path and the send parameters with the http requests. In the Constant Time, we can set up Thread Delay to control the frequency of the http requests.


測試計畫_httprequest.jmx (/home/hswchen/Evaluation/測試計畫_httprequest.jmx) - Apache JMeter (2.3.4)

檔案 編輯 執行 選項 輔助說明

0 / 0

測試計畫
執行緒群組
HTTP 要求
HTTP 要求預設值
固定時隔
固定值計時器
工作台

執行緒群組

名稱 執行緒群組

備註

取樣錯誤後採取的動作

[✓ 繼續] 停止執行緒 停止測試 Stop Test Now

執行緒屬性

執行緒數量 100

啟動延遲(秒) 0.1

迴圈次數: ☑ 永久

☐ 定時器

測試計畫_httprequest.jmx (/home/hswchen/Evaluation/測試計畫_httprequest.jmx) - Apache JMeter

檔案 編輯 執行 選項 輔助說明

0 / 0

測試計畫
執行緒群組
HTTP 要求
HTTP 要求預設值
固定時隔
固定值計時器
工作台

HTTP 要求

名稱 HTTP 要求

備註

Web 伺服器

主機名稱或 IP 140.113.241.3 埠口號碼 3010

Timeouts (milliseconds)

Connect: Response:

HTTP 要求

協定 方法 POST Content encoding:

路徑 /httptest

☑ 自動跟隨重導 ☐ 跟隨重導 ☑ Use KeepAlive ☐ Use multipart/form-data for POST

送出含參數的要求

名稱	值	編碼？	包含相等？
topic	Patient/Patient${__threadNum()...	☐	☑
data	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx...	☐	☑
timestamp	${__time()}	☐	☑

測試計畫_httprequest.jmx (/home/hswchen/Evaluation/測試計畫_httprequest.jmx) - Apache JMeter

檔案 編輯 執行 選項 輔助說明

0 / 0

測試計畫
執行緒群組
HTTP 要求
HTTP 要求預設值
固定時隔
固定值計時器
工作台

固定值計時器

名稱 固定值計時器

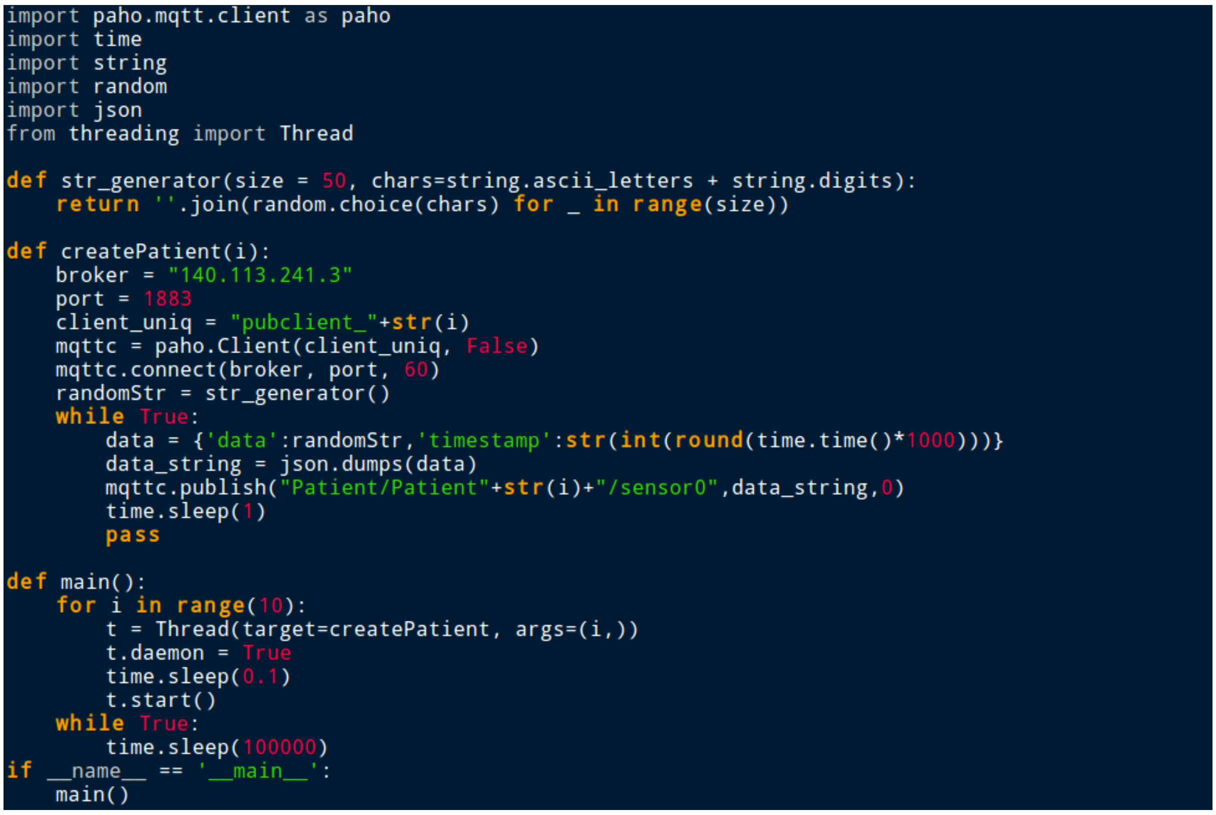
備註

延遲時間(單位是千分之一秒) 1000

Figure 15: Configuration of Apache JMeter

The MQTT data generator was created by Paho python client [19]. The Paho project has been created to provide open source implementations of messaging protocols for IoT inside the Eclipse M2M Industry Working Group. In Paho python client, it provides MQTT client class, so we can easily implement our MQTT client applications by using its helper functions. The code of MQTT data generator is depicted in Figure 16. The Paho Python library and required libraries are imported at the beginning, and we created number of threads with 0.1

second ramp-up period through a for loop in the main function. Each thread will call the function createPatient which publishes MQTT messages to the MQTT Proxy regularly by calling the function of mqttc.publish in an infinite loop. We can control the frequency of MQTT publish using the time.sleep function between each loop. Note that the third parameter of mqttc.publish indicates the QoS level of the message to be published. In our experiment, QoS 0 is used because HTTP does not support reliability of delivery.



```python
import paho.mqtt.client as paho
import time
import string
import random
import json
from threading import Thread

def str_generator(size = 50, chars=string.ascii_letters + string.digits):
    return ''.join(random.choice(chars) for _ in range(size))

def createPatient(i):
    broker = "140.113.241.3"
    port = 1883
    client_uniq = "pubclient_"+str(i)
    mqttc = paho.Client(client_uniq, False)
    mqttc.connect(broker, port, 60)
    randomStr = str_generator()
    while True:
        data = {'data':randomStr,'timestamp':str(int(round(time.time()*1000)))}
        data_string = json.dumps(data)
        mqttc.publish("Patient/Patient"+str(i)+"/sensor0",data_string,0)
        time.sleep(1)
        pass

def main():
    for i in range(10):
        t = Thread(target=createPatient, args=(i,))
        t.daemon = True
        time.sleep(0.1)
        t.start()
    while True:
        time.sleep(100000)
if __name__ == '__main__':
    main()
```

Figure 16: The code of MQTT data generator

Also, each thread pushes data once every second, and was created with 0.1 second ramp-up period. We changed the traffic to handler applications by tuning the number of data generator threads. Throughput can be calculated by the following equation:

$$\text{Throughput} = \text{Thread number} * \text{Carried data per second}$$

In the first experiment, we fix the size of carried data per request to 50 Bytes. The data generator keeps pushing data to the interworking proxy on the GSCL, and then data is delivered to the application on the NSCL though the OpenMTC. We did two experiment based on scenario I and scenario II of the handler application, and the results are shown in Figure 17. The X-axis indicates the number of threads created in the data generator, and the Y-axis indicates the average latency we measured from nine thousand successive samples.

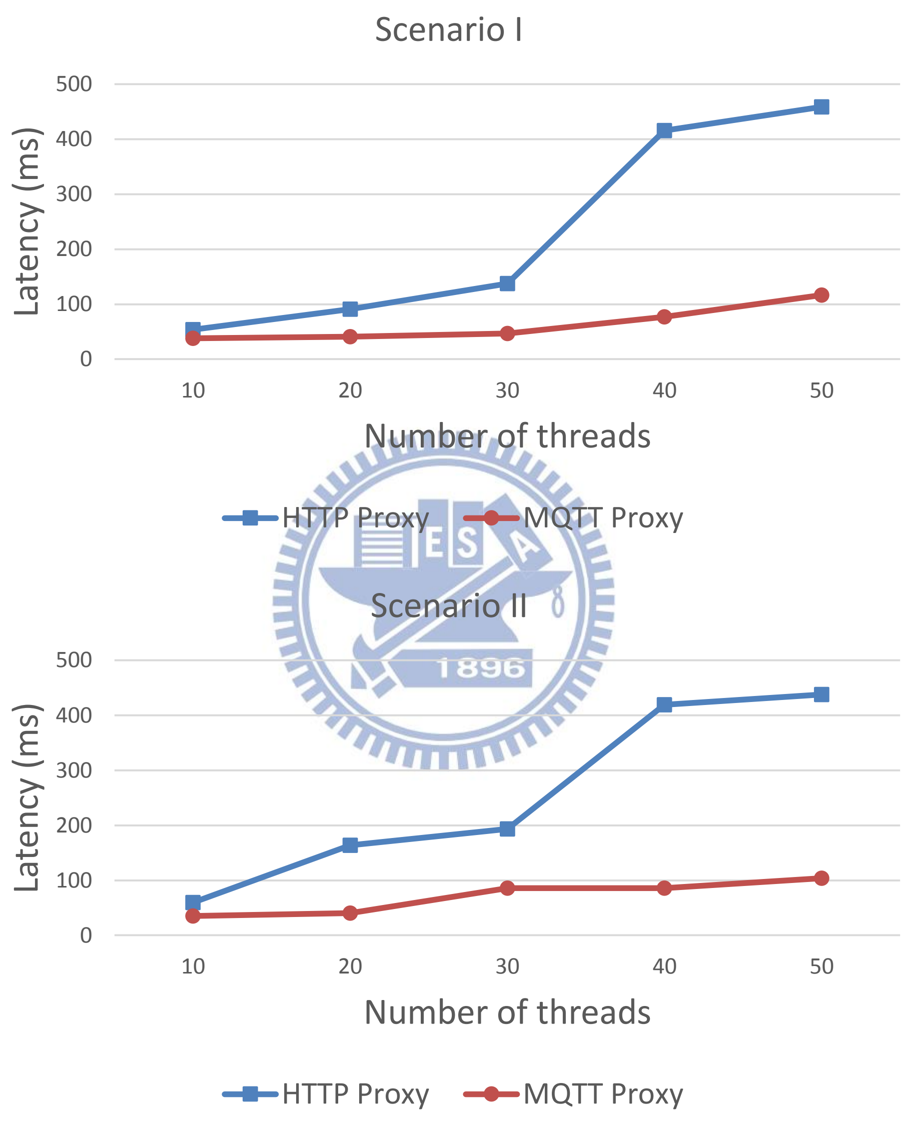
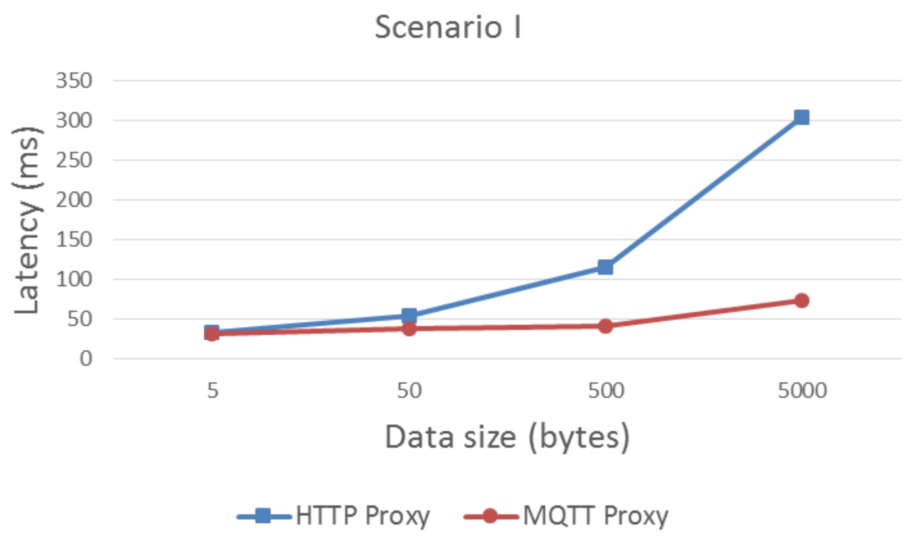


Figure 17: First latency experiment

In the second experiment, we fix the number of threads created in the data generator to 10. The data generator keeps pushing data to the interworking proxy on the GSCL, and then data will be delivered to the application on the NSCL though the OpenMTC. We did two experiment based on scenario I and scenario II of the handler application, and the results are shown in Figure 18. The X-axis indicates the size of carried data per request, and the Y-axis indicates the average latency we measured from nine thousand successive samples.



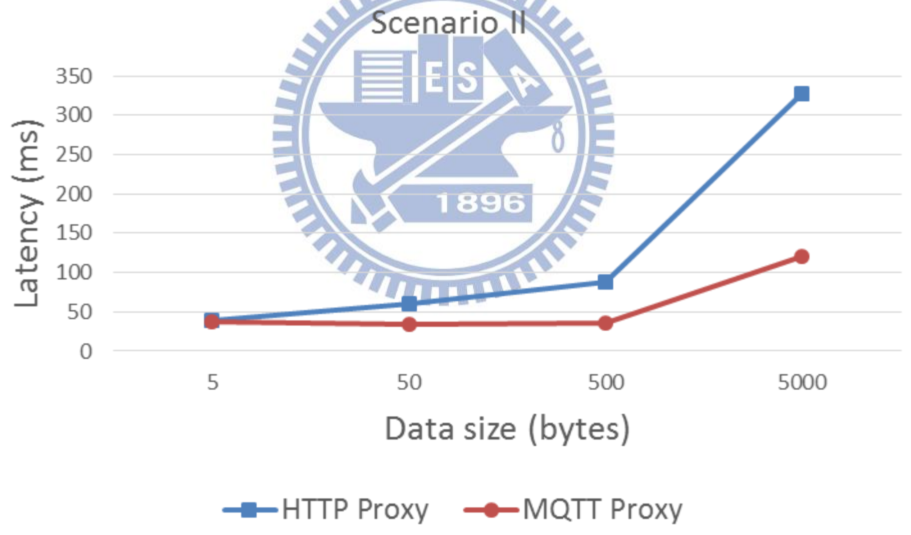


Figure 18: Second latency experiment

In both experiments the scenario I and II have similar result. It shows that the MQTT Proxy can deliver data to the network application on NSCL with lower latency compared to the HTTP Proxy. This is especially obvious when the number of threads or the size of data increases.

5.2 Comparison between the MQTT Proxy and the HTTP Proxy

The experiment in section 5.1 shows that the MQTT Proxy has lower latency to deliver data to network applications on NSCL compared to that of the HTTP Proxy. Besides, there are other advantages by using the MQTT protocol for interworking proxy:

- **Power consumption of the devices**

Having lower power consumption can increase the battery lifetime of devices. There is already a detailed experiment for power consumption comparison between HTTPS and MQTT on Android [20]. The result shows that the MQTT protocol perform significantly better than HTTPS in almost all tests, which include maintaining the connection, receiving messages, and sending messages. For maintaining the connection (keep alive period is set to 240 seconds), ~51.8% of battery can be saved for 3G case and ~92.1% of battery can be saved for WiFi case by using MQTT instead of HTTPS. For receiving messages, ~21.4% of battery can be saved for 3G case and ~91.5% of battery can be saved for WiFi case. For sending messages, ~91.6% of battery can be saved for 3G case and ~84.6% of battery can be saved for Wifi case.

- **Support features**

Having useful support features is also convenient for application developers. MQTT supports many useful features that HTTP doesn't have, for example: The "topic" based property of MQTT is very convenience to be mapped to the "container" in a resource tree. The "broker" based property of MQTT well maintains the connection between clients and the MQTT Proxy. The publish/subscribe module decouples message senders and receivers and supports one-to-many data transmitting. Three-level QoS provides reliable connection over unreliable networks. Keep-alive message and last will and testament can detect failed connections. If the applications we are developing require those features, using MQTT is helpful and has better performance in general.

The comparison between the MQTT Proxy and the HTTP Proxy shows that the MQTT Proxy has lower latency, better power-saving, and more support features than the HTTP Proxy we used before.

Chapter 6 Developing with the MQTT Proxy

In this chapter, we introduce how to develop M2M applications on OpenMTC with the MQTT Proxy. We focus on the application domain in section 6.1, and the device domain in section 6.2. Finally, we give an example of developing a smart lighting M2M application in section 6.3.

6.1 Application Domain

By the new network function, MQTT Proxy, all M2M applications on OpenMTC can leverage MQTT resources through built-in OpenMTC functions. If the applications wants to receive data from the device domain, they can subscribe on interested topics of data from MQTT clients by creating multiple instances of <subscription> with appropriate contact point information. The application will then get the notification from the NSCL if there is any incoming data of those subscribed topics. For the applications which would like to deliver data to MQTT clients, they just have to create a <contentInstance> under the "uniqueContainer" which is created in the initialization process and subscribed by the MQTT Proxy. The MQTT Proxy will get the notification from GSCL and then publish MQTT messages to clients who subscribe it.

6.2 Device Domain

Because of the diversity property of device domain, there are many kinds of devices been designed and developed in the past for different applications. Most of them are not using MQTT as the communication protocol, so they still need a protocol adapter. To push or receive data from the MQTT Proxy, the protocol adapter should have MQTT client function. Fortunately, numerous MQTT libraries [21] for many of programming languages running on every popular Operating System simplifies the application development. There is an open source MQTT clients available in Eclipse Paho Project, and it provides APIs for many famous programming languages like C/C++, Java, Javascript, Python, etc. By using those MQTT libraries, protocol adapters supporting MQTT can be built easily.

Instead of using legacy devices through the protocol adapter, we can build MQTT devices directly by open source hardware platforms, for example: Smart phones, Arduino or Raspberry Pi. Those open source hardware platforms have large community and support complete high-level APIs, so they are the best choice to build MQTT devices rapidly in the laboratory environment. Figure 19 shows some of our customized MQTT devices. The left of the Figure 19 is the sensors side including photocell which reacts based on the presence of light and DHT11 which is for measuring temperature and humidity. The right of the Figure 19 is the actuators side including four LEDs which can be controlled to turn on or off. The photocell sensor is connected to the Arudino first, and connected to the Raspberry PI through the USB port. DHT11 and LEDs are connected to Raspberry PIs directly through general purpose IO of Raspberry PIs. By developing MQTT functions on Raspberry PIs, this combination of hardware components can turn into MQTT devices.

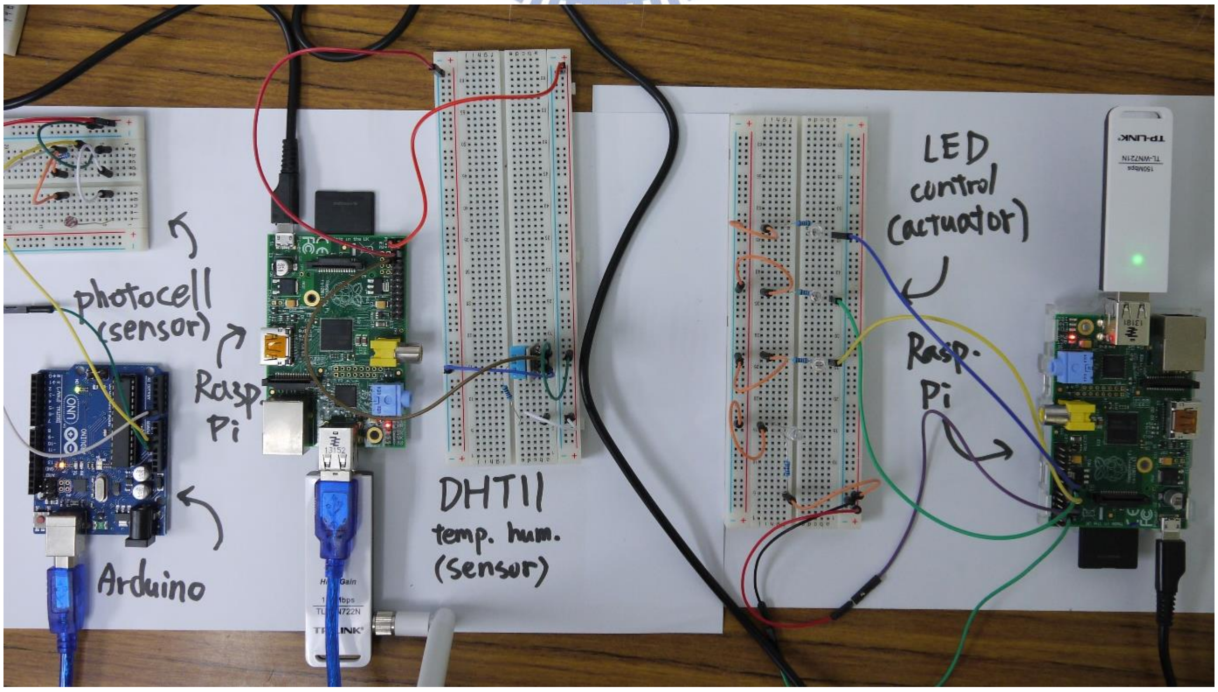


Figure 19: Customized MQTT Devices using Arduino and Raspberry Pi

6.3 Smart Lighting Application

In this section, we give a smart lighting application example to show how to implement M2M applications on OpenMTC with the MQTT Proxy. In our smart lighting application, we want to make the LED automatically turn on or turn off according to the ambient light in the room, as shown in Figure 20. To finish this application, we have to create two MQTT devices: one with a light sensor and another with a LED light, and develop a LED control application on NSCL.

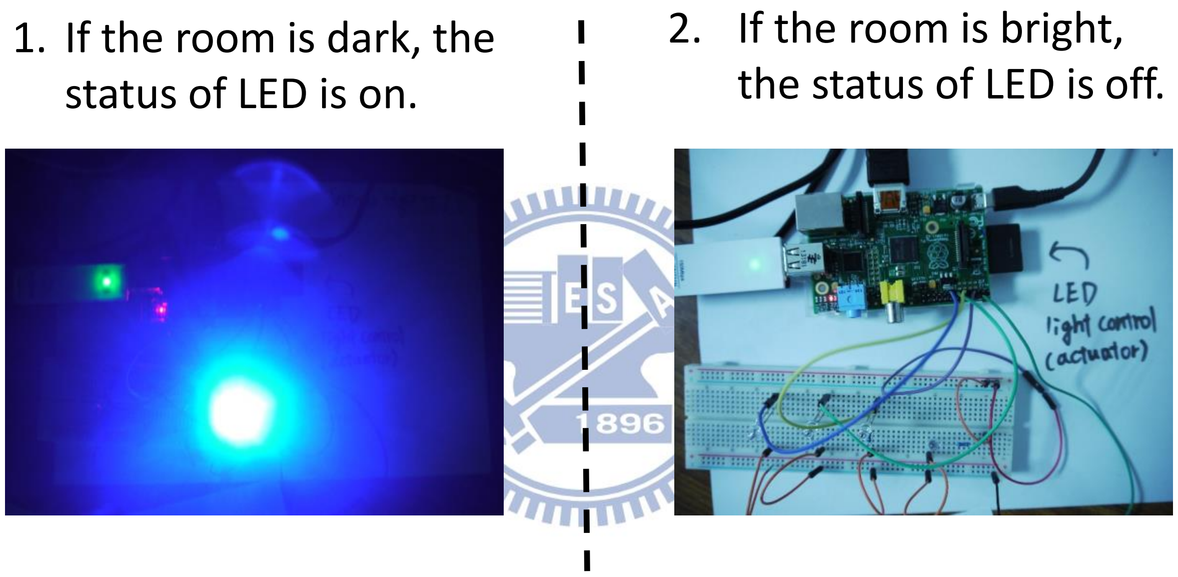

1. If the room is dark, the status of LED is on.
2. If the room is bright, the status of LED is off.
LED light control (actuator)

Figure 20: Two status in the smart lighting application

We built an MQTT device with a light sensor by using both photocell and Arduino [21]. The circuit diagram is shown in Figure 21. The resistance of photocell is influenced by the ambient light, so the analog voltage information which is read by the Arduino can be used to quantify the light. We connected the Arduino board to a Raspberry Pi board by USB port, and wrote a program running on Raspberry Pi to receive the analog voltage data from the Arduino and publish the data to the MQTT Proxy by MQTT protocol with the topic “Devices/photocell”.

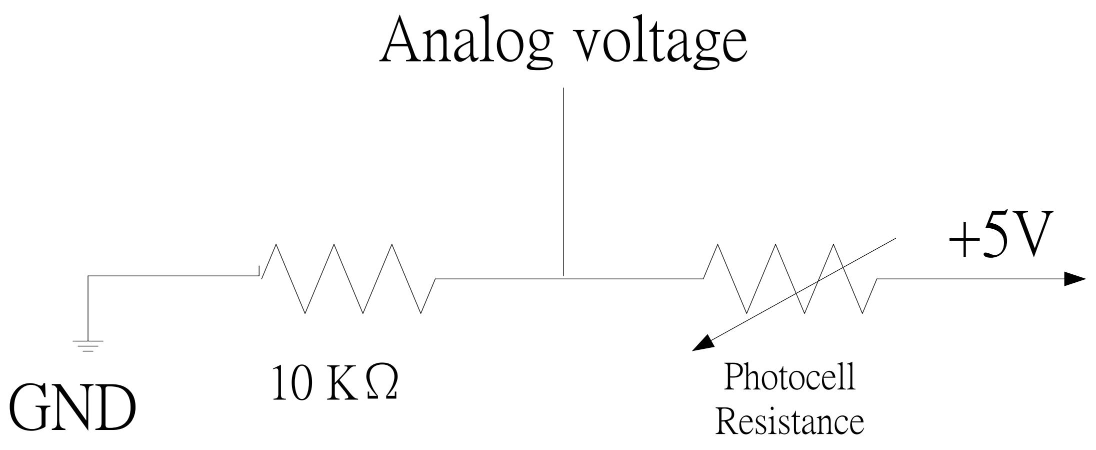


Figure 21: The circuit diagram of using the photocell

Then, we built another MQTT device to control the LED light by referencing the tutorial of getting started with Raspberry Pi GPIO (general purpose IO) and Python [23]. By installing the RPi.GPIO python library which takes some of the complexity out of driving the GPIO pins, writing a program on Raspberry Pi to control the LED light through GPIO becomes very easy. Our program running on the Raspberry Pi subscribed to the topic "Devices/LED1", and controlled the status of the LED light according to the received MQTT messages from the MQTT Proxy.

Finally, we developed a LED control application on NSCL. The control application subscribed to the "Devices/photocell" container, and created contentInstances in the "uniqueContainer" in the resource tree to deliver command to the LED. The whole system is completed after connecting our MQTT devices to the MQTT Proxy on GSCL and running the control application on NSCL. How system works are listed in the following steps and depicted in Figure 22:

1. The MQTT device with the photocell will publish the MQTT message with the analog voltage data with the topic "Devices/photocell" to the MQTT Proxy.
2. The MQTT Proxy will update the resource tree, and finally the NSCL will notify the smart lighting application with the analog voltage data.

3. The smart lighting application will determine the command of turning on or off the light according to a predefined threshold.

4. The smart lighting application will update the resource tree. As a result, the GSCL will notify the MQTT Proxy with the command and the topic “Devices/LED1”.

5. Finally, the MQTT Proxy will publish the command to the MQTT device who subscribed the topic “Devices/LED1”.

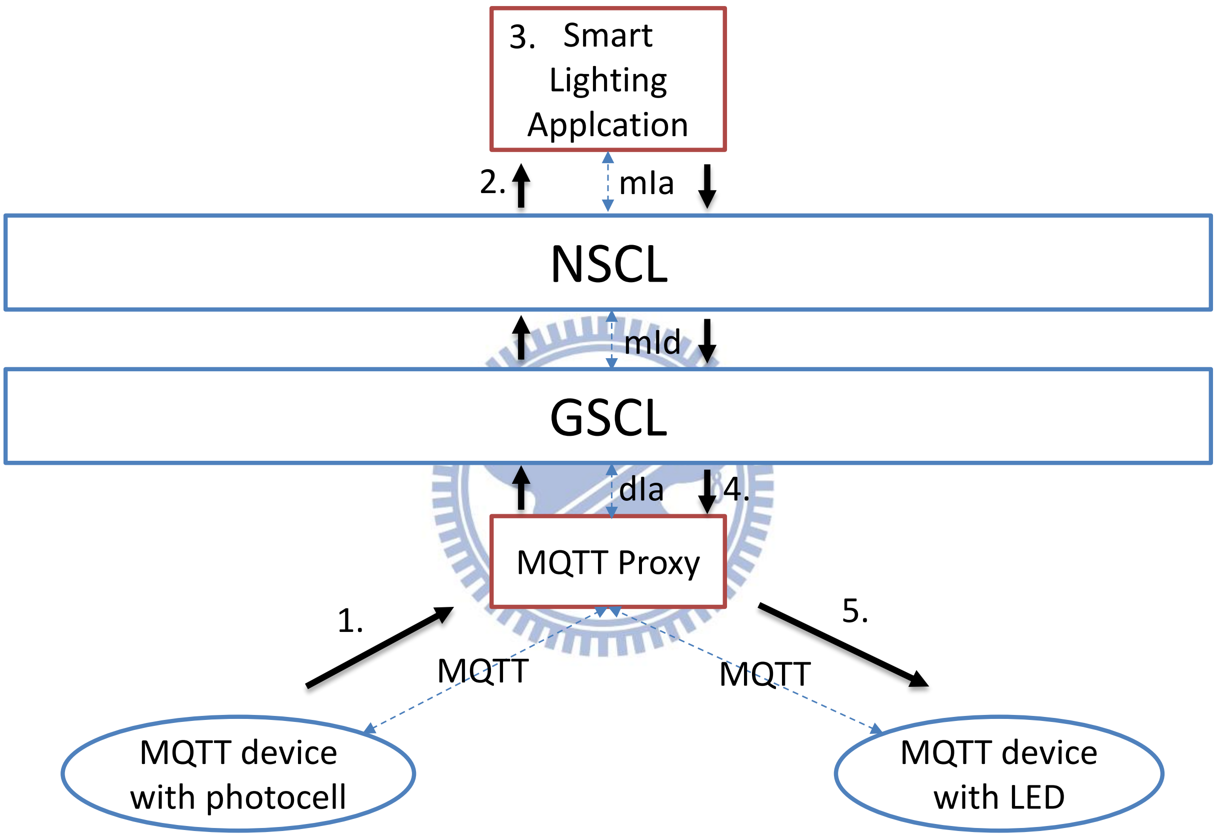


Figure 22: Whole system diagram of the smart lighting application

Chapter 7 Conclusions and Future Works

In this thesis, we explained some issues we faced in our 2013 MOST Deep Plowing Project, and proposed to converge MQTT resources in the ETSI standard based M2M platform. We developed the MQTT Proxy to allow M2M applications on OpenMTC to access and deliver data to MQTT devices through the built-in ETSI mechanisms on OpenMTC. We gave an example to show how to develop applications with the MQTT Proxy. The comparison between the MQTT Proxy and the HTTP Proxy shows that the MQTT Proxy has lower latency, better power-saving and more support feature than the HTTP Proxy we used before. Hence, if the sensors or actuators in the Devices domain can support MQTT protocol.

For the first latency experiment, data showed in the line graph is not a straight line. In the future, we plan to analyze the expected data based on mathematics theory. Besides, our comparison between MQTT and HTTP proxy is based on OpenMTC Release 1. As OpenMTC is an evolving platform, the testing will need to redone for later releases of OpenMTC such as Release 2 or 3 for further verification.

There are still many research opportunities in the future. First, in the thesis we don’t consider the security seriously, which is also a very important topic in the M2M area. Both OpenMTC and the MQTT broker have their built-in security function but totally different. The MQTT Proxy should design a way to combine two different security function. Second, there is an MQTT for sensor area network protocol called MQTT-SN [24]. The MQTT-SN protocol is designed to be as close as the MQTT but optimized for the implementation on devices with limited battery, processing and storage resources, and it is not based on TCP/IP but originally developed for running on top of the ZigBee APS layer. How to converge MQTT-SN resources in ETSI standards based M2M platform can be further explored. Third, seven standard development organizations, such as ARIB(Japan), ATIS(U.S.), CCSA(China), ETSI(Europe), TIA(U.S.), TTA(Korean) and TTC(Japan), set up the oneM2M global initiative to avoid the creation of competing M2M standards. The oneM2M standard inherits the experience from ETSI M2M, like the CSE (common service entity) in oneM2M is very similar to the SCL in ETSI M2M, but still has some new entities added in the oneM2M architecture, like AE (application entity) which contains the application logic of the M2M. There is an oneM2M standard draft for the MQTT protocol binding related to our work, but the draft is still in the early stage with little contents. The effort seems to only focus on the communication between CSE to CSE, and between AE to CSE. How to port our MQTT Proxy into oneM2M architecture is another interesting research topic.

References

[1] P. Warrior, “Connecting Everything: Cisco’s Padmasree Warrior”, Available: http://dowser.org/connecting-everything-ciscos-padmasree-warrior/

[2] Ericsson White Paper, "More than 50 Billion Connected Devices", 2011, Available: http://www.ericsson.com/res/docs/whitepapers/wp-50-billions.pdf

[3] Mqtt, Available: http://mqtt.org/

[4] Using MQTT Protocol Advantages Over HTTP in Mobile Application Development, 2012, Available: https://www.ibm.com/developerworks/community/blogs/sowhatfordevs/entry/using_mqtt_protocol_advantages_over_http_in_mobile_application_development5?lang=en

[5] Organization for the Advancement of Structured Information Standards (OASIS), Available: https://www.oasis-open.org/

[6] OpenMTC, Available: http://www.open-mtc.org/

[7] MQTT v3.1 Protocol Specification, Available: https://www.ibm.com/developerworks/webservices/library/ws-mqtt/

[8] ETSI, TS. "102 690 V1.1.1" Machine-to-Machine communications (M2M) (2011).

[9] ETSI, TS. "102 921 V1.1.1 Machine-to-Machine communications (M2M); mla, dla and mld interfaces." Technical Specification 1 (2012).

[10] ETSI, TR. “102966-V1.1.1” Machine-to-Machine communications (M2M) (2014)

[11] F. J. Lin, Y. Ren, E. Cerritos, A Feasiblity Study on Developing IoT/M2M Applications over ETSI M2M Architecture, IEEE Parallel and Distributed Systems (ICPADS), 2013.

[12] M. Collina et al., Introducing the QEST broker: Scaling the IoT by bridging MQTT and REST, IEEE 23rd International Symposium on Personal Indoor and Mobile Radio Communications (PIMRC), 2012.

[13] OSIOT, Available: http://osiot.org/

[14] Collina Matteo, “mosca”. 2013-2014. Available: https://github.com/mcollina/mosca

[15] OneM2M, Available: http://www.onem2m.org/

[16] nodejs, Available: nodejs.org/

[17] mongoDB, Available: www.mongodb.org

[18] Jmeter, Available: http://jmeter.apache.org/

[19] The Paho Project, Available: http://www.eclipse.org/paho/

[20] Power Profiling: HTTPS Long Polling vs. MQTT with SSL, on Android, Available: http://stephendnicholas.com/archives/1217

[21] MQTT libraries. Available: http://mqtt.org/wiki/doku.php/libraries

[22] Using a photocell, Available: https://learn.adafruit.com/photocells/using-a-photocell

[23] Lighting up a LED using your Raspberry Pi and Python, Available: http://www.thirdeyevis.com/pi-page-2.php

[24] Specification of MQTT-SN, Available: http://www.mqtt.org/new/wp-content/uploads/2009/06/MQTT-SN_spec_v1.2.pdf