Arm Mbed – AWS IoT System Integration.

Sergio Martin

School of Engineering

University of Glasgow

Supervised by:

Dr. Hadi Heidari

Dr. Ashkan Tousi

Professor Sandy Cochran

Thesis submitted for the degree of Meng

Mechatronics Engineering

January 2018

Abstract

This project explores the different Internet of Things (IoT) architectures and the available platforms to define a general IoT Architecture to connect Arm microcontrollers to Amazon Web Services. In order to accommodate the wide range of IoT applications, the architecture was defined with different routes that an Arm microcontroller can take to reach AWS. Once this Architecture was defined, a performance analysis on the different routes was performed in terms of communication speed and bandwidth. Finally, a Smart Home use case scenario is implemented to show the basic functionalities of an IoT system such as sending data to the device and data storage in the Cloud. Furthermore, a Cloud ML algorithm is triggered in real time by the Smart Home to receive a prediction of the current Comfort Level in the room.

Acknowledgements

I would like to thank my supervisors and colleges at Arm for all of the support and making this experience in industry great. I would also like to thank my partner for her unconditional love and support and to my family for granting me the opportunity to follow my dreams.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 The Internet of Things

The Internet of Things is a technological field that encapsulates a lot of different ideas, concepts and features. IoT started as a vision introduced by the International Telecommunications Union(ITU) in its seventh report [17] in 2005 and is currently in the peak of inflated expectations stage of the Gartner Hype Cycle [16]. The Internet of things, as its name indicates, is the connection of all common objects to the Internet, with the purpose of making them smart and interact with one another and people. A Thing (also known as endpoint or node) is a device embedded within an IoT system connected to the Internet, capable of being controlled, monitored and provide information to other devices or users. This devices are characterized by being constrained in terms of space, connectivity, computing capabilities and energy consumption. Another mayor challenge of IoT is the deployment of such a large number of devices, making them able to communicate securely with the Internet and have them interact with users via mobile applications and web pages. For that, Cloud Service Providers have developed Platforms to ease the deployment of IoT systems and connect Things to the internet.

1.1.1 Cloud Service Providers

It was common practice at the end of the twentieth century for technology companies to have data centres with servers and computer clusters to handle the company's both internal and external needs of storage, networking and computation. Over the last decade, a large number of companies have emerged that provide Cloud Services, more formally defined as Infrastructure as a Service (IaaS). This services offer computing resources available over the Internet, that erase the need of a company to have their own data center. The concept of Cloud Services revolutionized the way companies handled their computational needs, reducing the investment needed, the difficulties of managing a data center and the time of deployment of a product. Furthermore, it reduces the risk of wrongly estimating the size of the data centre, allowing the company to adjust the amount of resources they need and quickly scale up or down according to their needs.

Figure 1.1: The three main Cloud Providers in the current market: Amazon Web Service, Microsoft Azure and Google Cloud. (Image created from icons found on the Internet)

Figure 1.1 shows the four main Cloud providers: Microsoft Azure, Amazon Web Services (AWS) and Google Cloud. They all offer a great range of different services for web and mobile applications, data processing, computational resources and flexible storage and database solutions. Regarding IoT applications, they include device management services, to securely connect, manage and interact with IoT devices. It is important to note that the design of nan IoT system is currently heavily dependent in the Platform, since each company has its own required implementation to connect devices. When choosing a Platform for IoT, it is important to consider not only the IoT service, but the endpoint services that will use the data from the devices, like data Analytics, data processing, storage, and mobile and web application integration. Arm offers the Mbed IoT Platform, that provides the developer with all the necessary tools to design IoT Devices and connect them to their Device Management Platform shown in Figure 1.2. A greater description of the AWS and Mbed IoT Platform is given in chapter 2.2.

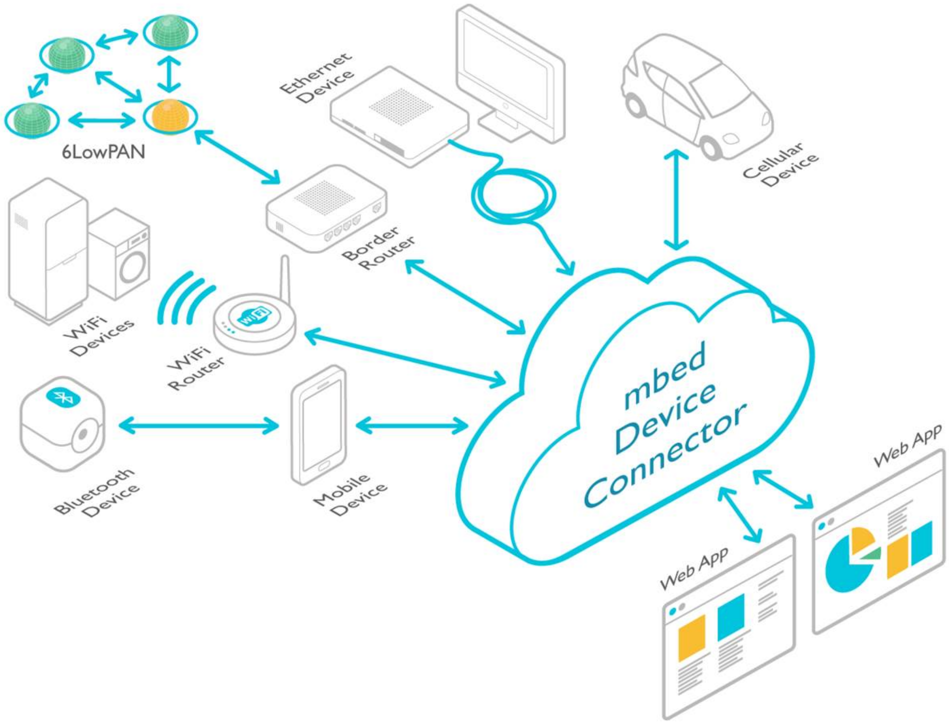


Figure 1.2: Diagram of the Mbed Device Connector: IoT endpoints connecting to web applications. (Figure extracted from Mbed Documentation [12])

1.2 Aim of the project

The aim of the project is to investigate the Mbed IoT Platform and its use in conjunction with AWS. From this investigation, a general IoT system using this platforms had to be defined from which other IoT Applications can be built upon. An example application has to be designed using the features offered by this platforms to serve as a proof of concept of the system.

Chapter 2

State of the Art

2.1 Literature Review

2.1.1 Internet of Things Architecture

In its simplest form an IoT System consists of four main components, as shown in Figure 2.1.

- Thing: embedded within the environment recording data and controlling actuators.
- Gateway: Internet access point for the thing to connect to the Internet.
- Cloud: with storage, computing and device connectivity capabilities.
- Human Interfaces: such as mobile and website applications for the user to interact with the system.

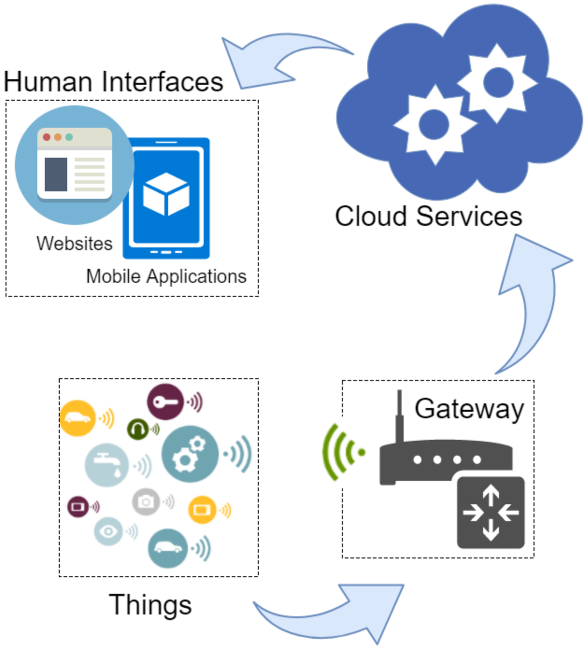
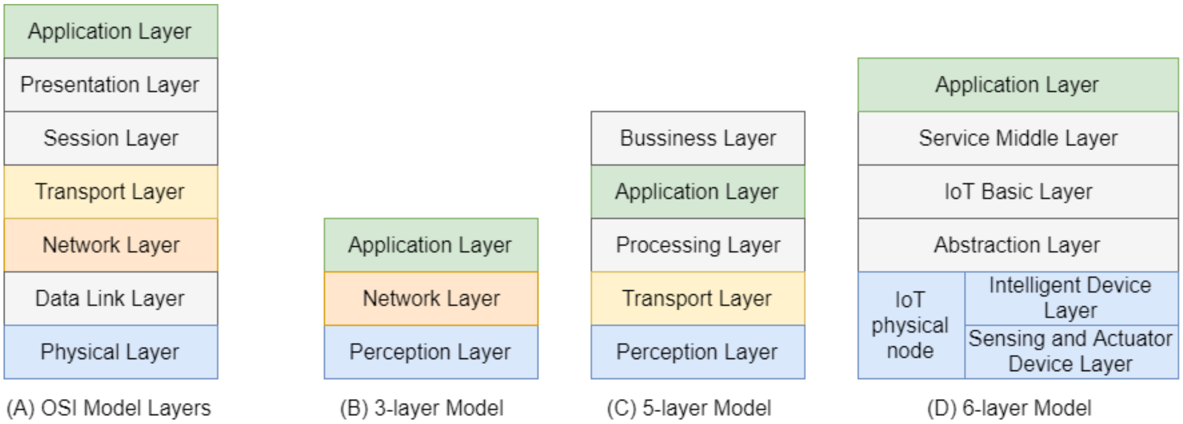


Figure 2.1: The four main components of an IoT System. (Image created from icons found on the Internet)

Since IoT systems are build on top of the current Internet Network, it is important to consider how generic Network systems are designed. Most computer communication networks follow the Open Systems Interconnection (OSI) model, shown in diagram A of Figure 2.2. This conceptual model standardizes the way network systems are designed in order to ensure they can interact with one another. There are standards and protocols from which network systems can base their design to ensure interoperability. There has been attempts to recreate similar layer models for IoT systems such as those in [33] and [4].

In [33], Sethi et all describe a five tier architecture compared to the common 3-tier architecture defined for IoT systems. We also find a more detailed architecture proposed by Weigong LV et all in [4], which includes important concepts in IoT design like service middle Layer and abstraction layer. This two architectures are shown in diagram B and C of Figure 2.2. The importance of this two layers is due to the challenges raised by managing million of devices in a system, so creating services that can immediately handle actions and automate the system is crucial. Furthermore, the abstraction layer can increase the interoperability of different IoT systems, by standardizing the data from a resource such a sensor (for example, standardizing a temperature sensor to have certain properties that the system can be aware of so that each device does not have to be designed for a specific application, such as resolution, update frequency, etc.). Nevertheless this Architecture does not show the transport or network layer as the others do.



(A) OSI Model Layers (B) 3-layer Model (C) 5-layer Model (D) 6-layer Model

Figure 2.2: The Different model Layers for Network and IoT Systems. Diagram A adapted from [30], B and C from [33] and D from [4]

Finding a unified solution for all IoT systems is complex due to the diversity of IoT applications and the requirements each of them impose in the system. The number of Things, their required capabilities (such as computation power) and the their constrains (such as connectivity or power constrains) are just some of the factors that can influence the structure of the whole system. This is mainly due to the fact that current IoT solutions lack interoperability, as described by Ahmed Banafa [1], and there must be a greater effort by companies and research to standardize IoT.

2.1.2 ASHRAE Project

The American Society of Heating, Refrigerating and Air-Conditioning Engineers (ASHRAE) Project was lead by Dr Richard de Dear [3]. During his investigation of *Thermal Comfort*, a large database with data collected from thermal comfort studies was gathered and is available in Sydney University's web page [35]. This comfort zone studies where performed in a series of buildings where the conditions in the building where monitored as well as having the people in the building fill in a questionnaire of their comfort. The full content and structure of the database can be found also be found on the web page, as well as links to download the database itself. The comfort level as a result of the analysis performed by the above study is given as the *ash* value: a value between negative three and positive three. The user was in complete comfort when the value was zero, while really warm at positive three and really cold at negative three.

Later in the project it was decided to use this database as a proof of concept to perform Cloud computing from data gathered by the device by using the AWS Machine Learning Service (Explained in Section 2.2.3). The database from this project is used to predict the comfort level in a room from its internal conditions (temperature and humidity).

2.2 Internet of Things Platforms

Below is a description of the IoT platforms available for developers that where used throughout the project. For each platform a general description is given, followed by some of the technical details of the platform. The main two platforms used are the Arm Mbed IoT Device Platform (Mbed) and Amazon Web Services (AWS).

Mbed Provides developers with C++ Software Libraries to accelerate the development process using Mbed OS, Mbed Device Connector and Mbed Cloud [1] (Cloud Device-Management Platforms). Hardware that complies with Mbed's criteria and technical requirements are said to be Arm Mbed Enabled [15]. Mbed supports this devices by providing Open-Source Software in the means of Libraries, Mbed OS compatibility and sample code. Figure 2.3 shows the main features of the Mbed Platform, and the sections below explore these features in more detail. The platform and its components are described in the Mbed Platform Website [15]. Developer resources such as Reference Manuals, list of supported platforms, Software Libraries, sample code and Forums can be found in the Mbed OS Developer website [14].

[1] Arm announced a new Platform in 2016 called Mbed Cloud, currently only available for Arm Partner companies.

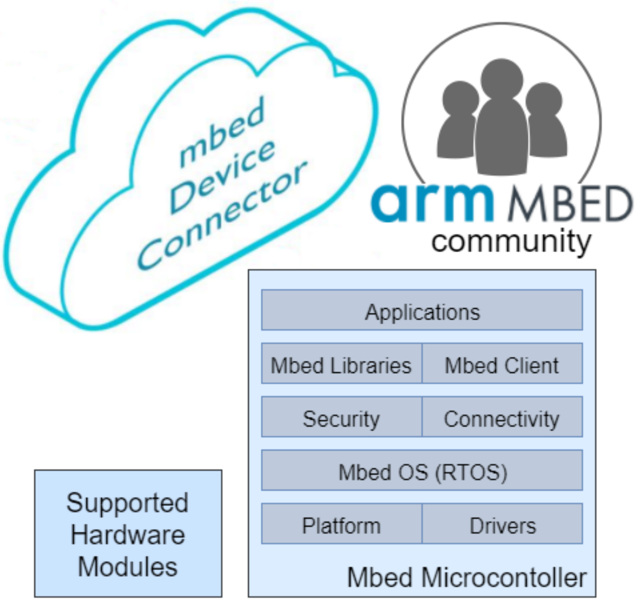


Figure 2.3: An Overview of the Mbed IoT Platform. (Adapted from Figure in Mbed OS Documentation [14])

Since the Mbed Platform does not include Cloud Services, to build an end-to-end IoT system, an Mbed device must connect to a Cloud provider, such as AWS, to add functionalities essential to IoT such as data analytics, data storage or Cloud computing. The main components of the AWS Platform are also explored below.

2.2.1 Mbed Operating System

Mbed OS is an Open Source Operating System designed by Arm. It is registered under the Apache 2.0 licence, providing all developers with a wide range of features and support for embedded applications including drivers, security, connectivity and Real-time features. Development boards [2] and hardware components from its partner companies which are Mbed Enabled (and thus compatible with Mbed OS) are listed on the Mbed OS developer Website [14].

Drivers and Platforms

Mbed simplifies the use of peripherals such as General Purpose Input Output (GPIOs), Timers and Interrupts providing a Hardware Abstraction Layer (HAL) [3] for Mbed Enabled microcontrolers. Mbed also provides Drivers for Serial Communications (such as UART, SPI and I2C). There exists a set of

[2] Development Board: A Printed Circuit Board (PCB) with a microcontroller and the surrounding hardware (such as pins, Power Supply Unit (PSU) and USB port) designed for rapid prototyping.

[3] HAL is a layer of software which allows the operating system (in this case Mbed OS) to interact with the hardware in the processor.

platform modules used to standardize the way the developer uses the different standard software libraries [4] and toolchains in Mbed. This in turn increases code portability, since libraries and programs implemented with Mbed can be simply used by two different Platforms by changing some minimum platform-specific parameters. Take as an example the mbed OS Blinky program in the Mbed Developer Website [14], the name of the pin for a specific Development Board must change, but they both use the same Drivers Library *DigitalOut* to blink a LED, meaning that the developer does not have to write the code at register level and can use the code across a range of platforms.

Real-Time Operating System (RTOS)

Mbed OS is based on the CMSIS-RTOS, which supports multi-threaded software operation for microcontrollers using Cortex-M processors. The CMSIS-RTOS API has numerous features to support multi-threading [5] operations such as: creating different threads, managing threads using semaphores and signals, and managing resources shared between threads.

Security

Mbed provides security both at the hardware level with the Arm Mbed uVisor and at the software level with Mbed TLS Library. Mbed TLS provides developers with a simple way to use cryptographic capabilities for embedded devices and is also used to establish a secure connection to MDC.

Network Connectivity and Protocols

Mbed provides libraries that implement several communication functions in different layers of the Network system in order to enable the device to connect to networks and access the Internet.

Mbed provides Network Interface classes used to connect over the Internet over Ethernet, WiFi, Cellular and Mesh Networking. Note that this Libraries have to be implemented for each hardware module. Figure 2.4 below shows the Inheritance tree of the NetworkInterface child classes implemented for different hardware modules.

[4] A software library contains a series of programming resources that aid the development of software. (Also referred to as library)

[5] Multi-threading is the ability of a program to run several tasks (Threads) at the same time. The OS creates an ilusion that the two threads are running simultaneously by rapidly switching between the two.

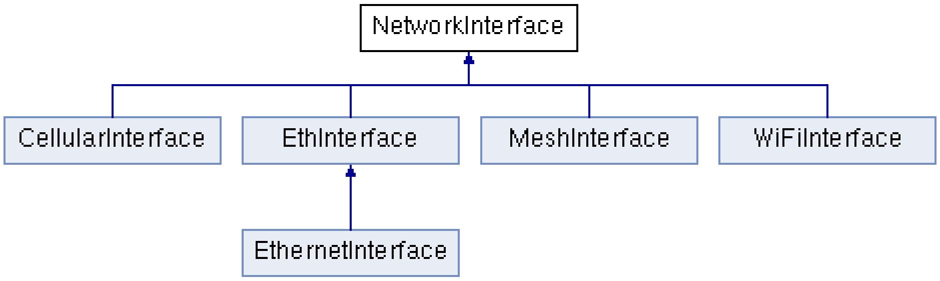


Figure 2.4: Mbed NetworkInterface Inheritance Tree. (Extracted from Mbed Documentation. [14])

For Mbed Enabled components, the child classes in Figure 2.4 are usually implemented by the hardware vendor or the Mbed Team and are available for the developer to use as libraries. Mbed Team also provides the easy-connect Library [23]: a collection of Network Interface implementations for some popular hardware components supported by Mbed.

- Transport Protocols:
  Mbed also provides libraries to send Internet data packets over TCP (TCPSocket Library) and UDP (UDPSocket Library) as well as a library for IP addressing to be used in combination with the previous two.

- Application Protocols:
  There are several Libraries used to communicate over the Internet using HTTP (*mbed-http* Library), MQTT (*MQTT* Library) and CoAP (*mbed-coap* inside Mbed OS). Several example projects are available at the Mbed developer website [14].

Device Connectivity

In many IoT systems, Things do not have Internet connectivity and frequently use non-IP communication protocols to connect to a gateway. Mbed OS offers other wireless connectivity methods such as Bluetooth Low Energy (BLE), Near-Field Communication (NFC) and Radio-frequency identification (RFID).

Mbed OS has a BLE library that abstracts the complexity of the BLE stack and can be used to create a BLE device that advertises GATT Services and Characteristics. Similarly to the Network Interface class explained earlier in this chapter, the BLE library requires an implementation specific to each Bluetooth hardware module. This is provided by vendors of BLE modules supported by Mbed in the form of a library. See Section 2.3.3 for an explanation of how the BLE protocol works.
Mbed provides a wide range of examples on how to implement BLE devices on Mbed OS, with examples on BLE services such as: Heart rate monitor, button and led [11].

Mbed Client

The Mbed Client library is used to establish a connection with MDC. Mbed provides a wrapper library called *simpleclient* to give an example implementation of the library and simplify its use for new developers. This Library abstracts the developer from all of the connection dependencies to the MDC Server, using many of the Mbed components mentioned above, such as:

- Secured Internet connection over Mbed TLS.
- Socket over TCP or UDP to establish the connection to MDC.
- Implement the LWM2M (Lightweight Machine to Machine) Protocol to share the device's resources with MDC.
- Network connection using a Network Interface implementation or the easy-connect library.

A general description of Mbed Client can be found in the Mbed Website [15], including links to sample code on GitHub to connect Mbed Enabled Device [22] and Linux Machines [21]. A guide and the Mbed Client API can be found on the Mbed Cloud Developers Website [13]. Mbed Client communicates with Mbed Device Connector using the Constrained Application Protocol (CoAP) [2]. An explanation of the CoAP protocol is given in Section 2.3.1

2.2.2 Mbed Device Connector

Arm Mbed Device Connector is an IoT Device Management Platform. The Platform offers device connectivity, end-to-end security, endpoint data collection and control.
The Thing is configured as an endpoint with LWM2M Resources, as shown in Figure 2.5 [6] .

[6] Device Resources: any asset related to an embedded system such as information, input devices (sensors) or output devices (actuators)

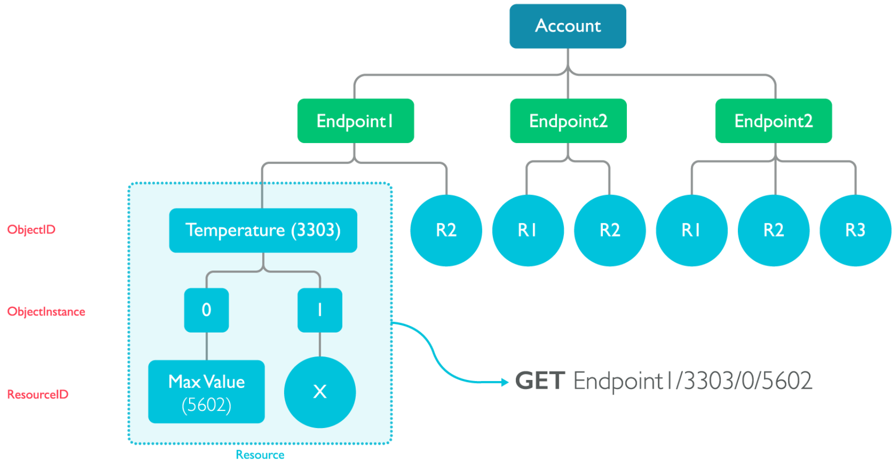


Figure 2.5: Diagram representing the LWM2M Resources of an endpoint device connected to MDC. (Extracted from Mbed Documentation. [14])

The Platform exposes these resources using a REST Model (explained in Section 2.3.4) and allows MDC Clients [7] to interact with the device. Table 2.1 summarizes the extensions supported by MDC as well a description of what they do as described in the API Console section of the MDC Dashboard web page [28]. To access the URL, the authorization token from MDC must be added in the body of the HTTP request in all transactions. Mbed Device Connector allows a developer to have 100 connected devices, 10000 transactions per hour and two access keys.

Table 2.1: The REST methods allowed by the Mbed Device Connector.

Method	Extension	Description
GET	/endpoints	List of connected devices
GET	/endpoints/endpoint-name	List of all the resources of a connected device
GET	/endpoints/endpoint-name/resource-path	retrieve the data of a specific device's resource
PUT	/endpoints/endpoint-name/resource-path	modify resource's data
POST	/endpoints/endpoint-name/resource-path	send data to a resource that doesn't modify it. [8]
DELETE	/endpoints/endpoint-name/resource-path	delete a resource and its data.

MDC Dashboard

A Thing connected to MDC can be managed and observed in the Mbed Device Connector Dashboard. From this webpage, the developer can download the security credentials for the device, the access keys

[7] MDC Client: Any device or application (such as web or mobile) able to perform HTTP requests in order to interact with connected devices via the MDC RESTful API

for applications accessing MDC through the REST API as well as observe the connected devices. This on-line portal also has an API Console which provides a way to test all of the functions of the REST API and interact with a connected device like an application would. This tool can become very useful when trying to debug the system.

Python mbed-connector-api Library

The mbed-connector-api-python library [29] can be used to manage, monitor and access devices connected to MDC. The library abstracts the developer from the HTTP requests necessary to interact with MDC. The Library formats the HTTP address for each operation, attaches the MDC access key to the header of each HTTP requests and inserts the message at the body. Most importantly, it handles the asynchronous responses of the MDC REST interactions, as described in the MDC API [13]. The developer only needs to write some callback functions [9] to handle the events happening at the MDC.

2.2.3 Amazon Web Services

AWS provides a wide range of Cloud Services, from the classic storage and computation services to Data Analytics, Application Services and IoT Tools [5]. The services used during this project are outlined below. Although AWS targets big organizations, providing services that aid the company's fast deployment and scalability, it is also a great solution for small projects and developers, expanding the capabilities of any system being developed at an affordable cost.

AWS for Developers: AWS SDKs

AWS has an extensive set of developer tools and Software Development Kits (SDK) available for developers. There are three main categories under the SDKs, Services SDKs, AWS IoT Device SDKs and AWS Mobile SDKs. There is extensive documentation available for all AWS Services and SDKs, as well as example applications and extensive documentation for each Service [6], sample code and Qwiklabs [9](Learning Portal for AWS Services) .

- Services SDKs: This SDKs are developed to access AWS Services, available in multiple different programming languages and with an extensive API and online documentation. The available languages are: C++, Java, python, JavaScript, .NET, PHP, Ruby and GO.
- AWS IoT Device SDKs: This SDK is specific for the AWS IoT Service, to simplify the connection to the MQTT Broker, the devices authorization through certificates and messaging func-

[9] A callback function is a function that gets executed when a specific event occurs, interrupting the current thread of the program momentarily.

tionalities. There are SDKs available for generic devices in a set of Languages (C, C++, python, JavaScript and Java) and a device-specific SDK devices for Arduino Yun.

- AWS Mobile SDKs: AWS provides two SDKs specfic to Android and iOS Devices to use all AWS Services with which to develop Mobile Applications.

AWS IoT

AWS IoT is a Device Management Platform from which devices can be registered creating individual device certification. The Platform can be configured to allow devices to interact with other Cloud Services using a Rule Engine to trigger actions. AWS IoT uses the MQTT Protocol to communicate with IoT Devices (see a description of this protocol in Section 2.3.2). There is a sample code written by Klika-Tech Team [19] that connects a specific Mbed Device (the STM32 Nucleo F401RE board [24]) to MQTT directly by implementing the Mbed MQTT Library. [10]

- AWS Security: In order for a device to be able to access AWS IoT, it must have the appropriate authorization. Certificates are created and downloaded into each device to ensure only authorized devices have access to the MQTT Broker. Policies are attached to this certificates that describe what the device with this certificate can do on AWS IoT, such as enable it to connect, publish, subscribe, etc. AWS offers the possibility to generate certificates for the developer or upload the developer's own certificates.

- AWS Manage: In order for a device to connect to AWS IoT, it has to be added to the registry and have certificates with the correct policies attached to it. AWS IoT offers ways to create device's groups that define a set of common attributes that this will have. The developer can also monitor the devices with statistics about the incoming messages and test devices using the MQTT Client in the *Test Console*.

- AWS Rules & actions: AWS IoT Rules [10] are configured to be triggered only for specific messages using SQL-like syntax to filter the incoming messages by Topic, a specific attribute inside the message and for a condition of this attribute. To each rule, a set of actions can be attached. The developer can attach more than one action for the same rule, choosing from a list of actions provided by AWS, as shown in the Documentation. [7].

  For example, for a given device publishing the room conditions in JSON format, a rule can be configured for the specific Topics "name-of-device/room-conditions/" to trigger an action when

[10] This sample code is not supported by all Mbed Devices and uses a previous version of Mbed OS and outdated wifi drivers.

the attribute "temperature" is greater than 30. The message can then be send as an SNS push Notification to, for example, an email address to alert the user.

- Device Shadows: When a device is registered in AWS IoT, the service automatically creates a series of MQTT Topics that holds the devices information even when it is not connected, called Shadows. [8] MQTT Messages are not hold anywhere, they are just received by an MQTT Client subscribed to them. The Shadow holds the "reported" state of a device's attribute, and an MQTT client can update the "desired" state of the attribute. This allows an AWS Service or MQTT Client to update an attribute when the device is not connected, and the device will receive the "desired" value when it re-connects.

Cloud Computing Services

- AWS Lambda: AWS Lambda runs *Serverless* applications, in which the user does not provision or manage the Servers in which the application is running, but rather lets AWS handle the resources necessary to handle the application. This applications are not constantly running, they have to be triggered in some way: such as from AWS IoT Actions, another AWS Service, from a Web or Mobile Application or using one of the SDKs.
- AWS Elastic Cloud Computing (EC2): The EC2 is a Cloud computing service that allows developers to deploy virtual machines and configure memory, CPU and instance storage to enable rapid scalability of cloud computing solutions.

AWS Machine Learning (ML)

AWS ML service is capable of running predictions from an ML model in the Cloud. The service guides the user step by step in the process, generating a data source from a csv file, adjusting the ML model type and parameters and performing an evaluation of the accuracy of the ML Model. The service features Real-time predictions (with which Services SDKs can be used to obtain a prediction for some given input values) and Batch predictions (for a given input dataset, the out is predicted). This Service uses the Simple Storage Service (S3) AWS to load and store the data sources to train and evaluate ML Models as well as generate and retrieve batch predictions.

The ML AWS Service allows the developer to select a range of possible algorithms including regression and classification algorithms. Which type to use, depends on the type of data trying to predict: Regression models are used to predict continuous values and classification is used to predict a field of categorical data or a binary value (which is essentially like a category with only two possible fields.

2.2.4 Other Mbed Solutions

Arm Mbed Team have developed other software solutions related to IoT Systems. Both of the solutions explained below are experimental software and are not suitable for production (prototype solutions).

Mbed Bridge Connector

Mbed Device Connector does not provide Cloud Services for storage, processing and manipulating data. Mbed Bridge Connector provides a way for Mbed Device Connector to interface with a range of Cloud Providers: AWS (Amazon Web Services), Microsoft Azure and IBM Watson IoT. Mbed Bridge Connector is a Docker Container Application. It can run in any computing environment capable of running Docker, independently of the computer's OS and any other dependencies. After running the application, the user must configure the Connector Bridge inputting AWS private and public keys as well as an MDC access key.

When configuring the Bridge Connector to work with AWS, the Application observes all of the resources being registered to MDC and configures AWS IoT to handle traffic between MDC and AWS. The way it does this is by having an Access Token to use the MDC REST API and at the same time have authorization to connect to the AWS Broker as an MQTT Client. When a Resource is updated in MDC, the application automatically publishes to an AWS Topic with a Specific format. AWS Services and other MQTT Clients can interact with MDC and connected devices, by subscribing to this Topics. An MQTT Client can perform the REST methods supported by MDC (See Table 2.1 by publishing to specific topics with a specific payload. The format of this Topics and Payloads is explained below.

For a given endpoint connected to MDC, the MQTT Topics to which the Bridge connector publishes and subscribes have the format:

Topic: *mbed/[operation]/[endpoint-type]/[endpoint-name]/[resource-name]*

The *endpoint-type*, *endpoint-name* and *resource-name* are specific to the users endpoint. The possible operations are *get, put, post, cmd-response, notify*. Messages arrive automatically to the *notify* Topic when an observable resource is updated. When performing the *get* operation, the data is received asynchronously in the *cmd-response*. When performing the *put* or *post* operations, the payload of the message being send must be as followed:

Payload: *"path":"[resource-name]", "ep":"[endpoint-name]",*
"coap_verb":"[operation]", "new_value":"[your-value]"

To get familiar with the format of the topics, it is recommended to subscribe to the *mbed/#* Topic in the Test Console of the AWS IoT webpage. The '#' sign is a *wildcard*, so all of the traffic between the

Connector Bridge and AWS is picked up here. As an example, to perform a PUT operation to update the value of a resource, the MQTT Topic would be:

Topic: *mbed/put/test/83a4d6e0-fcc4-469f-96d4-6f3a33ecd636/13/0/3*

Payload: *"path":"/13/0/3", "ep":"83a4d6e0-fcc4-469f-96d4-6f3a33ecd636",*
"coap_verb":"put", "new_value":"1995"

DeviceLink BLE

Mbed DeviceLink is an experimental software to connect BLE (Bluetooth Low Energy) Devices to Mbed Device Connector and Mbed Cloud. The software is deployed in a Linux Machine that acts as the Gateway to the Internet for the BLE device, enabling Cloud connectivity of a non-IP device. Mbed DeviceLink translates the data traffic between the device and the Cloud. This has to be configured for each resource by the developer.
The experimental Software developed by John Jongboom [18] provides a method for connecting any BLE device to MDC. Since BLE is a non-IP protocol, a gateway must be used to route the information to MDC. This is accomplished by two pieces of software: a JavaScript application that connects to BLE devices and a script running Mbed Client. The Linux machine must have both Bluetooth and Internet access capabilities as well as an Internet connection, such as a Raspberry Pi 3. The developer is required to translate the data traffic through the Linux machine by writing JavaScript functions that interpret the data coming from each end of the device. To summarize, the Platform automatically routes messages coming from the BLE device to MDC and vice versa.

2.3 Background Information

This section summarizes the information and concepts necessary to understand some of the Platforms and technologies used.

2.3.1 Constrained Application Protocol (CoAP) Protocol

This protocol was implemented specifically for IoT applications and it is considered to be similar to HTTP only more lightweight and simpler. This specially suits the storage constrain of IoT nodes, since the protocol can be implemented in devices with really low RAM and Flash memory units. Another significant benefit of the CoAP protocol is that it uses the Representational state transfer (REST) Model, which becomes extremely practical when integrating the device on web applications and internet resources. The REST Model is explained further at the end of Section 2.3.4. The Internet

Engineering Task Force (IETF) has reviewed this protocol extensively and published a Request for Comments (RFC) document with the protocol's specifications [36]. RFC documents are used for documenting Internet standards.

2.3.2 Message Queuing Telemetry Transport (MQTT) Protocol

The MQTT Protocol defers from the common Client-Server configuration in the way transactions between the Server and the Client are handled. In the MQTT Protocol there is what is called a Broker instead of a Server. In the Client-Server configuration the Client establishes a connection with the server and sends requests messages, for which the Server can send a response. The Client-Broker on the other hand defines message Topics which can be seen as channels of communication through which Clients send messages. Clients can Publish messages to this Topics, and Subscribe to a Topic to receive messages being published to this Topic. The Broker handles the filtering of the topics, routes messages to subscribed Clients and handles the connections with Clients. Figure 2.6 below shows a representation of the two configurations.

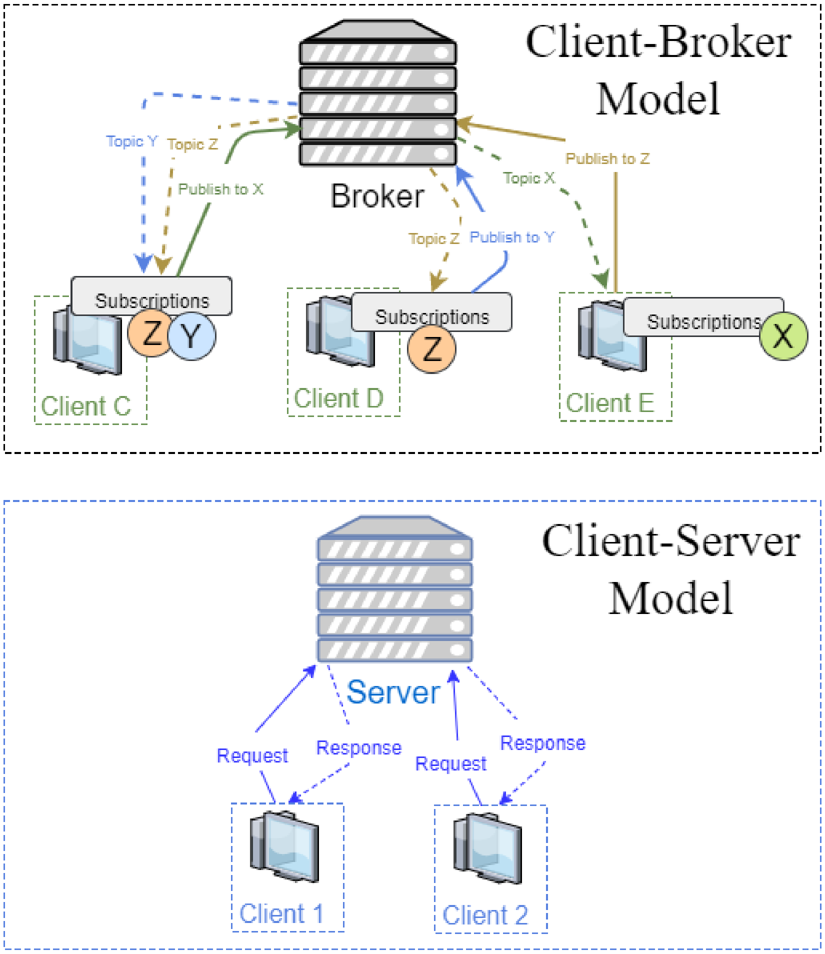


Figure 2.6: Client-Server Model and Client-Broker Model. The colors of the messages show three different MQTT Topics X, Y and Z.

2.3.3 Bluetooth Low Energy (BLE) Protocol

The BLE Protocol follows a Client Server configuration. A BLE Device can only act either as a Server or as a Client. It acts as a Server when the device contains information that some other device can access while the Client is the device accessing that information. A BLE device can be in two different modes: advertising and connected mode. When the device is in advertising mode the device is sending generic information to the surrounding devices. The Generic Access Profile (GAP) is the standard format in which this information is being advertised periodically. Any BLE can read this advertising message and identify the device and read some basic information from it. BLE devices can only establish a one-to-one connection, defined as connected mode. Once the connection is established, the Server can share information with the Client using the Generic Attribute Profile (GATT). The GATT Profile structures data in three layers: a Server can have a number of Services, that in turn have a number of differen characteristics. This characteristics in turn can have a Descriptor with a description of the characteristic to which they correspond. Figure 2.7 shows the structure of a GATT Profile [14]).

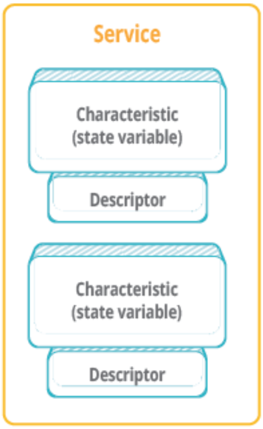

Service
Characteristic (state variable)
Descriptor
Characteristic (state variable)
Descriptor

Figure 2.7: Diagram Representing the GATT Profile of the BLE Protocol. (Extracted from Mbed Documentation. [14])

2.3.4 HTTP RESTful Requests

The Representational State Transfer (REST) Model is a series of constrains set upon designing systems to ensure the way they interact with other systems is standardised. It has become really popular when designing web applications to use the REST model, to simplify the way resources are accessed, modified and managed by users. Thi is done through a set of defined methods such as (GET,PUT,POST and DELETE). To use RESTful HTTP applications, it is as simple as making an HTTP Request as it would be done when opening a website, only the HTTP Uniform Resource Locator (URL), also known as web address, contains extensions with the specific resources and methods being accessed.

Chapter 3

IoT System

3.1 System Overview

Chapter 2 outlines some of the software implementations available to connect to MDC adnd AWS. It was decided to explore the other solutions from other IoT Platforms, and the System in Figure 3.1 was defined as a general system to connect Mbed Devices to AWS.

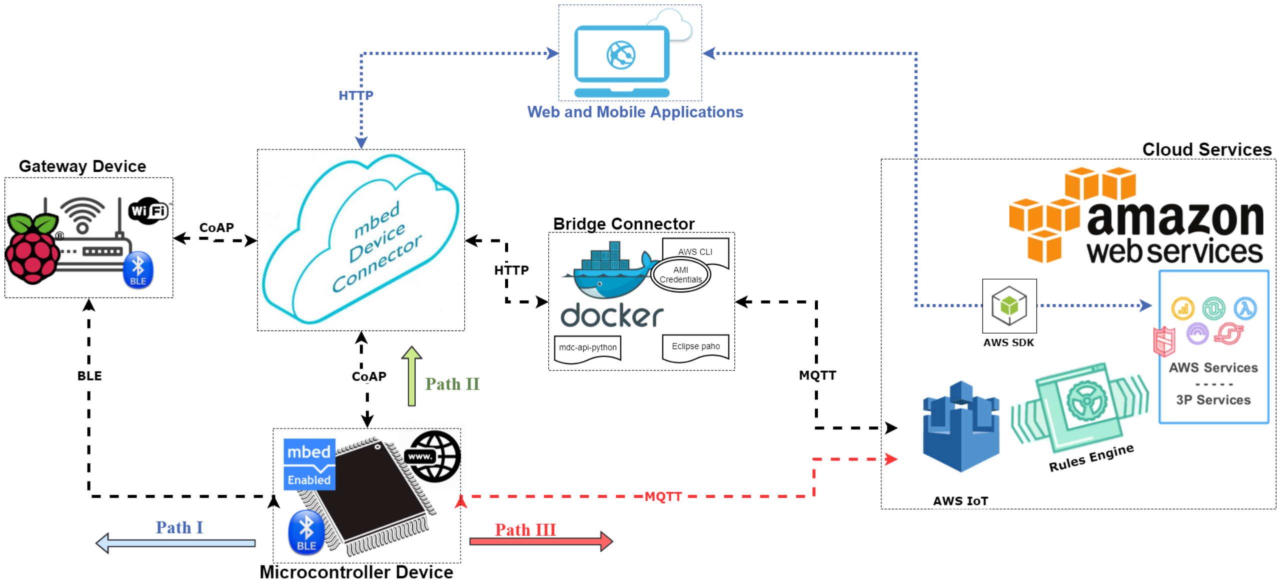


Figure 3.1: System Implementation for Mbed Devices connecting to AWS. (Icons and logos where extracted from Internet Resources)

The different solutions where tested in terms of speed and message size. The sections below outline the different components and the way they interconnect. It is important to note that Path III is solely used for comparison, as it is currently not an implementation that is supported by all Mbed Enabled devices, but rather an example implementation running a previous version of Mbed OS. From the literature review in section 2.1.1 the IoT System Layers defined by research where identified. From this definitions, the following layers in figure 3.2 where identified for this system.

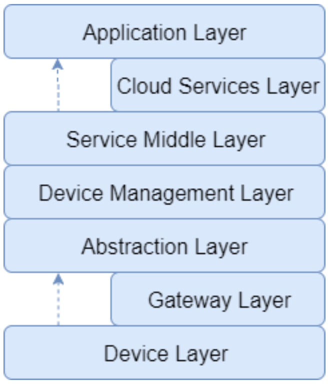


Figure 3.2: The Layers of the proposed System to connect Mbed devices to AWS

Endpoint Layer

These devices require appropriate hardware modules and software implementations to enable Internet access (such as Wifi, Ethernet and Cellular) and have the correct implementation to establish a connection with the Server. In the case of Mbed Devices, Internet access is accomplished using hardware modules supported by Mbed and porting a software implementation of a Network Interface, as explained in section 2.2.1

In the system we can define three different device implementations: one that connects to MDC via a Gateway (Path I), another one that connects directly to MDC (Path II) and a last One that connects directly to AWS IoT (Path III). Path I uses the DeviceLink project mentioned in Section 2.2.4, in which a Raspberry Pi acts as a Gateway that translates the messages between MDC and the BLE device. Path II is based on the example application provided by Mbed Team [22]. It uses the easy-connect library to gain internet access and the Mbed Client to establish a connection to MDC. This two implementations are then Bridged to AWS IoT using the Connector Bridge application described in section 2.2.4. Using the Connector Bridge enables this devices to use AWS Services in the same way any other AWS IoT device would. Implementation C based on the example application made by Klika-Tech [19] which uses the Mbed MQTT Library to connect to the MQTT Broker from AWS IoT.

Gateway Layer

Connecting a device to the Internet is an expensive feature in terms of the device's power consumption, price and design complexity compared to other Non-IP low-range communication methods such as

BLE, RFID and NFC. Connecting a Non-IP device to the Internet adds an additional device in the system with Internet access: a gateway.
The Gateway must have the capability to communicate with the Non-IP Device as well as have the ability to communicate with the MDC Server. In the case of this system, the Raspberry Pi will have to be configured to translate the data from the BLE communication protocol to CoAP protocol. It does so by having to programs that communicate with each end of the bridge and information is passed between the two pieces of software.

Abstraction Layer

This layer defines the format of the information, to ensure the interoperability of devices: declaring the structure of the data for a specific resource, ensures that two different implementations behave in the same way and that the data from both of these devices follows certain standards that guarantee the quality of the data being collected. For example, standardizing the data structure of a room thermometer, ensures that a Cloud service could perform data analysis from collecting two devices implemented by different companies, without having to mind about incompatibilities.

Device Management Layer

This is the part of the Platform to which the connection is established and the message arrives. The Endpoints must open a Socket to communicate with the Server with a specific Socket Address. The Protocol Stack must implement the corresponding Application Protocol that the Server can interpret and must have Certificates that authorize its connection to the server. In the case of MDC, the communication occurs using the CoAP protocol [36], while for the AWS IoT its the MQTT Protocol [34]. The credentials to allow the device to connect to the Cloud is also specific to the Service.

Service Middle Layer

While the Device Management layer receives and sends the messages to the device, the Middle Service actually handles what happens with that message. This layer of the system between the Device Management and Cloud Services. It organises the messages incoming to MDC or AWS IoT. In the case of MDC, the Server stores the new value and exposes the data via the RESTful API and therefore an application can interact with the device directly using this interface. With AWS IoT on the other hand, messages arrive to the MQTT Broker and the developer must take further action to collect this data or store it.

Cloud Services Layer

Cloud Services are accessed by devices via AWS IoT. This is done by triggering AWS IoT actions that are configured for messages incoming to the *notify* Topic of the Resource. Cloud Services can communicate back with the device by publishing to the device's resource MQTT Topics, as outlined in Section 2.2.4.

Applications Layer

Applications are designed to interact with the System by utilizing the information collected at the Devices, controlling the Devices, triggering actions and even Managing the Devices themselves. Some applications would access the Device directly by interacting with MDC via the REST API. In some other applications, it is desired to access processed data, whether it is data stored in the cloud or the output of some computation regarding data collection, in which case the Application accesses the corresponding Cloud Service using the AWS SDKs. Note that even though it might be redundant, an application could use the AWS IoT SDK to publish and subscribe to Topics from the device, that would in turn have the same effect as using the MDC REST API.

3.2 System Evaluation

Three different solutions where tested in terms of its ping time and packet size. The ping time of a connection is a measure of how responsive a server is and it's the time taken for a server to echo a message originated from a client. The packet size is the maximum size in bytes that a client can send to the server on a single transmission. The System can be divided into three paths with different implementations as shown in Figure 3.3.

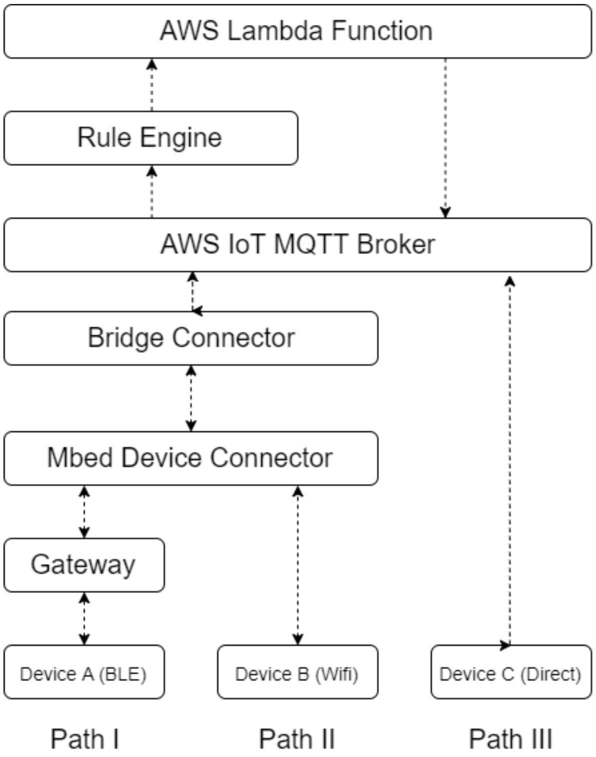


Figure 3.3: The three different paths shown in Figure 3.1 simplified to clearly show the experiment's objective. Note that all this connections and platforms have been previously described in Section 2.2

To evaluate the system, programs and scripts where designed for the three different device implementations, including the appropriate configuration of AWS IoT Actions. The code implementations can be found in the Project's GitHub account [31]. The hardware components used for the evaluation are:

- ST Nucleo F401RE [24] development board
- ST X-NUCLEO-IDB05A1 BLE shield [25]
- ST X-NUCLEO-IDW01M1 Wi-Fi shield [26]
- Raspberry Pi 3

Experiment 1: Device to AWS Service Ping Time

The first experiment carried out is the ping time to access an AWS Service. To measure the ping time, the system must be set up to automatically respond when the message arrives to the AWS Service. The ping time was found to vary greatly depending on how often messages where sent, so for each route the frequency at which outgoing messages are send was varied and three different system states are defined:

- Stressed: The frequency with which messages are sent is set very low (0.01 seconds). Nevertheless, the microcontroller is configured to wait for the sent ping message to arrive before sending the next one, so in this state the device just sends a message as soon as the previous one arrives.
- Close to stress: The period between ping time messages is slightly higher than the ping time recorded under stress.
- Unstressed: The period is set to be significantly greater than the ping time under stress.

Table 3.1 shows the 3 different intervals of the system: 50 samples where taken at each of the three different periods.

Table 3.1: The Periodic intervals in Experiment 1 for each of the paths.

System Path	Period 1 (seconds)	Period 2 (seconds)	Period 3 (seconds)
Path I	0.01	7	10
Path II	0.01	7	10
Path III	0.01	3	5

For the three different device implementations, the device has two resources: an outgoing resource and an incoming resource. At set intervals, the device updates the outgoing resource with the value of a counter and starts a timer. When the Device receives the response from the AWS Service, the counter is increased, the timer read and ping time stored. The Device is configured to wait until the incoming messsage has arrived before sending another outgoing message. The messages are sent a set number of times for each of the three frequencies mentioned above. A *start* resource is created to control the program and a *time* resource with a comma-delimited string with all of the ping times from the test. To start the program, a POST Request with a payload string "START" must be performed and at the end of the program, the device updates the *time* resource with all of the recorded ping times.

For the three paths, the MQTT Topics for the outgoing resources have an AWS IoT Rule attached that triggers an AWS Lambda function. This Lambda function is the example AWS Service being accessed, and it carries out the echoing, by publishing back to the incoming message. In the case of the resources of the two Bridged paths, the Lambda function also formats the payload for the *return* resource correctly for the Bridge Connector to interpret, as explained in section 2.2.4. There is a Republish Action for AWS IoT Rules which could be used for Path III, but for consistency, a Lambda function is also used to carry out the echoing.

Experiment 2: Ping Time for different Packet sizes

The three different paths use different protocols: MQTT, BLE and CoAP. Note that this s not a comparison of the three protocols, but rather of the three different device solutions being used. Each of this solutions have a maximum packet size. In this experiment, each of the three paths the ping time is recorded for minimum (1 byte) and maximum data Packet sizes shown in table 3.2, and the ping times for this two sizes are compared.

Table 3.2: The maximum packet sizes for the different paths of the system.

System Path	Protocol	Maximum Packet Size
Path I	BLE	26
Path II	CoAP	1000
Path III	MQTT	300

This test could not be performed from end to end of the system using the Connector Bridge, since that would limit the maximum package size of all three paths to be 300. For the Devices A and B connecting to MDC, a *MDC Python Script* was created using the *mdc-python-api* [20] to perform the echoing, as shown in figure 3.4. In the case of Device C, the Republish Action mentioned in experiment 2 is attached to the AWS IoT Rule for the outgoing MQTT topic. The ping times for each path are compared per byte and percentage increase from minimum to maximum bandwidth.

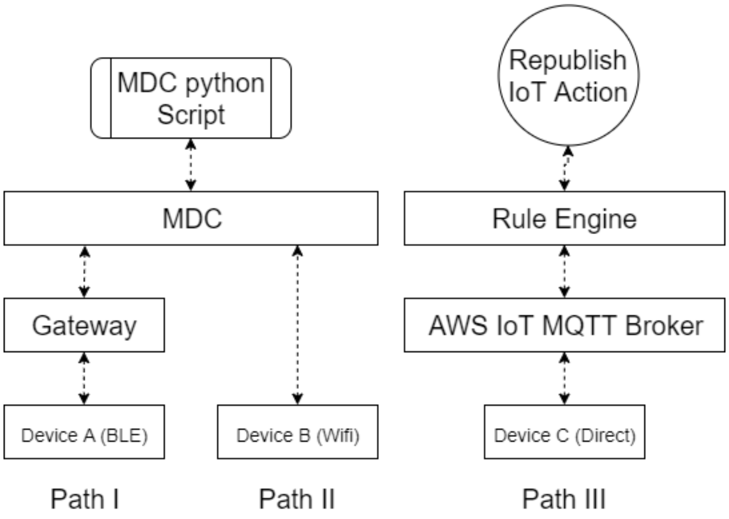


Figure 3.4: System configuration for the second experiment.

Experiment 3: Path II time per system component

This experiment measures the time spend in each of the 3 components of the system: from the device to MDC, from MDC to AWS IoT and from AWS IoT to Lambda. This measurements where accomplished by writing a script that uses the mdc-python-api to listen to the MDC server and the AWS iot-python-sdk to subscribe to AWS IoT. As the messages pass through the system, the script is being notified of the outgoing resource update and has a subscription to the MQTT Topics for the outgoing resource and incoming resource. Using a timer, the script calculates the time spent at the Bridge and AWS Lambda. The time from the device to MDC can be calculated by subtracting the two recorded times from the ping time. Figure 3.5 shows the system and the places where measurements are taken by the application. The labeled time intervals where calculated as followed:
$T_2 = t_1 - t_0$
$T_3 = \frac{t_2 - t_1}{2}$
$T_1 = \frac{P}{2} - T_2 - T_3$

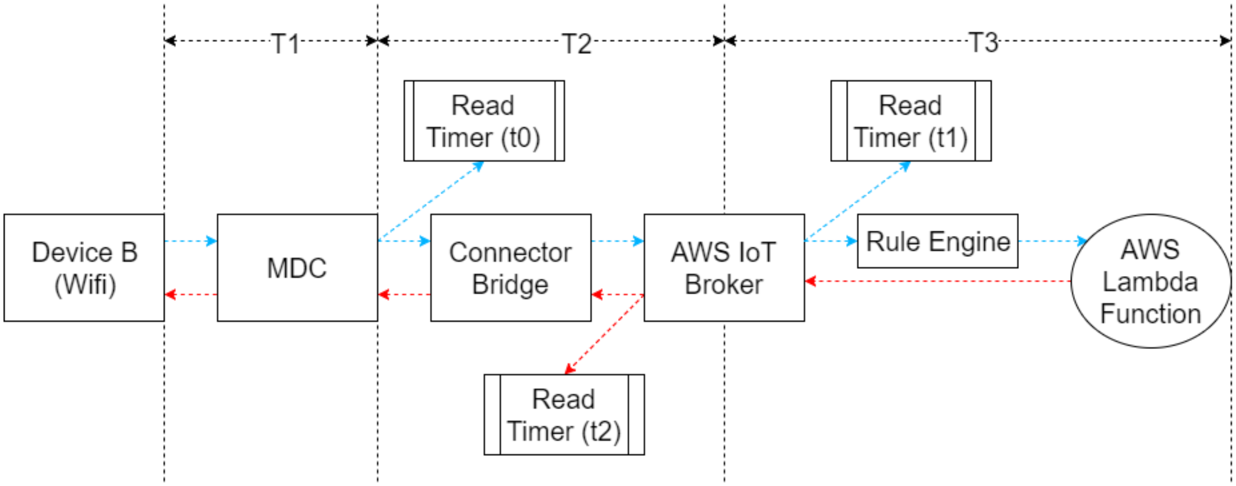


Figure 3.5: System configuration of Experiment 3: The place at which the Timer is read are labeled, as well as the three different sections of the system being measured.

Experiment 4: Comparing Path A and Path B ping time.

This experiment compares the connection of Path A and B to MDC, unlike Experiment 1 that compared the ping time for the three different paths to AWS. To do so, the *Bouncer MDC Script* mentioned in Experiment 2 was used to echo the messages from A and B devices. The system configuration is the same as that Path I of Figure 3.4.

3.3 Evaluation Results and Discussion

Experiment 1: Device to AWS Service Ping Time

The Periodicity at which ping messages where sent for each of the three implementations is shown in Table 3.1. Note that the system was designed for the *return* message to arrive so the 0.1 period just sends messages at the greatest frequency possible. The test runs continuously recording 50 ping times for each interval. The packet size is maintained to the minimum: just a counter that increases with every ping time sent.

Figure 3.6 and Table 3.3shows the results from the test. The continuous sending of values is described as the system being under stress, while the state with the interval close to the ping time is referred to as *close to stress* and finally the last one *unstressed*. See appendix A with the figures with the graphs of ping time against samples, showing the ping times for all 50 ping time recordings.

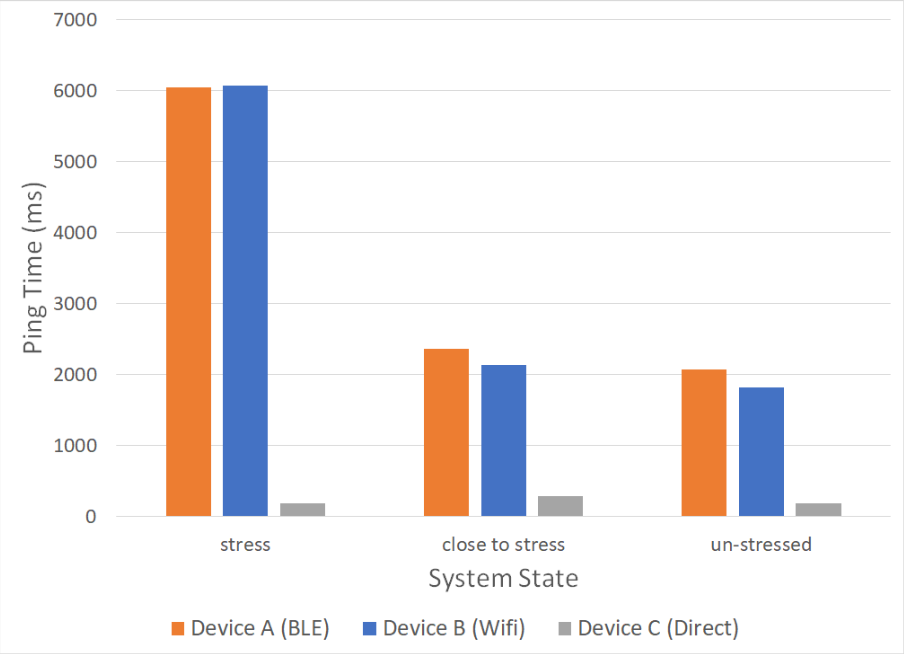


Figure 3.6: Bar chart with Experiment 1 results: the average ping time of 50 samples taken at the three system states.

The results show that Path III is 10 times faster than the other two Paths. This result does not come as a surprise, since the MDC Server would take a significant amount of time to carry out the REST notification service compared to the time taken for the MQTT Broker to automatically forward an incoming message to the Rule engine. From Paths I and II, the ping time was seen to significantly increase with the system under stress rather than unstressed. For some unknown reason, when the

Table 3.3: Table containing the data from Fiure 3.6.

System Path	Stress State	Close to Stress State	Unstress State
Path I	6044 ms	2369 ms	1819 ms
Path II	6072 ms	2136 ms	2075 ms
Path III	188 ms	284 ms	194 ms

device sent the ping time message as soon as the previous one arrived, the message gets significantly delayed. Furthermore, when close to the stress state, the all three paths performed slightly worse than when unstressed. From the graphs in Appendix A it can be seen that the the three paths exhibit a more erratic behaviour when close to stress. Further investigation would be needed to understand the reason of this behaviour.

Experiment 2: Ping Time for different Packet sizes

During this test the Packet sizes being send where varied from the minimum packet size (counter value increased with every ping time send) to the maximum for each specific system. Figure 3.7 shows the ping times for the test. See appendix A with Figure A.2 with all 50 ping time recordings.

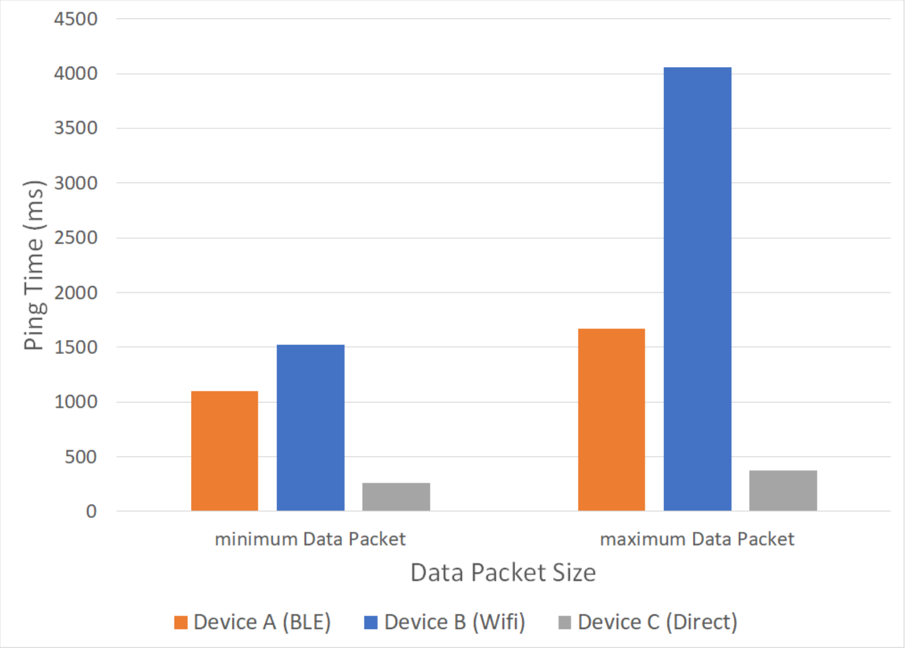


Figure 3.7: Bar chart with Experiment 2 results: the average ping time of 50 samples taken at the maximum and minimum packet sizes for each of the three paths.

Note that this is not a fair comparison of the system since they are traveling different paths as explained in section 3.2, so the data has to be manipulated to find a measure proportional to the increase in data packet size. The percentage increase from minimum to maximum data packet was calculated and then divided per byte of maximum data packet resulting on the percentage increase per byte. The results for the test are shown in Figure 3.8.

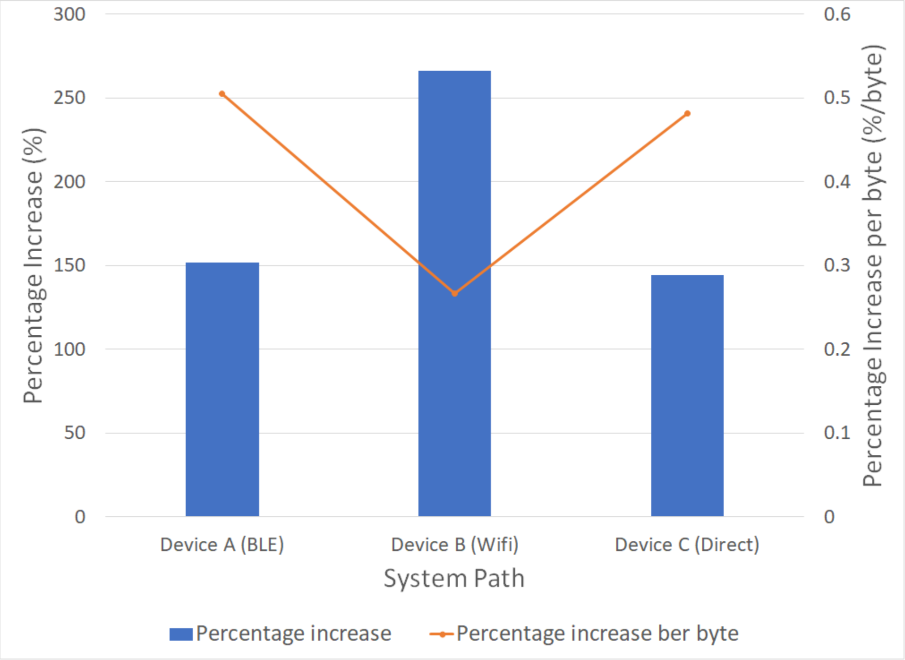


Figure 3.8: Comparing the percentage increase in ping time from sending the maximum data packet compared to the minimum data packet. The percentage increase is then compared per number of bytes transmitted when sending the maximum data packet.

The results of this experiments show that Path II is the fastest one in terms of ping time per second while the other two have similar results. This shows that it is important to consider the packet sizes in the system, since having to send more than one separate message for the same piece of information adds a latency as well as complexity since the message has to be reconstructed at the other end. It can be seen from Figure A.2 that the behaviour is more erratic when sending the maximum data packet (specially for Path II) showing that the larger the messages, the more unstable the ping time becomes.

Experiment 3: Path 1 time per system component

The results from the test are shown in Figure 3.9. Note that the test was performed using the minimum data packet, and the state of the system was changed from stress to unstressed (10 second period). The percentage of time spent in each component of the system was calculated and the results are shown in Table 3.4.

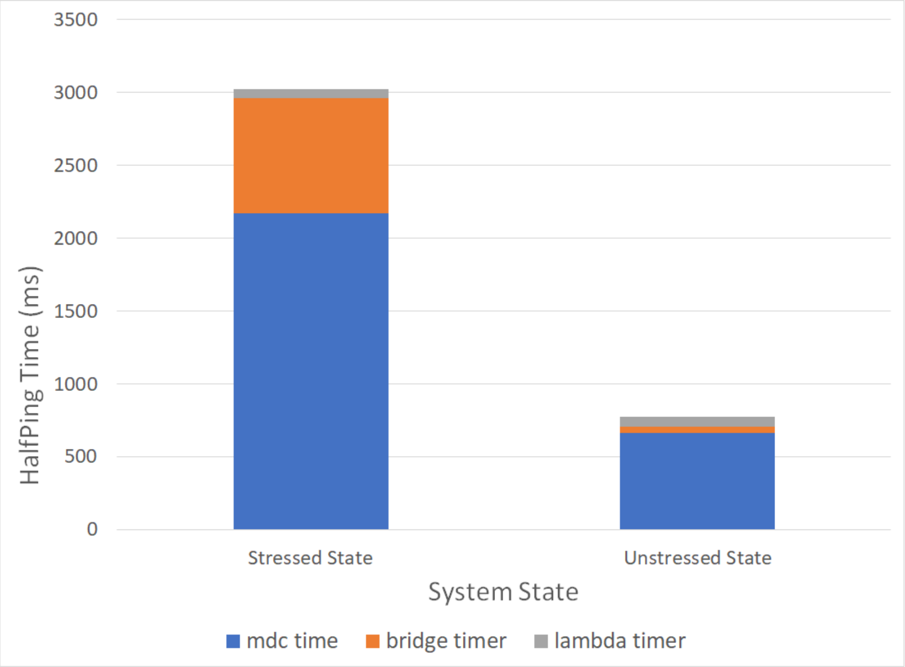


Figure 3.9: Bar chart with Experiment 3 results: The time taken at each component of Path II. Note that the Ping time is the time to travel to the Lambda function (half ping time)

Table 3.4: Table containing the percentage of time taken by each component when travelling through Path II

System State	MDC Time (%)	Conector Bridge Time (%)	Lambda Time (%)
Stressed	72	26	2
UnstressedS	95	5	10

This experiment tries to clarify nature of the unknown behaviour in Experiment 1 by analysing the time taken by each component in the system. From the results, it can be seen that the signal takes the most time to travel through MDC and be notified to MDC Clients. A possible explanation on why this part of the system takes so long is because the Bridge Connector and the Python Script taking the measurements both use long polling. A possible solution to improve the performance is explored later in the Conclusion in Chapter 5.

Experiment 4: Comparing Path A and Path B ping time.

From the results in of Experiments 1 and 2 there was ambiguity on whether the ping time for Path I was greater than Path II. Therefore on this last experiment, the two paths are tested up to the MDC Server and back, to eliminate the Connector Bridge from the test to see if that was affecting the results. The results are shown in Figure 3.10

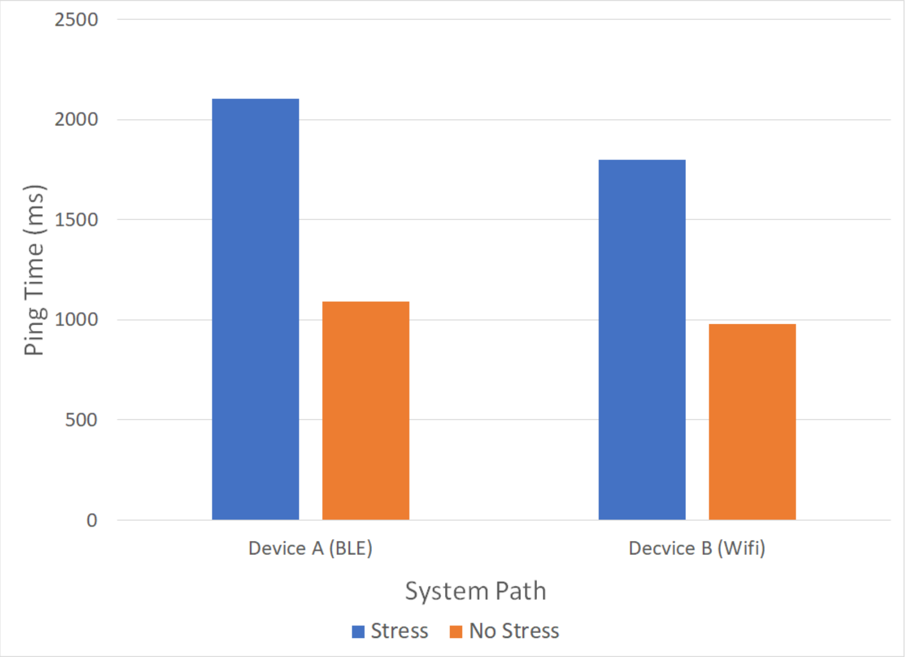


Figure 3.10: Bar chart with Experiment 4 results: Average ping time on Paths I and II when travelling to the MDC Python Script and back.

This experiment clarifies the ambiguity presented by the results of experiment 1 and 2, showing that indeed Path II is faster than Path I. This is as expected, since Path I requires the extra steps of the device communicating with the gateway, the gateway processing the information to finally forwar the data to MDC.

Chapter 4

Example IoT Application

To demonstrate the operation and functionalities of the system and the platforms, an example use case of the system was designed. The system was implemented that mimics the functionalities that a smart home would exhibit. An analysis of the system's operation is provided, defining the interactions between the different components of the system.

In order to explore the operation of the system proposed in the previous chapter, an IoT application was created which mimics the functionalities that a smart home would exhibit. Firstly, an analysis of the system's operation is provided, defining the possible interactions between the different components of the system and how they would be implemented for the proposed system. The specific operation of the smart home application is then explained in more detail.

The use case is implemented using the following hardware:

- ST Nucleo F401RE [24] development board
- ST X-NUCLEO-IDW01M1 Wi-Fi shield [26]
- ST X-NUCLEO-IKS01A1 sensors shield [27]

The use case explores Path I of the system proposed in Section 3.1: an Mbed Device with a Network Inteface hardware that allows it to connect to the MDC Server. The Connector Bridge is used to route the traffic from MDC to AWS IoT, from which AWS IoT Actions are triggered to access other AWS Services.

4.1 System Operation

The System can operate in a different number of ways, depending on the needs of the application. Operations in the system can be undertaken by three different clients: the Device (mbed Enabled device connected to MDC), the User (Application with MDC Access Key capable of performing HTTP REST Requests) and at the Cloud (AWS Services interacting with the rest of the system via the AWS IoT MQTT Topics generated by the Connector Bridge). Figure 4.1 shows the operation of the system in its simplest form, in which the three clients carry out all interactions via the MDC Server.

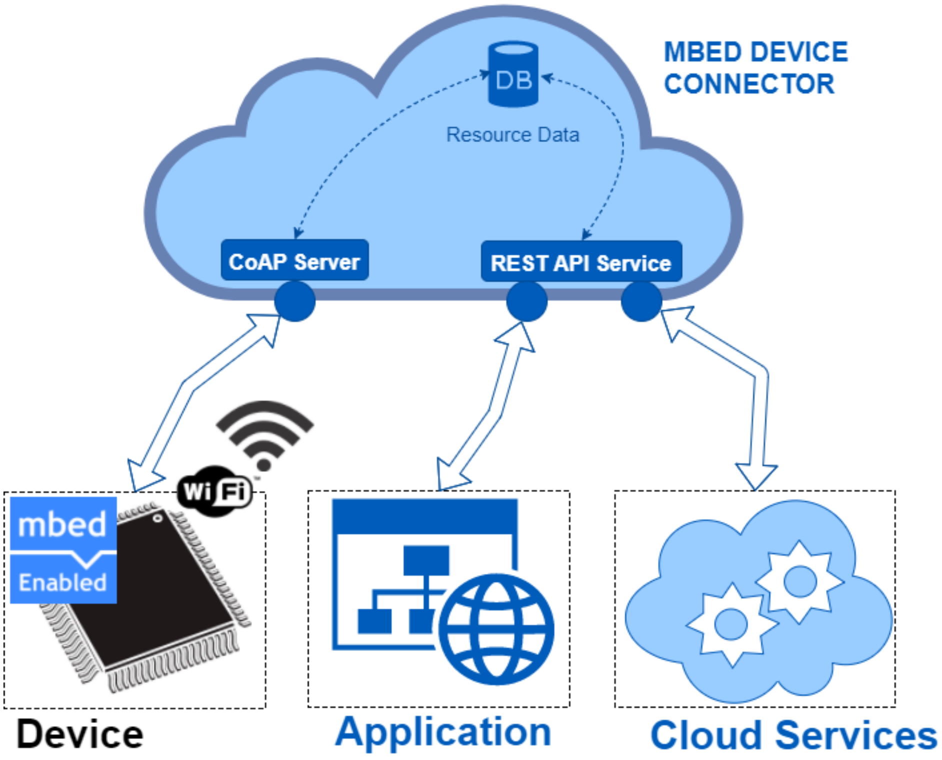


Figure 4.1: Graphical Representation of teh Mbed-to-AWS System. (Icons adquired from Internet resources)

An MDC Client can perform HTTP requests to interact with the system using the four CoAP REST verbs: GET, PUT, POST, DELETE. This HTTP Requests can be performed using the mdc-api-python Library [20]. In order for the Client to perform any of this requests, the must be configured at the device to allow them. The device doesn't perform the CoAP REST verbs like the Client does: the update, retrieval and notifications of a value are handled by the Mbed Client library. Furthermore, the device can have callback functions defined that stop the program momentarily to handle POST and PUT events from a client. The AWS Services on the other hand, interacts solely through the AWS IoT MQTT Broker by configuring AWS IoT Actions for messages coming to specific resources MQTT

Topics [1] .

4.1.1 Device Operation

- Retrieve Resource Data from MDC Server: The Device uses the *get_value_string* method of the M2M resource class, which returns the value stored on the server as a string.
- Update Resource Data on MDC Server: The Device uses the *set_value* method of the M2M resource object, which sends the data in the form of an array.
- Observe Resource Changes: A callback function must be implemented that gets called when the value is updated. It is then attached when the resource is being configured using the *set_value_updated_function* method of the M2MResource class. This function then retrieves the value as explained above. Note that the resource must be configured as observable for the other clients to be notified.
- Handling Requested Action: The Device can attach a callback function using the *set_execute_function* method of the M2MResource class. This function gets called when a POST operation is performed and the Device must then have to be configured to interpret the message and carry out the appropriate action.
- Command Client Action: The device can only communicate with the client by updating the value of a resource. Therefore to request the action of a client, the system must be preconfigured so that when a specific resource gets updated, the Client observes that change, interprets the message and takes appropriate action.
- Command Cloud Action: Similarly to that for the client, commanding an action to the Cloud requires the system to be preconfigured. The resource gets updated and an AWS IoT Action is configured for the specific *notify* MQTT Topic, that triggers other AWS Services and further actions.

4.1.2 MDC Client Operation

An MDC Client can perform the four CoAP REST verbs: GET, PUT, POST, DELETE. In order for the Client to perform any of this requests, it must be configured at the device to allow them.

- Retrieve Resource Data from MDC Server: The Client can retrieve the value from the MDC

[1] Note that, for a resource of an Mbed device connected to MDC, we refer to the MQTT Topic of a resource as one of those created by the Bridge connector, as explained in section 2.2.4.

Server using a GET HTTP Request. The GET request is asynchronous so the data is not retrieved from the request response, but from the *async* callback function.

- Update Resource Data on MDC Server:
- Observe Resource Changes: The Client can be automatically be notified when an observable resource is updated by using the *putResourceSubscription* function.
- Command Device Action: The Client can request the device to take an action by performing a POST HTTP Request. The message send in this operation does not modify the value in the MDC server: it is forwarded to the device, which interprets the command and takes the apropriate action.
- Command Cloud Action: When a PUT HTTP Request is send by an MDC Client, the rest of the MDC Clients don't get notified. Therefore, a client can not directly trigger the action of the Cloud via the MDC Server. A possible workaround would be to send the request to the device, that could then send the action request to the Cloud.

4.1.3 Cloud Operation

In order to access AWS Services, the Connector Bridge is used to route the Device's Resources from MDC to AWS IoT. Once it arrives to this service, the Mbed Device is seen by AWS as any other IoT Device that publishes and subscribes to MQTT Topics. Section 2.2.4 describes the operation of the Connector Bridge and the MQTT Topics of the resource.

- Retrieve Resource Data from MDC Server: The Cloud can publish to the resource's *get* MQTT Topic and receive the answer by subscribing to the resource's *cmd-response* MQTT Topic.
- Update Resource Data on MDC Server: The Cloud can receive the update the data by publishing to the resource's *put* MQTT Topic and formatting the payload of the message as described in Section 2.2.4.
- Observe Resource Changes: The Cloud can observe the changes to a resource by subscribing to the resource's *notify* MQTT Topic where the value is automatically published by the Connector Bridge when it changes.
- Command Device Action: The Cloud can request an action by publishing to the resource's *post* MQTT Topic and formatting the payload of the message as described in Section 2.2.4.
- Command Client Action: The same problem is encountered as that explained above in the Client's *Command Cloud Action* operation.

- Handling Requested Actions: AWS IoT Rules can be used to trigger AWS Actions. Nevertheless this actions are limited to a set of functionalities. To use any of the functionalities offered by other AWS Services, a Lambda Function can be triggered that uses the AWS SDKs. The message with attributes can be passed to the Lambda function, and be the input to the other AWS Service.

  - Cloud Computing: The Lambda function could be configured to take different actions depending on the value of the attribute, perform calculations or process the data. The IoT Rule can be configured to be triggered by any incoming message to a topic or to be triggered for a specific event: a message with an attribute meeting a defined condition for a specific Topic. The Lambda function can then access the full range of AWS Services, enabling powerful computational features.

    AWS ML can be triggered as explained above. The message incoming in the Topic must contain the inputs of the prediction as attributes (key pairs of a JSON string). The Lambda function extracts this attributes and uses one of the AWS SDKs to trigger an ML Real-time Prediction. The prediction value is retrieved and the Lambda can take further action to store the value or send it back to the device.

  - Storage: The Shadow of an IoT Device contains the values to the device's predefined attributes. In the case of an Mbed Device Bridged to AWS, the device can not publish to the Shadow Topic, and to update it, a more complex action has to be taken. The notification resource must trigger a Lambda function that extracts the value, and updates the Shadow either publishing to the Shadow Topic or using the AWS SDK. Alternatively, the DynamoDB AWS can be used to store resources in a table database. This Service can be triggered by AWS IoT Rules directly, passing the message with the attributes to be stored to this Service.

4.2 Smart Home Example Application

An example Application of a simple smart home was created to explore the operations mentioned above. To simulate a smart home, some resources where created to show some of the features that this specific IoT application would have. The device has some output devices such as an RGB LED to show the comfort zone of the room, and a status LED to show a connected application. Furthermore, there are temperature, humidity and pressure sensors, that measure the conditions in the room and motion sensors (accelerometer, gyroscope and magnetometer) that can simulate motion detection when the device is shaken. There is also a button that can trigger events and maintains a count of how many times it has been pressed. To simulate the POST Resquests, a timer recording the elapsed time can be controlled by a Client. A prediction resource triggers an AWS ML Model that calculates the comfort level which is send back to the device.

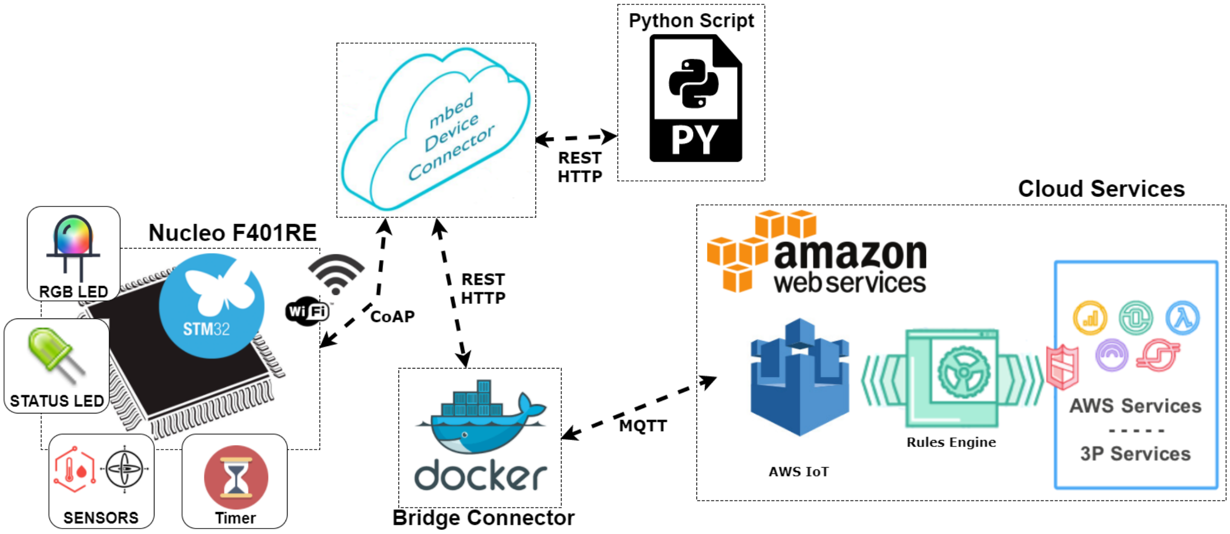


Figure 4.2: System Diagram for smart Home Application.(Icons adquired from Internet resources)

4.2.1 Endpoint Resources

There are two events at the Device in which resources are updated in the MDC Server: some resources are configured to update every 10 seconds and others by the button being pressed. When a Application performs a PUT and POST operation, this triggers a callback function that interrupts the operation of the program. All of the resources are set to observable, since this is a requirement for the Connector Bridge to function appropriately.

The following resources are available in this example application:

- LED Resource: Status (11/0/1)
  The Application initialises the value of the built-in LED to 0 (Off state) and thereafter the LED is only controlled by an MDC Application. Only PUT is allowed and the values that the Device can interpret are "ON", "OFF" or "TOGGLE"

- Button Resource: Count (12/0/1)
  The Application triggers an interrupt event every time the Button is pressed but also shares the number of times it has been pressed with the Cloud. Only GET is allowed, the number of times the button has been pressed is received as a string.

- RGB Resource: Color (13/0/1)
  The RGB resource is being updated by the device and the cloud. The application is changing through the different RGB states and updating the current color to the cloud. every 10 seconds when no prediction has been received. When a prediction arrives, the RGB lights red it the

comfort value is too high, blue if its too low and green if its acceptable. When pressing the button, the RGB LED changes to white for 10 seconds. At any time, the Application can change the state of the RGB.
The values that can be GET from the Resource or PUT to the Resource are: "RED", "GREEN", "BLUE", "WHITE" and "OFF".

- Sense Resource: Sensors (14/0/1 to 14/0/6)
  The values of the sensor shield get updated every 10 seconds, but only the Temperature and the Humidity get updated in the MDC Server. When the button is pressed all of the sensor values are updated to MDC Server. Only the GET operation is allowed in these Resources.

- Sense Resource: Motion (14/0/7)
  There is a separate thread reading the accelerometer every 100 ms and detects whether the device has been shaken or not. Every 10 seconds, the flag gets checked and the value of this resource is updated to "true" if there has been motion in the last 10 seconds or to "false" if there hasn't. AWS IoT has been configured to listen to this resource, and when the value is true it is configured to automatically send an email to notify that the device has been moved.

- Timer Resource: Elapsed Time (15/0/1)
  The Timer gets initialised when the application starts. The Application updates the value of time elapsed everytime the Button is pressed. The Application can command the Device to update the time elapsed by performing the POST operation and writing "UPDATE". The Timer can be reset to zero by writing "RESET" in the POST operation.

- Prediction Resource: Current conditions (16/0/1)
  The Application requests a prediction from AWS every time the button has been pressed. The temperature and humidity values are sent as a string in JSON format. When this arrives at AWS IoT through the Bridge Connector, a Lambda function is triggered that extract the temperature and humidity data and performs an AWS ML real-time prediction. The ML model used is described in Section 4.2.4

- Prediction Resource: ASH Value (16/0/2)
  The result of the prediction performed in the Lambda function is retrieved and send to this resource. It does so by writing to the specific topic with the specific payload to perform a PUT operation. A callback function is triggered at the device, and the application updates the ASH value in the device, and updates the comfort level variable. Depending on the comfort level, the RGB lights a different color. Table 4.1 shows the ranges of the ASH value that define each comfort level and the RGB LED color that corresponds to it.

Table 4.1: value of comfort variable depending on ASH value

ASH<-1.5	-1.5≥ASH≤1.5	ASH>1.5	
comfort	cold	ok	hot
RGB color	blue	green	red

4.2.2 Application

An Example MDC Application is created to interact with the Endpoint. This is an example on the functionalities that any application or device with Internet connection and able to perform HTTP Requests, can interact with an endpoint.

Observe Resources

The Application can be notified every time a resource value is updated by the device. To do so, it must use long polling and add the resource to the notification handler, as explained in Section X. Alternatively, the Application can perform a GET operation on the resource to retrieve the information. Note that to receive notifications the resource must be observable and be GET enabled to perform this operation. Since all resources of the Smart Home are observable, the application gets a notification every time any of them is modified.

Example Actions

As an example, The application lights up the status led whenever it first starts running. It also switches off the RGB led periodically, and when it receives a notification that motion is detected, a POST operation is send to the timer resource to request the device to carry out a reset action.

4.2.3 Cloud Services

In order to increase the functionalities of the system, some AWS Services where used to carry out more complex functions than just storing and retrieving the last value of a resource. Figure 4.3 shows the three example actions taken at the Cloud. AWS IoT Rules have been configured for the *notify* MQTT messages of the Motion (14/0/7), Current conditions (16/0/1) and Temperature Sensor (14/0/6) resources.

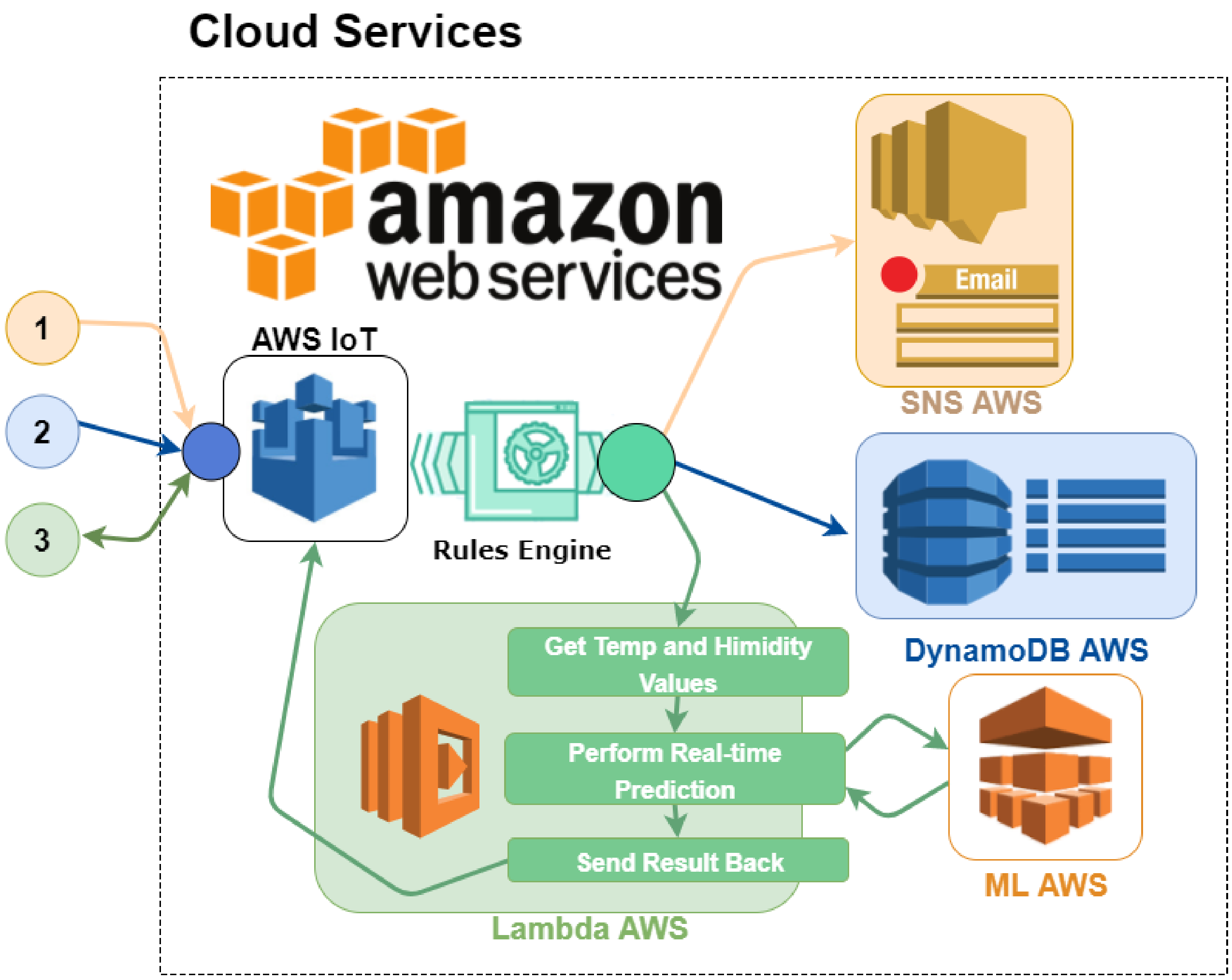


Figure 4.3: Graphical representation of Smart Home Cloud Services Actions

1. Motion Action: When motion is detected the resource is set to *true*. The AWS IoT Rule can be configured to filter the incoming message to trigger the SNS AWS service when the value is equal to *true*. The SNS AWS Service can then be used to send an email to a specific email address, and alert that the device has been moved.

2. Temperature store Action: When the device updates the temperature, an AWS Action can be set that triggers the AWS DynamoDB. This service stores values into tables, including the timestamp when the value arrived

3. Prediction Action: The device sends the temperature and humidity a JSON string to Current conditions (16/0/1) resource. The AWS IoT Rule is configured to trigger a Lambda AWS function. This function reads the values from the JSON string and carries out a Real-time ML prediction using the Python AWS SDK (Boto3). The result of the prediction is then returned to the device by publishing to the *put* MQTT Topic of the ASH Value (16/0/2) resource.

4.2.4 Thermal Comfort ML Model

The AWS ML Service was used to train an ML Model that could predict the Thermal comfort in a room. To do so, the service must be provided with a file containing all of the fields for the training. From the data set of the ASHRAE Project [35] 47 different Comma Delimmited Files *(CSV)* where downloaded. This data had to be combined and then processed to extract only the necessary fields. Most of the fields are not relevant for the model, since they are parameters that the system cannot measure (such as metabolic rate, user dissatisfaction or number of windows in the room). ML AWS provides some statistical insights about the datasets, such as the distribution of values and invalid parameters. The AWS ML Service can then be used to train and evaluate the system, as well as test the predictions by inputting some data into the ML model.

Extracting data from database

The three fields of interest for the ML model are:

- *taav*: Average indoor temperature.
- *rh*: relative humidity.
- *ash*: ASHRAE Thermal Sensation (-3, +3)

The steps taken to get a data set with which to train the model are:

- Merge the 48 different csv files.
- Add an extra column containing the Climate Zone where the study was carried.
- Select only the fields that are relevant for the prediction.
- Clean the data set (remove empty fields).

The data set used can be found in the Project Github [31].

AWS ML Evaluation Results

AWS ML Model scores the ML Model to be better than the baseline. Root Mean Square Error (RMSE) equals 0.9302, which measures the deviation of the model from the real values. The lower the score the better the ML model is considered to be. The ML model performance is shown in Figure 4.4. From the results, it can be observed that the ML model tends to predict the *ash* value to be lower than it should. Nevertheless, no further improvements were carried out since the goal was to provide an example cloud computing application, not the actual accuracy of the results.

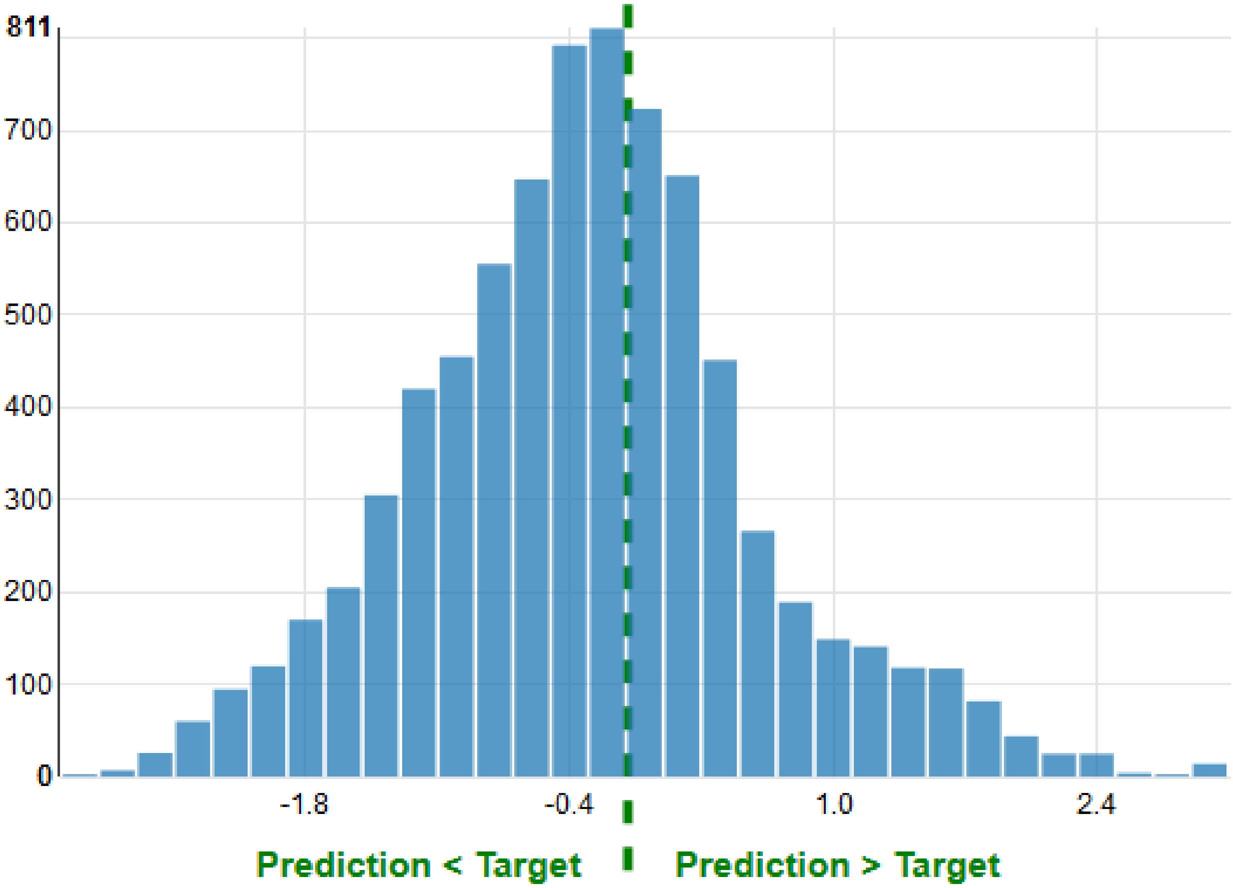


Figure 4.4: Screen-shot taken from the Evaluation performed by AWS on the ML Model in the AWS ML Console. It shows the number of values that presented the inaccuracy shown in the horizontal axis.

Chapter 5

Conclusion

After having explored the different Platforms and designed implementations for the proposed system in Chapter 3, some important remarks can be made that can aid future prototyping of IoT Systems. It was identified that the implementation of the system and its behaviour is heavily dependent on the Platform used since each Platforms adds each own series of constrains such as supported devices and protocols to use.

Path II is the method best supported by Mbed, since the Mbed Client Library handles the registering and communication of the device seamlessly. Even though any Mbed Enabled device can be used it was identified that selecting a device that had could support one of the network hardware modules supported by Mbed largely accelerated the implementation process. This is because the appropriate libraries to handle the connection to the networks have already been implemented for supported modules and they have been tested extensively with some specific boards. For this reason, the ST Nucleo board is used throughout the project, since STMicroelectronics offer a range of Shields that can fit into a wide range of microcontrollers. This is important during the prototyping phase, since a developer can choose the microcontroller with the processor that best suits its computational power and efficiency needs.

Path I offers a very interesting implementation for IoT systems, since it uses a non-IP protocol (BLE) and a Gateway device. This topology is popular in IoT applications such as smart cities where it is more efficient and cost-effective to have hundreds of low-power devices hidden in the environment that communicate with a more computationally powerful device that handles the traffic with the Cloud. In terms of the actual implementation of this system, the DeviceLink software in the Raspberry Pi accelerates the prototyping for a proof of concept design. Nevertheless, to model real life applications, the Gateway would have to automatically connect the device to the internet without needing to manually configure each device. Furthermore, using the BLE Protocol has clear limitations for IoT applications, since its a one-to-one connection between the device and the Gateway. On the other hand, the BLE Protocol was found to be very well supported by Mbed OS, with a lot of sample applications and extensive documentation.

Path III uses the MQTT Protocol to connect directly to AWS. Even though it seems like this approach clearly makes the other two redundant, there are a number of reasons why that is not the case. Firstly, even though the Software Libraries being used is compatible with Mbed devices, the implementation is using a previous version of the Mbed OS, it does not use the latest software library for the Wifi module and the sample code has currently only been tested with the Nucleo F401RE board. Furthermore, when implementing the system complications where experienced trying to connect the device to AWS IoT. Nevertheless, it clearly proofs that a connection of an Mbed enabled device to AWS IoT is possible and was an interesting solution to compare this path with the other two. An essential feature that is missing in MDC compared to AWS IoT, is the fact that two devices connected to AWS IoT can easily communicate with one another, while on MDC there is currently no way of doing this.

5.1 Future Work

As Future Work, it is proposed to try to improve the performance of the system by substituting the way the messages flow from MDC to the Connector Bridge. Using a web-hook instead of long polling could significantly improve the performance. To expand the performance analysis of the system, the power consumption of the different Paths could be measured using a power monitoring shield such as the STM32 Power Shield [32]. Furthermore, the rest of the Network Interface solutions available on Mbed OS (such as Cellular, Ethernet and Mesh) can be implemented and compared.

In terms of continuing with other IoT use cases, it is proposed to move into areas of IoT that explore the constrains of the system. For example, extreme low-power applications using technologies such as NFC or systems with high data traffic demands, such as performing image processing on the Cloud. For the latter, the block-wise messages feature of Mbed Client could be explored, which offers a method for handling a set of large concatenated messages.

Appendix A

Experiment Sample Data

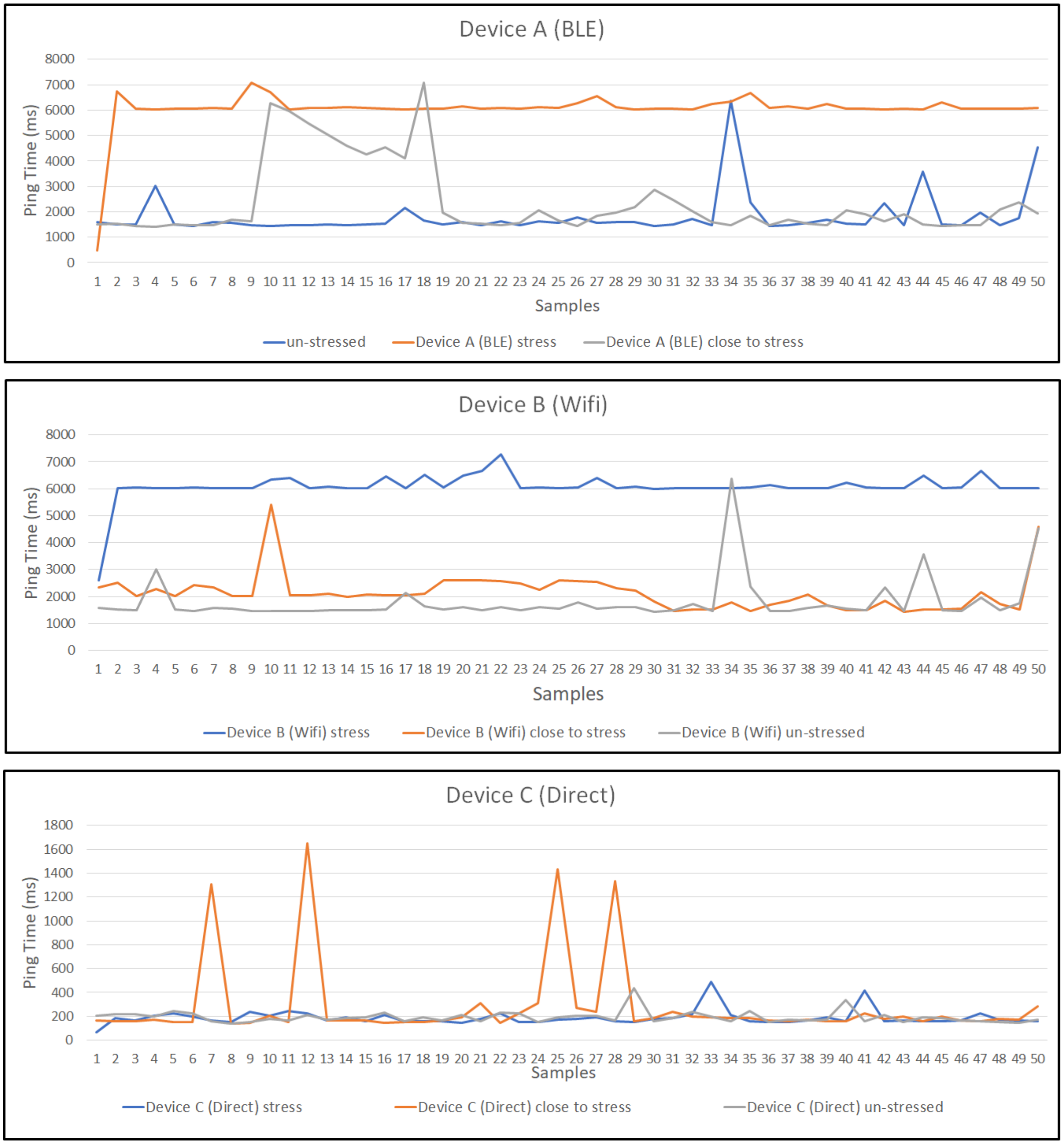


Figure A.1: All 50 samples for the three system states during experiment 1.

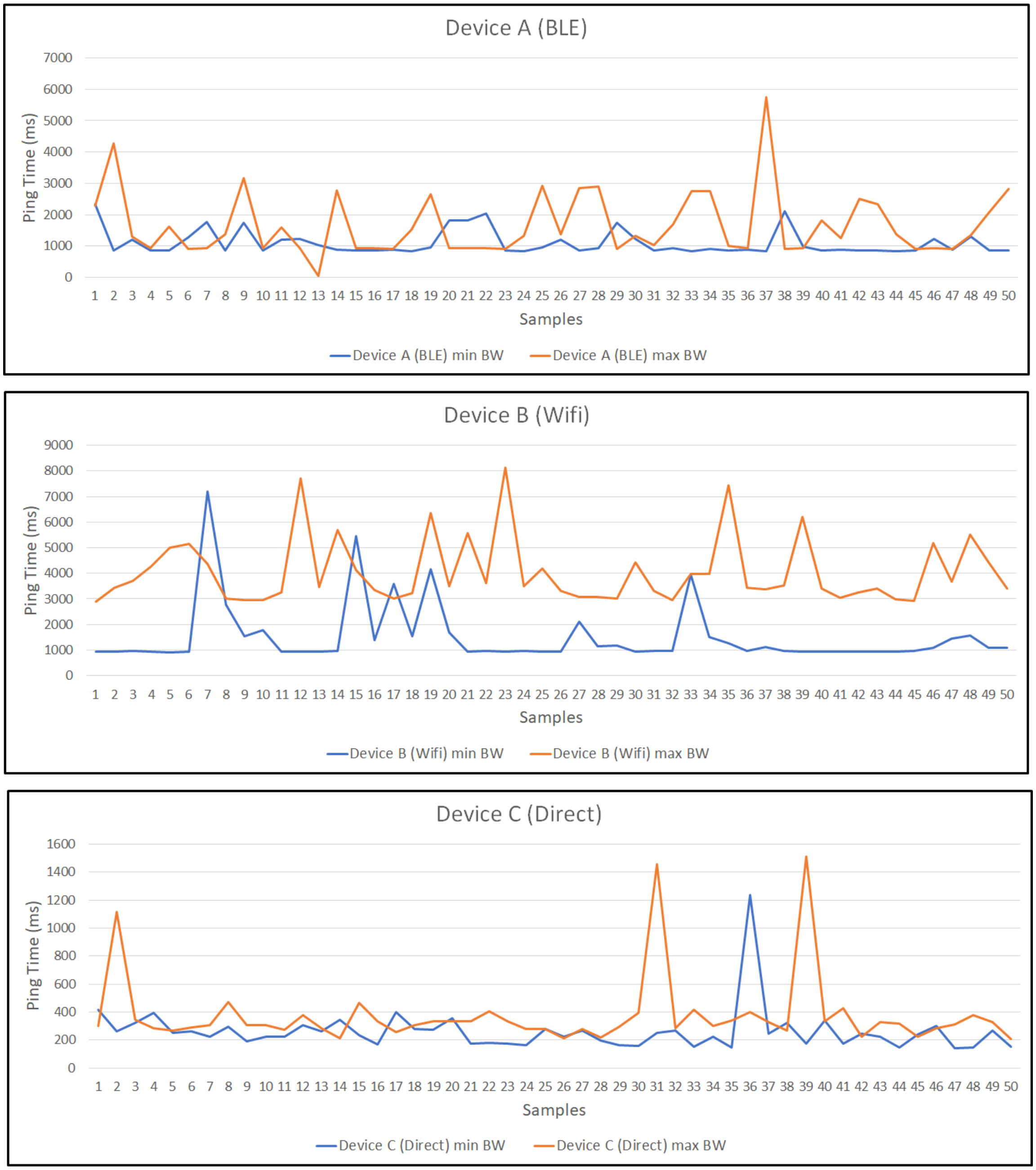


Figure A.2: All 50 samples for maximum and minimum data packages during experiment 2.

Bibliography

[1] Ahmed Banafa. Iot standardization and implementation challenges newsletter. Available at https://iot.ieee.org/newsletter/july-2016/iot-standardization-and-implementation-challenges.html (2017/12/13).

[2] C. Bormann, A. P. Castellani, and Z. Shelby. Coap: An application protocol for billions of tiny internet nodes. *IEEE Internet Computing*, 16(2):62–67, March 2012.

[3] Richard de Dear, Gail Brager, and Cooper D. Developing an adaptive model of thermal comfort and preference - final report on rp-884., 01 1997.

[4] W. Lv, F. Meng, C. Zhang, Y. Lv, N. Cao, and J. Jiang. A general architecture of iot system. In *2017 IEEE International Conference on Computational Science and Engineering (CSE) and IEEE International Conference on Embedded and Ubiquitous Computing (EUC)*, volume 1, pages 659–664, July 2017.

[5] Amazon Web Services. Amazon web services: Services offered main page. Available at https://aws.amazon.com/ (2017/12/12).

[6] Amazon Web Services. Aws iot documentation. Available at https://aws.amazon.com/documentation/ (2017/12/22).

[7] Amazon Web Services. Aws iot documentation: Aws iot actions. Available at http://docs.aws.amazon.com/iot/latest/developerguide/iot-rule-actions.html (2017/12/22).

[8] Amazon Web Services. Aws iot documentation: Aws iot shadows. Available at http://docs.aws.amazon.com/iot/latest/developerguide/iot-thing-shadows.html (2017/12/22).

[9] Amazon Web Services. Aws iot documentation: Aws labs. Available at https://qwiklabs.com/ (2017/12/22).

[10] Amazon Web Services. Aws iot documentation: Rules for aws iot. Available at `http://docs.aws.amazon.com/iot/latest/developerguide/iot-rules.html` (2017/12/12).

[11] Arm Holdings. Arm mbed github. Available at `https://github.com/ARMmbed/mbed-os` (2017/12/12).

[12] Arm Holdings. Introduction to mbed device connector (web page). Available at `https://docs.mbed.com/docs/getting-started-with-mbed-device-connector/en/latest/Connector-intro/` (2017/12/12).

[13] Arm Holdings. Mbed cloud developers website. Available at `https://cloud.mbed.com/` (2017/12/12).

[14] Arm Holdings. Mbed os developers website. Available at `https://os.mbed.com/` (2017/12/12).

[15] Arm Holdings. Mbed os product description. Available at `https://www.mbed.com/en/development/mbed-os/` (2017/12/12).

[16] Inc. Gartner. Top trends in the gartner hype cycle for emerging technologies, 2017. Available at `https://www.gartner.com/smarterwithgartner/top-trends-in-the-gartner-hype-cycle-for-emerging-technologies-2017/` (2017/12/13).

[17] IntâĂŹl Telecommunication Union ITU. Itu internet reports 2005: The internet of things. Available at `https://www.itu.int/net/wsis/tunis/newsroom/stats/The-Internet-of-Things-2005.pdf` (2017/12/13).

[18] Jan Jongboom. Blog post: Connecting ble devices to the cloud. Available at `https://os.mbed.com/blog/entry/Connecting-BLE-devices-to-the-cloud/` (2017/12/12).

[19] Klika-Tech Team. Sample code to connect to aws iot using the stm32 nucleo board. Available at `https://docs.mbed.com/docs/mbed-connector-api-python/en/latest/` (2017/12/13).

[20] Mbed Team. Api documentation for the mbed-connector-api-python library. Available at `https://docs.mbed.com/docs/mbed-connector-api-python/en/latest/` (2017/12/13).

[21] Mbed Team. Example implementation to connect a linux machine to mbed device connector. Available at `https://github.com/ARMmbed/mbed-client-linux-example` (2017/12/13).

[22] Mbed Team. Example implementation to connect an mbed device to mbed device connector. Available at https://github.com/ARMmbed/mbed-os-example-client (2017/12/13).

[23] Mbed Team. Library that simplifies the use of network interfaces for specific hardware modules. Available at https://github.com/ARMmbed/easy-connect (2017/12/13).

[24] Mbed Team. Mbed developer reference page: St nucleo-f401re development board. Available at https://os.mbed.com/platforms/ST-Nucleo-F401RE/ (2017/12/13).

[25] Mbed Team. Mbed developer reference page: St x-nucleo-idb05a1 bluetooth low energy expansion board. Available at https://os.mbed.com/components/X-NUCLEO-IDB05A1-Bluetooth-Low-Energy/ (2017/12/13).

[26] Mbed Team. Mbed developer reference page: St x-nucleo-idw01m1 wi-fi expansion board. Available at https://os.mbed.com/components/X-NUCLEO-IDW01M1/ (2017/12/13).

[27] Mbed Team. Mbed developer reference page: St x-nucleo-iks01a1 mems and environment sensors. Available at https://os.mbed.com/components/X-NUCLEO-IKS01A1/ (2017/12/13).

[28] Mbed Team. Mbed device connector webpage. Available at https://connector.mbed.com/ (2017/12/13).

[29] Mbed Team. mdc-api-python docs. Available at https://docs.mbed.com/docs/mbed-connector-api-python/en/latest/ (2017/12/13).

[30] Microsoft. The osi model's seven layers (microsoft support page). Available at https://support.microsoft.com/en-gb/help/103884/the-osi-model-s-seven-layers-defined-and-functions-explained (2017/12/11).

[31] Sergio Martin. Github repository containing the project. Available at https://github.com/smysergio (2017/12/13).

[32] STMicroelectronics. Stm32 power shield (x-nucleo-lpm01a) product webpage. Available at http://www.st.com/en/evaluation-tools/stm32-nucleo-expansion-boards.html?querycriteria=productId=SC1971 (2017/12/13).

[33] Pallavi Sethi and Smruti R. Sarangi. Internet of things: Architectures, protocols, and applications. 2017:1–25, 01 2017.

[34] OASIS Open Standards. Mqtt website of oasis open standard. Available at `https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=mqtt` (2017/12/11).

[35] Design & Planning. The University of Sydney School of Architecture. The adaptive model of thermal comfort: Ashrae rp-884 project (webpage). Available at `http://coap.technology/` (2017/12/11).

[36] K. Hartke Z. Shelby, Arm Holdings and C. Bormann. The Constrained Application Protocol (CoAP). RFC 7252, Internet Engineering Task Force (IETF), June 2014.

Glossary

IOT: Internet of Things

AWS: Amazon Web Services

IaaS: Infrastructure as a Service

MDC: Mbed Device Connector.

ML: Machine Learning

ASHRAE: American Society of Heating, Refrigerating and Air-Conditioning Engineers

GPIO: General Purpose Input Output

PCB: Printed Circuit Board

OS: Operating System

HAL: Hardware Abstraction Layer

LED: Light Emitting Diode

HTTP: Hypertext Transfer Protocol

CoAP: Constrained Applications Protocol.

I2C: Inter-integrated Circuit

UART: Universal asynchronous receiver-transmitter

SPI: Serial Peripheral Interface

LWM2M: Lightweight Machine 2 Machine

API: Application Programming Interface

RTOS: Real Time Operating System

CMSIS: Cortex Microcontroller Software Interface Standard

REST: Representational State Transfer

SDK: Software Development Kit

MQTT: Message Queuing Telemetry Transport

GATT: Generic Attributes

GAP: Generic Access Profile

URL: Uniform Resource Locator