UNIVERSIDADE ESTADUAL DE CAMPINAS

Faculdade de Engenharia Elétrica e de Computação

JORGE BENAVIDES ASPIAZU

MIDDLEWARE DESIGN FOR APPLICATION INTEGRATION IN IOT NETWORKS

PROJETO DE MIDDLEWARE PARA INTEGRAÇÃO DE APLICAÇÕES EM REDES IOT

CAMPINAS

2020

JORGE BENAVIDES ASPIAZU

MIDDLEWARE DESIGN FOR APPLICATION INTEGRATION IN IOT NETWORKS

PROJETO DE MIDDLEWARE ARA INTEGRAÇÃO DE APLICAÇÕES EM REDES IOT

Dissertation presented to the School of Electrical and Computer Engineering of the University of Campinas in partial fulfillment of the requirements for the degree of Master in Electrical Engineering, in the area of Telecommunications and Telematics.

Dissertação apresentada à Faculdade de Engenharia Elétrica e de Computação da Universidade Estadual de Campinas como parte dos requisitos exigidos para a obtenção do título de Mestre em Engenharia Elétrica, na área de Telecomunicações e Telemática.

Supervisor/Orientador: DR. HUGO ENRIQUE HERNANDEZ FIGUEROA

Este exemplar corresponde à versão final da dissertação defendida pelo aluno Jorge Benavides Aspiazu, orientada pelo(a) Prof(a). Dr(a). Hugo Enrique Hernandez Figueroa.

CAMPINAS

2020

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca da Área de Engenharia e Arquitetura
Luciana Pietrosanto Milla - CRB 8/8129

B431m Benavides Aspiazu, Jorge, 1984-
Middleware design for application integration in IoT networks / Jorge Benavides Aspiazu. – Campinas, SP : [s.n.], 2020.

Orientador: Hugo Enrique Hernández-Figueroa.
Dissertação (mestrado) – Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação.

1. Internet das coisas. 2. Sistemas embarcados. 3. Software. 4. Aplicação. 5. TCP/IP (Protocolo de rede de computador). I. Hernández-Figueroa, Hugo Enrique, 1959-. II. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. III. Título.

Informações para Biblioteca Digital

Título em outro idioma: Projeto de middleware para integração de aplicações em redes IoT
Palavras-chave em inglês:
Internet of things
Embedded systems
Software
Application
TCP/IP (Computer network protocol)
Área de concentração: Telecomunicações e Telemática
Titulação: Mestre em Engenharia Elétrica
Banca examinadora:
Hugo Enrique Hernández-Figueroa [Orientador]
Luis Geraldo Meloni
Carlos Silva Cardenas
Data de defesa: 28-01-2020
Programa de Pós-Graduação: Engenharia Elétrica

Identificação e informações acadêmicas do(a) aluno(a)
- ORCID do autor: https://orcid.org/0000-0003-1630-3615
- Currículo Lattes do autor: https://wwws.cnpq.br/cvlattesweb/PKG_MENU.men

COMISSÃO JULGADORA – DISSERTAÇÃO DE MESTRADO

Candidato(a): Jorge Benavides Aspiazu RA: 108812

Data da defesa: 28 de janeiro de 2020

Título da Tese: "Middleware Design for Application Integration in IoT Networks"

Prof. Dr. Hugo Enrique Hernandez Figueroa (Presidente)

Prof. Dr. Luis Geraldo Pedroso Meloni

Prof. Dr. Carlos Bernardino Silva Cárdenas

A Ata de Defesa, com as respectivas assinaturas dos membros da Comissão Julgadora, encontra-se no SIGA (Sistema de Fluxo de Dissertação/Tese) e na Secretaria de Pós-Graduação da Faculdade de Engenharia Elétrica e de Computação.

ABSTRACT

The internet of Things (IoT) is a field with increasing development due to its effect in all aspects of society. Within this context the amount of data and devices that are integrated into this new internet is exploding and will continue to do so in the foreseeable future. One of the main and prevalent challenges to the IoT is the ease of integration be-tween applications and the general infrastructure underneath; namely a simple way of allowing horizontal integration of devices and applications. The most important challenge to accomplish this is the fact that the models for sensors and actuators that face applications are heterogenous, which means that not all sensors provide data in the same for-mat and not all actuators are operated in the same way. From an application's point of view this makes any kind of integration an important challenge. Another pain point for integration is the fact that communication protocols used by many devices are many times different, which means that any application hoping to integrate different type devices needs to make sure it can work with many protocols, which increases the complexity required. From these challenges the appearance of a middle component that facilitates the interaction of devices and applications is necessary. The work presented here pro-poses a solution to these integration problems by means of a middleware as an intermediate layer that bridges applications and devices in a transparent way, which in turn al-lows real-time data collection and historic data analysis. To achieve this, we propose the use of a virtual device model that presents a simpler version to the application, while ex-tending its capabilities, such as memory and processing power. Another effect of this virtual device model definition is that it creates a homogenous model that allows the search for devices within the middleware based on specific attributes; for example, it is possible to search for all temperature sensors, irrespective of any other characteristics. All these allow any application to integrate into any IoT network without prior knowledge of the specific devices that exist in it, which fosters greater integration.
Key Words: Internet of Things. Embedded Systems. Software. Application. TCP/IP.

RESUMO

O Internet das Coisas (IoT) é um campo em franco desenvolvimento devido ao seu efeito em todos os aspectos da sociedade. Dentro deste contexto, a quantidade de dados e dispositivos que são integrados neste novo internet está explodindo e continuará desse jeito no futuro previsível. Um dos desafios principais para o IoT é uma integração fácil e simples entre aplicações e a infraestrutura geral: dispositivos, comunicações etc.; em outras palavras, um jeito simples de conseguir uma integração horizontal dos dispositivos e aplicações. Um dos pontos mais importantes de resolver para conseguir essa integração horizontal é o fato que os modelos para os sensores e atuadores que interagem com as aplicações não são heterogêneos, isso quer dizer que não todos os dados que os sensores proveem existem no mesmo formato, e não todos os atuadores trabalham do mesmo jeito. Desde o ponto de vista da aplicação, esse tipo de integração é um desafio grande. Além disso, os protocolos de comunicação utilizados por muitos dispositivos são muitas vezes muito diferentes, o que significa que qualquer aplicação que tente integrar diferentes dispositivos, precisa segurar que pode trabalhar com múltiplos protocolos, o que aumenta a complexidade necessária. Devido a estes desafios, é necessário um componente intermediário que que ajude na interação dos dispositivos e aplicações.

O trabalho apresentado propõe uma solução para estes problemas de integração por meio de um Middleware como uma camada intermediaria que faça uma conexão entre aplicações e dispositivos de um jeito transparente para as aplicações, que por sua vez permita coletar dados em tempo real, assim como permitir análise de dados históricos: “Big Data”. Para conseguir isso, propomos o uso de um modelo virtual de dispositivo que apresente uma versão mais simples para a aplicação, enquanto estende as suas capacidades, tal como a memória e o poder de processamento. Um outro efeito desta definição de modelo virtual de dispositivo é que cria um modelo homogêneo que permite uma pesquisa simples de dispositivos dentro do Middleware, baseado em atributos específicos; por exemplo, é possível procurar todos os sensores de temperatura, independentemente de outras características. Tudo isso permite a qualquer aplicação se integrar dentro de qualquer rede IoT sem ter um conhecimento anterior dos dispositivos específicos que existem nesta rede, o que ajuda a promover grandemente a integração.

Palavras-chave: Internet das Coisas. Sistemas Embarcados. Software. Aplicação. TCP/IP.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

AAL	Ambient Assisted Living
ABE	Attribute-Based Encryption
AES	Advanced Encryption Standard
AMQP	Advanced Message Queueing Protocol
API	Application Programming Interface
AWS	Amazon Web Services
CA	Certificate Authority
CDF	Cumulative Distribution Function
CP-ABE	Ciphertext-Policy Attribute-Based Encryption
CoAP	Constrained Application Protocol
DNS	Domain Name Server
ETSI	European Telecommunications Standards Institute
HTTP	Hypertext Transfer Protocol
IEEE	Institute of Electrical and Electronics Engineers
IP	Internet Protocol
IPv4	Internet Protocol Version 4
IPv6	Internet Protocol Version 6
IoT	Internet of Things
JSON	JavaScript Object Notation
KP-ABE	Key-Policy Attribute-Based Encryption
LED	Light Emitting Diode
LTE	Long-Term Evolution
M2M	Machine to Machine
mDNS	Multicast Domain Name Server
MOM	Message Oriented Middleware
MQTT	Message Queue Telemetry Transport
NoSQL	Non-Structured Query Language
PIR	Passive Infrared Sensor

QoS	Quality of Service
REST	Representational State Transfer
SoC	System on Chip
SSL	Secure Sockets Layer
TCP	Transfer Control Protocol
TCP/IP	Transfer Control Protocol/Internet Protocol
TLS	Transport Layer Security
UDP	User Datagram Protocol
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
VPS	Virtual Private Server
XML	Extensible Markup Language

CONTENTS

1 INTRODUCTION

The Internet of Things (IoT) is an increasingly developed field because of its direct impact to every aspect of society. Several critical aspects regarding how the IoT operates are being researched currently such as Machine to Machine (M2M) Communication, ap-plication deployment, data organization and many more(Evans, 2011)(Al-Fuqaha, Guizani, et al., 2015).

It is often assumed that the IoT should work as the regular internet currently works, how-ever, due to details like low power, battery operated devices, many of which use different type of radio standards for communication such as WiFi, IEEE802.15.4, ZigBee, etc. a direct implementation for these is extremely difficult and complex. Also, most devices within a typical IoT application have different requirements; for instance, sensors usually only communicate to the cloud (sending data) and rarely do they require to receive data from the cloud, in that regard working differently as how clients within the regular internet operate(Al-Fuqaha, Guizani, et al., 2015).

To complicate matters further due to differences in type of devices being connect-ed, how they operate, the type of data that they send or receive and the M2M communication scheme they use, it becomes virtually impossible in today's landscape to deploy applications that target the same IoT networks and that are written by different people.

One of the greatest drivers of our current form of internet is its ability to interconnect different applications through a common language. This is possible, because of a single topology: client-server, and that prior knowledge of outside systems (such as complementary services of third parties) only require reading the methods that are presented by such system. In turn, this allows complex systems to be built on top of others with each of them not being necessarily developed by the same entity(Al-Fuqaha, Khreishah, et al., 2015).

Because of these reasons, the IoT application landscape remains heavily fragmented, with different applications being unable to be reused and services being tied directly to a single application (Al-Fuqaha, Khreishah, et al., 2015)(Yokotani & Sasaki, 2016). Most applications tend to follow a vertical approach which makes it extremely difficult to integrate with others.

To allow such interconnectivity to happen there should be a method that allows any IoT application to be able to communicate easily and reliably to any network and gather only the data it requires from two specific sources: the individual devices that generate such data in real time (if the application requires or supports real time data generation) and from a central repository that contains data gathered through an undetermined period.

This method must consider that a typical IoT deployment consists of devices that likely contain low memory capacity as well as low processing power, due to their low energy consumption requirements. This is considered in conjunction with the fact that the greatest benefit of using an IoT solution is the accumulation of data over time, which these devices probably are unable to do individually.

Finally, since the data generated from any IoT application tends to be really diverse, be-cause of the different types of sensors that could be used and the data of interest that could be desired to be recorded, such method has to be able to handle that any third party application be flexible enough to use only the data of interest within a universe of different types of data gathered(Al-Fuqaha, Guizani, et al., 2015).

The present work proposes a method that attempts to ease the application deployment in IoT networks through the use a simple data arrangement that allows the use of only the parameters relevant to the application.

Furthermore, consideration will be taken regarding the different capabilities that IoT de-vices might have and how that plays into the needs of any application that can be deployed along the network. Those needs include the use of historic data as well as real time data. One final consideration will be the volume of information and interconnection of different networks within one application.

1.1 MOTIVATION

The current landscape of IoT solutions show a complete fragmentation, making any project deployment extremely difficult and costly. Even worse, reusing existing infrastructure, while only deploying new applications on devices and data already in place, currently requires heavy customization in the application’s software, and, most often than not, modifications on the devices to allow full integration with the application.

Another important issue is the tools being used in current IoT applications are the same as the ones used in regular software application deployment. This poses a problem, in the sense that these tools were not designed for the heterogenous and resource-constrained environment of any IoT application. Adapting these tools to work on an IoT environment is a complex and difficult task, which decreases the incentives to adopt and implement IoT solutions.

From a personal point of view, my previous work, both in an academic as well as in an industry setting, have made me realize the difficulties in adopting IoT solutions with the existing tools and technologies. In order to advance and increase adoption of IoT solutions, and create a more interconnected world, a new set of tools are required. This work is an attempt at creating a new tool for this specific purpose.

1.2 DEFINITION AND GROWTH EXPECTATIONS

The Internet of Things is a concept in which everyday objects are given processing capabilities as well as connectivity so they can communicate with each other and the internet as a whole (Evans, 2011). The idea is that data shared between these objects allows for an increased productivity in human activities and greater comfort for all society.

One of the biggest drivers of research into the IoT is the fact that its growth is expected to continue in an exponential way, in 2010 the number of connected devices has surpassed the earth's human population (Evans, 2011)(Al-Fuqaha, Guizani, et al., 2015). In 2020, it is expected that the number of IoT objects will reach 212 billion deployed globally (Al-Fuqaha, Guizani, et al., 2015). Five years after that it is expected that traffic from such devices will account up to 45% of the total internet traffic (Al-Fuqaha, Guizani, et al., 2015)(Taylor, 2013).

On top of that it is estimated that IoT applications will affect several different markets as vertical applications; it is expected that these applications will interconnect with each other in the future, bringing even greater benefits to its deployment. This integration is also expected to be horizontal, meaning that computing services and third-party applications could be deployed through any segment, or all segments in an independent manner (Al-Fuqaha, Guizani, et al., 2015). Figure 1.1 shows the different vertical markets and how a possible horizontal integration might be possible.

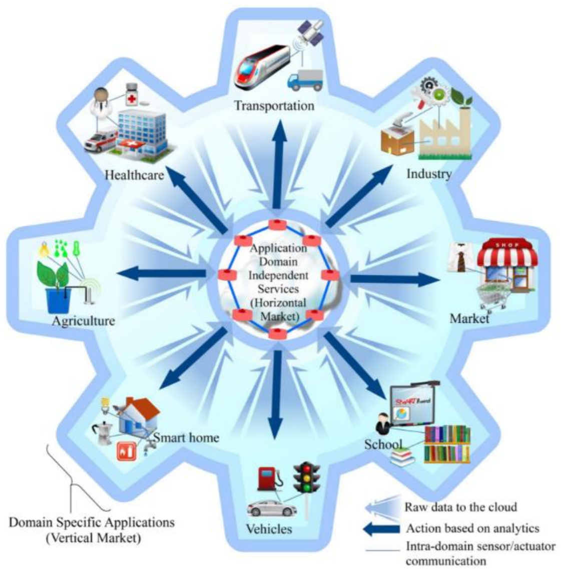

Transportation
Industry
Market
School
Vehicles
Smart home
Agriculture
Healthcare
Application Domain Independent Services (Horizontal Market)
Domain Specific Applications (Vertical Market)
Raw data to the cloud
Action based on analytics
Intra-domain sensor/actuator communication

Figure 1.1– Different vertical markets where IoT is expected to deploy and its horizontal integration (Al-Fuqaha, Guizani, et al., 2015)

1.3 ARCHITECTURE

The current state of deployment uses a vertical integration, where all elements of the system are designed in a tightly integrated fashion, this helps create a really optimized system, with the application being tailored exactly to the requirements for that individual deployment. The problem with this approach is that the whole system must be developed individually one at a time, with little chance of third-party application or services being capable of integrating to it (Yokotani & Sasaki, 2016).

In contrast to this, a horizontal implementation, the middle layers of the system are common to any implementation and are treated as a shared resource. This allows for the applications that are written on top to be able to access any system implemented, specially the sensor data. This in turn allows for a much wider and easier third-party integration (Yokotani & Sasaki, 2016). Both approaches are shown in figure 1.2.

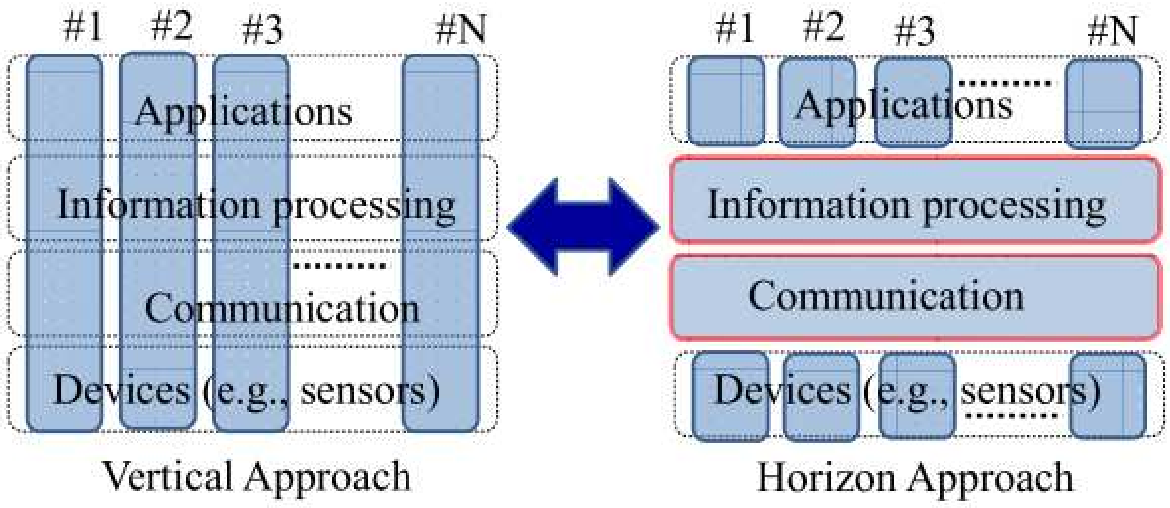


Figure 1.2 – Vertical and Horizontal approach for an IoT deployment. (Yokotani & Sasaki, 2016)

Due to this potential interest in several markets and research of new technologies regarding the Internet of Things are being carried out by many individuals and organizations. On this regard healthcare applications related to IoT, including services in the same area are expected to create about \$1.1-\$2.5 trillion in growth annually by the global economy by 2025, with the whole economic impact caused by the IoT is estimated to be in range of \$2.7 trillion to \$6.2 trillion by the same year (Al-Fuqaha, Guizani, et al., 2015). Figure 1.3 shows the market share of the of the projected dominant IoT applications by 2025.

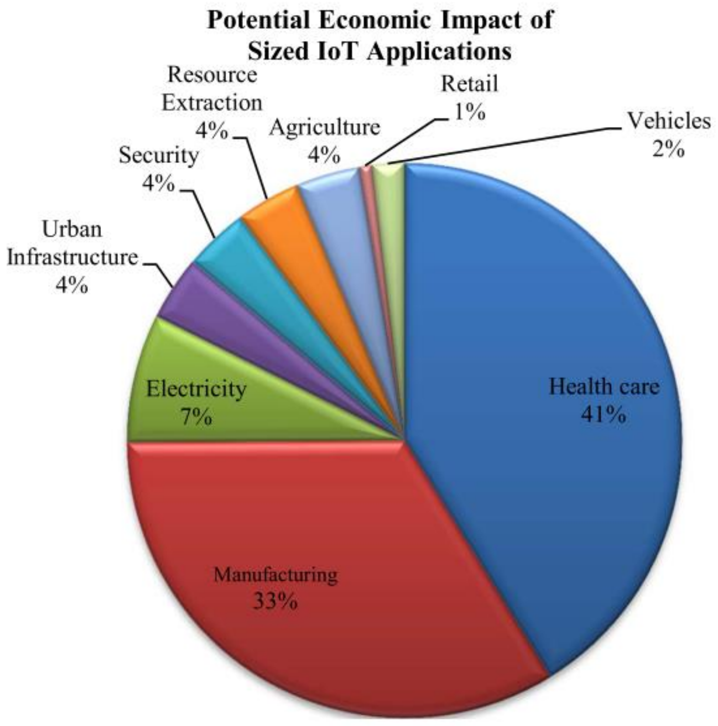


Figure 1.3 – Projected market share of dominant IoT applications (Al-Fuqaha, Guizani, et al., 2015).

Since any IoT deployment is a combination of different technologies and standards, a model or architecture must be considered for a successful implementation. Several models have been proposed to allow a better design of any IoT deployment, this considering the amounts of devices that could be connected to any one application; according to (Al-Fuqaha, Guizani, et al., 2015) several architecture models have been proposed. However, a five-layer model seems better suited for the needs of flexibility and abstraction the whole IoT stack requires.

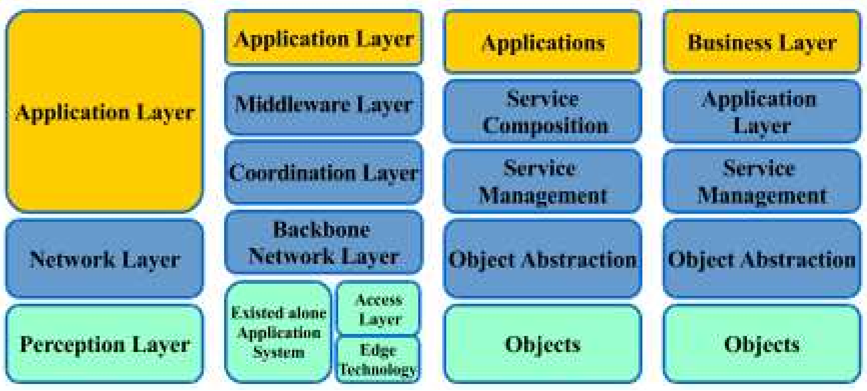


Figure 1.4 – The different architecture models for the IoT (Al-Fuqaha, Khreishah, et al., 2015).

To understand the construction of this five-layer model, a description of each layer will be presented, with the intention of providing a better understanding of all the building blocks involved.

Also, this description will be provided as a starting point to understand what parts of the stack need to be addressed to accomplish the objectives of this work.

The first layer within this architecture is the object layer, also called the perception layer, which represents the physical objects, typically sensors, that operate within the IoT (Al-Fuqaha, Guizani, et al., 2015). This layer includes the different sensors and actuators that constitute any IoT applications, with the consideration that such different devices will generate heterogenous data, one of the main strengths of the IoT, but also one of the main problems for general integration between applications.

On top of this layer comes the object abstraction layer, which transfers data from the objects to the top layer, usually through secured channel. Next comes the services management layer, which is a middleware that pairs services with requesters based on addresses and names. This is the layer that allows application programmers to work with heterogenous objects without considering any specific hardware platform (Al-Fuqaha, Guizani, et al., 2015).

Next comes the application layer which is the gateway of interaction between users and the services. The users in this case could be human beings as well as other machines that make use of these services. This layer is important because of its ability to provide any user with smart services at their request (Al-Fuqaha, Guizani, et al., 2015). Finally, the top layer of this architecture is the business layer, whose job is to manage the overall IoT system activities and services. This is done in a way that returns information in the best way possible, that is in the form of graphs, flowcharts, etc (Al-Fuqaha, Guizani, et al., 2015). The structure of this architecture is shown in figure 1.4.

In addition to understanding the architecture of the IoT, one other important abstraction to be made is the separation of critical building blocks that are inevitably present in any deployment. Figure 1.5 shows these elements to create the IoT. As can be seen the abstraction at this level involves six different elements that allow for separate development and more importantly to match different technologies and different standards, depending on the use case.

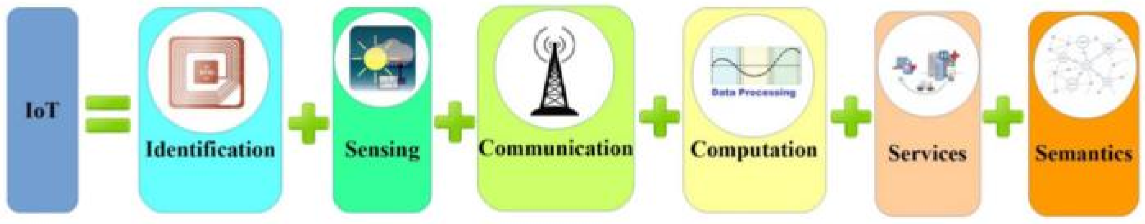


Figure 1.5 – Abstraction of IoT into different categories (Al-Fuqaha, Guizani, et al., 2015).

In addition to the five-layer architecture model presented, another abstraction of the IoT solution can be synthesized in six elements that are universal to any IoT implementation, regard-less of any architecture model being used as a template. This six-element synthesis should not be viewed as a replacement of the five-layer architecture model, but complementary to understand universal requirements that should be met, regardless of architecture, technology or implementation details.

To understand this six-model distribution, a description of the respective elements will be done, these elements can be catalogued are: Identification, sensing, communication, computation, services and semantics (Al-Fuqaha, Guizani, et al., 2015). On the identification category, this is in relation to the methods that allow the matching of services to objects, addressing and identification fulfill two different but important roles, one allows identification within the realm of the application while the other helps differentiate objects within a communications network (Al-Fuqaha, Guizani, et al., 2015).

One of the main drivers behind the IoT is the ability to gather data, the sensing elements in any deployment are responsible for gathering data as well as sending it to a data warehouse, database or cloud. This is closely related to the communication element, which corresponds to the interconnection of heterogenous elements together to deliver smart services (Al-Fuqaha, Guizani, et al., 2015). Different communication technologies are used in this stage, depending on the requirements, although most IoT nodes are expected to operate using low power. Typical technologies used for this purpose are WiFi, Bluetooth, IEEE 802.15.4, Z-Wave and LTE.

Computation refers to any processing done on the IoT, this can be either the individual objects that require an element of processing to carry out the data gathering operations, or the cloud component which provide facilities for the objects to send their data as well as the processing of said data to recognize patterns and generate the services. Regarding these services, there are different categories, with each using information passed from the next, specifically we can fit any service within one of these four categories: identity-related, information aggregation, collaborative-aware and ubiquitous (Al-Fuqaha, Guizani, et al., 2015).

The identity-related services can be considered the most basic and important ser-vice, since every other service are built on top of them, their job is to allow identification of individual objects and allow them to form part of the application. Information aggregation services work on top of identity-related services, and their job is to collect and summarize raw sensor measurements that need to be processed and reported by the IoT application (Al-Fuqaha, Guizani, et al., 2015). Collaborative-Awareness services use the information from the service below to make decisions and react according to the raw data available. Finally, ubiquitous services are responsible for providing any Collaborative-Aware service at any time it is needed. As can be seen, the structure of these services work on top of other services to provide a more modular approach, this in turn promotes easier integration, figure 1.6, shows how this service structure works.

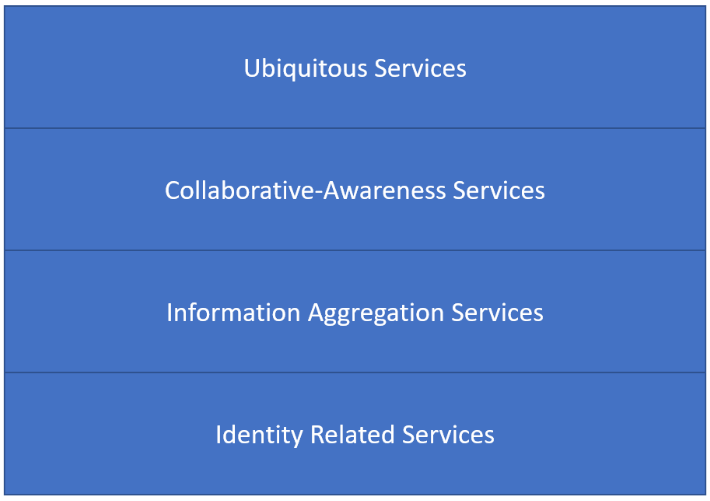
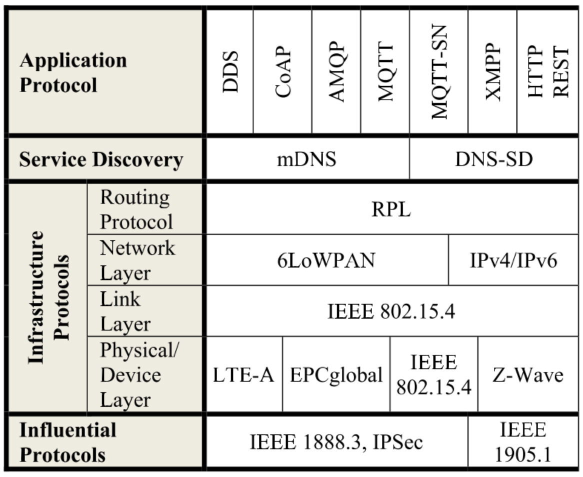

Ubiquitous Services
Collaborative-Awareness Services
Information Aggregation Services
Identity Related Services

Figure 1.6 – Service structure with the most critical services at the bottom.

The last IoT element to regard is semantics, which consists of extracting knowledge smartly by different machines to provide the required services, in other words, raw data is converted into useful information (Al-Fuqaha, Guizani, et al., 2015)(Wagle, n.d.). There are several techniques that can accomplish semantic extraction, one way is using data formats that allow for easy data extraction, with XML and JSON as the two most important, both data representation for-mats possess a lightweight version, considered for resource-constrained environments. In the case of XML, it is called EXI and converts the XML structure into binary, to reduce bandwidth (Al-Fuqaha, Guizani, et al., 2015). In the case of JSON, a format is JSON-LD which is also extremely light-weight and beneficial for constrained devices (Datta & Bonnet, 2016).

1.4 STANDARDS AND PROTOCOLS

Since the IoT consists of several heterogenous devices with different requirements, especially regarding power consumption, different standards exist at different levels that allow integration of such heterogenous devices between each other and the ap-plications that run on top. Figure 1.7 shows a table of different standards and their use at different levels. It is important to note that all protocols must be bundled together to deliver an IoT application (Al-Fuqaha, Guizani, et al., 2015).



Application Protocol		DDS	CoAP	AMQP	MQTT	MQTT-SN	XMPP	HTTP REST
Service Discovery		mDNS				DNS-SD		
Infrastructure Protocols	Routing Protocol	RPL						
	Network Layer	6LoWPAN					IPv4/IPv6	
	Link Layer	IEEE 802.15.4						
	Physical/ Device Layer	LTE-A	EPCglobal	IEEE 802.15.4	Z-Wave			
Influential Protocols		IEEE 1888.3, IPSec					IEEE 1905.1	

Figure 1.7 – Standards for the IoT (Al-Fuqaha, Guizani, et al., 2015).

At the application level, there are several protocols that can be used, most of which were created for application rich devices, some of them have characteristics that make them suitable for typical IoT devices, which are resource constrained. At the application level the prominent protocol is the Hypertext Transfer Protocol (HTTP) which is widely used in conventional inter-net because of its client-server computing model, which allows for easy transactions between all the clients. Figure 1.8 shows an arrangement of HTTP in an IoT solution, where several devices connect to the server and the user (application) also has a connection in the same way.

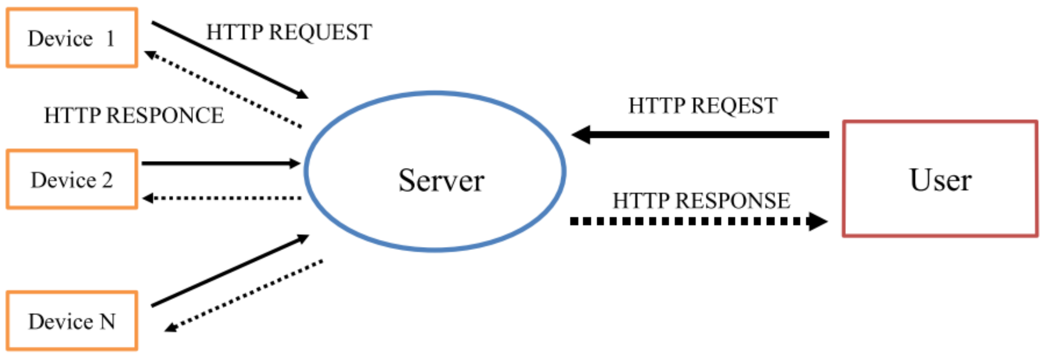


Figure 1.8 – Diagram of an IoT deployment with HTTP as the application protocol

For HTTP, any message being sent passes through a series transmission, figure 1.9 shows how an HTTP transaction works, with several requests and responses being sent through the channel. Due to HTTP working over TCP/IP, this sequence is necessary to make sure messages sent are being received correctly (Yokotani & Sasaki, 2016).

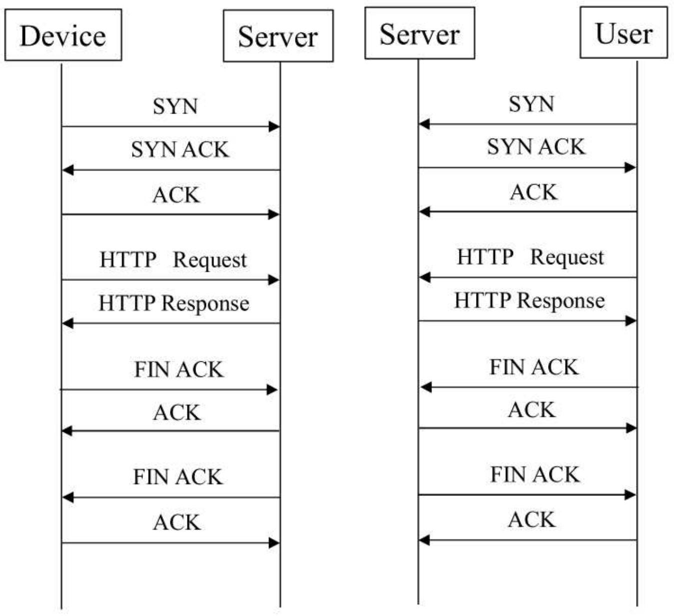


Figure 1.9 – An HTTP transaction going from device to user (application) (Yokotani & Sasaki, 2016).

One of the main uses of HTTP is through Representation State Transfer (REST) Application Programming Interfaces (API), which allows interoperability of web services by using just four operations: GET, POST, PUT and DELETE. This simplifies the operations between applications and make sure integration between them is possible and al-most universal, this is how the conventional internet was able to grow and exist in its current form.

HTTP has some disadvantages for IoT applications: firstly, because every HTTP transaction is symmetrical, it requires a request and response from the client and server respectively, which means that the connection has to be kept open, which in turn generates a greater power consumption. The second disadvantage is that identification on HTTP works by way of IP addresses or URL, which require a connection to the larger internet, which might not be possible for some applications, this limits the use cases for HTTP. Because of this, and as figure 1.9 shows, there is a serious overhead due to headers to make sure that the messages are being received (Yokotani & Sasaki, 2016). Because IoT devices usually operate on battery power, which in turn involve several constraints regarding its use, other type of protocols is necessary. Figure 1.10 shows a time diagram of how an HTTP connection operates in a possible IoT scenario.

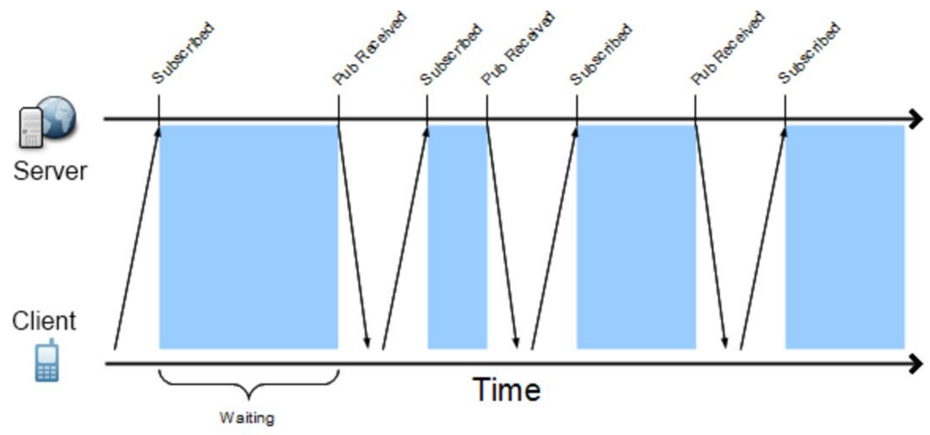


Figure 1.10 – Communication diagram of an HTTP communication between a client and server (Datta & Bonnet, 2016).

A suitable direct alternative for HTTP is Constrained Application Protocol (CoAP); CoAP defines a web transfer based on REST, which in a way allows for an easier integration of applications. Unlike HTTP, CoAP runs on UDP, this allows for faster wake-up and transmit cycles due to its connectionless datagrams, as well as smaller packets with less overhead (Thota & Kim, 2016); this allows for the device to remain in a sleep state and only transmit when needed without too much battery consumption.

Another benefit of CoAP is that the identification used in CoAP are URI (Uniform Re-source Identifier), this allows a degree of independence in the message packets, since the destination node's capabilities are partly understood by its URI details (Thota & Kim, 2016). This makes it possible that a battery powered sensor node may have one type of URI while a line powered flow-control actuator may have a different one.

CoAP can handle the request/report model as well as a publish/subscribe model more commonly associated with other types of protocols such as MQTT and AMQP. Use of any of these models require an adaptation of the protocol since it runs on top of UDP which doesn't allow for any acknowledgement of a correct message reception.

CoAP is not as mature a protocol as MQTT for instance, this impacts on the number of implementations and library ports available for its use. The other issue is that CoAP works particularly well for M2M communications because of its UDP implementation, but integration between a user application (such as web app) and these devices might not be direct. Fortunately, because CoAP has been designed based on REST, a conversion between HTTP and CoAP is straightforward. Figure 1.11 shows an example implementation of several IoT devices communicating through CoAP, and a REST-CoAP proxy that bridges these devices to any external application.

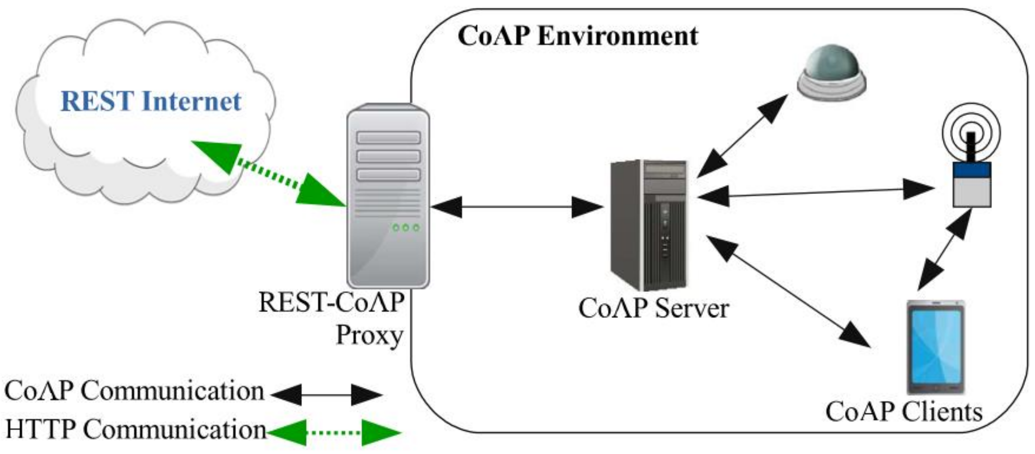


Figure 1.11 – Example of CoAP deployment (Al-Fuqaha, Guizani, et al., 2015).

A typical CoAP message can be between 10 to 20 bytes, with the packets having the form depicted in figure 1.12. The header takes four bytes indicating the version of CoAP, the type of transaction, the option count and the request method to be used (Al-Fuqaha, Khreishah, et al., 2015).

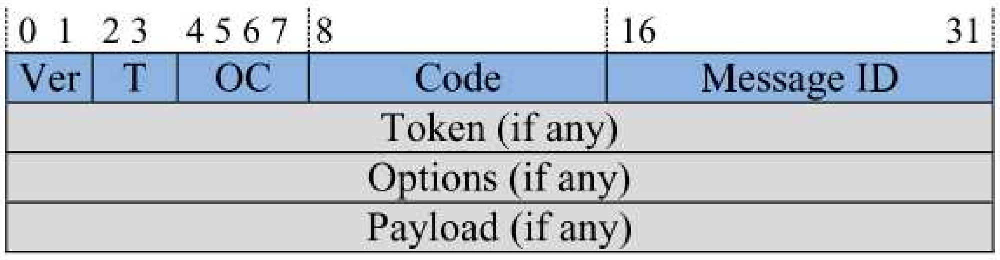


Figure 1.12 – CoAP message format (Luzuriaga, Perez, et al., 2015).

An alternative to CoAP is the Message Queue Telemetry Transport (MQTT) protocol, which was proposed by Andy Stanford-Clark of IBM in 1999 and was standardized in 2013 (Al-Fuqaha, Guizani, et al., 2015). MQTT is designed around a publisher/subscribe model, in which devices subscribe to “topics” that are relevant to their operation; whenever a device publishes a message under a topic, every single device subscribed to that topic will receive it. Figure 1.13 shows the architecture of an MQTT application.

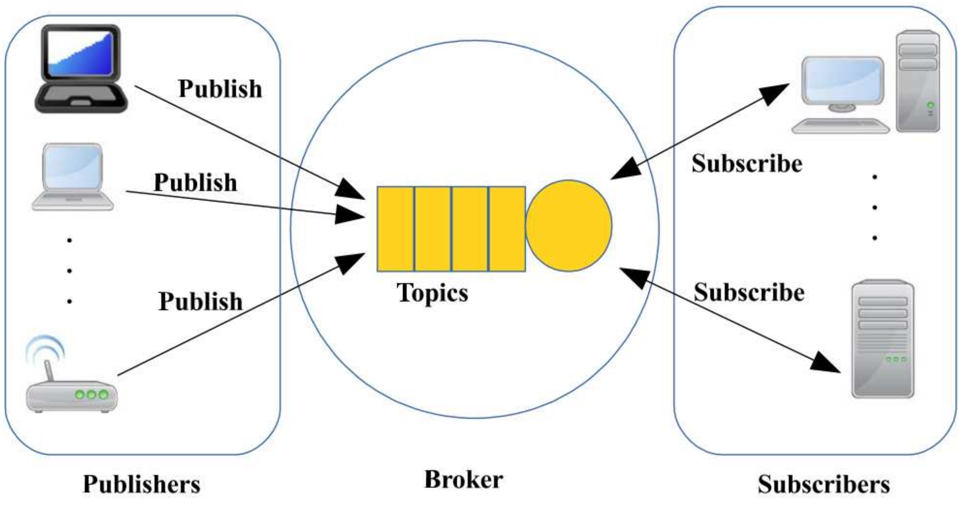


Figure 1.13 – MQTT architecture (Al-Fuqaha, Guizani, et al., 2015).

MQTT runs on top of TCP which is connection oriented and more reliable than UDP, because it has error detection mechanisms; MQTT has a small overhead because most of the "frame" data is sent upon connection, after that a connection is maintained and only the payload is sent any time a message needs to be published. The protocol has 3 components: publisher, subscriber and a broker; an interested device would register as a subscriber for specific topics to be informed by the broker when publishers publish topics of interest, when a message under a topic of interest is published, the broker distributes it among all the subscribers of that topic (Al-Fuqaha, Guizani, et al., 2015)(Yokotani & Sasaki, 2016)(Luzuriaga, Perez, et al., 2015)(Nicholas, n.d.)(Singh et al., 2015)(Luzuriaga, Cano, et al., 2015). Security can be achieved by the broker by checking authorization of the publishers and subscribers connected (Al-Fuqaha, Guizani, et al., 2015)(Singh et al., 2015). Figure 1.14 show the dynamic of the communication in an MQTT application.

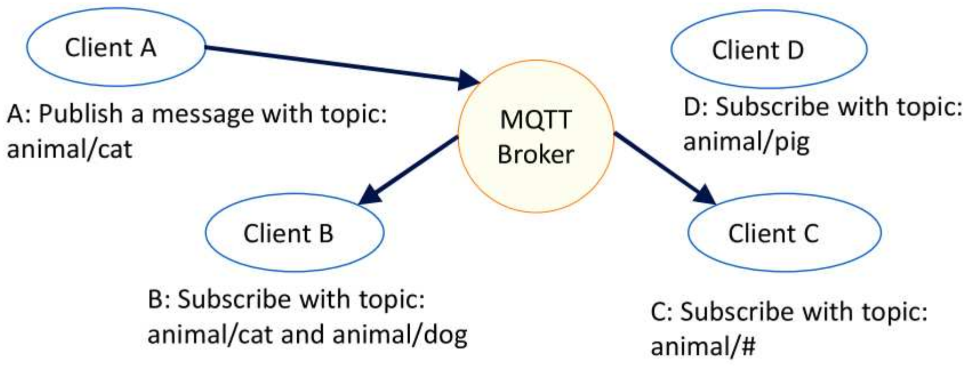

Client A
A: Publish a message with topic: animal/cat
MQTT Broker
Client D
D: Subscribe with topic: animal/pig
Client B
B: Subscribe with topic: animal/cat and animal/dog
Client C
C: Subscribe with topic: animal/#

Figure 1.14 – MQTT communication model (Klauck & Kirsche, 2012).

MQTT is a popular protocol being used in such diverse applications as healthcare, monitoring, energy metering and Facebook notifications (Al-Fuqaha, Khreishah, et al., 2015). The reason for such wide use in in small, cheap, low power and low memory devices that operate in vulnerable and low bandwidth networks is due to the message format which requires very little header data to route the messages to its destinations. Figure 1.15 shows the message format, as it can be seen the number of bytes required for control of the protocol is of 5 bytes maximum, with the first indicating the type of message, the Quality of Service (QoS), if the message is a duplicate and if it should be retained by the server; the remaining bytes indicate the length of the topic plus payload, with the payload and the message being of a variable size (Al-Fuqaha, Guizani, et al., 2015).

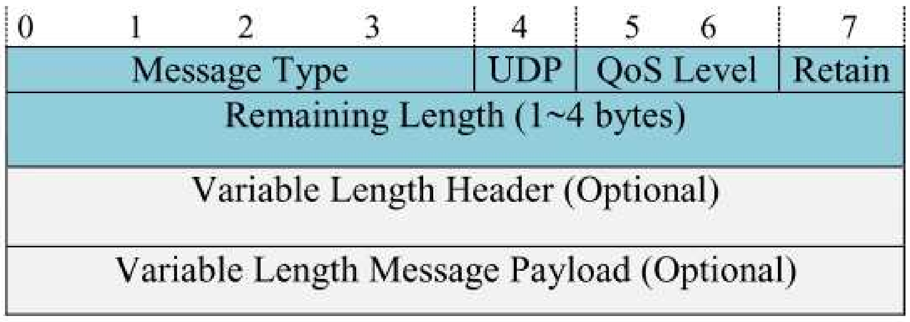

0
1
2
3
4
5
6
7
Message Type
UDP
QoS Level
Retain
Remaining Length (1~4 bytes)
Variable Length Header (Optional)
Variable Length Message Payload (Optional)

Figure 1.15 – MQTT message format (Luzuriaga, Perez, et al., 2015).

Regarding the QoS, MQTT establishes three levels of QoS for delivery assurance of publish messages, these levels can be identified in the QoS Level field (this in reference to figure 1.15). QoS Level 0, commonly known as "Send and forget", has a single transmission of the message with no guarantee of its arrival. This level can be used for highly repetitive or non-mission critical messages. QoS Level 1 tries to guarantee a message is received at least once by the intended recipient, once a published message is received and understood by the recipient, it acknowledges the message with an acknowledgment message (PUBACK). Until this message is not received, the message is stored, the publisher stores the message and retransmits it periodically. Finally, QoS Level 2 attempts to guarantee the message is received and decoded by the intended recipient. This is the most secure and reliable level of QoS (Thota & Kim, 2016).

Besides application protocols, another important element for any IoT deployment is the use of discovery protocols, to allow any application to discover resources and ser-vices in a self-configured, efficient and dynamic way (Al-Fuqaha, Guizani, et al., 2015). Once again protocols already in use in the conventional internet are of interest for the IoT, the two most important protocols at this level are multicast DNS (mDNS) and DNS Service Discovery (DNS-SD).

mDNS inquires names by sending an IP multicast message to all the nodes in the local domain; by this query, the client asks devices that have the given name to reply. When the target machine receives its name, it multicasts a response message which contains its IP address. All devices in the network that obtain the response message up-date their local cache using the given name and IP Address, figure 1.16 shows how mDNS operates. In the case of DNS-SD, clients discover a set of desired services using standard DNS messages. DNS-SD uses mDNS to send DNS packets to specific multicast addresses through UDP. Here there are two steps: finding host names, like a printer, and pairing IP addresses with their host names using mDNS. Figure 1.17 shows how the discovery works in DNS-SD. (Al-Fuqaha, Guizani, et al., 2015)(Klauck & Kirsche, 2012)(Jara et al., 2012).

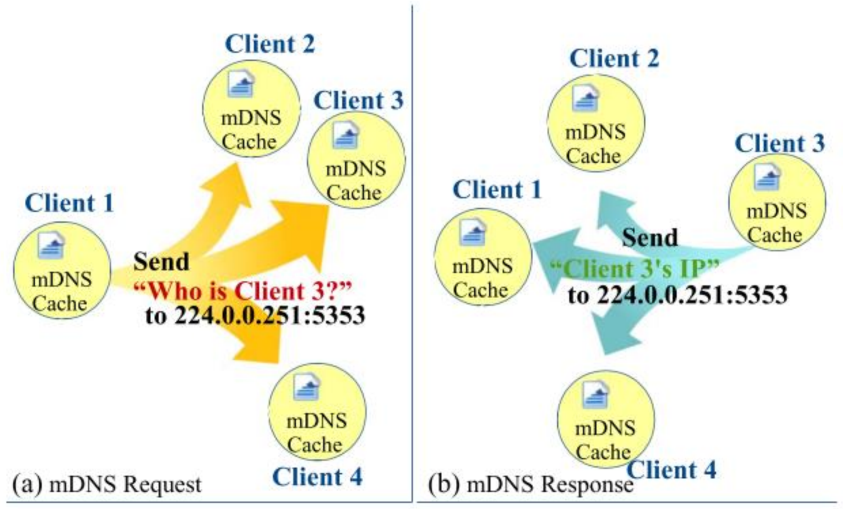


Figure 1.16 – mDNS operation (Al-Fuqaha, Guizani, et al., 2015)

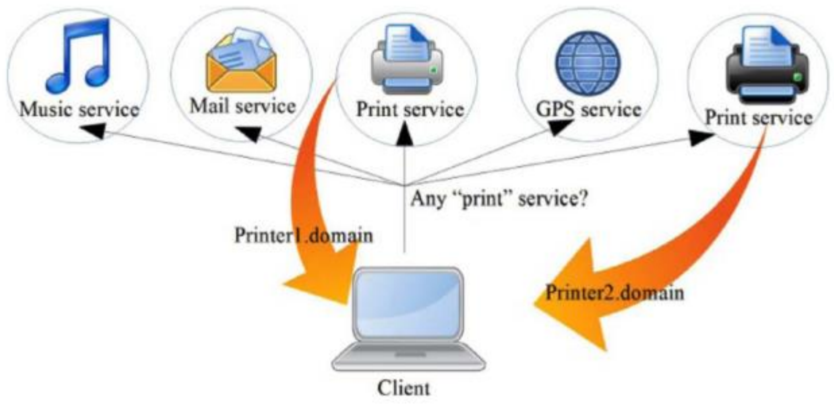


Figure 1.17 – Discovery of a print service by DNS-SD (Al-Fuqaha, Guizani, et al., 2015)

1.5 CURRENT CHALLENGES IN IOT DEPLOYMENT

As has been previously discussed, one of the main drivers of IoT is that different objects can gain connectivity and processing capabilities; this creates huge opportunities, but also brings one challenge: the existence many heterogenous devices that will share disparate data formats with each other. On top of this, the devices that will directly interact in IoT networks can be of different types, (Al-Fuqaha, Guizani, et al., 2015) defines two types resource rich, which have hardware and software capable of supporting the TCP/IP protocol Suite, and resource-constrained which have smaller processing capabilities (such as microcontrollers), which are unable to support the whole TCP/IP suite.

The advantages of TCP is that many known application protocols run on top of it, making sure that application is infrastructure independent (the underlying devices could be communicating via WiFi, IEEE 802.15.4, or any other protocol that supports TCP); the expectation is that this would allow horizontal applications, which in turn would allow IoT applications to work side by side with conventional internet application and legacy applications that are already running in this conventional internet (Yokotani & Sasaki, 2016). Figure 1.18 shows how this approach would work.

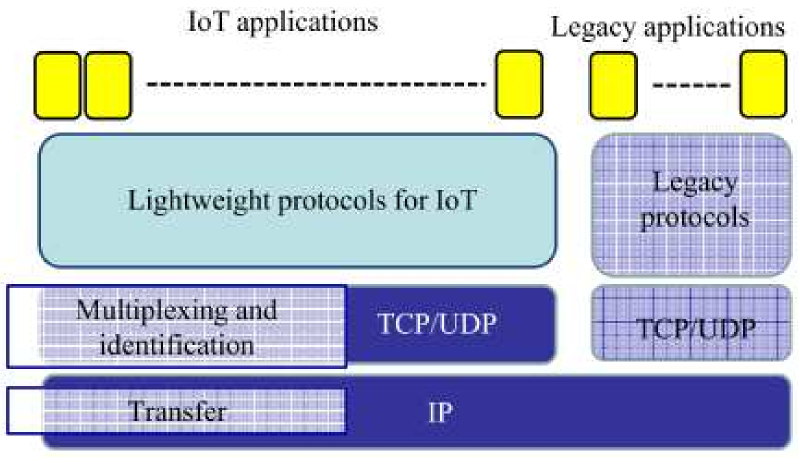


Figure 1.18 – How a horizontal approach for IoT would enable faster integration (Yokotani & Sasaki, 2016)

It is evident that the landscape for IoT applications is deeply fragmented between proto-cols used for communication within, and across, resource-constrained and resource-rich devices, this fragmentation is not expected to change soon (Al-Fuqaha, Guizani, et al., 2015). This inability to create a real interoperation between IoT devices and their applications is what keeps the whole structure from becoming a horizontal one and hinders wider adoption.

For a real and easy integration between applications to happen some element must exist that exposes all services and devices to the application in a transparent manner. This would allow that the application be written with existing and familiar tools to the programmer, and with all the flexibility of using the application protocols that are best suited for the job. Moreover, third parties could build application for existing IoT devices without the need for them to be deployed by that same party. This last part is important since right now little to no integration can be done by third parties, and any application must be designed explicitly for the system being deployed.

The current work proposes a middleware component for IoT that allows several types of IoT devices to connect to an application and to other devices. The middleware being proposed will allow connection between different IoT devices, applications and al-low it to be protocol independent at the application level.

2 STATE OF THE ART

Given how the IoT is one of the topics of most interest these days, it is natural that a lot of research is going into it lately. Also, as was discussed in the previous chapter there are many aspects crucial to the IoT, with protocols, interoperability, applications and security being the top of them. This chapter will focus on showing the most recent research done in these areas.

Regarding application protocols, the analysis and research topics are diverse. On (Luzuriaga, Perez, et al., 2015) MQTT and AMQP are evaluated for use over unstable and mobile networks, as mobile networks (such as GSM, 3G and LTE) are common infrastructure protocols for IoT devices (Al-Fuqaha, Guizani, et al., 2015). The choice of AMQP and MQTT is due to the fact that they are the most suitable for implementing Message Oriented Middleware (MOM); MOMs are beneficial because they provide abstraction of the different participating entities (i.e. devices, applications, etc.), easing the programmers job of communicating these different components.

The basic concept of MOMs is that messages are added to a queue so that they are delivered to the interested components of the system, the advantages of this type of arrangement is that the identity of the elements can be defined in an abstract way and not needing to know the physical details of it. The protocols used (AMQP and MQTT) are the most relevant for use in MOMs and have similar approaches with some key differences. AMQP has a producer/consumer model that is like MQTT but that allows messages of any size and have a fixed header (unlike MQTT with the minimum header being 2 bytes but capable of having 5 bytes in total).

The tests conducted in (Luzuriaga, Perez, et al., 2015) measured jitter in the messages sent by producer/publisher and received by the consumer/subscriber on two different devices simulating unstable or mobile networks. From the tests MOMs not only allow for a layer of abstraction that helps integration of different IoT elements easier, but that they are a reliable form of communicating information even in unreliable networks, such as mobile ones. An interesting conclusion is that while both protocols have a similar robustness to operate under these networks AMQP offers more aspects related to security, in other words, assuring the correct reception of messages, while MQTT is more energy efficient, and most likely could have a better throughput.

Even though AMQP and MQTT are very similar in many respects, MQTT has the advantage of being a more mature protocol with implementations in many languages and more research being done on it. The works presented in (Yokotani & Sasaki, 2016), (Chen & Lin, 2014), (Asghar & Mohammadzadeh, 2016), (Luzuriaga, Cano, et al., 2015), (Thota & Kim, 2016), (Nicholas, n.d.), and

(Doukas et al., 2015) study how MQTT compares to different protocols, how it operates in different types of networks and some elements like power requirements that are critical to the operation of the IoT.

In (Yokotani & Sasaki, 2016) MQTT is compared against HTTP, the most widely used protocol in the conventional internet. HTTP has a symmetrical nature that works well in standard web applications, a client requests data, the server returns the request a moment later, this operation is not suitable to the IoT due to some of the end nodes (the devices) just sending data, like sensors. Because MQTT has an asymmetrical architecture, if a device needs to send a message it only requires sending it and doesn't have to wait for a reply from the other device. The configuration (Yokotani & Sasaki, 2016) proposes for illustrates exactly how MQTT works and how it could be deployed for a given IoT application, figure 2.1 shows this.

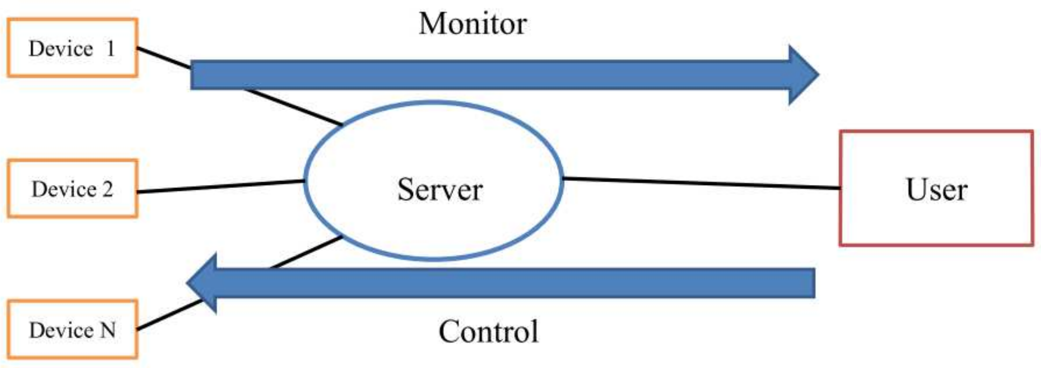


Figure 2.1 – System configuration for an IoT application using MQTT (Yokotani & Sasaki, 2016)

It can also be seen in (Yokotani & Sasaki, 2016) that MQTT requires less bandwidth to send a message, this is because of the overhead HTTP has due to headers, even when payload is incorporated into the messages, the difference of bytes required between MQTT and HTTP is big. It is important to note that for both protocols the number of devices operating affect the number of bytes being used, which is an intuitive effect.

Other advantage observed in MQTT is the fact the server resource usage for MQTT tends to be lighter than HTTP which means that for a system configuration MQTT would be able to support more devices connected to the broker. An interesting phenome-non can be seen when the size of the topic increases, there's a point when there is a sufficiently large topic, in the case of (Yokotani & Sasaki, 2016)is shown to be 680 bytes for 10 devices connected and a payload size of 0, the overhead of HTTP is smaller than that of MQTT.

Another important advantage of MQTT is the fact that since it operates over TCP, it can be integrated on standards that naturally work with HTTP. On (Chen & Lin, 2014) an MQTT broker is added to a OneM2M server to act as a gateway between devices that work with MQTT and those that work with the standard. OneM2M is a standard supported mainly by ETSI (oneM2M, 2019), which aims to be network independent and able to interconnect different devices and enable communication between them, the most popular implementations of OneM2M are openMTC (OpenMTC, 2016) and OM2M (OpenM2M, 2015).

Even though OneM2M does support operations with MQTT, the frameworks described before don't have an implementation of the MQTT binding described in the standard. Once again it can be seen that from an end device standpoint, the use of an asymmetrical protocol like MQTT allows for a more seamless integration, since a lot of IoT devices are usually delivering data (in the form of sensor data). The bridge proposed by (Chen & Lin, 2014) maintains a lot of the important characteristics of OneM2M, like the data containers and a resource tree that can be translated from the standard to all the devices communicating via MQTT.

Once again comparing the performance of MQTT with HTTP (both being used within the confines of the OneM2M standard), the latency generated from MQTT is much lower than the one for HTTP with two different parameters: first with a lot of threads that emulate different devices, and later with an increased payload size (fixing the number of threads). In both tests the results are consistent, MQTT experiences less latency.

Continuing with the study of MQTT, (Luzuriaga, Cano, et al., 2015) proposes a decoupling mechanism of an inter-mediate buffer to the conventional implementation of MQTT. The choice of MQTT is supported by the work done in (Collina et al., 2014) which suggests the use of this protocol in the presence of high delays, and the use of CoAP when the data traffic increases significantly. The modification proposed uses an intermediate buffer to store the messages that the broker will need to send, if a connection is broken the messages aren't lost because they are stored in the buffer. This arrangement benefits greatly MQTT messages that are being sent with a QoS level of 0, since this creates a new level of security for the message delivery.

As can be seen from the results in this work (Luzuriaga, Cano, et al., 2015), the messages are delivered without loss, and most importantly with a network that might not be completely stable. Furthermore, the Jitter generated from such a network falls within acceptable values.

One last work comparing MQTT is presented in (Thota & Kim, 2016) in which it is compared with CoAP, another lightweight protocol commonly used in resource-constrained devices, that runs over UDP instead of TCP. The authors in this work used a resource-constrained device, the ESP8266 which has a 32-bit processor as well as a WiFi radio, which makes it an ideal platform for an IoT end device.

The tests show that both protocols work with 100% reliability, even when packet loss is induced, which shows the capabilities of both for retransmission. One caveat with the retransmission capabilities of MQTT is that by increasing the QoS level, the application performance suffers for an overhead increment (Luzuriaga, Cano, et al., 2015).

Other crucial point of research within the IoT is the application side, a lot of work has been put into deploying diverse types of applications. This research is important since it highlights the needs in terms of how different computing platforms are used in these applications, as well as the data structures and reliability needs.

As was discussed earlier the IoT is expected to affect every aspect of society, so different markets and segments have diverse needs; the work presented in (Koch & Ph, 2017) shows how the IoT could be beneficial for the electrical grid, the so-called smart grid. A Raspberry Pi (*Raspberry Pi - Teach, Learn, and Make with Raspberry Pi*, n.d.) is used as the end device to monitor the status of a specific part of the grid, and report it to a server, the cellular connection also allows for the device to receive up-dates if there are any. The advantage of using a Raspberry Pi is that it is a resource-rich device which means that it is compatible with the whole TCP/IP suite, so for this scenario the use of HTTP is possible.

It can be seen that several segments are still in a more immature stage of development, one example of this is agriculture which is explored and developed in (Kamilaris et al., 2016) by proposing a IoT framework. This framework attempts to model real sensors in the middleware through semantic extraction of the values, to allow for an easy integration at the application level. Since there are no middleware deployments targeting the specific agriculture segment, it is a novel an interesting approach, even though it reveals a more generalized problem, of application protocol independent solutions.

In the work presented in (Campo et al., 2016) a solution is proposed in the healthcare segment, once again the use of middleware is presented as the best solution of integrating all the devices working in Ambient Assisted Living (AAL) for elderly people with some sort of mental disorder (in this particular case dementia). The authors analyze different frame-works used for the healthcare segment, finally concluding that all application cases require active user input (e.g. performing a blood pressure movement), also as in the case of (Kamilaris et al., 2016) no middleware implementation presents a truly universal integration, which contributes to fragmentation in the IoT.

The authors of (Campo et al., 2016) propose a middleware using MQTT, once again because its asymmetric architecture allows for an easy transmission of messages. Also, because the destination is a topic and not a physical address, MQTT allows the decoupling of the network protocol from the application, which helps the programmer focus more on application logic. Of note is that the authors show two different type of applications for IoT: applications for ambient data collection and analytics, and real time reactive applications. The former collects data for later analysis, with emphasis on statistics and trends, while the latter deals with real time decision making based on the most recent data. Even though this classification exists, both categories are not exclusive amongst themselves, which allows for an application to require both types of operation.

Also, on the topic of applications the authors of (Bondarevs et al., 2017) propose the concept of virtual things, which are a virtual model of a real IoT device. The main advantages of this approach are the extended capabilities the virtual object has (e.g. larger memory), because the hardware limitations that the real object has are eliminated within the virtual environment. The second advantage to this is that any application will interact with the virtual object and not the real one, once again decoupling the underlying hardware components from the application, allowing for an easy integration.

In this work, this characteristic is more prominent because the architecture is laid out in a way that makes any interface with the IoT devices irrelevant to the application and allowing a faster and seamless integration. The approach proposed in (Bondarevs et al., 2017) is shown in figure 2.2, where separation of the applications and devices is done at the middleware level, thus allowing direct integration. The downside of their approach is that IP addresses (both IPv4 and IPv6) are re-quired to identify each virtual object, which means that some type of discovery protocol is need-ed to work alongside the middleware, adding to its complexity.

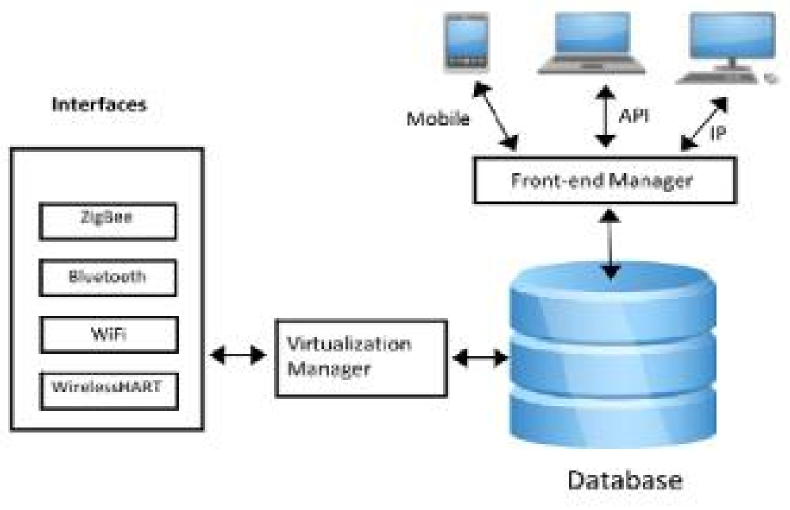


Figure 2.2 – Proposed architecture for virtual IoT (Bondarevs et al., 2017).

In (Kamilaris et al., 2016), (Campo et al., 2016), (Bondarevs et al., 2017) the concept of semantic extraction is mentioned and as it has been dis-cussed, semantic extraction is a crucial topic to convert the raw data from the end devices into information that can be used in services. Another approach for semantic extraction is developed in (Wagle, n.d.) in which a classification-like algorithm is used to extract relevant information for a weather application. The algorithm has a lot of similarities with a machine learning project in that there's a training set that allows the algorithm improves its performance over time. The semantic extraction was made easier by the usage of MQTT, in which the topics that come with every message can be also parsed to determine the type of data being received.

On the same subject of semantics (Mohalik et al., 2016) addresses the critical fact of interoperability, referring to the interoperability in data and not in protocols as their work assume that part has been achieved. The concept of interoperability as defined by them is "the ability to exchange usable data between two systems and to invoke their services using the appropriate parameters", this means that all devices within the IoT, including the applications, have a common understanding of data flowing throughout every element in the network. (Mohalik et al., 2016) also proposes a system that is dynamic since not only interoperability is important but also the ability to adapt to constant changes in the application requirements that need to be reflected in the middleware.

One other important topic to consider in IoT and that recently has been getting a lot of attention is security. Although security is a problem that have different angles of solutions, there are several works trying to solve the various parts of the security issues that are prevalent in IoT. For instance, one of the significant issues is firmware update of the devices; this is critical since software will inevitably present bugs that could be used as exploits to access the system in an unauthorized way, or even worse, use of these devices as part of a botnet to cause damages to other system; this actually has happened and had serious consequences on several aspects of modern life (Brian Krebs, 2016). One of the works that addresses this particular issue is (Koch & Ph, 2017), where the case for software updates on the end devices is considered on its design, this intrinsically secures the system in terms of correcting bugs in any deployed device as soon as they're found.

Other approaches to security are done at the protocol level. From previous works it has been shown that MQTT is one of the most viable application protocols for the IoT, due to its simplicity in generating a network, and the asymmetrical architecture allows for almost in real time data for devices that are connected and assurance that messages will get to the devices that are unavailable at the time but will connect later.

In that regard, (Singh et al., 2015) proposes a modification to the MQTT and MQTT-SN (a version of MQTT for sensor networks) that aims to lower the processing overhead required by some other proposals by using encryption based on attributes, this allows that only de-vices that satisfy those attributes get the messages they are subscribe to. This is possible because in ABE (Attribute Based Encryption), the key is generated with those attributes as inputs, the other aspect of this approach is that encryption uses 128-bit AES. The results on this work shows that depending on the type of encryption (use KP-ABE vs. CP-ABE), the key size and the number of attributes used at the time of the encryption can increase considerably. Even though this approach is an interesting one, it might not work very well on resource-constrained devices, where memory and processing resources are extremely limited.

Another approach is proposed in (Mektoubi et al., 2016) in which MQTT is coupled with RSA type encryption which allows for a more secure communication. The reasoning behind the approach is that SSL and TLS (which MQTT supports due to being run on top of TCP), are insufficient to support a truly secure communication. In order to address these issues, the authors propose the use of a Certificate Authority (CA) as means of authentication and the elliptic curve algorithm for encryption to ensure that all communications remain safe.

3 METHODOLOGY AND REQUIREMENTS FOR THE PROPOSED SOLUTION

As it has been previously discussed, one of the main factors for greater adoption of the IoT is its ability to create horizontal integration which allows for applications being developed by third parties, with no required knowledge of the underlying components. This approach is what allowed the traditional internet to grow to the point to where it is right now.

This integration also needs to work with different types of end devices, both resource-rich and resource-constrained, with the consideration that in the case of resource-constrained devices it might not be possible to use the same type protocols to communicate between them. Regardless of that our solution should be capable of guaranteeing interoperability between de-vices and allowing the applications to interact with them regardless of the underlying networks.

One other important requirement for this solution to work correctly is that it should sup-port both categories of applications: data collection and analytics, and real time, if the applications require or support it.

With all these considerations, the proposal we present is a middleware component that allows several types of IoT devices to connect to applications and to other end devices, with the capabilities of extending the end devices in terms of memory mainly, especially when hardware doesn't allow for the storage of large amounts of data. Also, the middleware being proposed will allow connection between different IoT devices, applications and allow it to be protocol independent at the application level.

The best approach as can be seen from previous works is to use a middleware as a bridge between end devices and applications. Specifically, the middleware needs to fulfill the following requirements:

- Ease of integration for both applications and end devices. This requires that programmers on both ends of the solution don't need to consider any other element with-in the system.
- Decoupling of application logic with the underlying workings of the system. Since the goal of the system is allow for an effortless application integration, this element is of importance.
- Consideration of heterogenous devices participating within the system. Considering how many diverse types of devices can operate in the IoT, and that each device might be sending different types of data and more importantly, different types of data for-mats, the middleware should have a way to model this behavior while allowing devices and applications reading this data structures to extract only the relevant parts for them.

- Capacity of every element in the system to allow real time applications as well as big data applications. This can help reuse data from sensors into different types of applications which might have different requirements; furthermore, this keeps the system as flexible as possible, which in turn fosters integration.

All these requirements will be considered as the proposed solution. The design of such solution is presented in the next section.

3.1 MIDDLEWARE DESIGN

The middleware proposed in this work contains two main parts: a data warehouse, or repository, that contains a data structure that models the devices and extends the memory on them to allow historic sensor data stored in the middleware for any application or device needing it to retrieve it. The other component is a protocol connector to allow different types of application protocols to operate with the middleware directly. Figure 3.1 shows a diagram of the proposed middleware.

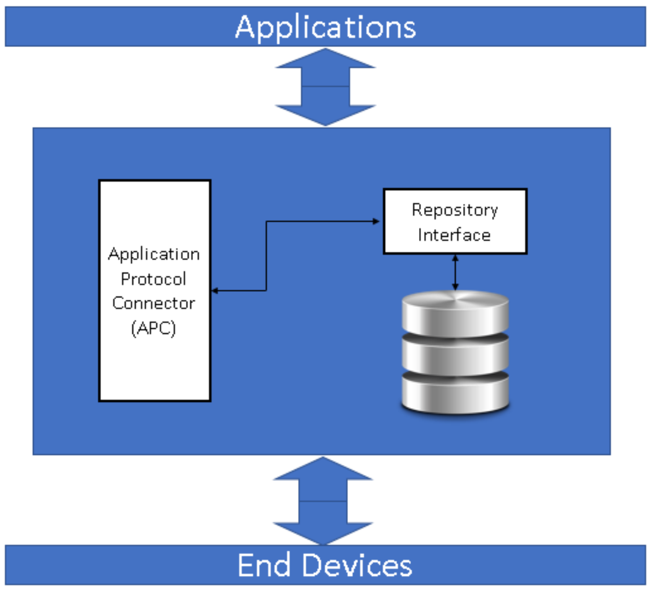


Figure 3.1 – Middleware functional diagram

The connector operates at the application protocol level since it is assumed that any end device running on the network will operate over TCP/IP as most devices, either resource-rich or resource-constrained, support any of the application protocols that run on top of TCP. Because of this connecting between devices is straightforward since all the different types of application ports talk to a central component that converts the specificities of the protocol to generate a common device and data model. Figure 3.2 shows how this connector operates.

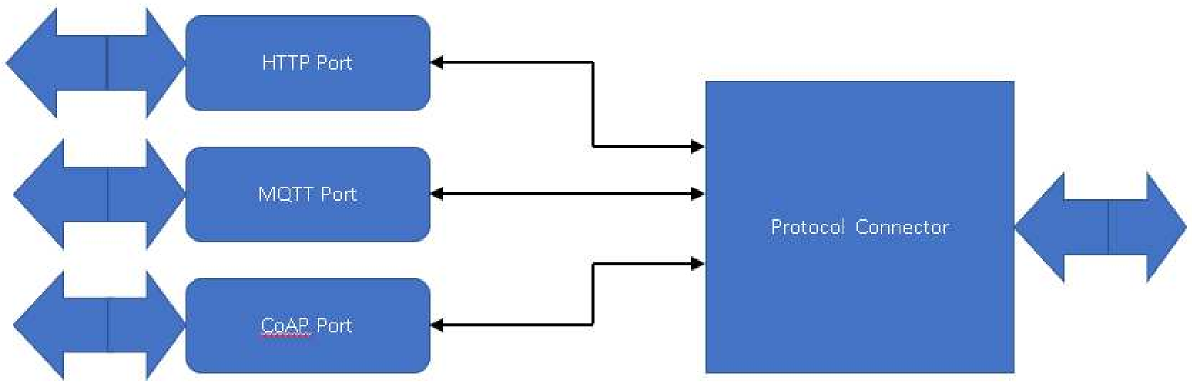


Figure 3.2 – Connector component architecture.

Although the middleware supports different types of protocols, special emphasis is made on MQTT as the main protocol for the end devices, and real time operation of the applications. The selection of MQTT over any other application protocol comes from the fact that it is currently widely used in the literature and accomplishes the following goals:

- Minimum number of components to start a network of IoT devices.
- Small code footprint and complexity to accommodate devices of any size in terms of memory and processing power.
- Ease of reach in regards of deployment over different locations.
- Because the addressing is done in regards of topics and not IP addresses, identification of individual devices can have a simpler structure and allows for a better organization of devices individually and in groups.
- Easy application deployment over different networks and different groups of devices, the data structure should be enough to know device capabilities.
- Storage capabilities of gathered data for any application that requires it. With different QoS it is possible to maintain the messages in the broker until all recipients get them.

Because of all of this it is assumed that most end devices will communicate over MQTT since the requirements for their operation are minimal. A quick survey of available libraries in different architectures reveal implementations for the most common one, of interest is the one for Arduino based on the AVR architecture. The PubSubClient library is particularly popular and well suited for IoT operation (*Arduino MQTT Client*, n.d.).

For deployments where all end devices are communicating via MQTT, any other proto-col connectors can act as bridges between the protocol in question and MQTT. Due to the low bandwidth operation of MQTT this doesn't impact negatively on the operation of the whole network and simplifies the structure of the central component as this would merge the MQTT port as well as the component that passes on the received data to the repository for storage or registration.

Independently of the protocol used the message's payload will be encoded in the JavaScript Object Notation (JSON). JSON is a format for storing and exchanging data with a human readable format, with the data organized as a key/value pair and an ordered list values (*JSON*, n.d.); JSON is a data format that is easy for devices to parse, and easy to programmatically assemble and send. The criteria used for selection of this data format is that is supported in most languages and in most architectures; it is also less verbose than XML, which means that it uses less memory, a condition critical for operation in resource-constrained devices. Finally, most JSON parsers tend to be faster and with a clearer syntax for a JSON structure than XML which alleviates the programmer's job.

The use of JSON allows the modeling of the end devices and applications with a com-mon language, while allowing for flexibility to accommodate heterogenous devices, which are common in IoT environments. To be able to model the devices and the data associated with them, the repository handles a resource tree structure, where every de-vice or application has a base element associated to it that contains all the information regarding that element. Figure 3.3 shows this data structure.

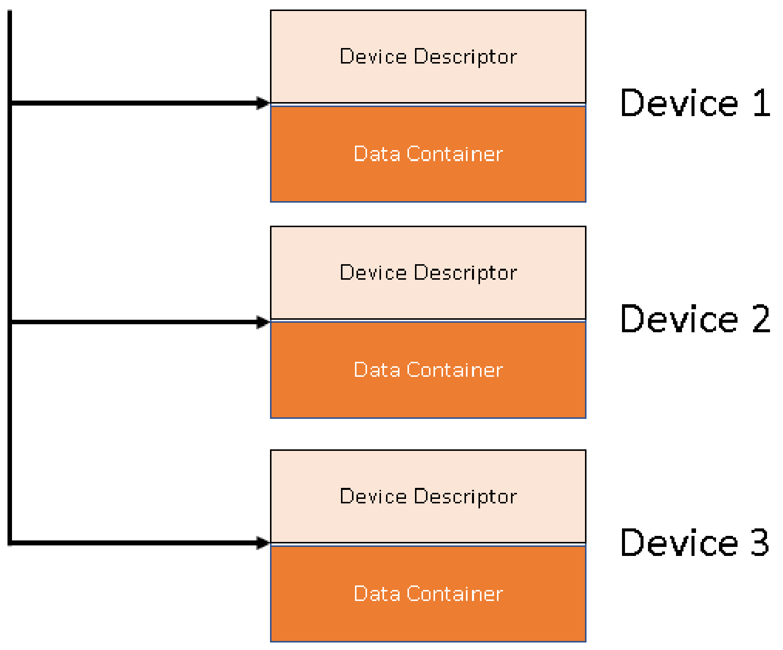


Figure 3.3 – Resource tree data structure

Each element within the resource tree contains two categories: an element descriptor which models within the middleware a real object or application, and a data container that has each data entry that the device or application wishes to write to the repository. The descriptor only has a few fixed elements, that indicate the id of the device, type of device and the actions that the device can receive (this would correspond to the commands); besides these fields, any device can add additional fields that increase the description of the device. Any end device or application that requests information on devices matching the criteria, would simply need to parse the fields of interest to know about that device.

The data container groups all individual data entries that any element (end device or application) sends to the middleware, the idea is that the element model extends the memory capacity of the real element through the data container, which allows for applications that require data aggregation and analysis through access to a single data repository. This will accomplish two things: all applications can access one point of entry without having to identify the individual containers of data, the second part is that through the use of the container the methods for accessing the data are uniformed which once again makes application integration easier.

Regarding the communication of end devices and applications to the middleware, a simple method for communicating with the middleware is employed that is reminiscent to the RESTful approach of HTTP; in it there are a limited number of simple operations that allow to read and write to the software (usually a server). The middleware uses two types of operations: set and get; with set being used to write data to the middleware and get being used to reading data from the middleware.

The process for every element within the network to operate follows this procedure: each element upon initiating is required to register with the middleware by sending a message encoded in JSON that has information of the element. Upon reception, the middleware sends an Acknowledgement indicating that the registration has been confirmed and that the element's information is stored for any other element or application to request it in the future. Figures 3.4 and 3.5 show the registration process and the contents of the JSON.

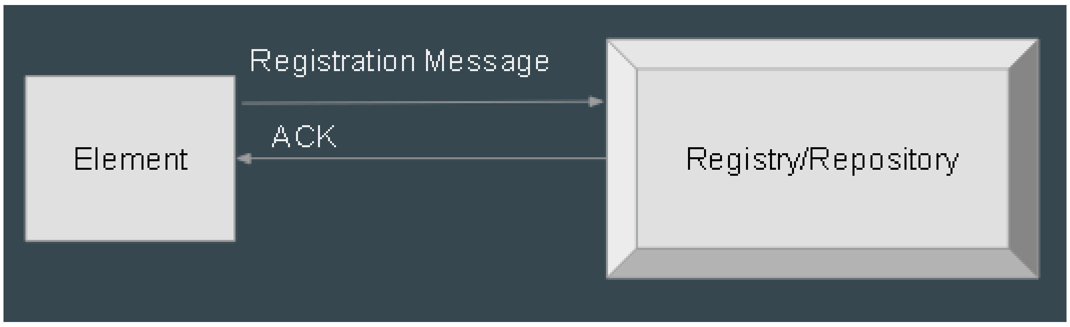
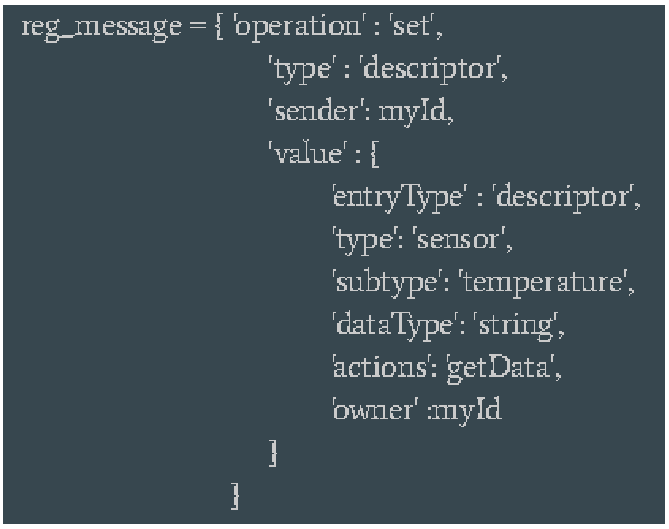


Figure 3.4 – Registration operation with the middleware.



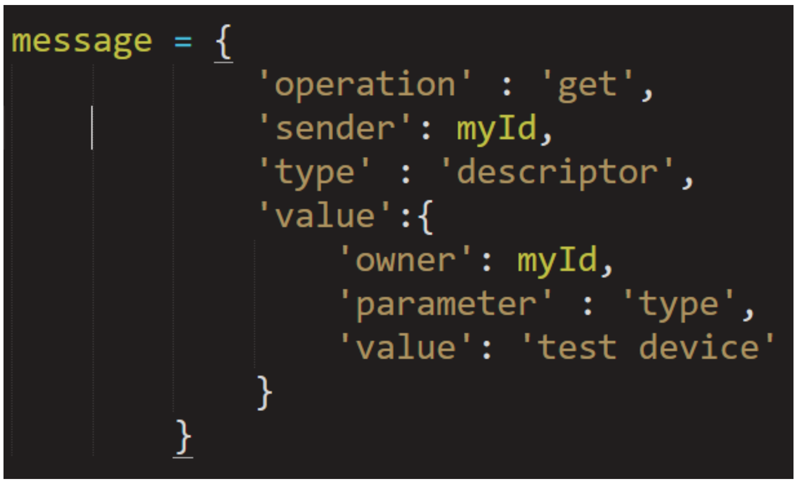
```
reg_message = { 'operation' : 'set',
                'type' : 'descriptor',
                'sender': myId,
                'value' : {
                     'entryType' : 'descriptor',
                     'type': 'sensor',
                     'subtype': 'temperature',
                     'dataType': 'string',
                     'actions': 'getData',
                     'owner' :myId
                }
             }
```

Figure 3.5 – Example of message encoded in JSON.

From figure 3.5, a registration is marked by the 'type' field of the data the encoding has, which shows that the element is trying to write a descriptor to the middleware which would signify a registration. The reply from the middleware would be an acknowledgement of the registration or a message indicating that the device has already been registered within the middleware. After the registration is done, the device can proceed to read and write data from the repository. The 'value' field in the message contains the actual descriptor that will be stored, in other cases this field will contain other types of objects that can be written in the middleware.

With the case of reading data from the middleware there are two different scenarios that must be considered: real time data sent directly by end devices, and historic data that is contained within the middleware.

The first step to do any type of operation with the devices and data in the network is to set up a registration or to send a get operation to the middleware. For example, to get real-time data from any device it is necessary to send a get message to the middle-ware with the 'type' field having the value "descriptor" in it, this instructs the middleware to return all devices that match the criteria located under the 'value' field. Once the application gets the information of all devices meeting the criteria it can then subscribe to their ids as topics on MQTT and receive the data readings directly. Figure 3.6 shows the message encoding for this type of search. It is important to note that MQTT is the protocol of choice for real time data operation due to the low overhead and asynchronous nature of the protocol, which makes any message sent from a device to be instantly available to any subscribers.



```
message = {
            'operation' : 'get',
            'sender': myId,
            'type' : 'descriptor',
            'value':{
                'owner': myId,
                'parameter' : 'type',
                'value': 'test device'
            }
        }
```

Figure 3.6 – Data encoding for searching devices.

The second type of get operation gathers information from the middleware directly from historical data that end devices might have sent. In this scenario, a get message is sent to the middleware with the parameters of the search as the 'value' field. The middleware will respond with the data in the 'value' field encoded in JSON inside of that field, after this point the application can extract the data for its own use.

Regarding the IDs of every element of the network, a three-field ID is used in the form of domain/group/device. This arrangement allows messages to be broadcasted to the whole domain of element (when using MQTT) or group or direct communication with the device.

4 IMPLEMENTATION AND ANALYSIS

To ensure that the proposals made in the previous chapter are valid, it is necessary to implement them and test the functionality of the proposed components, as well as its performance relative to other platforms with similar objectives.

This chapter presents such implementation and analysis of the workings of the proposal in the following manner: first the implementation of the proposed solution of the previous chapter is shown, this implementation considers a real world scenario that uses a combination of resource-rich and resource-constrained devices to understand the impact of the solution on a variety of conditions.

The second part includes an analysis of the results on this implementation to understand the impact the implementation decisions play in subjective analysis only. This implementation will be carried on to the next chapter as well, where more objective tests will be performed to fully asses the implications of the proposal.

In order to evaluate the middleware design, an IoT system will be implemented with several components that could be considered typical within any IoT application. To this end the middleware will connect to two different sensors: one temperature sensor that will report to an application, and a PIR to report to an actuator, to simulate a motion sensor and a lighting controller. Finally, a web application will serve as a Human Machine Interface (HMI) to the entire system.

To test the design in a more real environment the middleware will be hosted in a server operating in a different network, specifically in an Amazon Web Services (AWS) Virtual Private Server (VPS), which would allow IoT end devices from different locations to connect an interact on the same IoT application.

On the part of the end devices the implementation will consider two different sensors and one actuator. The idea is to provide different type of devices that would normally appear on regular IoT applications; towards this end, the two sensors will be resource-constrained devices and the actuator will consist of a resource-rich device. For the re-source-constrained devices two ESP8266 modules will be used because of the current popularity of such SoCs which makes them a great example of how a resource-constrained device operates within any IoT application.

For the actuator, a Raspberry Pi is selected; this sort of device can be considered re-source-rich because the processing and memory capabilities of the device allow it to run an operating system. A diagram showing a layout of the entire system being implemented in figure 4.1.

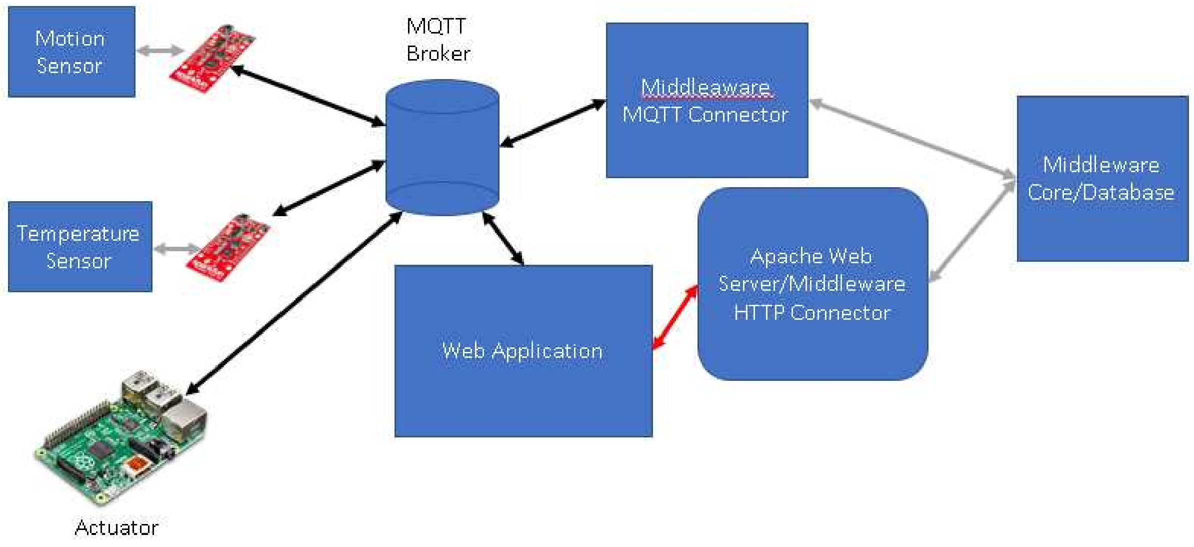
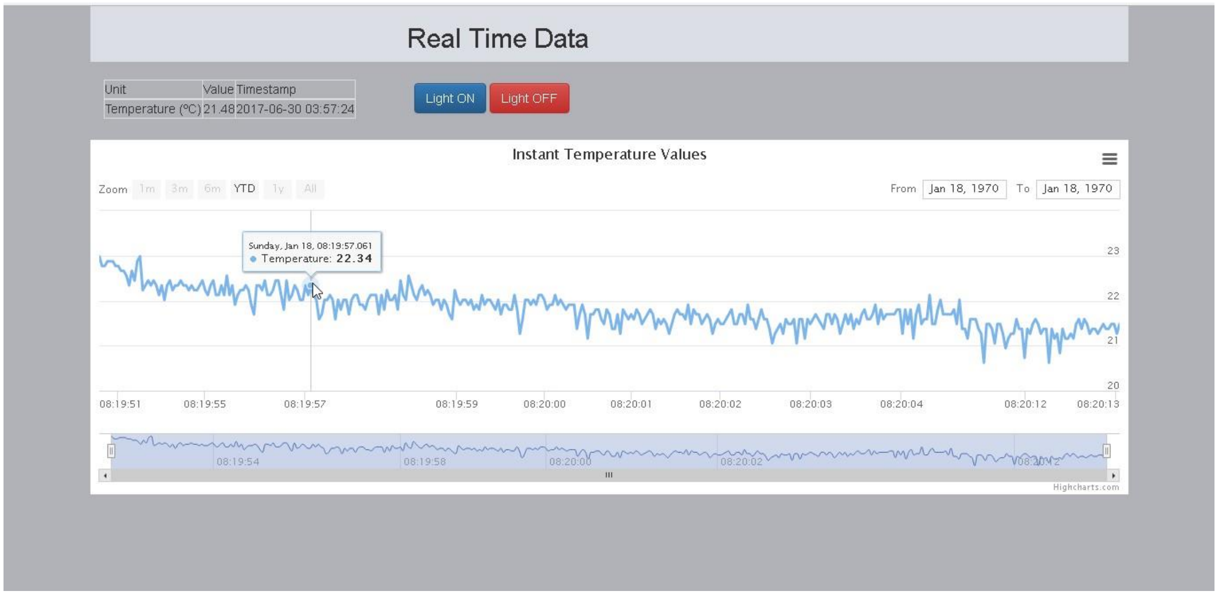


Figure 4.1 – Layout of the design's implementation.

As can be seen in figure 4.1 a web application will be used as the user interface for the system, with the ability to operate the actuator directly, as well as receiving temperature information in real time and getting historic data, in this case in graph form, any other input required could be easily programed into the web application directly. The web application interface is shown in figure 4.2.


Real Time Data

Unit	Value	Timestamp
Temperature (ºC)	21.48	2017-06-30 03:57:24

Light ON
Light OFF
Instant Temperature Values
Zoom 1m 3m 6m YTD 1y All
From Jan 18, 1970 To Jan 18, 1970
Sunday, Jan 18, 08:19:57.061
● Temperature: 22.34
Highcharts.com

Figure 4.2 – Web application interface.

The web application was written in JavaScript, which is the only language that natively supports front end web application development. The web application uses both HTTP and MQTT to interact with the middleware, leveraging both protocols strengths to different use cases, in the case of historic data which most likely will have a large payload, something for which HTTP was designed for. For the

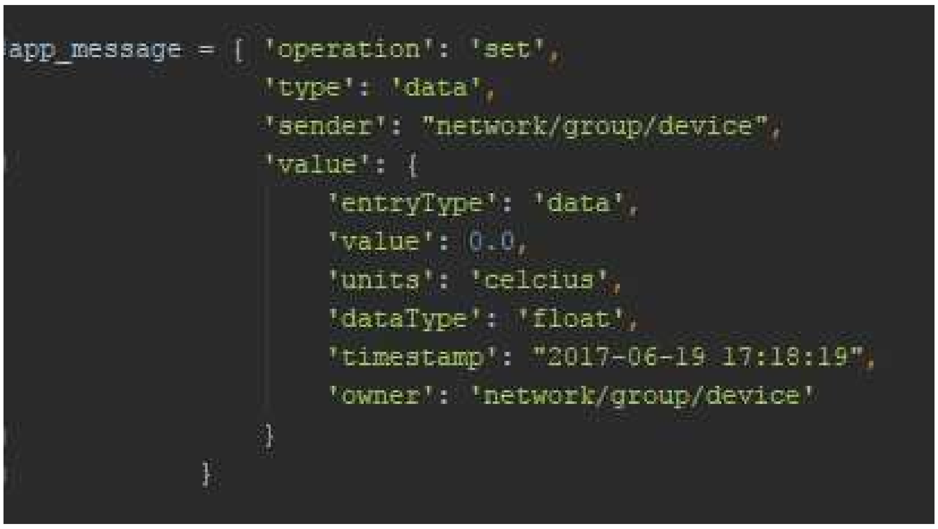
MQTT components, a broker must be used along with the MQTT connector. For the broker the Mosquitto broker was selected (A Light, 2017) due to it being open source, greatly documented and having ease of integration with different types of programming languages.

In the case of MQTT special consideration must be taken due to JavaScript not working on regular sockets and requiring web sockets to be implemented. This poses a problem to the middleware design as elaborated in chapter 3, since most traffic will come from port 1883 as a regular TCP socket. In order to overcome this, the MQTT broker that is implemented can listen to regular TCP sockets as well as web sockets, when a MQTT message is published via the TCP socket, it is also retransmitted to the web sockets. One issue with this approach is that the MQTT broker is unable to retransmit MQTT web socket messages to the regular TCP socket subscribers, one solution could be an external MQTT bridge to connect both types of messages, for the purposes of this solution such arrangement is unnecessary because the web application will send messages through HTTP and only listen to real time data over MQTT.

On the side of the middleware, this is implemented on an AWS VPS running Ubuntu, with two different connectors implemented: one for HTTP and one for MQTT. The HTTP connector was developed using the Flask framework for Python, because of Python's versatility and its ease of integration with the repository core, with an Apache server integration so it can handle traffic without issues.

The MQTT connector was implemented using the Paho MQTT module for Python (Python, n.d.). Once again, a Python implementation was chosen for ease of integration with the repository core. The MQTT connector works a subscriber to the whole MQTT application network using a topic in the form "network/#" where the # symbol implies that the sub-scriber will get any messages that start with the topic "network/". To determine whether the message was intended for the middleware, the full topic will be analyzed within the connector logic and compared to complete identifier for the middleware.

Since MQTT is not a protocol that directly supports a request/reply for a one on one communication, it must be adapted to allow steps like registration, that uses a request from an end device or application to the middleware and expect a reply from the middleware, to be per-formed along the normal operation of MQTT. The proposed modification uses the topic as the destination indicator, this would work as the equivalent of the address in similar protocols, which allows the subscribers to instantly know when a message is intended for them; while the sender identification is embedded within the payload as part of the message, this way if a reply is necessary, the client only needs to check the message to verify to whom the message must be sent to, this last part can be seen in figure 4.3 where part of the message indicates an owner field.



```
app_message = { 'operation': 'set',
                'type': 'data',
                'sender': "network/group/device",
                'value': {
                    'entryType': 'data',
                    'value': 0.0,
                    'units': 'celcius',
                    'dataType': 'float',
                    'timestamp': "2017-06-19 17:18:19",
                    'owner': 'network/group/device'
                }
            }
```

Figure 4.3 – Message structure for the application.

The repository core is the final part implemented in the middleware, this component is responsible for accessing the stored data on the middleware as well as registering new devices and fulfilling requests being forwarded by the different connectors. The core was written in Python which allows direct integration between the connectors and the core.

One of the main elements of the repository core is the database that contains all end de-vices and applications registered, as well as all the data sent by all the end devices. To accommodate the flexible and variable data that has to be stored, a NoSQL data-base is selected since it is better at handling unstructured and unpredictable data, even though it is expected that all the data will follow some sort of format, as explained before, because there are different types of end devices that generate data in different ways many more fields could appear for any one data entry, therefore it can be said that the data won't follow an extremely specific format.

To implement the database MongoDB was selected (MongoDB, 2018). MongoDB treats data entries as documents, with a group of documents being called a collection. The main advantage of this specific database for the implementation is that documents are stored in a dictionary or JSON format, which is a native type of data for Python; this makes integration of the database straight-forward while allowing the flexibility that the solution requires.

Communication between components of the implementation will proceed as follows: the temperature sensor will send its measured data to the middleware for storage, the movement sensor will send data whenever movement is detected, this will not be stored in the middleware; to signal this behavior a field within the payload of the message will indicate this. This difference in behavior shows that different applications require different ways of handling data, should every instance of presence needs to be stored in the middleware for analysis purposes, a change in the payload field is the only change needed for the payload to modify how this sensor data is handled.

The Raspberry Pi working as an actuator will, immediately after registering in the system, request all the registered sensors, matching the type of sensor and group that the actuator requires. In this case, there is only one movement sensor so the only criteria that will be searched is the type of sensor, which is expected to return just one element; however, it might be necessary to narrow the search criteria by location or other property defined in the descriptor, which would allow the actuator to react only to certain presence sensors that are near it. Once these initial steps are complete, the actuator operates under two different scenarios: the first is to modify the status of the LED depending on the values of the movement sensor, this allows for an "automatic" operation of the actuator. The other scenario is by commands sent from the web application; in this case the communication is sent directly from the application to the actuator via the HTTP connector which translates it to MQTT for transparent operation. Figure 4.4 shows a diagram illustrating the actuator's behavior.

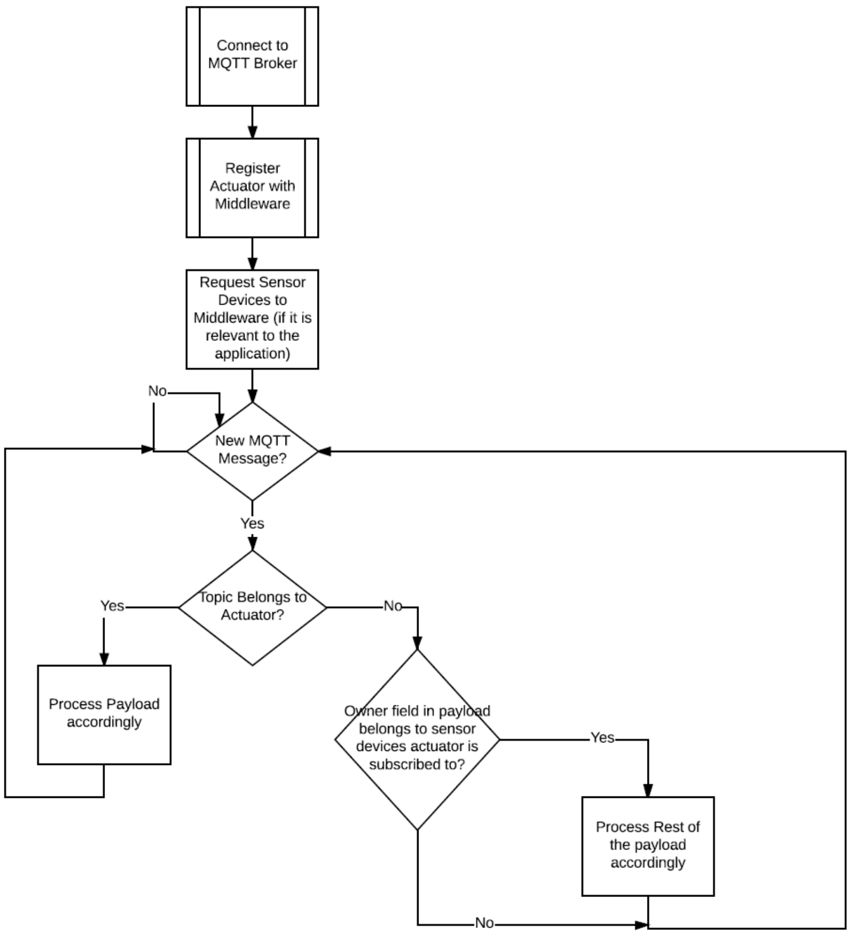
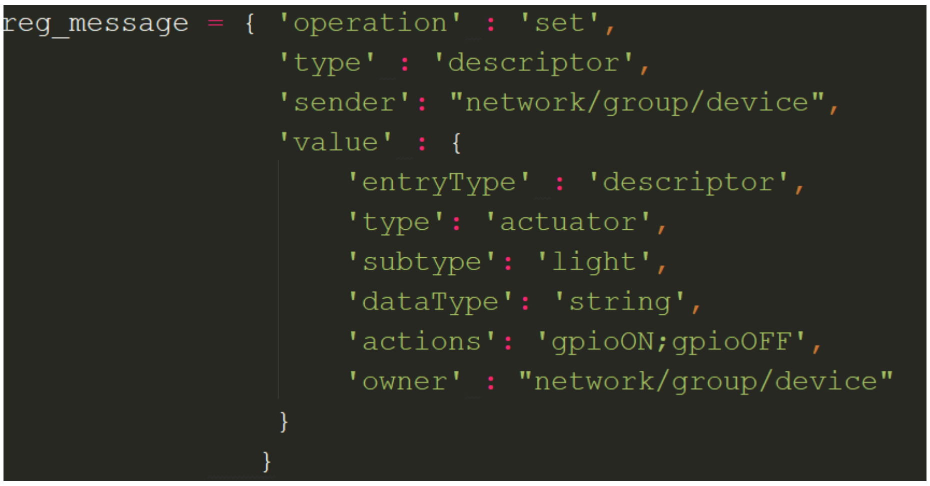

Connect to MQTT Broker
Register Actuator with Middleware
Request Sensor Devices to Middleware (if it is relevant to the application)
No
New MQTT Message?
Yes
Topic Belongs to Actuator?
Yes
No
Process Payload accordingly
Owner field in payload belongs to sensor devices actuator is subscribed to?
Yes
Process Rest of the payload accordingly
No

Figure 4.4 – Actuator's operating behavior

The last component is the web application, which is the interface with the users. The web application requests data on all temperature sensors available to the middleware, in this case the query would return only one element, as well as all actuators. In the case of the actuator, its registration exposes all the commands that the actuator can receive, figure 4.5 shows how this information is received by the web application.



```
reg_message = { 'operation' : 'set',
                'type' : 'descriptor',
                'sender': "network/group/device",
                'value' : {
                    'entryType' : 'descriptor',
                    'type': 'actuator',
                    'subtype': 'light',
                    'dataType': 'string',
                    'actions': 'gpioON;gpioOFF',
                    'owner' : "network/group/device"
                }
              }
```

Figure 4.5 – Actuator registration structure

As can be seen from this, the actuator has an actions field in its descriptor object that in-forms of the possible actions that can be performed on it. This allows any application requiring using a specific type of device to search for devices that not only match the type and subtype criteria, but that also have the actions desired for said application.

Regarding the temperature information, the web application parses each message received to determine whether the data is coming from the temperature sensor (since the sensor itself has the middleware id as its topic) and if it’s a message from the sensor uses it to display the current temperature. The web application also can get historical temperature data, this is done via the HTTP connector in the middleware, using a range of timestamps as the search criteria inside the repository.

5 RESULTS AND DISCUSSIONS

5.1 LATENCY AND JITTER TESTS

To ensure the implementation and design support the hypothesis presented, some tests have been conducted on the implemented system from the previous chapter, these tests focus mainly on the performance of the components. In this section latency and jitter, as well as power consumption tests, will be conducted to determine how the components perform within the IoT application.

One of the requirements for this IoT application is the ability to gather information in re-al time, this is particularly critical to actuators that act depending on conditions reported by other sensors. Real time requires that information from sensors arrive at the actuators almost instantly; because of this, any protocol to be used must ensure a very small time to arrival.

In order to verify that the protocols proposed in the implementation satisfy this re-al-time requirement a series of latency tests are conducted. The protocols to be analyzed are HTTP and MQTT, for both a python script runs several messages with the data structure showed in the previous section and measures the time the message arrives.

To ensure that latency is measured correctly and that the result can be comparable be-tween both protocols the MQTT implementation will consider a round trip for the message, that is, the publisher will be subscribed to the same topic it will be publishing to; this is required since every HTTP transaction has a request with a reply, something that MQTT was not de-signed to do.

Two different tests have been performed on both protocols: the first test will be done in a local area network, the objective in this case is to account for any effects elements outside the network could contribute to further latency. Figures 5.1 and 5.2 show the effects of the Latency in the local network settings.

In the case of the local area network test, the results will show a better performance with the HTTP protocol as opposed to MQTT. This is result is counterintuitive, since MQTT is expected to have better performance, however the explanation and results on the external networks will reinforce the preconception that MQTT performance is better, which will show that the asymmetrical nature of HTTP can improve its performance in local networks.

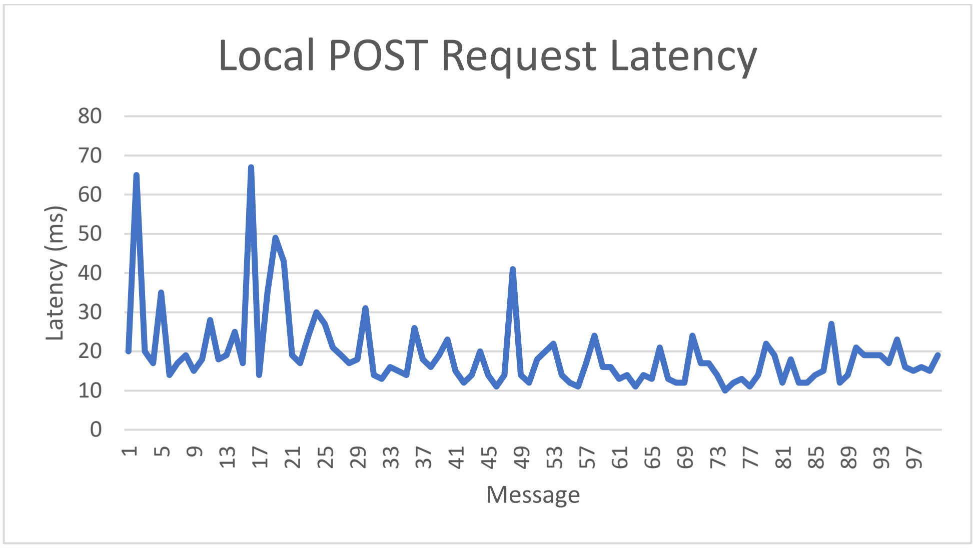


Figure 5.1 – HTTP POST request latency measured over local network.

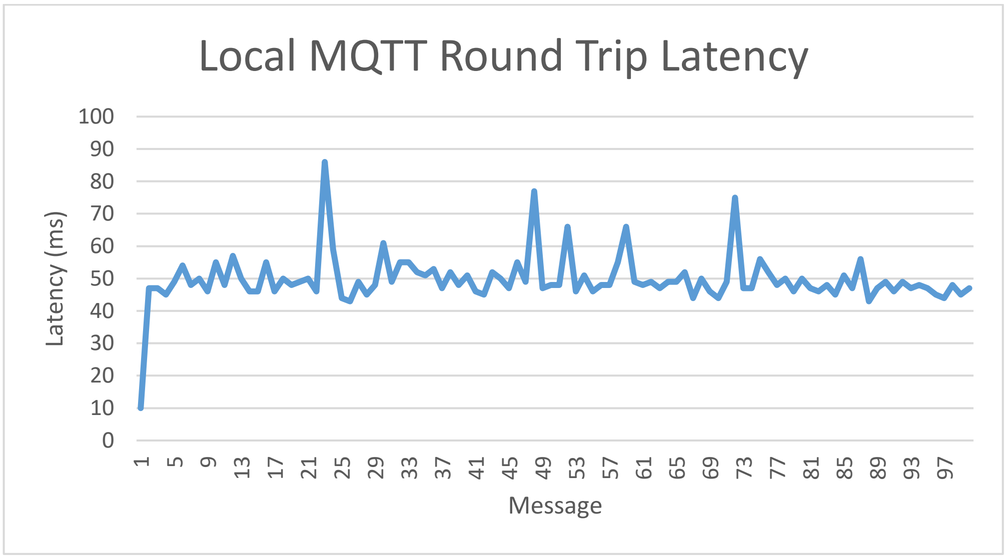


Figure 5.2 – MQTT round trip latency measured over local network

Over the local network the testing shows HTTP with a lower average value of 19.2 ms, compared to the MQTT roundtrip average value of 49.72 ms. As expressed before, this result seems to contradict what other works have already discover, such as (Yokotani & Sasaki, 2016). The reason for these results comes from the following fact: the payload in the MQTT operation is symmetrical, meaning that the amount of data in the response's body is the same as in the request's body. This doesn't happen in the HTTP request, since HTTP already considers two messages in every transmission: one for the client and one for the server, making the operation asymmetrical. In the case of MQTT we need to create this functionality.

An interesting metric that is also observed is that the minimum value for both protocols is 10 ms, which would suggest that even with more payload bytes on the round trip, the small overhead of MQTT messages is ideally suited for sensor applications and real-time information. These results complement the findings of (Yokotani & Sasaki, 2016) in which the overhead of HTTP is always greater than that of MQTT, with a comparative number of devices. The maximum values for both protocols show once again that since the HTTP operation is asymmetrical it will show a lower maximum value. Table 5.1 shows the results of latency over local network for both application protocols.

Table 5.1 – Latency comparison for both application protocols over a local

	HTTP	MQTT
Average Value (ms)	19.2	49.72
Minimum Value (ms)	10	10
Maximum Value (ms)	67	86

The second round of tests operate with a server/broker on a different network. Once again in the case of MQTT the tests will measure latency of a round trip, with the amount of payload bytes being sent the same as those being received. As in the case of the local network the software being run on both client/server publisher/broker/subscriber is the same so a comparison can be made.

In this second test, the layout of the experiment was done as follows: the client/publisher/subscriber devices were on the same network, while the server/broker was hosted on an Amazon Web Services Server, located in a different location. This layout was done to simulate what a more likely IoT application would encounter, since objects could be dispersed on different locations and the application would need to gather data from them remotely. One of the main objectives for this test is to check the reliability of MQTT for real-time operations; since it must be considered that because of its inherent characteristics, MQTT serves better for small low-bandwidth transmissions.

Figures 5.3 and 5.4 show the results for latency over different networks, once again for both protocols. On these tests, it can be shown that MQTT shows lower latency than HTTP, with values as low as 23 ms. The average latency value for MQTT is of 287.41 ms which is consider-ably lower than the 443.46 ms average value for HTTP; once again the reply bytes are lower than the transmission bytes for HTTP and both tests for the protocols are using the same payload when transmitting data.

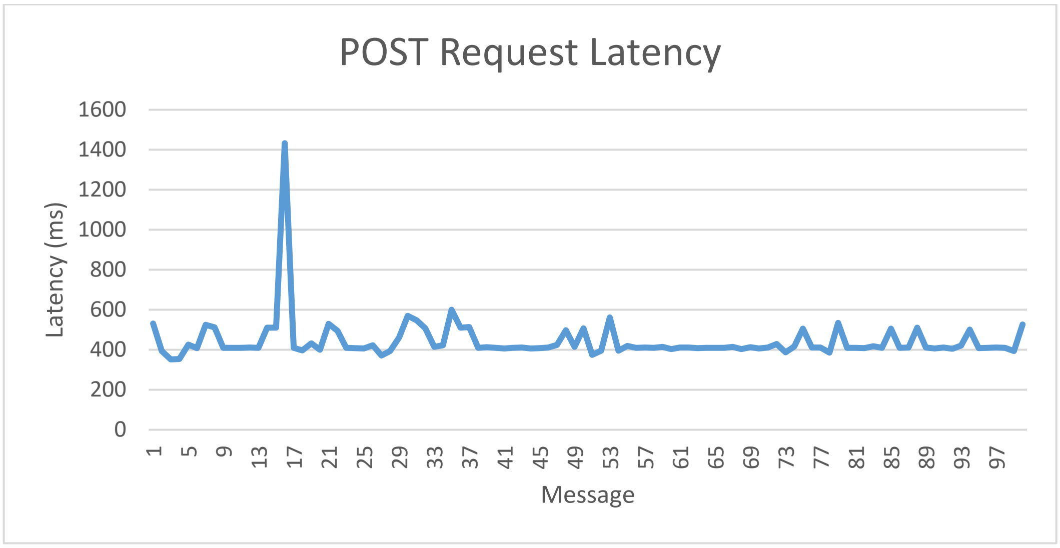


Figure 5.3 – POST Request latency over different networks.

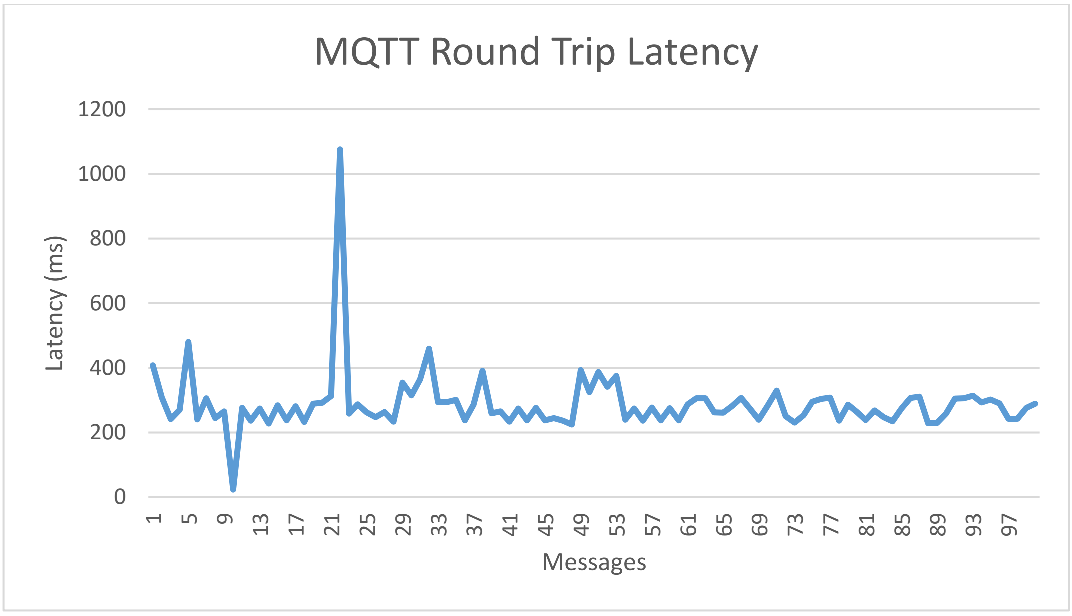


Figure 5.4 – MQTT Round Trip latency over different networks.

As can be seen from both figures the latency remains at a relative constant value for both protocols, with MQTT showing an overall smaller value, something desirable for real-time data delivery. Table 5.2 shows also the relevant information of these tests.

Table 5.2 – Latency measurements of HTTP and MQTT over different networks.

	HTTP	MQTT
Averge Value (ms)	443.46	287.41
Minimum Value (ms)	352	23
Maximum Value (ms)	1433	1076

As can be seen from comparing the values of table 5.1 with table 5.2, when used over different networks, MQTT has lower latency values; this most likely is due to the fact that the connection of MQTT is established once and any messages sent after are over an open connection, this reduces overhead which has a direct impact on latency. HTTP, on the other hand, has more overhead since for every message sent a connection must be established, over different networks this can increase latency.

Another parameter of interest is Jitter, which is the deviation from periodicity for every message sent, this is important for real-time applications because data is expected to be received at regular intervals. These tests were only conducted over MQTT to further analyze the capabilities of the protocol for resource-constrained end devices. A set of 100 messages were sent with-out delay between them and calculate the time passed between each message received, figure 5.5 shows each message and the time difference between them. As can be seen the jitter remains relatively constant with about 100 ms variation.

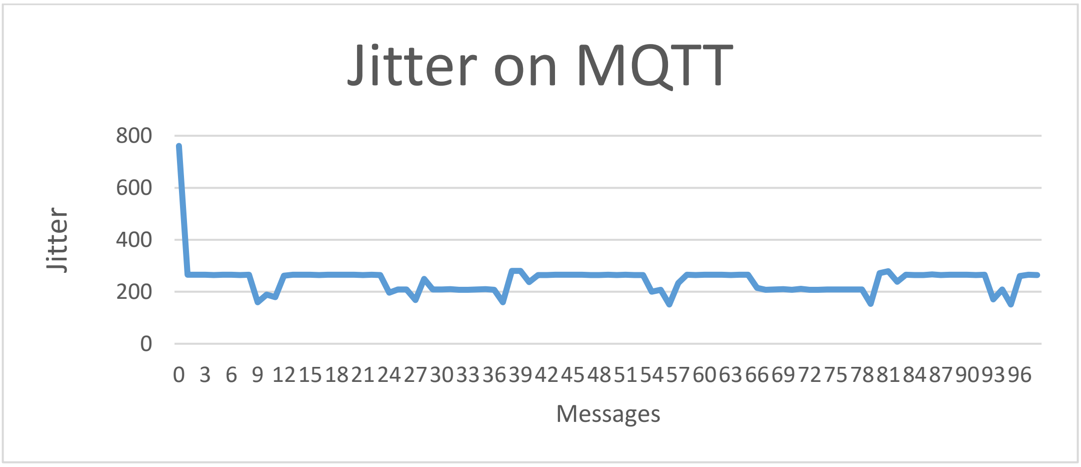


Figure 5.5– Jitter over MQTT.

With those same results, we use the Cumulative Distribution Function (CDF) to analyze the behavior of the jitter. The result of applying CDF to the data shows the percentage of messages that fall within a time difference. As can be seen, most messages have the same periodicity, something desirable for real-time operation, since the time of arrival for any message is relatively predictable. The CDF has been calculated for each message and plotted in figure 5.6.

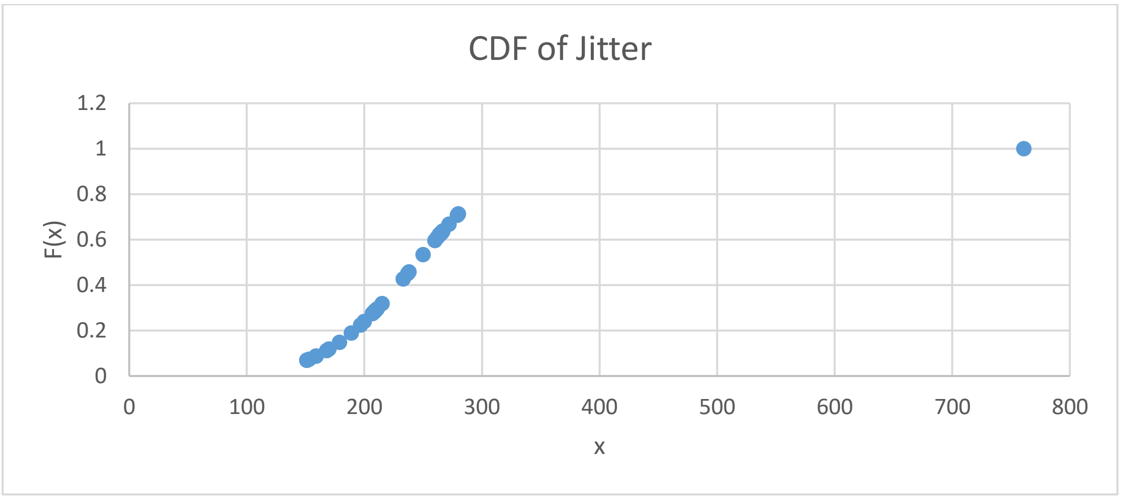


Figure 5.6 – CDF of Jitter.

Finally, energy consumption tests are performed on both application protocols to demonstrate their reliability for low-power applications. To conduct these tests, the Sparkfun ESP8266 Thing Dev board will be used; the choice for this board lies on the fact that currently the ESP8266 is one of the most popular System on Chip (SoC) being used in IoT applications, it has a microcontroller and a WiFi radio, it has low power modes and falls within the resource-constrained device category. Since the device can't run an operating system, power consumption can be attributed only to the software programed expressly for the tests, which allows for a clearer power consumption analysis.

For the test, a readout of the power line going to the device is made, using a shunt resistor to get the current flowing to it. Since the voltage values relative to current on the device are low (on the order of mV) the AD620 instrumentation amplifier will be used (Analog Devices, n.d.), with the circuit being shown in figure 5.7.

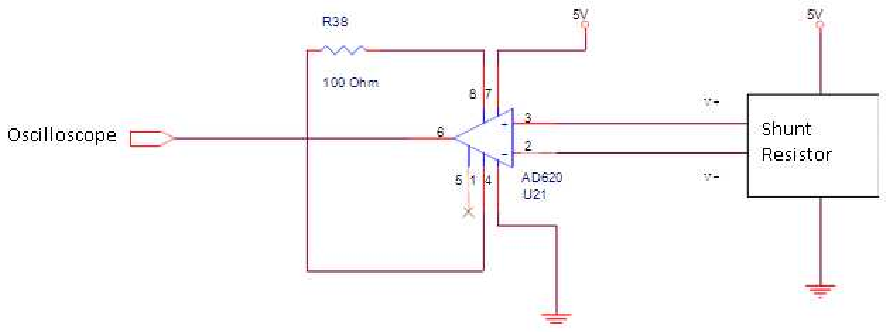


Figure 5.7 – Energy measurement circuit.

Since the element transmitting the data is the same for both application protocols, readings on electrical current while transmitting will be the same, which was measured to be of 16.21 mA. With the input voltage of 5 V, the power for the device when transmitting would be calculated by equation 5.1.

$$P = V * I \dots \dots eq. 5.1$$

Where V is the input voltage and the I is the electrical current flowing through the circuit, from the values measured it can be calculated that the power consumption when in operation to be of 81.05 mW, this will be independent of the protocol used, since this is the power consumed by the SoC itself when transmitting or receiving.

Since the interest lies in power consumed over time, because this metric can be used to calculate autonomy of the device on battery power, it is one of the more telling aspects for the choice of application protocol. In this case the time measured for the HTTP request was of 452 ms and the time for a round trip MQTT message was measured to be 151 ms, both values are consistent with the latency tests performed earlier.

With those values, the total power consumed during an HTTP request is equivalent to 36.6346 mW, in contrast an MQTT round trip only consumes 12.23855 mW of power, which is roughly a third of the value for HTTP. It is important to consider that the messages used for these tests are the same that were used for the latency tests, meaning that in the case of MQTT the message received was of the same size of the message being sent, a situation that wouldn't happen on the middleware implementation due to the reply message coming from the middleware being of a smaller size than the message being sent by the device.

A couple of observations that are important at this point: first is that the electrical current values obtain from the experiments differ from what can be found in the literature ((User), 2015). The explanation for this can be related to the design of the board used for the experiments, since capacitors placed on the board charge at the startup, they are probably creating a "cushion" for the current peaks on packet operations. Despite this "issue" with the test hardware, the results still maintain the trend observed in other works, in which the power requirements of MQTT are lower than HTTP, which supports the validity of the tests.

The second observation is that since MQTT maintains an active connection via a keepalive packet, it is completely feasible to use sleep functions in devices for a period smaller than the keepalive (which usually is around 5 minutes) and just wakeup to send the data. For specific values, the ESP8266 SoC has a deep sleep mode which has a current drain of about 10 uA, which is an extremely low drain, allowing battery applications to operate even with WiFi (which is usually considered a power intensive communications standard).

5.2 COMPARISON TO OTHER IOT MIDDLEWARE

In the current landscape there are many IoT platforms that promise some level of integration with different sensor types. Most major cloud vendors currently offer some IoT platform alternative, such examples are AWS IoT Core (Amazon, 2018) and Azure IoT Hub (Microsoft, 2016); while these commercial solutions exist, they present a challenge for comparison testing with our proposed middleware since they are closed sourced and operate on remote hardware that is unavailable for this work, this presents a challenge for testing since the ideal scenario for a comparison in the software implementation lies in using equivalent hardware.

Fortunately, there are open source implementations of IoT platforms that can be installed in any hardware of choice that can accomplish the same basic functionality as the commercial solutions. Table 5.3 shows the most popular IoT platforms to date.

Table 5.3 – Open source current IoT Platforms

Platform	Device Management	Protocols for Data Collection	Installation of Additional Protocols
Thingspeak	No	HTTP	No
DeviceHive	No	REST API, Websockets or MQTT	No
Thingsboard.io	Yes	MQTT, CoAP, and HTTP	No
OM2M	Yes	CoAP and HTTP	Yes

From all the platforms shown, of interest is the OM2M platform due to two rea-sons: first, it is a standard which is desirable in the sense that there’s a better chance of adoption and can have a larger user base. Second, the modular nature of OM2M allow for the inclusion of more protocols as plugins for the middleware, this in turn is comparable to what the proposed middleware allows.

Because of these reasons the platform of choice to make the comparison tests will be OneM2M [20], which currently is one of the most popular standards in use. Since OneM2M is a standard, an implementation of said standard is necessary for the tests, in this case the OM2M implementation will be used since it’s open source and has support from the Eclipse foundation.

In order to compare both middleware, qualitative and quantitative analyses will be performed like the ones presented in (Pereira et al., 2017). On the qualitative level, we can show that like OneM2M the middleware implemented in this work there is a communication model implemented that integrates different protocols with a single method of passing information, likewise the viability of the system is considerable as was demonstrated by the tests. Unfortunately, the middleware does not comply with the IoT-A requirements which OneM2M does.

However, for a more detailed approach some quantitative tests will be performed on both OM2M and the middleware described here and see how they compare. The information provided in (Pereira et al., 2017) details some tests that are performed, here some of those tests will be run that can provide information on the performance of the proposed solution in contrast to OM2M.

Two types of tests will be performed: the first will test the response time of the middleware with sequential requests, meaning that at any given point there will only be one request sent at the middleware. The round time for each request will be measured to understand how the middleware handles this. It is important to note that each request will simulate a sensor sending a data point, this is the most common case so it will illustrate better the performance of both middlewares.

The second test will evaluate how both platforms handle multiple requests simultaneously. In this test, response time will be measured again as well as successful requests and failed requests and response per second (RPS) to understand how much load both platforms can handle. It is important to note, although OneM2M supports the MQTT protocol, OM2M (which says it also supports it) has no simple way to implement it, and the documentation is not clear in this regard. Because of this all the tests will be done via HTTP requests, which shows similarities with the tests done in (Pereira et al., 2017).

The hardware used for both tests is a t2.micro server from Amazon Web Services (AWS), located in Oregon, United States. As per AWS specifications a t2.micro Virtual Private Server (VPS) has the following: 1 GiB of RAM Memory, 8GiB of storage and 1 Xeon E5-2670 v2 (Ivy Bridge). For comparison purposes both platforms will be running on the same VPS, to account for latency (distance from the server), and hardware capabilities.

For the first test 100 requests are sent sequentially to both platforms and the response time was measured, the results are shown in tables 5.4 and 5.5 and figures 5.8 and 5.9 for OM2M and the middleware respectively. As can be seen results for both platforms are close, with the response time for the proposed platform being slightly lower than the one for OM2M. Both requests use JSON as the payload's format. The difference in times responds to a lower number of headers being used in the proposed middleware as well as the number of bytes used in the response in OM2M, which is larger. It is worth mentioning that according to the tests conducted previously, it is more suitable to use MQTT for sensor data due to its lower response times.

Table 5.4 – Results for sequential testing of OM2M.

	Response Time (ms)
Averge Value	484.82
Minimum Value	460
Maximum Value	720

Table 5.5 – Results for sequential testing of proposed middleware.

	Response Time (ms)
Average Value	470.61
Minimum Value	451
Maximum Value	675

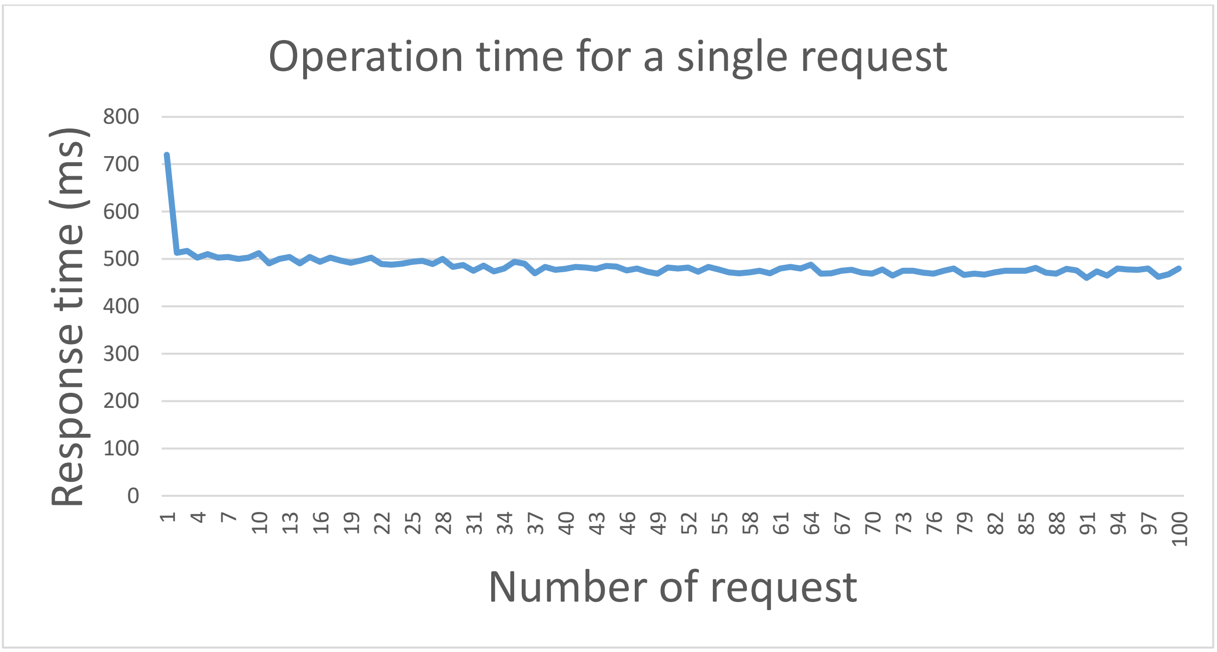


Figure 5.8 – Results for sequential testing of OM2M.

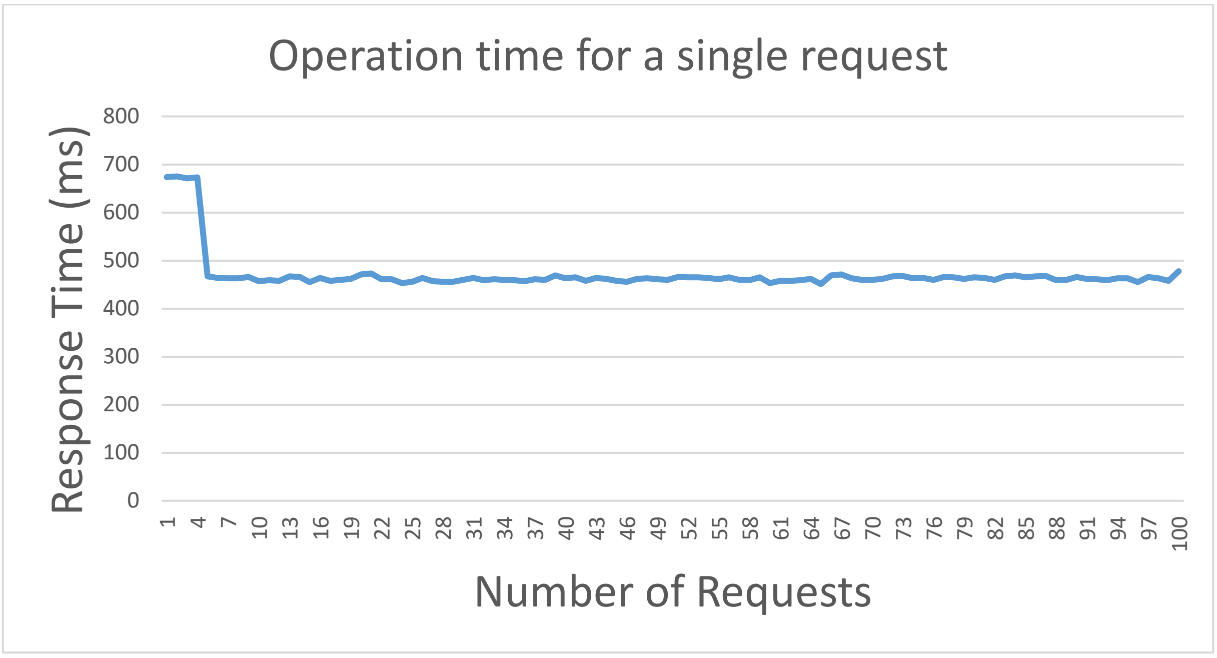


Figure 5.9 – Results for sequential testing of proposed platform.

For the parallel tests, multiple requests are sent in a predefined number of re-quests per second until the number of total requests sent is 1000 in all scenarios. The first scenario will send batches of 10 simultaneous requests for every second until 1000 requests have been sent, while the second scenario will consider 100 simultaneous requests per second. Tables 5.6 and 5.7 show the results for both scenarios on both platforms.

Table 5.6 – Results for parallel testing (10 requests per second).

Parameter	OM2M	Proposed
Successful Requests	746	1290
Failed Requests	1295	1234
Median Response Time	310	470
Average Response Time	1166	732
Min. Response Time	254	229
Max. Response Time	12980	9536
Average Content Size	571	64
Requests Per Second (RPS)	8.07	13.97

Table 5.7 – Results for parallel testing (100 requests per second).

Parameter	OM2M	Proposed
Successful Requests	86	1314
Failed Requests	998	22
Median Response Time	410	470
Average Response Time	864	479
Min. Response Time	269	229
Max. Response Time	3973	1518
Average Content Size	571	64
Requests Per Second (RPS)	2.89	156.86

For these tests, the locust.io (Heyman et al., n.d.) tool was employed. This is a Python based tool that allow multiple clients to be simulated over a period with as many simultaneous clients as needed, and all the behavior is developed via a Python script. The results shown con-sider not only response time, but also successful and failed requests, this part is important since response time would be measured only on successful requests, which could lead to incomplete results. From the tables it can be seen that the proposed middleware gets a better performance than OM2M, especially in terms of successful vs failed requests, one of the main reasons for this difference could lie in the way both platforms are implemented: while OM2M has a complete Java implementation, that also considers the web server and database internally as Java code, the proposed middleware uses off the shelf software such as the Apache software for the web server and MongoDB as the database. While the middleware core is designed in a modular way, which allow different elements to be swapped and could play into creating further delays (this explains in part why the median values for OM2M seem to be better), it also makes it better suited for handling higher traffic.

It is worth mentioning that hardware limitations also play an important role in the in the numbers presented in the results, since a bigger amount of memory and a higher number of processors could handle a higher traffic. However, from these results and the difference trend that both middleware handle traffic at different situations with the same hardware, it can be concluded that the proposed middleware handles client data with the same, or even greater, level of performance than a standard implementation such as OM2M for the OneM2M standard.

6 CONCLUSIONS

As it is presented in this work, the Internet of Things is poised to become one of the most important drivers in just about every human activity by interconnecting objects and providing more data and information about ourselves and our environment. Unfortunately, the current state of affairs make horizontal integration of applications extremely difficult, because of the different types of protocols and data formats that exist.

In this work, a framework for horizontal integration is proposed, with a middleware acting as the core component for such an integration, with the capacity to allow different data structures to be used thanks a to a simple but flexible format. As is seen from the design, the middleware can support multiple protocols thanks to its modular design which allows to adapt different types of protocol connectors to the middleware core. This is one of the first elements necessary for horizontal integration.

From the tests and implementation, it can be concluded that this framework also allows different types of devices to operate within the same system, another requirement for horizontal integration, since applications running on different types of platforms (web applications, mobile phone applications, etc.) need to coexist with the different types of sensors, which can be running on resource-rich or resource-constrained devices.

It is concluded as well that the framework works extremely well for resource-constrained devices due to the use and integration of different application protocols, which allows to select the most appropriate protocol for each type of device, depending on its function and resources available. Also, it is concluded as well that the support for different types of applications, namely real-time and ambient collection and analytics type of applications can be achieved again due to selection of appropriate protocols and the flexibility of the middleware core to store different types of data formats within a simple structure.

When compared to another mainstream platform, such as OneM2M, the framework, and the middleware that operates as the core component, is a suitable alternative for IoT deployment. It is of interest since compared to OneM2M, the code footprint for end devices, which are commonly resource-constrained sensors, is smaller and the need for middle elements such as gateways becomes less critical or necessary.

Finally, it is concluded that the framework, thanks to the use of a middleware as a central component, allow horizontal integration of applications to IoT systems without the need of deployment of all the layers of the application, something currently available in the traditional internet, which will allow even better and more widely use IoT applications.

7 RECOMMENDATIONS AND FUTURE WORK

Even though it has been shown that the middleware works as expected and pro-vides an effective way of integrating application into existing IoT infrastructure, improvements on the middleware and the framework can be made.

One such improvement should be the use of an element to assign an ID to every element that registers in an automatic way, this could work similarly as how DHCP operates in an Ethernet or WiFi network. Since there is a registration phase within the framework, it is entirely possible to add an auto-assign element that could give IDs to every element that requests such functionality.

Another future work to be consider is the addition of more application protocols and the corresponding analysis of how additional protocols impact performance on the middleware. Finally, the use of other standards such as IEEE 802.15.4 to analyze ease of integration to IoT applications using more widely used standards such as IEEE 802.3 and IEEE 802.11.

BIBLIOGRAPHY

(User), E. (2015). *ESP8266 Power Consumption - ESP8266 Developer Zone*. Http://Bbs.Espressif.Com Forum. http://bbs.espressif.com/viewtopic.php?t=133

A Light, R. (2017). Mosquitto: server and client implementation of the MQTT protocol. *The Journal of Open Source Software*, *2*(13). https://doi.org/10.21105/joss.00265

Al-Fuqaha, A., Guizani, M., Mohammadi, M., Aledhari, M., & Ayyash, M. (2015). Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications. *IEEE Communications Surveys and Tutorials*, *17*(4), 2347–2376. https://doi.org/10.1109/COMST.2015.2444095

Al-Fuqaha, A., Khreishah, A., Guizani, M., Rayes, A., & Mohammadi, M. (2015). Toward better horizontal integration among IoT services. *IEEE Communications Magazine*, *53*(9), 72–79. https://doi.org/10.1109/MCOM.2015.7263375

Amazon. (2018). *AWS IoT Core Overview - Amazon Web Services*. Aws. https://aws.amazon.com/iot-core/

Analog Devices. (n.d.). *AD620 Datasheet*. Retrieved June 1, 2017, from http://users.ece.utexas.edu/~valvano/Datasheets/AD620.pdf

Arduino MQTT Client. (n.d.). Retrieved May 18, 2017, from http://www.hivemq.com/blog/mqtt-client-library-encyclopedia-arduino-pubsubclient/

Asghar, M. H., & Mohammadzadeh, N. (2016). Design and simulation of energy efficiency in node based on MQTT protocol in Internet of Things. *Proceedings of the 2015 International Conference on Green Computing and Internet of Things, ICGCIoT 2015*. https://doi.org/10.1109/ICGCIoT.2015.7380689

Bondarevs, A., Huss, P., Ye, Q., & Gong, S. (2017). Universal Internet of Things Solution : Protocol Independent. *Industrial Technology (ICIT), 2017 IEEE International Conference On*, 1313–1318.

Brian Krebs. (2016). *Hacked Cameras, DVRs Powered Today's Massive Internet Outage — Krebs on Security*. Krebs on Security. https://krebsonsecurity.com/2016/10/hacked-cameras-dvrs-powered-todays-massive-internet-outage/

Campo, A. Del, Gambi, E., Montanini, L., Perla, D., Raffaeli, L., & Spinsante, S. (2016). *MQTT in AAL Systems for Home Monitoring of People With Dementia*. 86–91. https://doi.org/10.1109/PIMRC.2016.7794566

Chen, H. W., & Lin, F. J. (2014). Converging MQTT resources in ETSI standards based M2M platform. *Proceedings - 2014 IEEE International Conference on Internet of Things, IThings 2014, 2014 IEEE International Conference on Green Computing and Communications, GreenCom 2014 and 2014 IEEE International Conference on Cyber-Physical-Social Computing, CPS 20*, 292–295. https://doi.org/10.1109/iThings.2014.52

Collina, M., Bartolucci, M., Vanelli-Coralli, A., & Corazza, G. E. (2014). Internet of Things application layer protocol analysis over error and delay prone links. *2014 7th Advanced Satellite Multimedia Systems Conference and the 13th Signal Processing for Space Communications Workshop, ASMS/SPSC 2014*, *2014-Janua*, 398–404. https://doi.org/10.1109/ASMS-SPSC.2014.6934573

Datta, S. K., & Bonnet, C. (2016). Describing things in the Internet of Things: From CoRE link format to semantic based descriptions. *2016 IEEE International Conference on Consumer Electronics-Taiwan, ICCE-TW 2016*, *i*, 0–1. https://doi.org/10.1109/ICCE-TW.2016.7520965

Doukas, C., Capra, L., Antonelli, F., Jaupaj, E., Tamilin, A., & Carreras, I. (2015). Providing generic support for IoT and M2M for mobile devices. *Proceedings - 2015 IEEE RIVF International Conference on Computing and Communication Technologies: Research, Innovation, and Vision for Future, IEEE RIVF 2015*. https://doi.org/10.1109/RIVF.2015.7049898

Evans, D. (2011). The Internet of Things - How the Next Evolution of the Internet is Changing Everything. *CISCO White Paper*, *April*, 1–11. https://doi.org/10.1109/IEEESTD.2007.373646

Heyman, J., Hamrén, J., Heyman, H., & Byström, C. (n.d.). *Locust - A modern load testing framework*. Retrieved September 3, 2017, from http://locust.io/

Jara, A. J., Martinez-Julia, P., & Skarmeta, A. (2012). Light-weight multicast DNS and DNS-SD (lmDNS-SD): IPv6-based resource and service discovery for the web of things. *Proceedings - 6th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing, IMIS 2012*, 731–738. https://doi.org/10.1109/IMIS.2012.200

JSON. (n.d.). Retrieved May 18, 2017, from http://www.json.org/

Kamilaris, A., Gao, F., Prenafeta-Boldu, F. X., & Ali, M. I. (2016). Agri-IoT: A semantic framework for Internet of Things-enabled smart farming applications. *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)*, 442–447. https://doi.org/10.1109/WF-IoT.2016.7845467

Klauck, R., & Kirsche, M. (2012). Chatty Things - Making the Internet of Things Readily Usable for the Masses with XMPP. *Proceedings of the 8th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing*, 60–69. https://doi.org/10.4108/icst.collaboratecom.2012.250464

Koch, D. B., & Ph, D. (2017). *An Internet of Things Approach to Electrical Power Monitoring and Outage Reporting*. 5–7.

Luzuriaga, J. E., Cano, J. C., Calafate, C., Manzoni, P., Perez, M., & Boronat, P. (2015). Handling mobility in IoT applications using the MQTT protocol. *2015 Internet Technologies and Applications, ITA 2015 - Proceedings of the 6th International Conference*. https://doi.org/10.1109/ITechA.2015.7317403

Luzuriaga, J. E., Perez, M., Boronat, P., Cano, J. C., Calafate, C., & Manzoni, P. (2015). A comparative evaluation of AMQP and MQTT protocols over unstable and mobile networks. *2015 12th Annual IEEE Consumer Communications and Networking Conference, CCNC 2015*, 931–936. https://doi.org/10.1109/CCNC.2015.7158101

Mektoubi, A., Hassani, H. L., Belhadaoui, H., Rifi, M., & Zakari, A. (2016). New approach for securing communication over MQTT protocol A comparaison between RSA and Elliptic Curve. *2016 Third International Conference on Systems of Collaboration (SysCo)*, *0*, 1–6. https://doi.org/10.1109/SYSCO.2016.7831326

Microsoft. (2016). *Azure IoT Hub | Microsoft Azure*. https://doi.org/10.1038/sj.bjp.0703864

Mohalik, S. K., Narendra, N. C., Badrinath, R., Jayaraman, M. B., & Padala, C. (2016). *Dynamic Semantic Interoperability of Control in IoT-based Systems : Need for Adaptive Middleware*. 199–203.

MongoDB. (2018). *MongoDB for GIANT Ideas | MongoDB*. MongoDB. https://www.mongodb.com/

Nicholas, S. (n.d.). *Power Profiling: HTTPS Long Polling vs. MQTT with SSL, on Android*. Retrieved May 14, 2017, from http://stephendnicholas.com/posts/power-profiling-mqtt-vs-https

oneM2M. (2019). *oneM2M - Home*. http://www.onem2m.org/

OpenM2M. (2015). *Eclipse OM2M - Open Source platform for M2M communication*. Eclipse.Org. http://www.eclipse.org/om2m/

OpenMTC. (2016). *Boosting the development of innovative M2M and IoT applications*. http://www.openmtc.org/

Pereira, C., Aguiar, A., & Morla, R. (2017). Benchmarking IoT Middleware Platforms. *2017 IEEE 18th International Symposium on World of Wireless, Mobile and Multimedia Networks (WoWMoM)*.

Python. (n.d.). *paho-mqtt 1.1 : Python Package Index*. Retrieved June 17, 2017, from https://pypi.python.org/pypi/paho-mqtt/1.1

Raspberry Pi - Teach, Learn, and Make with Raspberry Pi. (n.d.). Retrieved May 16, 2017, from https://www.raspberrypi.org/

Singh, M., Rajan, M. A., Shivraj, V. L., & Balamuralidhar, P. (2015). Secure MQTT for Internet of Things (IoT). *Proceedings - 2015 5th International Conference on Communication Systems and Network Technologies, CSNT 2015*. https://doi.org/10.1109/CSNT.2015.16

Taylor, S. (2013). The next generation of the internet: Revolutionizing the Way We Work, Live, Play, and Learn. *CISCO White Paper*, *April*, 1–17.

Thota, P., & Kim, Y. (2016). *Implementation and Comparison of M2M Protocols for Internet of Things*. https://doi.org/10.1109/ACIT-CSII-BCD.2016.20

Wagle, S. (n.d.). *Semantic Data Extraction over MQTT for IoTcentric Wireless Sensor Networks*.

Yokotani, T., & Sasaki, Y. (2016). Comparison with HTTP and MQTT on Required Network Resources for IoT. *The 2016 International Conference on Control, Electronics, Renewable Energy and Communications (ICCEREC) In*.