

Analysis of Power Consumption of the MQTT Protocol

by

Abhishek Viswanathan

Bachelor of Engineering – Electronics & Telecommunication, Mumbai University, 2015
MS - Telecommunication, School of Information Sciences, University of Pittsburgh, 2017

Submitted to the Graduate Faculty of
School of Information Sciences, in partial fulfillment
of the requirements for the degree of
Master of Science Telecommunications

University of Pittsburgh

2017

UNIVERSITY OF PITTSBURGH

School of Information Sciences

This thesis was presented

by

Abhishek Viswanathan

It was defended on

April 25, 2017

and approved by

Dr. David Tipper, PhD, Professor

Dr. Mai Abdelhakim, PhD, Visiting Assistant Professor

Thesis Advisor: Dr. Prashant Krishnamurthy, PhD, Associate Professor

Copyright © by Abhishek Viswanathan

2017

Analysis of Power Consumption of the MQTT Protocol

Abhishek Viswanathan, MST

University of Pittsburgh, 2017

With the exponential growth of the Internet of Things (IoT), there is a need to assess the different tradeoffs that exist in this realm of resource-constraints. Since it is touted as the protocol of IoT, it is imperative to explore MQTT in depth, analyzing the different conditions under which it might function favorably. Given the high importance of power in IoT devices, this thesis aims to shed light on some of the factors that might affect the power consumption and the different tradeoffs that exist when using the MQTT protocol.

MQTT, or MQ Telemetry Transport, is an open source protocol that operates on the publish/subscribe model for constrained devices. It provides messaging transport on top of the Transmission Control Protocol (TCP) in environments where networks have low bandwidth and high latency.

This thesis contains the results and inferences after varying the Quality of Service levels, Payload Sizes and implementing Authentication Mechanisms while using the MQTT protocol on a Raspberry Pi. It is hoped that the data from these experiments can be used to better predict the requirements of IoT systems.

TABLE OF CONTENTS

1.0	INTRODUCTION.....	1
2.0	BACKGROUND	8
2.1	IOT AND POWER	8
2.2	HARDWARE	10
2.2.1	Raspberry Pi	10
2.2.2	Power Measurement Devices	13
	2.2.2.1 Belkin Conserve Insight Energy Use Monitor: F7C005Q.....	14
	2.2.2.2 P3 P4400 Kill A Watt Electricity Usage Monitor.....	14
	2.2.2.3 DROK Pocket Digital Multimeter USB	14
2.3	PROTOCOLS	15
2.3.1	MQTT	15
	2.3.1.1 Quality of Service (QoS)	16
	2.3.1.2 KeepAlive	19
	2.3.1.3 Clean Session / Persistent Session.....	20
	2.3.1.4 Publishing and Subscribing to Topics	20
	2.3.1.5 TLS	22
	2.3.1.6 Authentication Mechanism	23
2.4	TRADEOFF BETWEEN PERFORMANCE AND SECURITY IN IOT	23

2.5	SOFTWARE	25
2.5.1	Raspbian Jessie	25
2.5.2	Mosquitto.....	25
2.6	RELATED WORK.....	25
3.0	EXPERIMENT DESIGN	29
3.1	SETUP	29
3.2	LIMITATIONS.....	31
3.3	PARAMETERS VARIED.....	32
3.3.1	Quality of Service.....	32
3.3.2	Number of Publishers.....	35
3.3.3	Payload Size.....	36
3.3.4	Authentication Mechanism	36
4.0	RESULTS, ANALYSIS & DISCUSSION	38
4.1	RESULTS	38
4.1.1	Quality of Service.....	38
4.1.1.1	Scenario: 1 Subscriber, 1 Publisher for Different QoS Levels	39
4.1.1.2	Scenario: 1 Subscriber, 2 Publishers for Different QoS Levels.....	41
4.1.1.3	Scenario: 1 Subscriber, 3 Publishers For Different QoS Levels.....	42
4.1.1.4	Scenario: 1 Subscriber, 5 Publishers for Different QoS Levels.....	43
4.1.2	Average Number of Messages Received Per Minute.....	44
4.1.3	Number of Publishers.....	45
4.1.4	Amount of Data Received by a Subscribing Client	47
4.1.5	Average Energy Consumed Per Publisher	48

4.1.6	Total Energy Consumed by Broker	49
4.1.7	Payload Size.....	50
4.1.8	Authentication Mechanism.....	55
4.2	ANALYSIS & DISCUSSION	57
5.0	CONCLUSION.....	61
	APPENDIX.....	66

LIST OF TABLES

Table 1: Raspberry Pi 3 Model B Specifications.....	11
Table 2: Current Drawn by Raspberry Pi 3 Model B	12

LIST OF FIGURES

Figure 1: IoT Layers	4
Figure 2: Quality of Service Levels in MQTT.....	17
Figure 3: QoS 2 Flow.....	19
Figure 4: MQTT Publishing.....	21
Figure 5: MQTT Subscribe Flow.....	22
Figure 6: Architecture of Experiment Setup.....	30
Figure 7: QoS-0 Communication Messages	33
Figure 8: QoS-1 Communication Messages	34
Figure 9: QoS-2 Communication Messages	35
Figure 10: Control Readings	39
Figure 11: QoS-0 Subscription	40
Figure 12: QoS-1 Subscription	40
Figure 13: QoS-2 Subscription	40
Figure 14: QoS Comparison for 1 Subscriber, 1 Publisher	41
Figure 15: QoS Comparison for 1 Subscriber, 2 Publishers.....	42
Figure 16: QoS Comparison for 1 Subscriber, 3 Publishers.....	43
Figure 17: QoS Comparison for 1 Subscriber, 5 Publishers.....	44

Figure 18: Average Number of Messages Per Minute.....	45
Figure 19: QoS-0: Number of Publishers Comparison.....	46
Figure 20: QoS-1: Number of Publishers Comparison.....	46
Figure 21: QoS-2: Number of Publishers Comparison.....	47
Figure 22: Data Received by Subscriber	48
Figure 23: Average Energy Consumed Per Publisher	49
Figure 24: Total Energy Consumed.....	50
Figure 25: 1MB Payload Power.....	51
Figure 26: 2MB Payload Power.....	51
Figure 27: 5MB Payload Power.....	52
Figure 28: 10MB Payload Power.....	52
Figure 29: Variable Payload Power Comparison.....	53
Figure 30: Average Amount of Data Transferred in 1 Minute	54
Figure 31: Average Data Transferred	54
Figure 32: Power Consumption with/without Authentication Mechanism	56
Figure 33: Data Transferred with/without Authentication Mechanism	56

1.0 INTRODUCTION

The MQTT protocol has been proclaimed as “the protocol” for the Internet of Things by the open standards body, OASIS [1] and a major technology company, IBM [2]. It has been touted as the lower power alternative to HTTP and other IoT protocols (Constrained Application Protocol - CoAP, Advanced Messaging Queueing Protocol - AMQP, etc.), but just how low-power is it? With a wide array of parameters to vary, how does MQTT perform in terms of power consumption, to meet different test environments? This thesis aims to answer some of those questions.

Invented in 1999, this protocol was not intended to be the protocol for what we know today as the Internet of Things [3]. It was invented to create a protocol that provided minimal battery loss and used minimal bandwidth for connecting oil pipelines over a satellite connection. Its goals were to be an easy to implement protocol that provided Quality of Service Data Delivery and to be bandwidth efficient and data agnostic while maintaining continuous “session awareness”¹. It also had to be lightweight and easy to implement.

While these remain the goals of the protocol, its application is not limited to connecting oil pipelines anymore, and now, it is a major driving protocol of IoT services and devices. Before

¹ If an edge-of-network device loses connectivity, all subscribed clients will be notified with the “Last Will and Testament” feature of the MQTT server so that any authorized client in the system can publish a new value back to the edge-of-network device, maintaining bidirectional connectivity.

looking at the protocol in depth, it is important to put it into context and look at the bigger picture of IoT, its prevalence, importance and impact on our world today and in the future, to understand why the protocols that drive it must be examined with rigor.

The seemingly sudden emergence of IoT has been many years in the making, as new technologies emerged and conditions become more favorable for enhanced connectivity.

The multinational technology conglomerate, Cisco, points out the reason for the emergence of IoT technologies succinctly [4]. Since the cost and size of wireless radios has significantly dropped and IPv6 expanded the number of devices that could be assigned a global communication address, more devices began to be shipped with inbuilt Wi-Fi and cellular wireless connectivity. With improvements being made to battery technology, devices are also becoming more power-efficient and location agnostic.

Predictions about the growth of IoT are plentiful, with almost every technological giant jumping onto the bandwagon to not miss out on the immense potential. Cisco's Internet of Things Group (IOTG) predicts that there will be over 50 billion connected devices by 2020. The American research and advisory firm, Gartner, Inc. forecasts that 8.4 billion connected things will be in use worldwide in 2017, up 31 percent from 2016, and will reach 20.4 billion by 2020. Total spending on endpoints and services related to IoT will reach almost \$2 trillion in 2017 [5].

With the industry growing at a rapid pace, there is an urgent need for risk assessments and a focus on the security and performance of IoT devices. The focus in digital security projects is moving toward detection and response. The increasing complexity of the environment requires a multifaceted approach to dealing with the security and performance of both individual devices as well as the system as a whole [6].

It is then imperative to break down IoT into its layers, to understand the security and performance requirements in each of them.

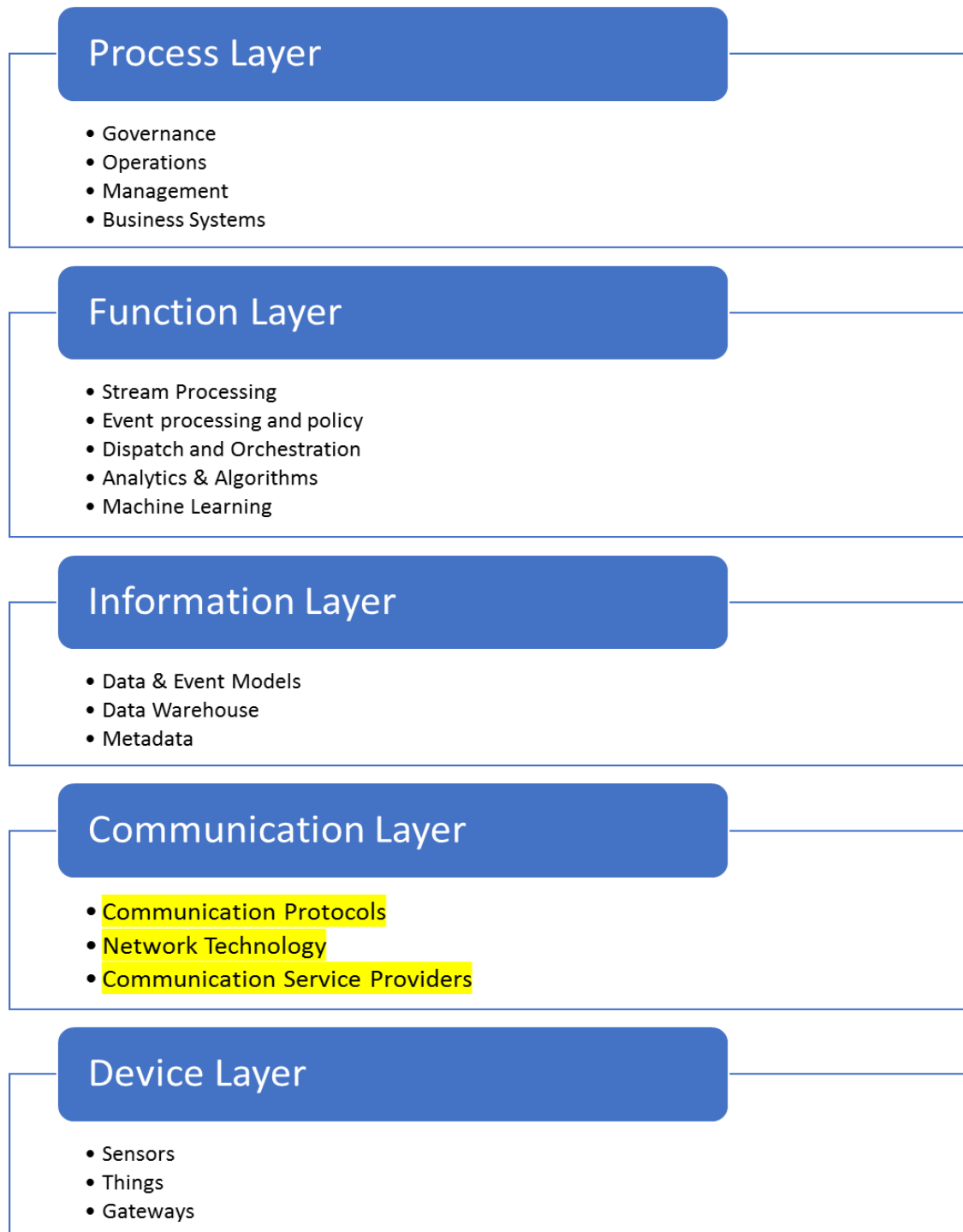


Figure 1: IoT Layers

We can refer to the Gartner IoT reference model [7], reproduced in Figure 1, to gain some clarity about the different functions of the various layers involved in the IoT architecture. Each layer highlights the major aspects through which data flows, to help us understand them.

In the IoT ecosystem, it is very difficult to create an end-to-end model given the diversity of systems being designed and the presence of *things* that are basically loose-ends in most models. However, by defining the various functions of components in IoT and grouping them together, we can analyze the individual layers better and optimize them for use.

Although a comprehensive end-to-end security and performance solution is ideal, this thesis focuses primarily on the communication layer of the IoT stack as depicted in Figure 1. The Communication Layer defines the communication protocols, network technologies and communications service providers (CSPs) necessary for the IoT system, along with the security protocols and mechanisms, if present. More specifically, this paper focuses on the MQTT protocol which is one of the data transfer protocols used commonly in IoT systems. MQTT is used in notifications for the social media platform, Facebook, for push-style messaging in low power mobile devices, monitoring and controlling SCADA equipment and a host of other real-world applications.

There are several issues concerning power consumption of IoT devices, considering a lot of Wireless Sensor Networks are deployed in remote locations where power is scarce and a lot of considerations need to be made to maximize the power efficiency of the devices. In this thesis, we will be examining the effect of the MQTT protocol on the power consumption of an IoT device. The experiments carried out are over a WiFi link; additional link-layer technologies like Bluetooth and Zigbee are out of the scope of this thesis, even though they are prevalent technologies that are used in IoT systems. A typical use-case of IoT systems is many sensors

publishing data to a broker. In this thesis, we have tried to emulate the sensors using multiple instances on a Raspberry Pi that publish data to the broker, instead of using multiple, distinct sensors. By measuring the power consumed by a Raspberry Pi that is running the Mosquitto MQTT broker, we can observe the changes in the power consumption when the test conditions change.

Thus, the real problem that this thesis aims to identify and elaborate is how the different parameters of the MQTT protocol affect the power consumption of a remotely placed Raspberry Pi based broker, and whether it can be quantified and analyzed.

In Chapter 2, this thesis explores the current state of the IoT ecosystem and the need to research power consumption in IoT devices given the different tradeoffs that need to be examined for IoT systems to function efficiently. It also elaborates on the background of the devices that are used in the experimental setup as well as the protocol that is being examined, MQTT. This chapter elucidates the different parameters that can be varied when using the protocol and explores related work.

In Chapter 3, the experimental setup is discussed. The reasons for picking the parameters that are varied are explained along with the results that are expected before performing the experiments. The limitations of the experiment design are also discussed in this chapter.

Chapter 4 contains the results, analysis of the results and some discussion about the trends in power consumption of the MQTT Broker device that are observed after analysis. The different readings are put forth along with observations and comparisons.

Chapter 5 concludes this thesis with a look at the findings from the experiments, the limitations of the results from them, and scope for further research in this sphere in the future.

There are many different energy consumption issues that need to be addressed in the realm of IoT. The objective of this thesis is not to address all of them or to compare the different protocols, different transport technologies or different devices, but to observe the differences in the power consumed when different parameters of the MQTT protocol are varied.

2.0 BACKGROUND

2.1 IOT AND POWER

Selecting a wireless network for an IoT device involves balancing many conflicting requirements, such as range, battery life, bandwidth, density (number of connected devices in an area), endpoint cost and operational cost. There is an important cluster of IOT networking devices that focuses on short-range, low-bandwidth, extended battery life, medium density devices, as in the case of smart homes or smart offices, that use star or mesh topologies. Some of these networks implement higher levels mechanisms, such as authentication and security.

It has been predicted that low-power, short-range networks will dominate wireless IoT connectivity through 2025, far outnumbering connections using wide-area IoT networks [8].

The key difference between the internet and IoT is that IoT devices are typically much more constrained in their resources than conventional internet devices. They typically have less memory, less bandwidth, less processing power, less available energy and thus, must use less power.

There are several ways in which IoT devices can be powered [9]:

- AC or DC lines: Although these supply a seemingly infinite source of power to IoT devices, they also severely limit the mobility of these devices. For AC lines,

an AC/DC converter will be required to power the device and these increase the costs of the system as well.

- Energy Harvesting: In systems where it can be implemented successfully, energy harvesting is a good solution for powering IoT devices. However, it is often impractical because there is no consistent or reliable source of energy that can be used.
- Battery: Although the eventual replacement of batteries as a power source for IoT devices makes them seem like an unattractive option, they provide the flexibility in placement as well as a stable power source for extended periods of time, if the battery is chosen correctly. Since most IoT devices that are deployed in the field typically draw minimal power, batteries are often chosen as the primary power source. After carefully selecting batteries based on their operating mode, temperature, self-discharge rate and its relation to the application of the IoT system, batteries can provide power to IoT devices for several years before they need to be replaced.

For this reason, it is important to know how much power these devices consume for different test-cases. Although the choice of hardware, software, protocols and link-layer technologies can have a significant impact on the power consumed by the setup, we can observe general trends for a specific hardware, software and protocol working over a specific link-layer technology and extrapolate the results and findings to similar use-cases. This is what this thesis aims to do.

2.2 HARDWARE

2.2.1 Raspberry Pi

With the advent of the digital age, it became necessary for more people across the world to have access to computers. Initially designed to teach computer science in schools in developing countries, the Raspberry Pi ² has grown much larger than the company expected, finding applicability in Robotics, Teaching, Astronomy and the Internet of Things.

The Raspberry Pi is a small, single-board computer, which has the capability to be used as a traditional computer with the right peripheral components. The Raspberry Pi, although built for other purposes, fits perfectly into the IoT ecosystem because of its low-cost, low-power and great potential for performing computing tasks and connectivity to various types of sensors.

IoT hobbyists use the Raspberry Pi extensively for projects in building smart systems to automate tasks. Since it is lightweight, inexpensive, easy-to-use and capable of connecting to networks (Bluetooth, Wi-Fi, Ethernet), it is used to perform processing of data from sensors (among other things) and either store it or upload it the internet.

For the experiments carried out in this thesis, the Raspberry Pi used is the Raspberry Pi 3 Model B. The specifications are outlined in Table 1.

² <https://www.raspberrypi.org/>

Table 1: Raspberry Pi 3 Model B Specifications

CPU	4× ARM Cortex-A53, 1.2GHz
GPU	Broadcom VideoCore IV
RAM	1GB LPDDR2 (900 MHz)
Networking	10/100 Ethernet, 2.4GHz 802.11n wireless
Bluetooth	Bluetooth 4.1 Classic, Bluetooth Low Energy

Unlike the previous models, this model comes with inbuilt Bluetooth and WiFi capabilities. For this thesis, we will explore the usage of the MQTT protocol over WiFi.

We remotely connect to the Raspberry Pi (wirelessly) over Secure Shell (SSH) to minimize the power lost through peripheral devices like monitors, keyboards and a mouse. The Raspberry Pi will act as a remote device: you can connect to it using a client on another machine. SSH is built into Linux distributions and Mac OS. For Windows and mobile devices, third-party SSH clients are available [10].

The Raspberry Pi 3 is powered by a +5.1V micro USB supply. The amount of current (in mA) that is used, depends on the application. A 2.5A power supply is sufficient for any applications that can run on the Raspberry Pi safely. Typically, the model B uses between 700-1000mA depending on what peripherals are connected. The maximum current the Raspberry Pi can draw is 1 Amp.

The power requirements of the Raspberry Pi increase as you make use of the various interfaces on the Raspberry Pi [11]. Table 2 compares the amount of power drawn in terms of the

current in amps under different situations, released by the Raspberry Pi Foundation [12], and the values in Watt derived if the device uses 5V.

Table 2: Current Drawn by Raspberry Pi 3 Model B

		Pi3 B (Amps)	Pi3 B (Watts)
Boot	Max	0.75	3.75
	Avg	0.35	1.75
Idle	Avg	0.30	1.5
	Max	0.55	2.75
Video playback (H.264)	Avg	0.33	1.65
	Max	1.34	6.7
Stress	Max	1.34	6.7
	Avg	0.85	4.25

The values in Table 2 were obtained under test conditions with the Raspberry Pi connected to an HDMI monitor, USB Keyboard and mouse, and connected to a WiFi access point. However, this does not provide any insight into how much power the Raspberry Pi will draw without any peripherals and when there is data being transferred over WiFi. In a typical IoT use-case, it is unlikely that each device will be connected to peripheral devices like an HDMI monitor, keyboard and mouse. Instead, one may expect a battery powered Raspberry Pi deployed potentially in remote areas.

This thesis tries to discover how changes in the protocol (MQTT) parameters affect the power consumption of a Raspberry Pi running without being connected by wires to any peripheral devices, except a power supply or battery pack.

2.2.2 Power Measurement Devices

To ensure reasonably accurate power consumption measurements, the values of power are measured using 3 different measurement devices that are available commercially. Since the current and voltage to be measured are relatively small (compared to household appliances), these three devices have been considered to ensure that the readings are verified across multiple instruments.

2.2.2.1 Belkin Conserve Insight Energy Use Monitor: F7C005Q

This device enables users to find out how much energy is drawn from a wall socket. This monitor provides the user with instantaneous power (watts). It also projects monthly and yearly power usage, based on actual values if plugged in over a period of time.

The continuous electrical rating is 15A/120V~/60Hz/1800W.

2.2.2.2 P3 P4400 Kill A Watt Electricity Usage Monitor

This device connects to a wall socket and allows users to plug in their devices to assess their power usage and efficiency by monitoring voltage, line frequency and power factor. It displays volts, amps and wattage within 0.2% accuracy.

The continuous electrical rating is 15A/125V~/60Hz/1875W

2.2.2.3 DROK Pocket Digital Multimeter USB

This device connects to a USB port, and allows users to measure the instantaneous power, current (0.5% accuracy), voltage (0.3% accuracy) and capacitance of any device being powered through the USB port in this device. By averaging out instantaneous readings over a period of time, users can calculate the average power drawn.

The continuous electrical rating is 3A/13V~/30W

2.3 PROTOCOLS

The protocols that are used in IoT (including MQTT), fit into the communication layer of the IoT stack, along with network technologies, communications service providers and in some cases, security mechanisms.

Wireless sensor networks & IoT systems often have overlapping definitions. Research has also stated that Wireless Sensor Networks are one of the most important elements in the IoT paradigm and there has been a call for integration [13] of Wireless Sensor Networks into IoT.

A typical Wireless Sensor Network consists of sensor nodes and gateways. The gateway receives data from the sensor nodes and then aggregates it and sends the data to a server or a broker [14]. This environment requires an energy and bandwidth efficient protocol that will effectively transfer data from a resource-constrained gateway to a server.

M2M or Machine-to-machine systems have specialized requirements for data transfer like multicast support, low overhead and simplicity for constrained environments [15]. This is where protocols like MQTT and CoAP come into the ecosystem. The widespread and quick evolution of devices that are ‘smart’ and have back-end applications has created the need for these protocols that specifically serve an M2M communication system [16].

There has been a fair amount of research comparing the different protocols that are used in IoT. This is further elaborated in section 2.6 of this document.

2.3.1 MQTT

MQTT is a machine-to-machine (M2M)/"Internet of Things" connectivity protocol that was designed as an extremely lightweight publish/subscribe messaging transport. It is extremely

useful in applications where a connection with a device in a remote location is required. In these cases, the protocols themselves must have low overhead and must use limited bandwidth, thus consuming lower power than other protocols in their class.

MQTT is a Client-Server publish/subscribe messaging transport protocol. MQTT is lightweight, open, simple, and designed to be easy to implement. The protocol runs over any protocol that provides ordered, lossless, bi-directional connections (mostly Transmission Control Protocol/Internet Protocol - TCP/IP). MQTT provides different Quality of Service levels for different use-cases, is data-agnostic, and provides a publish-subscribe architecture that allows for the decoupling of applications and provides multicasting of messages. Its most important feature is the low transport overhead it provides for efficient communication between devices.

MQTT has been called the protocol for the Internet of Things due to its ability to be bandwidth and power efficient, although it has a lot of parameters that are variable, so the degrees to which it consumes power could be very different. This thesis aims to identify which factors play an important role in determining how much power is used by the protocol for standard applications.

This section explores the different variable parameters that might have an impact on the power consumption of the device.

2.3.1.1 Quality of Service (QoS)

QoS is an important feature of MQTT since it simplifies communication in unreliable networks as the protocol is responsible for handling retransmissions and guarantees the delivery of a message regardless of the reliability of the underlying transport layer.

It also allows for clients to choose the QoS they desire based on their application and network infrastructure.

MQTT offers three different QoS levels [17]:

- **QoS 0** (At most once)
- **QoS 1** (At least once)
- **QoS 2** (Exactly once)

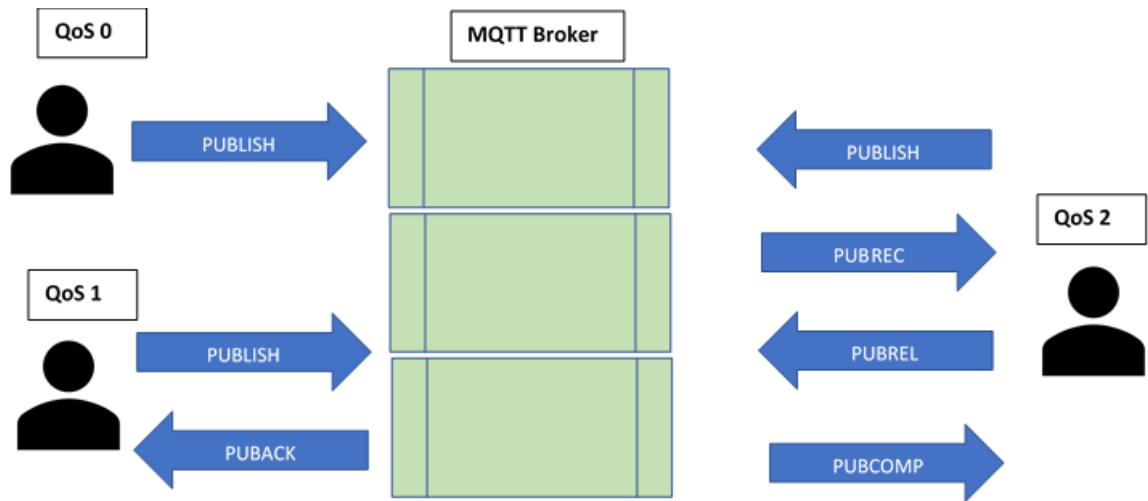


Figure 2: Quality of Service Levels in MQTT

Figure 2 shows a schematic of the way the various QoS levels works in MQTT [18].

It is important to note that the QoS level for publishing client to broker depends on the QoS that the client sets for a particular message. When the broker sends a message to a subscribing client, it is sent with the QoS level of the subscription made earlier by the client. Therefore, it is possible for a QoS level to be downgraded for clients that subscribe with a lower QoS. For the purposes of the experiments carried out for this thesis, the QoS levels of the subscribing clients and publishing clients are the same.

QoS 0

This level contains the least overhead and the protocol provides “best-effort” delivery. Messages are not acknowledged by the receiver and senders will not store and redeliver the messages. This level provides the same guarantees as that of the TCP protocol underneath it. It is the fastest mode of transfer.

QoS 1

This level ensures that the message is delivered at least once. If the sender does not receive an acknowledgement that the message has been received, the sender will set the DUP (duplicate) flag and repeatedly send the message until an acknowledgement is received.

Since the message must be sent repeatedly to the receiver in case of a failed transmission, the message must be stored locally at the sender. The message is deleted once the sender receives the acknowledgement from the receiver that the message has been received.

QoS 2

This is the highest level of QoS that MQTT offers. It guarantees that each message is received exactly once by the receiver. It is the slowest QoS level, but also the most reliable.

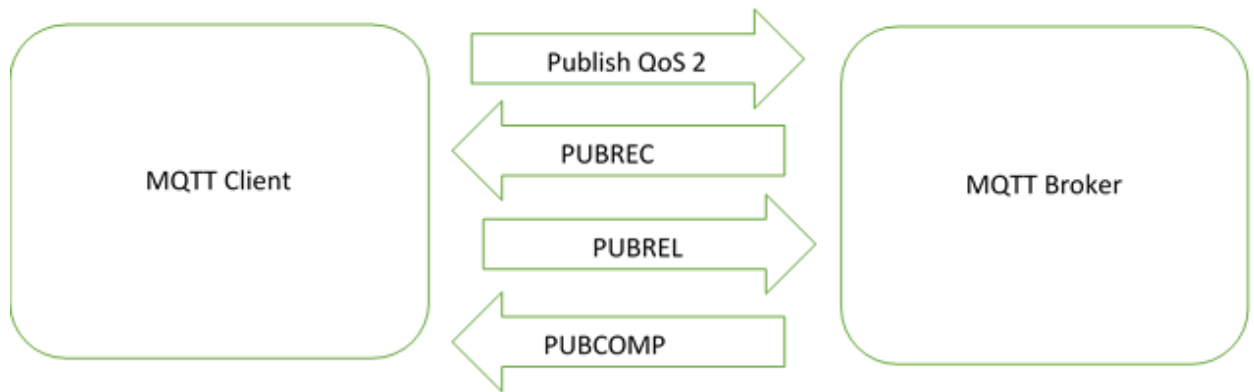


Figure 3: QoS 2 Flow

This level of QoS is used when it is critical to the application that each message is received exactly once. This level is used when a duplicate delivery would hinder the application of the system itself. There is a lot of overhead involved in the message exchange, however, the overhead is the cost of reliable delivery. The communication process is shown in Figure 3.

2.3.1.2 KeepAlive

The KeepAlive functionality of MQTT checks that the connection between the broker and the client is open, during periods when the messages being transmitted between them are relatively infrequent. The KeepAlive functionality may initially seem unnecessary over a TCP connection, however, Andy Stanford-Clark, the inventor of MQTT explains:

“Although TCP/IP in theory notifies you when a socket breaks, in practice, particularly on things like mobile and satellite links, which often “fake” TCP over the air and put headers back on at each end, it’s quite possible for a TCP session to “black hole”, i.e. it appears to be open still, but in fact is just dumping anything you write to it onto the floor [19].”

The broker must disconnect any client which does not reply to a KeepAlive message (PINGREQ) or any other message in one and a half times of the KeepAlive interval that is chosen. Similarly, the client must close the connection if it does not receive a similar reply from the broker in that same time interval. The KeepAlive value is set by the client based on its own notions of its signal strength and stability of the connection. If the KeepAlive interval is set to 0, the entire KeepAlive mechanism is deactivated.

2.3.1.3 Clean Session / Persistent Session

A client has the ability to request a persistent session when it is first connecting to the broker. If the *cleanSession* flag is set to True, then the client does not have a persistent session and any information is lost when the client disconnects from the broker for any reason (even accidental). However, when the *cleanSession* flag is set to False, this means that the client has requested a Persistent session and queued messages are delivered to a client on reconnection.

A persistent session is used when it is important for a client to receive all messages about a particular topic, even when it is offline. This is also useful in cases where clients have unreliable connections or limited resources.

A clean session is used in cases where it is necessary that clients only receive messages when they are online or when clients are only publishing messages and not subscribing to any topics.

2.3.1.4 Publishing and Subscribing to Topics

MQTT is a publish/subscribe protocol that allows clients to publish data to certain topics and subscribers of those topics to receive that data every time it is published, via the broker. The

broker is central to the protocol, handling all the published data and is responsible for transmitting the published data to all the subscribers.

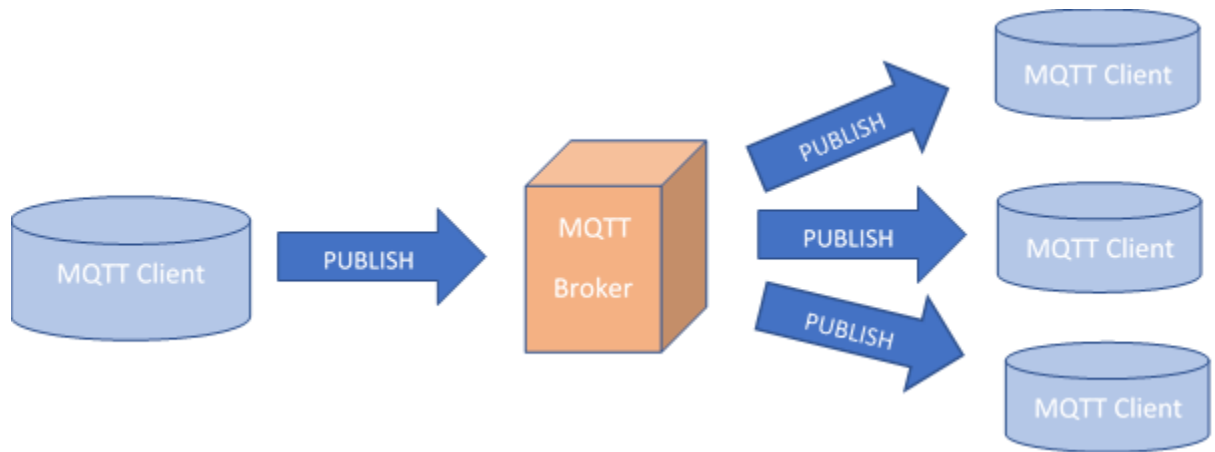


Figure 4: MQTT Publishing

The Publishing process is shown in Figure 4 [20].

MQTT filters content based on topics, so any data that is sent from the client to the broker must contain a topic or topic-hierarchy that the broker then uses to transmit that data to the subscribed clients. A typical MQTT Publish packet contains information about *Topic Name*, *QoS*, *Retain Flag*, *Payload* and *dupFlag*. After the publishing client successfully delivers all this information, it is the responsibility of the broker to deliver this information to the subscribed clients.

A client can subscribe to a topic, or a set of topics, by sending a message to the broker about with a unique packet identifier and a list of subscriptions. The packet identifier is a unique identifier between a broker and a client to identify a message in a flow of messages. The flow of subscribe messages is shown in Figure 5.

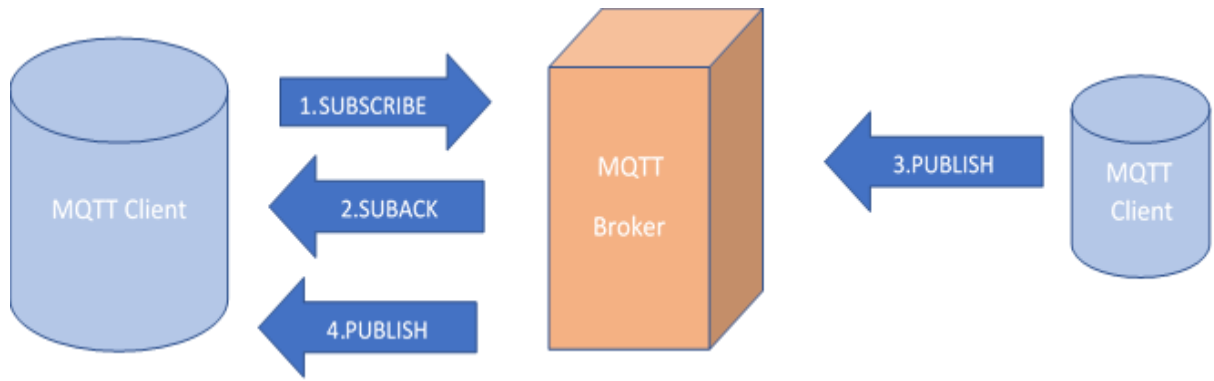


Figure 5: MQTT Subscribe Flow

Once an MQTT Client informs the broker of its subscription to topic A, the MQTT broker acknowledges the subscription and when an MQTT client publishes some content to topic A, the broker then publishes that content to the initially subscribed MQTT client.

The strain on a single broker (and subsequent power requirement) increases with the increase in the number of actively publishing/subscribing clients it is connected to. *This thesis aims to find out the extent to which the power requirement increases.*

2.3.1.5 TLS

By default, MQTT does not use encrypted communication since it relies on the underlying TCP architecture to provide encryption. However, it does provide for the option to use TLS for added security [21].

When using TLS with MQTT, it is called “secure-mqtt” and port 8883 is exclusively reserved for MQTT over TLS [22].

As with all security enhancements, TLS too comes with increased overhead and CPU usage. Techniques like *Session Resumption* can improve the performance of TLS. TLS Session

Resumption allow clients to use an already negotiated TLS connection when reconnecting to a server to avoid the overhead of a full handshake again.

2.3.1.6 Authentication Mechanism

On the application level, the MQTT protocol provides a mechanism for a client to authenticate itself using a username and password. When the client first connects to a broker, it has the opportunity to send a username and password along with the CONNECT request.

If the broker disallows anonymous connections and maintains a password list that it cross-references when clients attempt to connect to it, then it is necessary for the client to provide the username and password when seeking to connect with the broker, else the connection will be denied.

2.4 TRADEOFF BETWEEN PERFORMANCE AND SECURITY IN IOT

Ever since the advent of technology, performance and security have been at odds with each other, often competing for the same resources of services and devices. IoT is no different. If we take the example of *wearables*, a common use-case for IoT, there is an appalling lack of security built in and an indifferent attitude towards it from consumers.

The multinational professional services network, PricewaterhouseCoopers, reports that more than 20 percent of U.S. adults already own at least one wearable, and that there will be approximately 50 billion new connected devices by 2020. Due to the apparent lack of concern, many consumers fail to realize that wearable technology opens new avenues for security and

privacy invasions [23], with malicious entities collecting significant amounts of user data, sometimes without the user's knowledge.

Damien Mehers, a wearables developer who built the Evernote app for different devices has been quoted as saying, "Especially with the fitness [devices], if you read the license agreements, if people really realized what they are signing up for, they might be horrified at what they're allowing the companies to do with the data. I think there needs to be more clarity and perspective from the user [24]."

The reason for this, is the convenience. A threat researcher at the American security software company, Symantec, Candid Wueest has indicated the reality that wearable device developers do not even think about how to approach the security issue when the developing process starts. The overall consensus is to get the device ready to be produced and then "sprinkle some security on top" in the end [25].

Therein lies the need for research on just how much we are trading off when it comes to performance of IoT devices. For both enterprise, as well as consumer-based solutions, metrics need to be available for the power consumption of these devices under different use-cases, so that companies and consumers alike can make smart choices about the levels of security, reliability and conditions that need to be emulated to achieve a certain threshold of performance and power consumption. While this is not accomplished in this thesis, one of the motivations for this work is to eventually consider tradeoffs between power consumption and security protocols for IoT.

2.5 SOFTWARE

2.5.1 Raspbian Jessie

The Raspberry Pi Foundation's official supported operating system is Raspbian. It comes pre-installed with a lot of software like Python, Scratch, Java, Mathematica, etc.

The Operating system is based on Debian and has been optimized for Raspberry Pi Hardware, hence making it the optimal OS for the experiments conducted for this thesis.

2.5.2 Mosquitto

Eclipse Mosquitto™ is an open source (EPL/EDL licensed) message broker that implements the MQTT protocol versions 3.1 and 3.1.1. [26]

Along with the Paho Python library for the clients connecting to the broker, this will provide us with the necessary resources to carry out our experiments.

2.6 RELATED WORK

There has been a lot of research studying the performance and power consumption of IoT devices under different test conditions.

There has been work comparing the different data transfer protocols that are used in IoT systems, of which MQTT is one. *Yokotani and Sasaki* [27] compare the network resource usage

(required bandwidth and delay) of MQTT with HTTP on an IoT platform. Both the bandwidth and the delay are intrinsically related to the power consumed by the device, which is what this thesis aims to quantify.

Different ways of optimizing the power consumption of IoT devices over Wi-Fi has been researched by *Thomas, McPherson, Paul & Irvine* [28] where they first examine the feasibility of WiFi in IoT use-cases. They conclude that with low-powered processors, WiFi can practically be implemented in IoT systems and provides better range and security than 433MHz AM transmitters.

Since IoT devices are usually constrained in their power supply, it is important to know the correlation between different factors in the IoT stack and their effects on the power consumed by the devices, hence there has been work suggesting novel methods to reduce the power consumption while maintaining performance like reducing the packet size and using address clustering [29] and different models have been proposed to better understand which layers of the IoT stack affect the power consumption the most. *Gray, Ayre, Hinton & Tucker* [30] have shown that shared Wi-Fi access with Passive Optical Network (PON) backhaul is the overall most power efficient wireless access technology compared to Very-high-bit-rate digital subscriber line 2 (VDSL2) and Long-Term Evolution (LTE) for <1Mb/s data access rates. *Martinez, Monton, Vilajosana & Prades* [29] propose a model that takes a system-level perspective to account for all the energy expenditures: communications, acquisition and processing, focusing on the bigger picture. This thesis limits itself to examining the effects of just the communications layer, more specifically the MQTT protocol.

Various works have analyzed the use of MQTT and compare it with other protocols like HTTPS [30], AMQP [31], Representational State Transfer – REST over HTTP [32], CoAP [1], [33] and Simple Text Oriented Message Protocol - STOMP [34].

HTTPS seems less than ideal for an IoT use-case, because it cannot cater to some needs of an IoT environment like emitting information from one to many, listening for events whenever they may happen, distributing small packets of data in high volumes, pushing information over unreliable networks (as is the case with a lot of IoT applications), and scalability [35].

Advanced Message Queuing Protocol (AMQP) is sometimes considered an IoT protocol although it has its own use-case. It provides a rich set of messaging scenarios (as opposed to MQTT's small and minimalist design) and can be called the asynchronous complement to HTTP. AMQP permits many forms of messaging including round-robin, store and forward, classic message queues, and different combinations that you can choose based on the application, while MQTT is limited to its publish-subscribe model. *Cohn* [31] explores the different scenarios in which either of these protocols might be applicable based on their architecture, but does not perform any experiments.

CoAP is an application layer protocol developed for resource-constrained devices, which most IoT devices are. The main difference between MQTT and CoAP is that CoAP uses Universal Resource Identifiers (URIs) instead of the topics that MQTT uses and CoAP also runs on top of UDP as opposed to MQTT which runs over TCP. Since UDP is unreliable, CoAP compensates by offering its own reliability mechanism in the way of 'confirmable' and 'non-confirmable' messages. *Thangavel, Ma, Valera, Tan & Tan* [14] compare the performance of CoAP with MQTT, measuring factors like end-to-end delay and bandwidth consumption. They

find that MQTT messages have lower delay than CoAP messages at lower packet loss rates and higher delay than CoAP messages at higher loss rates. They also find that when the message size is small and the loss rate is equal to or less than 25%, CoAP generates lower additional traffic than MQTT to ensure message reliability. They conclude that the performance of the different protocols depended on the different network conditions. Although delay and bandwidth consumption are linked to the power consumed, they have not explicitly made any conclusions about the difference between the protocols in terms of the power consumed by them for certain applications.

STOMP is a simple and lightweight protocol that is text-based, but does not deal with queues or topics. It instead uses a “SEND” semantic with a destination string that other clients can then connect to. *Piper* [34] bills STOMP as simple and lightweight and offers interesting applications for STOMP in the IoT realm, but does not make a direct comparison to MQTT.

Most of the studies use certain configurations of MQTT and tweaking the different parameters that MQTT offers, is often out of the scope of their research. This thesis aims to understand the different parameters that affect the power consumption of the MQTT protocol.

3.0 EXPERIMENT DESIGN

3.1 SETUP

For the purposes of this experiment, we use two Raspberry Pi 3 Model B (RPi) with Raspbian Jessie OS installed. We further install *Mosquitto* to serve as the MQTT broker on the RPi and MQTT-Paho library to enable Python use in our experiments. The Paho Library is fully compatible with the Mosquitto broker and is used to enable the functionalities of the clients in the experiments.

Figure 6 shows the experimental setup. RPi_A and RPi_B are both Raspberry Pi 3 Model B devices. The power measurement devices are explained in section 2.2.2

RPi_A serves as the MQTT Broker device and multiple instances of RPi_B running Paho serve as the various clients. Both the Raspberry Pis are connected to 5V/2A power supplies. RPi_A is connected to the power-measuring devices (Belkin Conserve Insight Energy Use Monitor, P3 P4400 Kill A Watt Electricity Usage Monitor, DROK Pocket Digital Multimeter USB) subsequently for fixed time periods to get accurate readings.

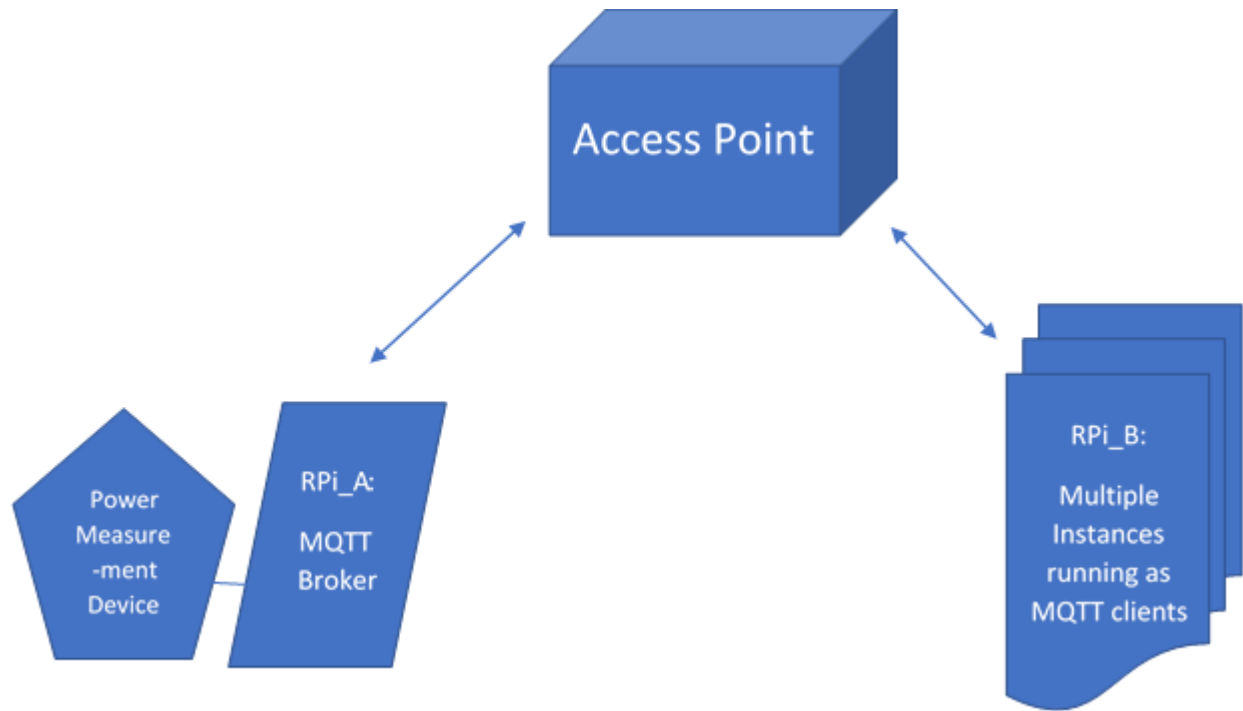


Figure 6: Architecture of Experiment Setup

Both Raspberry Pis are also connected to a local Wi-Fi Protected Access 2 - WPA2 - Personal WiFi network(2.4GHz) and are placed with line of sight access to the Access Point. A control measurement is carried out with the Raspberry Pis connected to the power-measurement devices, immediately after they are booted, with no processes running, just the exchange of WiFi management information with the Access Point. This is to establish a baseline power consumption of the Raspberry Pi when it is idle.

The setup executes as follows:

1. The broker is started on RPi_A
2. Client1 on RPi_B subscribes to a topic 'topic' with the broker
3. Content is published via Paho from Client2 on RPi_B on the topic 'topic'

4. Content is transmitted to the Access Point, then to the broker on RPi_A
5. Broker on RPi_A then sends out the content to all subscribers that are subscribed to 'topic'
6. Content is sent from the broker to the Access point and then received by Client1 on RPi_B.

Typically, clients on RPi_B will be individual devices, but for this test scenario, we are using the Raspberry Pi to emulate multiple clients. The power consumption of RPi_B is not being measured, so having multiple client instances running on RPi_B at the same time will not affect our results.

For the purposes of our experiments, all the clients have the default KeepAlive time of 60 seconds. They run Clean Sessions with no Session Resumption, no Last Will & Testament and no TLS implemented.

All measurements are in Watts, as displayed on the power-measurement devices, which are connected to RPi_A, where the broker is running.

3.2 LIMITATIONS

The limitations of this experimental setup are in the form of hardware. It is difficult to procure hardware that measures the slight changes in power that occur in the Raspberry Pi. The devices used for power measurement are all commercially available power monitors whose primary application is to test the power consumption of household devices that typically operate at higher voltages drawing more current.

Since we are using a Raspberry Pi to emulate all the clients that are connected to the network, it is possible that there are slight delays in processing the different simultaneous publish/subscribe requests, however, the Raspberry Pi features a quad-core 64-bit ARM cortex A53 clocked at 1.2GHz with 1GB of LPDDR2-900 SDRAM. This ensures that it can run 6 clients simultaneously, with minimal latency defects.

It should also be noted that the underlying WiFi network used in these experiments provided a stable connection and packet losses were not detected. A typical IoT environment often contains a lossy underlying network.

Finally, this thesis is limited to exploring the power consumption of the device running the MQTT broker only. No other protocol is studied.

3.3 PARAMETERS VARIED

3.3.1 Quality of Service

When conducting this set of experiments, Client1 on RPi_B is subscribed to the topic ‘topic’ with the broker on RPi_A with QoS level 0, 1 & 2. Client2 on RPi_B (and Client3, Client4, Client5 and Client6 in case of multiple publishers) runs a Python script publisher.py, constantly publishing messages of size 16 bytes to the topic ‘topic’ with QoS levels matching that of the subscriber.

We ensure that the subscriber and publisher have the same QoS level, even though it is possible for the QoS level to get downgraded if the subscriber is subscribed to a topic with a lower QoS level than that with which the publisher has published its content.

QoS 0: This is the level of Quality of Service with the least amount of transmissions as it is a fire and forget setting. Although the underlying transport is TCP/IP, no additional effort is made to transmit the message from publisher to broker or from the broker to subscriber. A PUBLISH message is transmitted and then the next message is sent.

If the message is lost due to the network, it cannot be retrieved.

We expect that the power consumed by the broker will be the least, or the data transferred will be the most when the publisher and subscriber are both using QoS level 0 (see Figure 7).

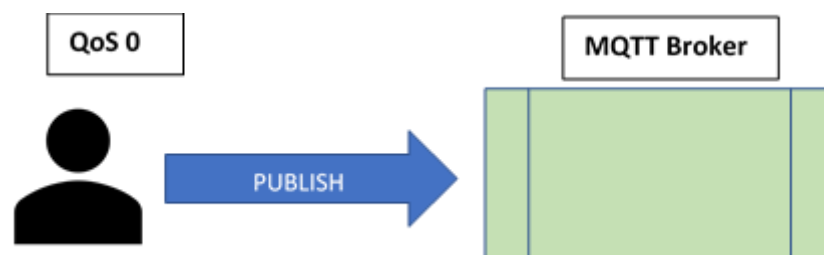


Figure 7: QoS-0 Communication Messages

QoS 1: This is the level of Quality of Service where the transmitting party receives an acknowledgement from the receiving party about its message. It is guaranteed with this QoS level that the message will be delivered at least once. If both the publishing and the subscribing client have set the QoS level to 1, the sender will store the message until it receives an acknowledgement from the receiver that the message has been received (see Figure 8).

We expect that the power consumed by the broker will be intermediate, or the data transferred will be less than that when compared with QoS 0, due to the additional requirement of the broker to acknowledge every Published message.

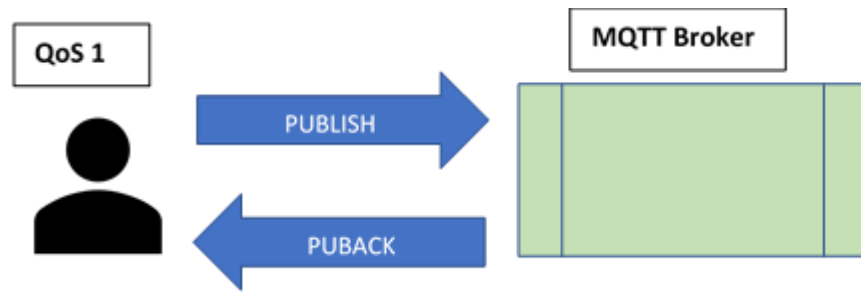


Figure 8: QoS-1 Communication Messages

QoS 2: This is the highest level of Quality of Service that MQTT offers, where the receiver receives the message exactly once. It is both the safest and slowest QoS level [17]. The guarantee of the message being received exactly once comes at the cost of two flows from the client to the broker and two flows in the opposite direction, 4 times as many flows as QoS 0 and twice as many as QoS 1 (see Figure 9).

For this reason, we expect that the power consumed by the broker will be the highest for this level of QoS, or the data transferred will be the least.

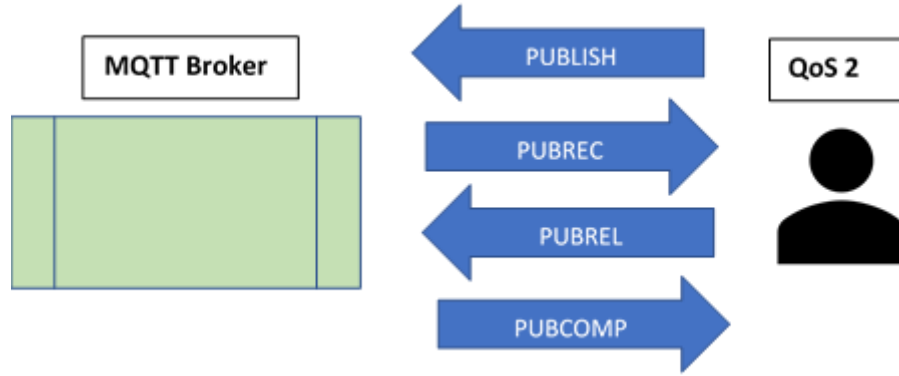


Figure 9: QoS-2 Communication Messages

3.3.2 Number of Publishers

A typical use-case scenario for IoT is Wireless Sensor Networks, where many sensors are constantly publishing the data to the broker device. In real-world IoT deployments, there can be hundreds or even thousands of such sensors, but in those cases, the load is also balanced by multiple brokers. For the purposes of our experiment, we test the power consumption of the broker when 1, 2, 3 and 5 clients are publishing to it simultaneously, for the different Quality of Service levels.

We expect the power consumed by the broker to be lowest for the case when there is 1 subscriber and 1 publisher, and highest for the case when there are 5 publishers and 1 subscriber. If this is not the case, we expect the data received by the subscribing client to be the most in case of 1 subscriber and 1 publisher and the least when there are 5 publishers and 1 subscriber. This is because the broker must simultaneously handle 5 different clients publishing data to it at the same time and it must also relay that data to the subscribing client.

3.3.3 Payload Size

MQTT is a data-agnostic protocol and its structure of payload is determined entirely by the user. Each message typically has a payload which contains the actual data to be transmitted in byte format [20]. In our case, we create files that are exactly 1MB, 2MB, 5MB and 10MB and measure the power consumption of the broker device and the data received at the subscriber when these files are repeatedly published with QoS 2. The code used to publish these files to the broker can be found in the Appendix.

We use QoS 2 because when sending large chunks of data, we prefer that there be minimal retransmissions of the data. Additionally, we also do not want to transmit data unnecessarily and have the receiver receive more than one copy of the file since it is wasteful of the bandwidth and energy resources of our already constrained devices.

We expect that as the payload size increases, the power consumed at the broker increases as well due to fragmentation of the packets. If this is not the case, we expect the data received at the subscriber to be the least when the payload is the largest. This is due to errors that occur when transmitting large files due to fragmentation and retransmissions.

3.3.4 Authentication Mechanism

MQTT allows for application level security, in the form of a username and password that a broker can implement for authenticating the clients that connect to it. The MQTT protocol provides for username and password fields in the initial CONNECT message that a subscriber sends to a broker when it is first connecting to it, to subscribe to topics. The username is a UTF-8

encoded string and the password is binary data with each 65535 bytes max. It is possible to send just a username without a password.

The experiment is conducted, publishing 1 MB files repeatedly at QoS level 1 at first without the authentication mechanism, and then with the authentication mechanism.

We measure the power consumed by the broker device as well as the amount of data received at the subscriber. To test whether any difference between the two sets of readings is not caused just by the initial authentication, we also begin measurements for the same time period with a 20 second offset, so as to give the broker enough time to authenticate a client and commence the data transfer. The objective is to test whether the authentication mechanism has any long-term effects on the data rate or the power consumed after a client has been authenticated.

4.0 RESULTS, ANALYSIS & DISCUSSION

4.1 RESULTS

4.1.1 Quality of Service

First, the Quality of Service Levels are varied and the power consumed by RPi_A (MQTT broker) is measured for 10 minutes and sampled every 10 seconds, with a constant payload message – “Hello World”. The publishing and subscribing clients are separate instances, both running on RPi_B. Each experiment is repeated 10 times to establish repeatability under these test conditions.

Initially, the power consumption is measured for 10 minutes on all 3 power-measurement devices, to obtain the baseline control reading. The only activity on the Raspberry Pi is the exchange of WiFi management messages with the Access Point. The measurements are shown in Figure 10.

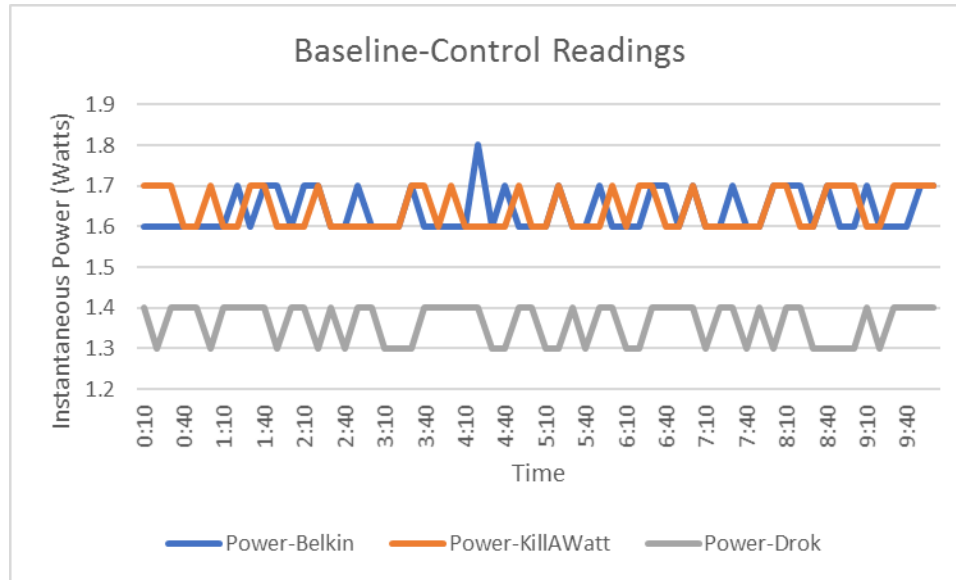


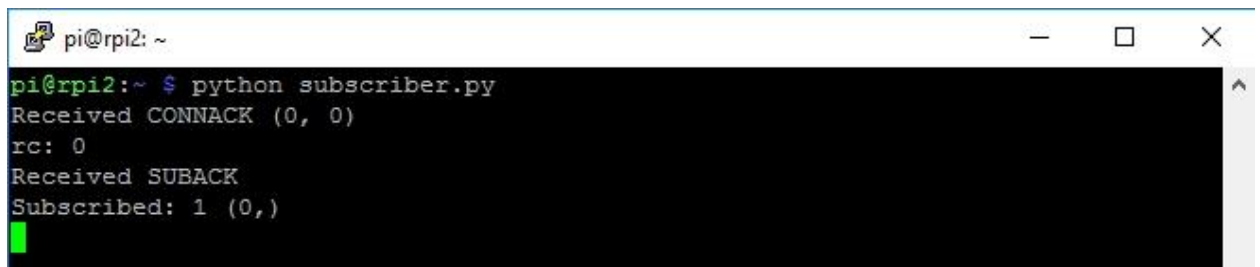
Figure 10: Control Readings

It is worth noting that since the Drok Power-Meter is a USB meter, it measures less power than the other two devices, which plug directly into the wall-outlets.

4.1.1.1 Scenario: 1 Subscriber, 1 Publisher for Different QoS Levels

Thereafter, the subscriber and publisher codes are run on the respective instances on RPi_B after starting the Mosquitto broker on RPi_A to get the readings for QoS 0 (default) with 1 Subscriber and 1 Publisher.

The code used in the experiments through the Paho-MQTT Python library can be found in the appendix. Figures 11-13 show the execution of the commands for the experiments in the SSH client, Putty, for the various QoS level subscriptions.

A terminal window titled 'pi@rpi2: ~' with standard window controls. The command 'python subscriber.py' has been executed, resulting in the following output: 'Received CONNACK (0, 0)', 'rc: 0', 'Received SUBACK', and 'Subscribed: 1 (0,)' followed by a green cursor.

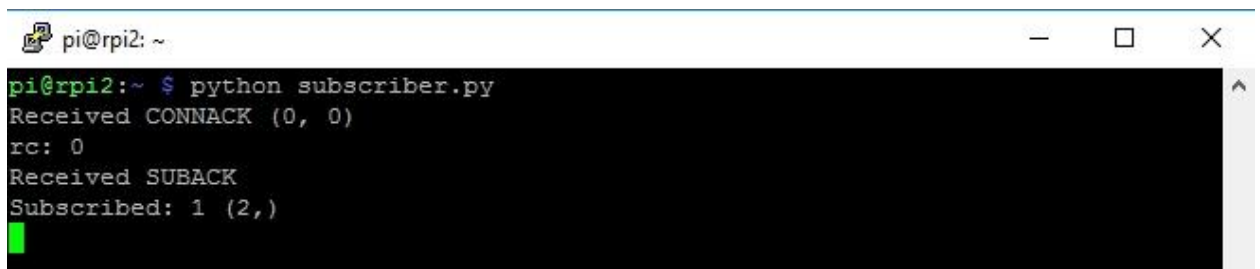
```
pi@rpi2:~ $ python subscriber.py
Received CONNACK (0, 0)
rc: 0
Received SUBACK
Subscribed: 1 (0,)
```

Figure 11: QoS-0 Subscription

A terminal window titled 'pi@rpi2: ~' with standard window controls. The command 'python subscriber.py' has been executed, resulting in the following output: 'Received CONNACK (0, 0)', 'rc: 0', 'Received SUBACK', and 'Subscribed: 1 (1,)' followed by a green cursor.

```
pi@rpi2:~ $ python subscriber.py
Received CONNACK (0, 0)
rc: 0
Received SUBACK
Subscribed: 1 (1,)
```

Figure 12: QoS-1 Subscription

A terminal window titled 'pi@rpi2: ~' with standard window controls. The command 'python subscriber.py' has been executed, resulting in the following output: 'Received CONNACK (0, 0)', 'rc: 0', 'Received SUBACK', and 'Subscribed: 1 (2,)' followed by a green cursor.

```
pi@rpi2:~ $ python subscriber.py
Received CONNACK (0, 0)
rc: 0
Received SUBACK
Subscribed: 1 (2,)
```

Figure 13: QoS-2 Subscription

Figure 14 compares the Average Power of the different QoS levels over a ten-minute period:

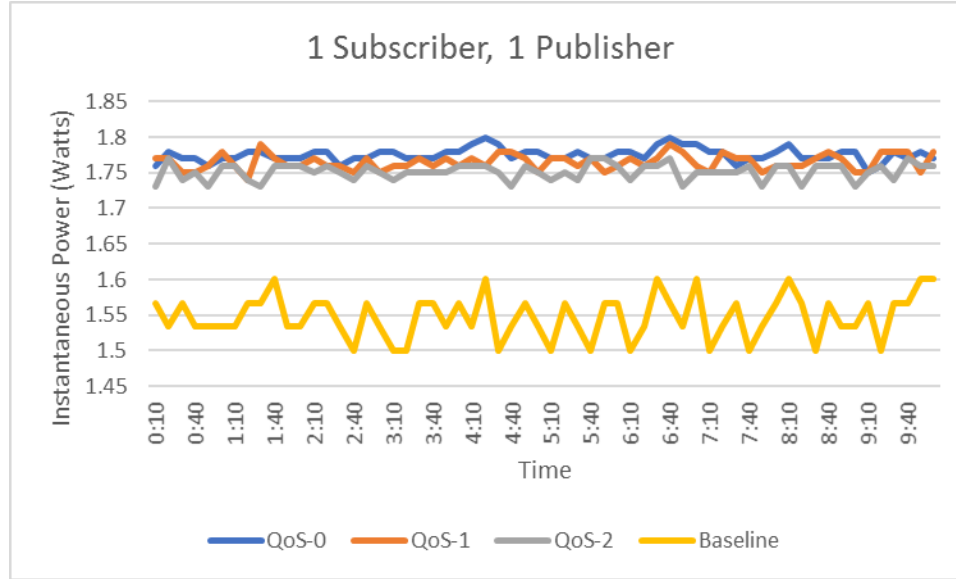


Figure 14: QoS Comparison for 1 Subscriber, 1 Publisher

Contrary to what we might expect, we see that the average power of QoS2 is lower than QoS0, even though the overhead is much higher in QoS2. This can be explained by the fact that fewer messages were actually published when QoS2 was implemented, as opposed to QoS0.

4.1.1.2 Scenario: 1 Subscriber, 2 Publishers for Different QoS Levels

In Wireless Sensor Network environments, we often find multiple sensors trying to publish data to a central node (broker). MQTT is a protocol that is widely used in these networks, thus, it is worth investigating how much power is drawn when there are multiple publishers.

Here, the 1 Subscriber and 2 Publishers are all instances on RPi_B, while the broker resides on RPi_A.

Figure 15 compares the Average Power of the different QoS levels when there are 2 Publishers simultaneously publishing to the broker.

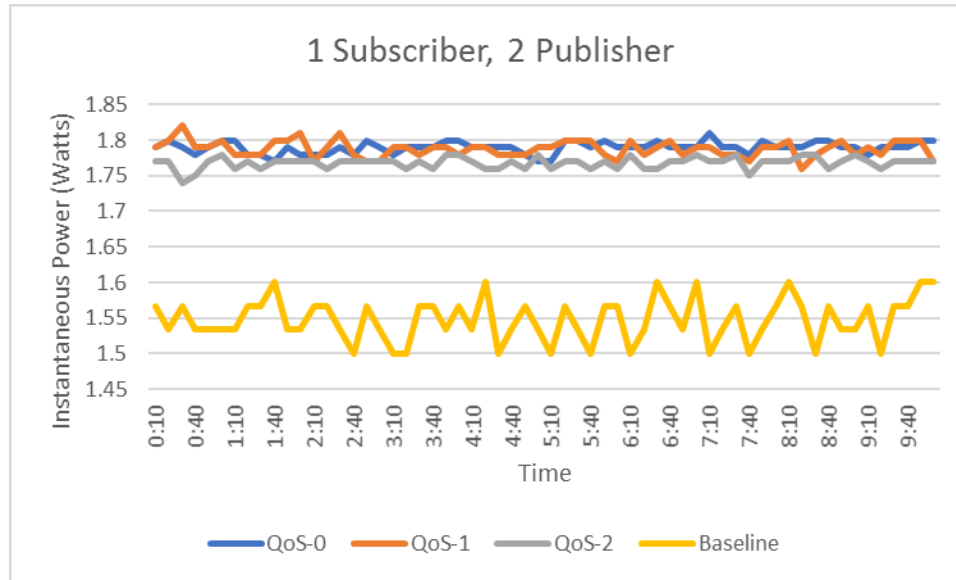


Figure 15: QoS Comparison for 1 Subscriber, 2 Publishers

4.1.1.3 Scenario: 1 Subscriber, 3 Publishers For Different QoS Levels

Figure 16 compares the Average Power of the different QoS levels when there are 3 Publishers simultaneously publishing to the broker.

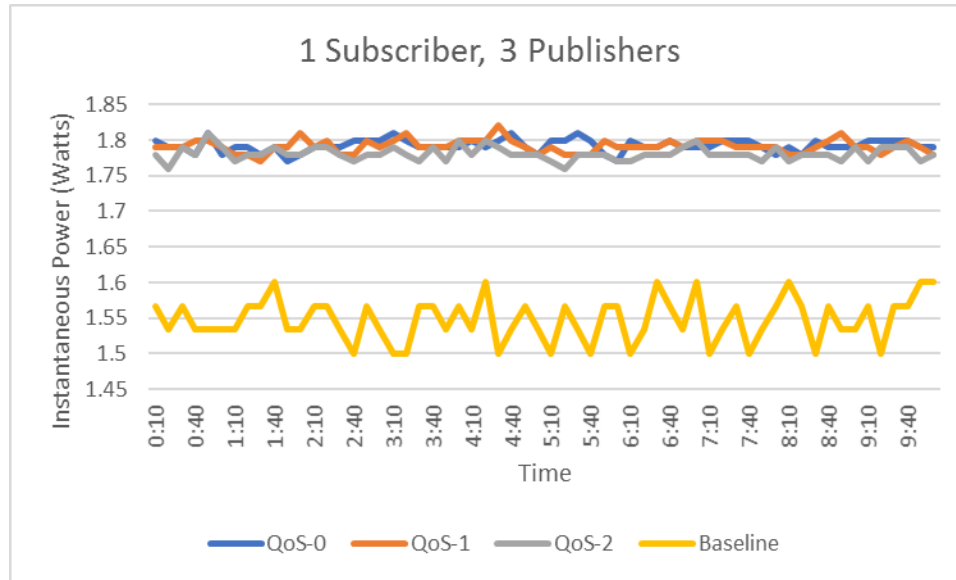


Figure 16: QoS Comparison for 1 Subscriber, 3 Publishers

4.1.1.4 Scenario: 1 Subscriber, 5 Publishers for Different QoS Levels

Figure 17 compares the Average Power of the different QoS levels when there are 5 Publishers simultaneously publishing to the broker.

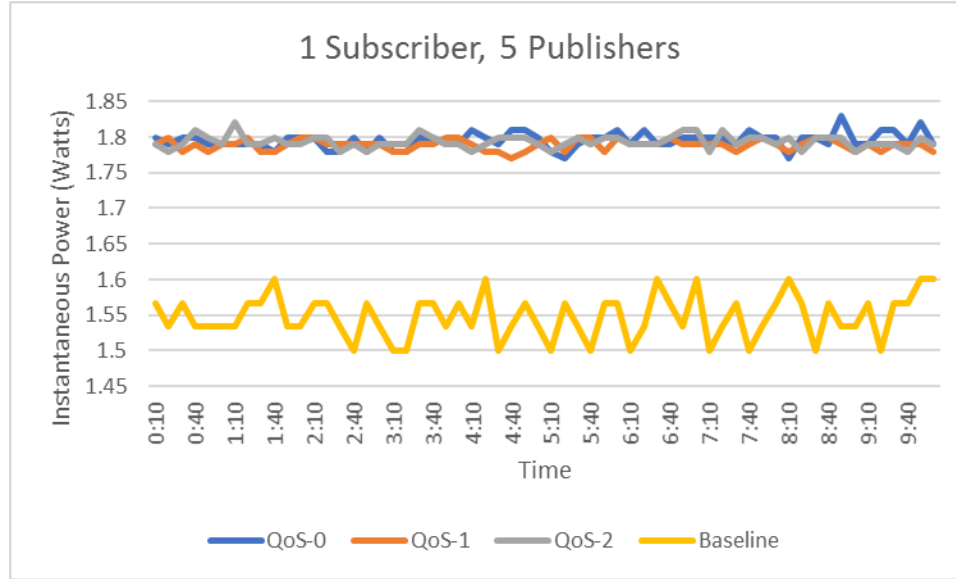


Figure 17: QoS Comparison for 1 Subscriber, 5 Publishers

4.1.2 Average Number of Messages Received Per Minute

If we consider the data received by the client that is subscribed to the topic, we can compute the average number of messages that it received for the different QoS levels. The confidence level over 10 runs indicates that although a trend is observed, mostly it may be hard to distinguish between the power consumption when using different QoS levels, as seen in Figure 18.

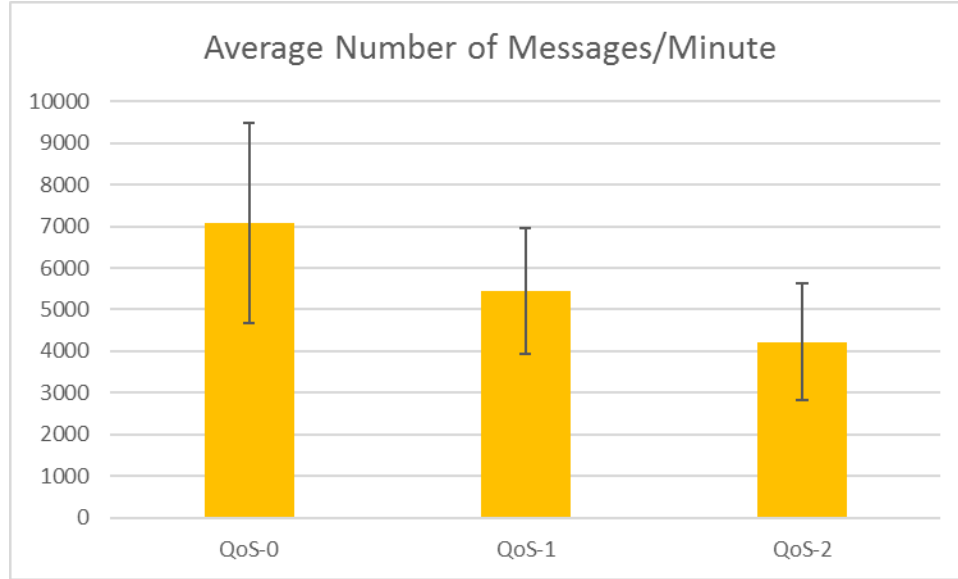


Figure 18: Average Number of Messages Per Minute

4.1.3 Number of Publishers

In typical wireless sensor networks, there are multiple sensors that routinely send data to a central device. To emulate this environment, the number of publishers is varied, to test the capacity of the broker and the power it consumes, to handle the incoming publish-messages from the clients and subsequently transfer those messages to all the clients that are subscribed to the topic. Since the number of messages the broker might have to store and process simultaneously increases with an increase in publishers, the number of publishers is varied and the results are observed. Figures 19-21 illustrate the variations in the instantaneous power for different number of simultaneous publishers (1, 2, 3, 5) for QoS level 0 (see Figure 19), QoS level 1 (see Figure 20) and QoS level 2 (see Figure 21).

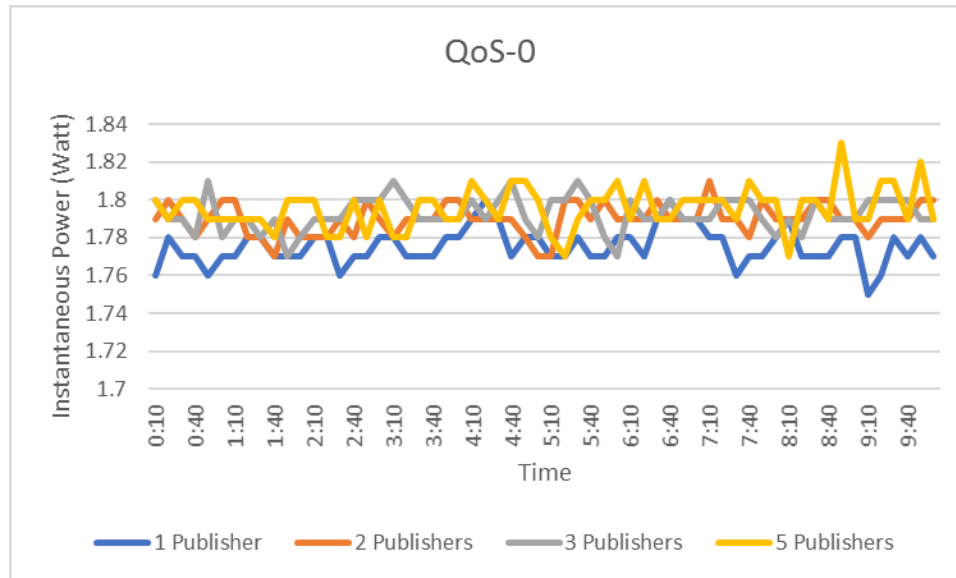


Figure 19: QoS-0: Number of Publishers Comparison

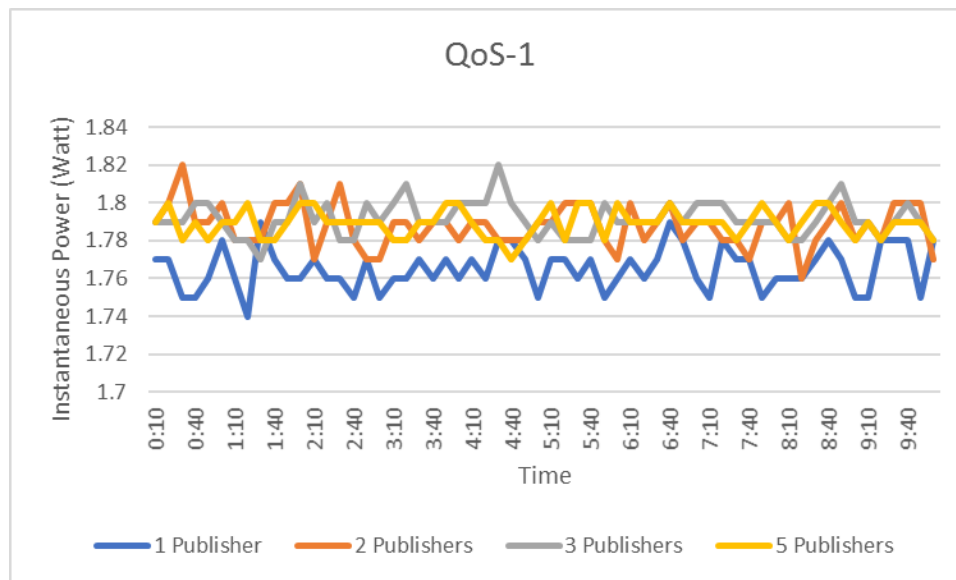


Figure 20: QoS-1: Number of Publishers Comparison

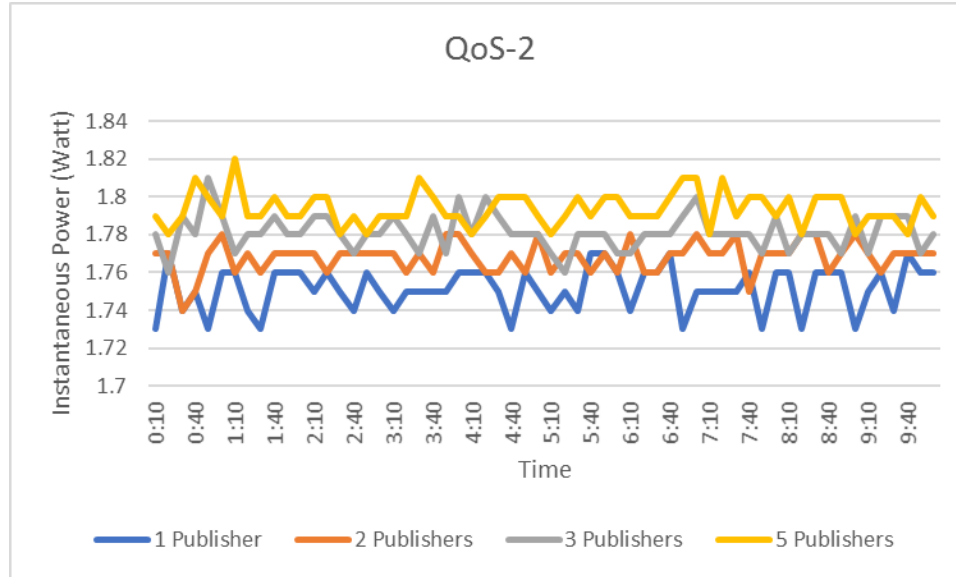


Figure 21: QoS-2: Number of Publishers Comparison

4.1.4 Amount of Data Received by a Subscribing Client

We observe the effects of varying the Quality of Service as well as the number of publishers, while measuring the total data that has been received by a subscribing client in a given time period (10 minutes). Figure 22 shows the variation, along with the standard deviation in the readings for the number of 16 byte messages received by the subscribing client in the period of 10 minutes. Due to the non-overlapping error-margins, we can say with confidence that there are observable trends in this case.

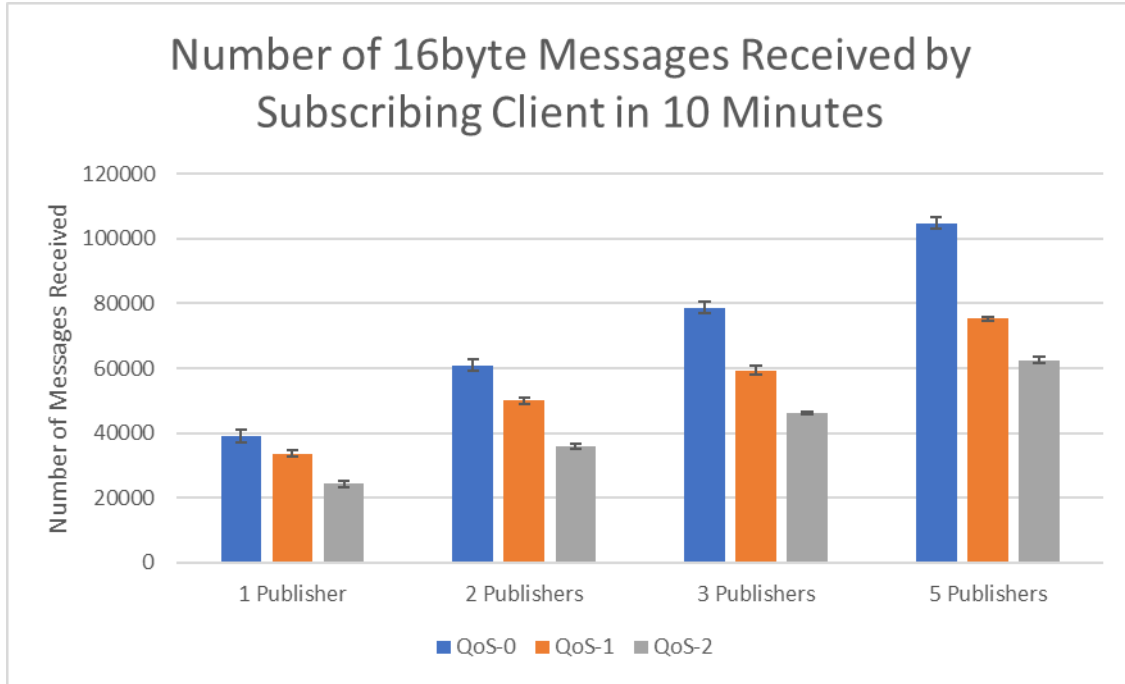


Figure 22: Data Received by Subscriber

4.1.5 Average Energy Consumed Per Publisher

In the given time period of 10 minutes with messages being sent continuously at different QoS levels, for varying number of publishers, we can analyze the average energy consumed by each publisher in the system. We assume that the power stays constant for the 10 second sampling interval. This is an important consideration when a system designer must decide the number of sensors or things to be placed with regard to the power available. Figure 23 illustrates the variation in the average energy consumed by each publisher along with the standard deviations for the readings. It is worth noting that the error margins for the different readings do not overlap and trends can be observed.

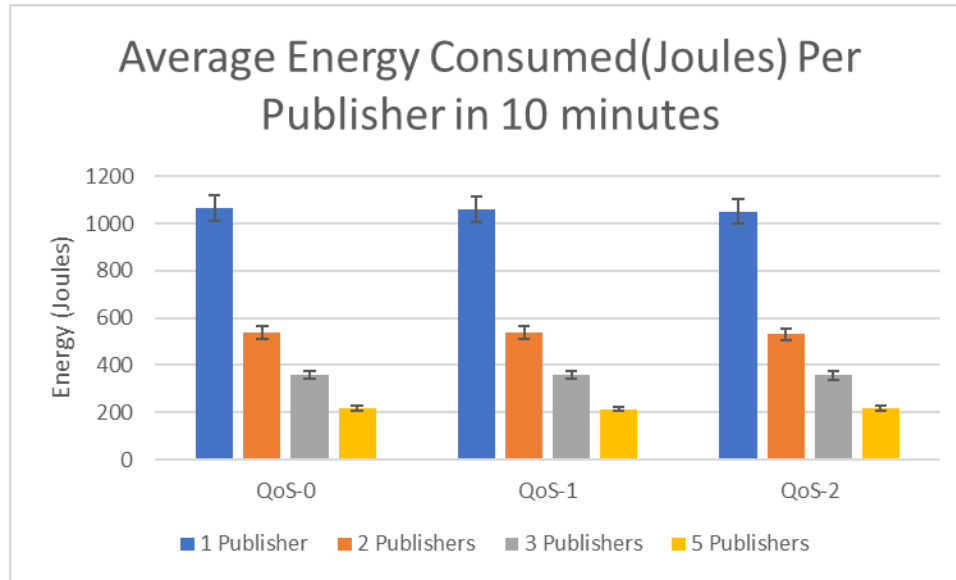


Figure 23: Average Energy Consumed Per Publisher

4.1.6 Total Energy Consumed by Broker

For the entire test period of 10 minutes, sampling the power every 10 seconds and assuming that the power stays constant for those 10 seconds, we obtain readings for the total energy consumed by the broker device in 10 minutes (see Figure 24).

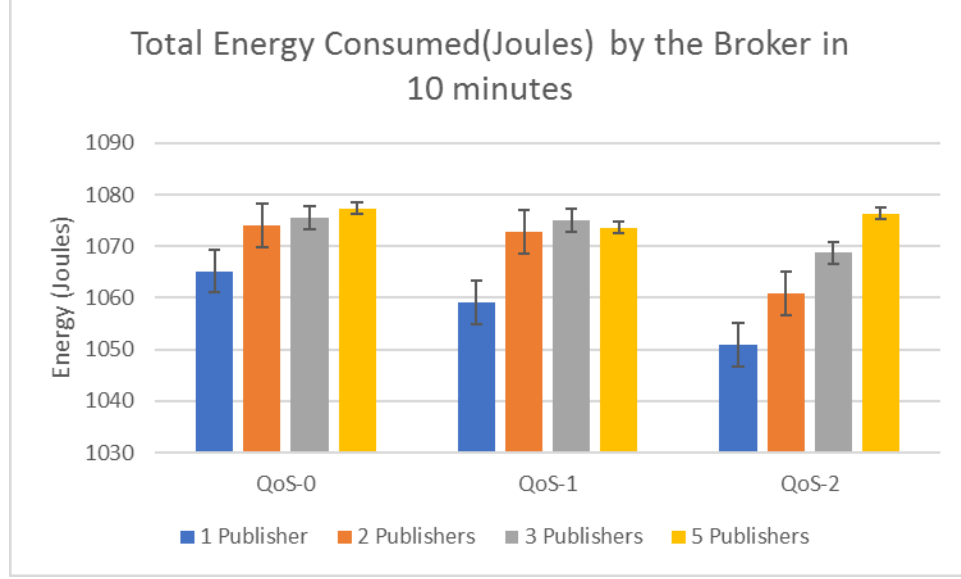


Figure 24: Total Energy Consumed

4.1.7 Payload Size

Next, we observe the effects of Payload size on the power consumed by the protocol/device.

We create 4 files, *test1MB*, *test2MB*, *test5MB* and *test10MB* of 1MB, 2MB, 5MB and 10MB respectively and publish it. For this particular application, we would prefer it if the file being published does not need to be re-published, and the subscribers are able to receive it as a whole. For this reason, we choose QoS=2, which ensures the file gets transferred exactly once.

We also measure the data that is transferred to the subscribers in the 1 minute time frame that we measure the power. The fluctuation in power consumption is more frequent and noticeable in this part of the experiment, thus we choose a smaller time frame (1 minute), with smaller intervals (1 second). Figures 25-28 illustrate that the error margins in the experiments are non-trivial and although a trend is observed, it may be hard to distinguish between the power consumed by the broker when files of different sizes are being transmitted.

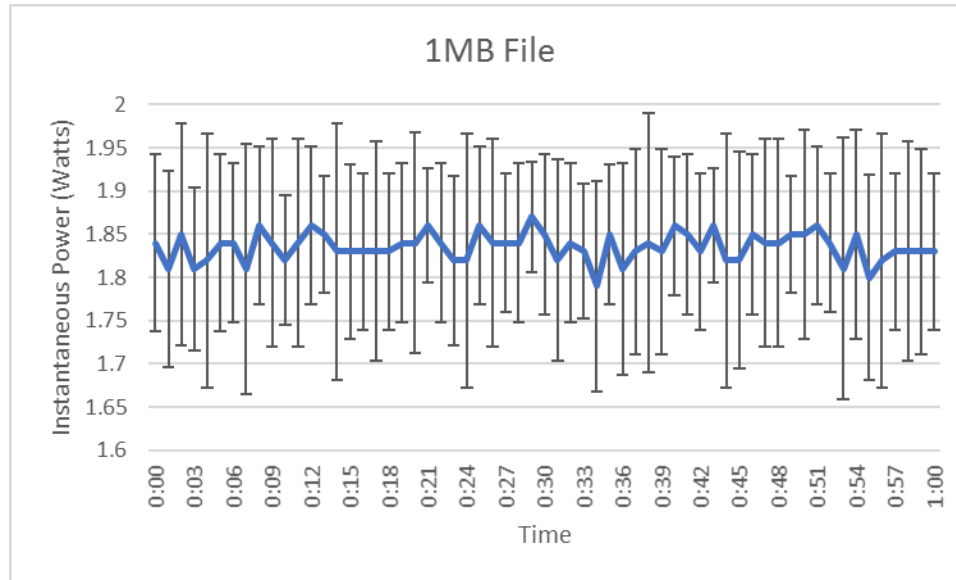


Figure 25: 1MB Payload Power

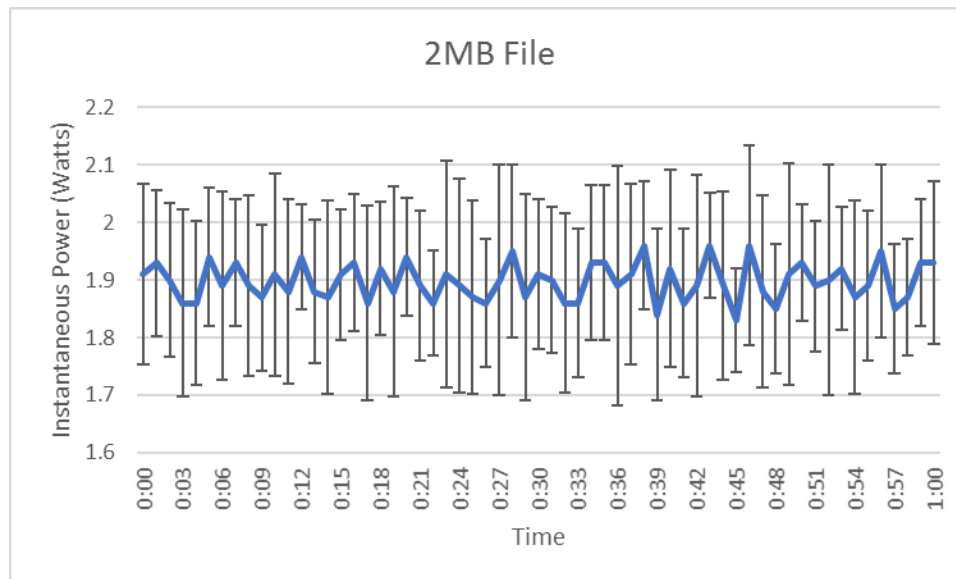


Figure 26: 2MB Payload Power

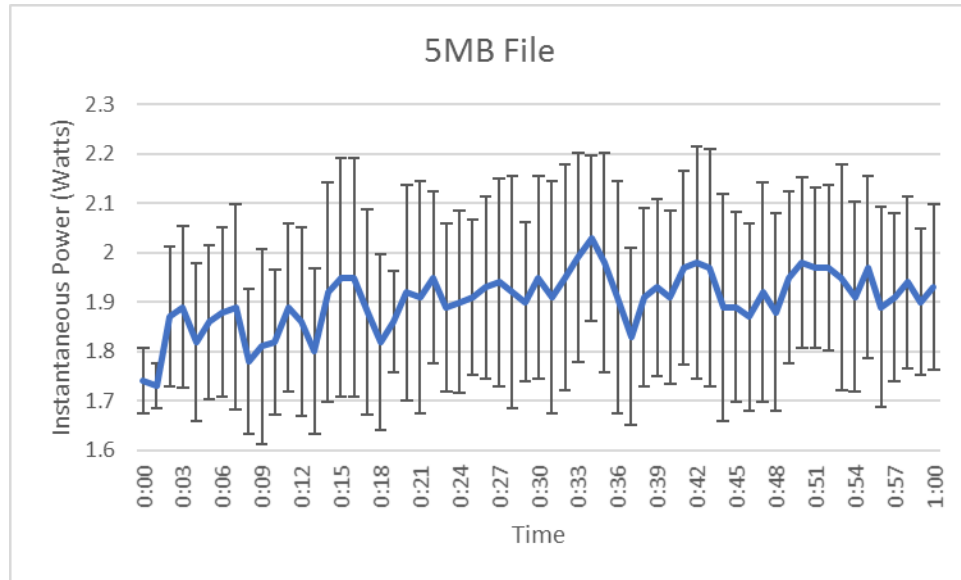


Figure 27: 5MB Payload Power

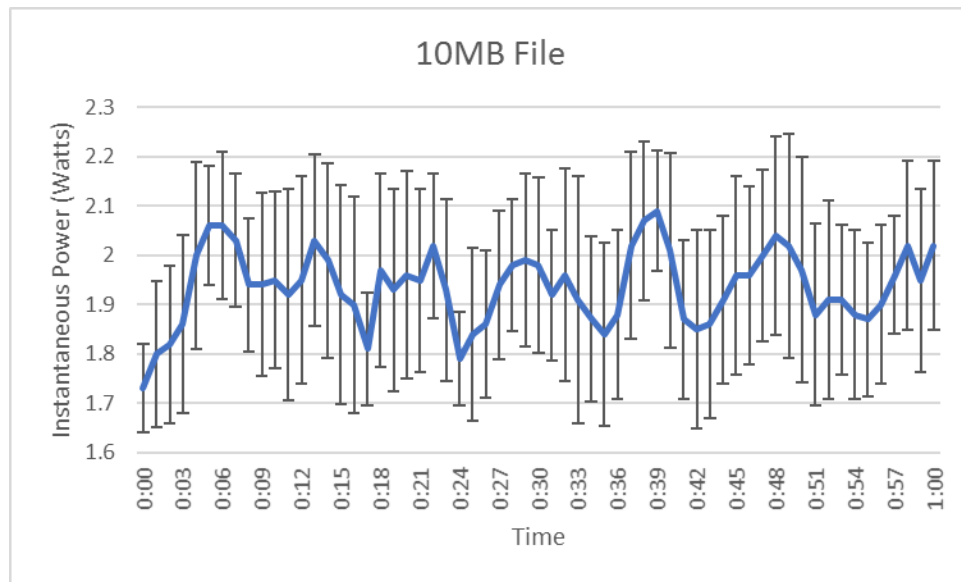


Figure 28: 10MB Payload Power

Figure 29 shows the comparison of the instantaneous power for the different payloads and compares them, although the trends that are noticed are only for the average instantaneous power.

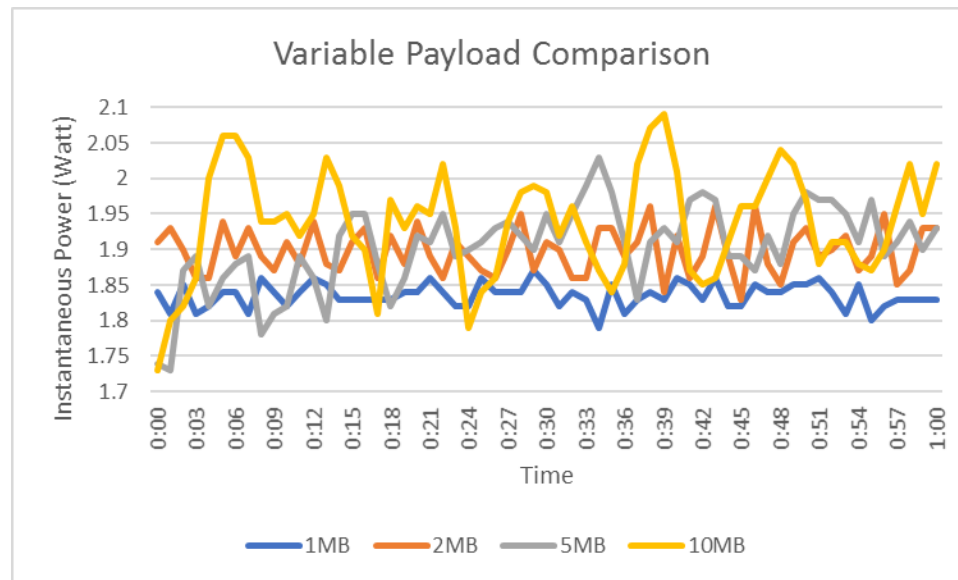


Figure 29: Variable Payload Power Comparison

The average amounts of data that were transferred in the given time frame (1 minute), using all the different payload sizes are shown in Figure 30.

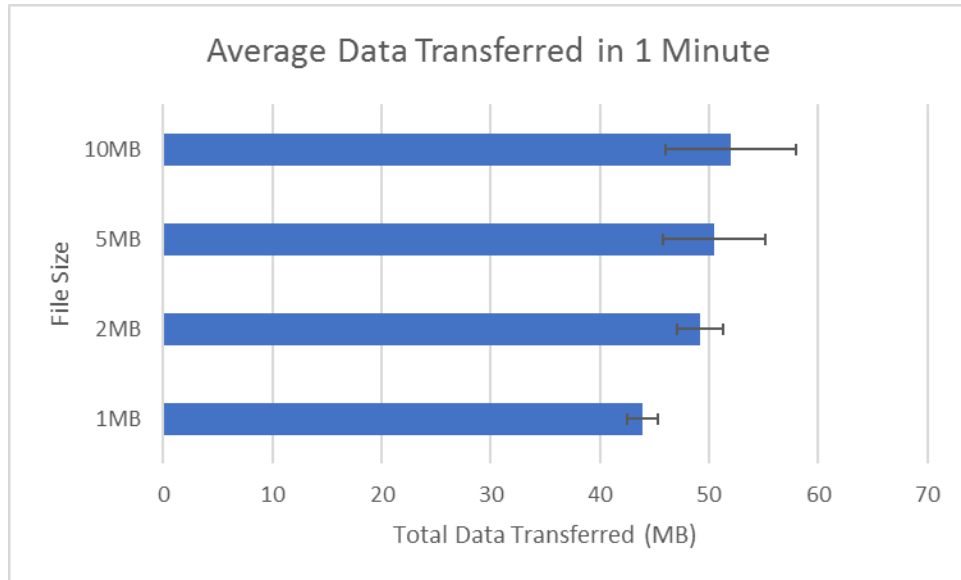


Figure 30: Average Amount of Data Transferred in 1 Minute

We can also measure how much data is received by the client as a function of the power consumed by the broker (see Figure 31).

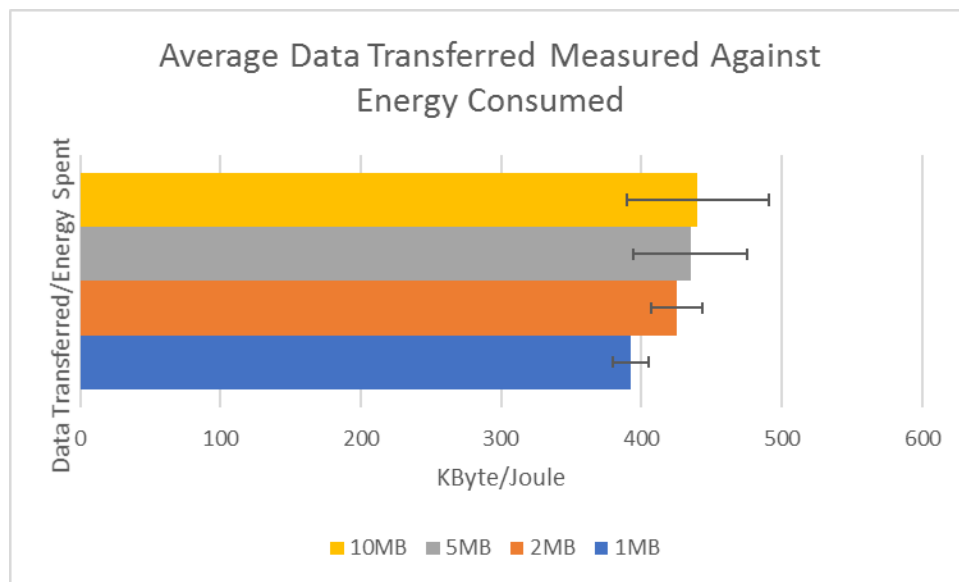


Figure 31: Average Data Transferred

4.1.8 Authentication Mechanism

This experiment sets up an Authentication mechanism for clients before they connect to the broker, by cross-referencing the username and password with a local file. Initially, power consumption is measured for a simple publish/subscribe of a 1MB file with QoS=1, without the authentication mechanism.

The power consumption is then measured once the authentication mechanism is in place.

The measurement time is 2 minutes and the interval of measurement is 1 second. The results can be seen in Figure 32.

To allow the broker to implement the authentication mechanism, the configuration file must be modified to disallow anonymous clients from connecting to the broker. The broker must also check if the login credentials used by the clients match those on the file as specified in the configuration file. The configuration file is amended as noted in the Appendix.

The Publisher and Subscriber scripts are also amended to include the login credentials for when the authentication mechanism is used. These can also be found in the Appendix.

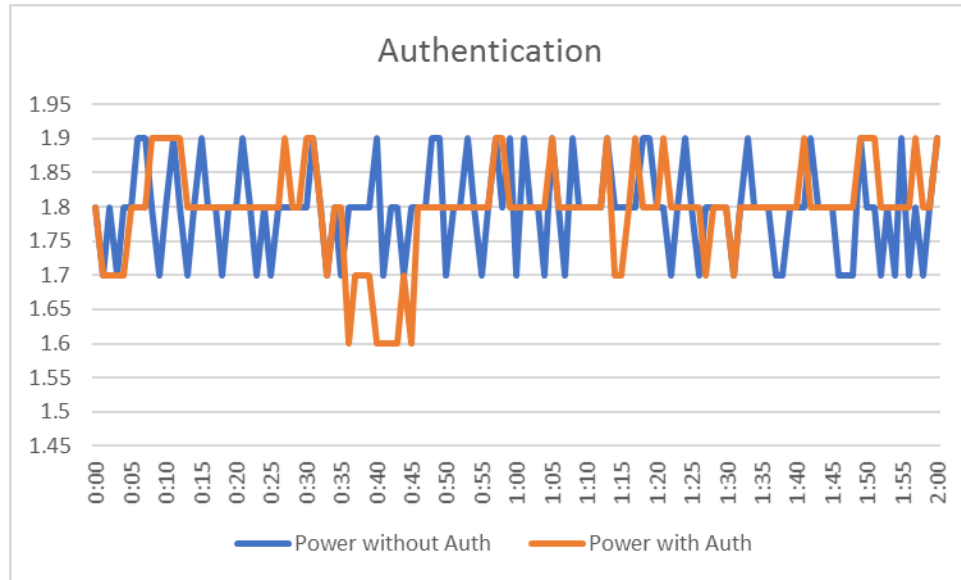


Figure 32: Power Consumption with/without Authentication Mechanism

The amount of data that is transmitted is also measured and the results are shown in Figure 33.

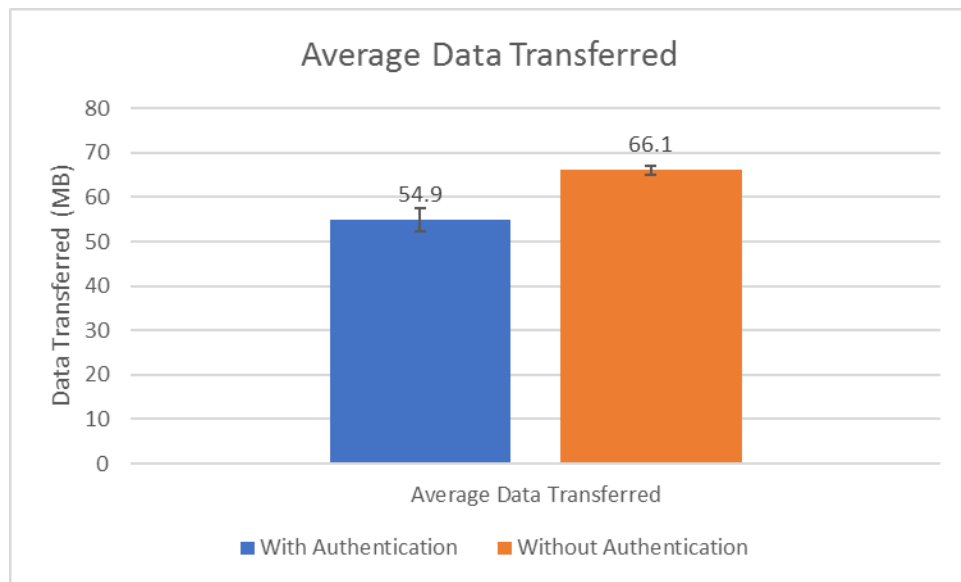


Figure 33: Data Transferred with/without Authentication Mechanism

4.2 ANALYSIS & DISCUSSION

For the three different QoS Levels in cases with 1 Subscriber and 1 Publisher:

- Average power consumed when the MQTT protocol is operating in QoS level 0 is higher than that of QoS2, even though the overhead in QoS2 is greater, but all three levels consume similar amounts of power.
- However, this difference is explained by the fact that in the given time period, more data is transferred in the QoS0 level (34,348 messages) than in QoS2(22,715 messages). The additional overhead of QoS2 slows down the protocol and thus, does not allow it to publish as many messages as if it were operating in QoS0.
- The reduced speed can be considered a tradeoff for reliability. Although in this experimental setup, no messages were lost, in cases where there are unreliable networks, QoS2 might work in delivering messages more reliably than QoS0.
- Figure 27 shows us the difference in the average number of messages per minute for the different QoS levels and we can observe that given its low overhead and fewer messages to publish, QoS0 transfers the most number of messages per minute, followed by QoS1, which has an additional acknowledgement message. QoS2 has the least number of messages per minute due to the additional back and forth messaging between the broker and client for each message.

When multiple publishers are introduced:

- For all 3 QoS levels, more publishers connecting to the broker and publishing simultaneously, makes the broker draw more power.

- This is most noticeable in the case of QoS2, which can be explained by the additional overhead of the handshake messages and the processing of simultaneous requests from different publishers.
- Figure 31 clearly shows us the difference and trends in the number of messages received by the subscribing client for the different number of publishers, for the different QoS levels. As the number of publishers increases (from 1 to 5), the number of data received by the subscribing client also increases, for all QoS levels. This indicates that perhaps the processing power is not being utilized completely and it is possible that additional publishers will yield even more data received at the client.
- We can also observe in Figure 31 that as we increase the QoS level, the number of messages reaching the subscriber in the given period of time, reduces. This can be explained by the additional messages for the higher QoS levels.
- Figure 32 illustrates how for different QoS levels, the average energy consumed by each publisher (for each case of varying number of publishers) is approximately the same. However, when there are multiple publishers in the system, each of them can only use a fraction of the total power available.
- In figure 33, for QoS 2, we can clearly see that the total energy consumed by the broker increases as the number of publishing clients increases. This is due to the fact that the broker has to service the extra clients, storing their messages when delivery is not possible, to attempt delivery at a later time. The broker operating at QoS2 must also handle simultaneous incoming published messages and at the same time also ensure that it is sending out those messages to clients that have subscribed to the topic.

For the different Payload Sizes:

- For a given period, when using QoS2, larger files (upto 10MB) offer higher power efficiency and data-transfer efficiency, rather than smaller files.
- The power consumed by the broker in the case of continuous transfer of different sizes of files is about the same, on average, for the 4 different file sizes that were tested.
- Figure 39 illustrates the amount of data transferred using the different file sizes as a function of the power that is consumed by the broker. Since our focus is the power consumption of the broker under different conditions, this metric allows us to pinpoint what decisions to make to optimize power usage, ie: use larger files (~10MB). We must note that this is not necessarily a trend, however, since a 5MB file shows poor power efficiency.

For the implementation of the authentication mechanism in Mosquitto:

- The amount of data that can be transferred in a given period of time is larger if we do not implement the authentication mechanism. Ie: the authentication mechanism slows down the transfer of data from publishers to subscribers, when using QoS1. This reinforces the notion that additional security mechanisms require additional time and resources from the protocol for their implementation and operation.
- The average power consumed in a given time period is approximately the same, with and without the implementation of the authentication mechanism. It follows that the authentication mechanism utilizes the resources of time and overhead from the protocol rather than power, ie: the protocol uses the time and data it would have used in transferring the data had the additional security mechanism not been present, in implementing the authentication mechanism.

- An additional set of measurements were taken 20 seconds after the transmissions began to observe if the authentication factor only played an initial role in determining the power consumption and vanished afterward. However, it was noted that there is no tangible difference between the delayed measurement and the original one, when using the authentication mechanism.

5.0 CONCLUSION

There are a number of directions for research in the sphere of IoT, considering the exponential growth of the industry. In the field of MQTT alone, there is scope for research in comparing MQTT to other protocols like CoAP, AMQP, HTTP, etc. Additionally, the security aspects of MQTT can be explored more in depth, with the implementation of TLS, with a secure certificate provisioning process.

This thesis was limited in its scope on hardware and software. MQTT can be implemented on devices other than Raspberry Pis as well, like Arduino, ESP8266, etc. In terms of software, there are plenty of MQTT brokers available for use, Mosquitto is just one of them. Other brokers might display different power consumption statistics for different use-cases, based on how they handle the incoming load.

In the future, it could be worth repeating the experiments using the broker as an ad-hoc WiFi device, thus allowing the clients to directly communicate with the broker, instead of through a WiFi access point. If the broker is configured properly, it could reduce the number of messages and the number of hops, possibly reducing the power consumed or increasing the throughput of the broker.

This thesis highlights some of the factors that affect the power consumption of devices that use the MQTT protocol for IoT use-cases. It can be observed that an increased number of publishers being serviced by the same broker, impose more of a power and processing burden on

the broker device. It can also be concluded in cases where clients use a higher level of Quality of Service, even if the power consumed is the same as in lower QoS levels, the amount of data that the protocol is able to transfer in stable networks is lower. In more lossy networks, the amount of data that is reliably transferred might be more, given the reliability of higher QoS levels.

It is observed that for different payload sizes between 1MB and 10MB, for a given period, it is more prudent to transfer larger files, since the protocol offers higher rates of transfer for larger files.

It is also noticeable that although the username and password authentication mechanism does not consume more power from the broker device, it slows down the transfer of data due to the additional overhead of authenticating clients.

This is not an exhaustive list of all the factors in the MQTT protocol that affect the power consumption or performance of the device running the protocol and there is need for further research in the area.

BIBLIOGRAPHY

- [1] J. Jackson, "OASIS: MQTT to be the protocol for the Internet of Things," 2013. [Online]. Available: <http://www.pcworld.com/article/2036500/oasis-mqtt-to-be-the-protocol-for-the-internet-of-things.html>.
- [2] R. Gupta, "5 Things to Know about MQTT - The Protocol for Internet of Things," 2014. [Online]. Available: https://www.ibm.com/developerworks/community/blogs/5things/entry/5_things_to_know_about_mqtt_the_protocol_for_internet_of_things?lang=en.
- [3] HiveMQ, 2015. [Online]. Available: <http://www.hivemq.com/blog/mqtt-essentials-part-1-introducing-mqtt>.
- [4] Cisco, "An Introduction to the Internet of Things (IoT)," 2013. [Online]. Available: http://www.cisco.com/c/dam/en_us/solutions/trends/iot/introduction_to_IoT_november.pdf.
- [5] Gartner, Inc., 2017. [Online]. Available: <http://www.gartner.com/newsroom/id/3598917>.
- [6] Gartner, Inc., "Securing the Internet of Things," 2016.
- [7] Gartner, Inc., "Architect Your Internet of Things System by Using the Gartner IoT Reference Model," 2017.
- [8] Gartner, Inc., "Top 10 IoT Technologies for 2017 and 2018," 2016.
- [9] B. Schweber, "Options for Powering Your Wireless IoT Device," 2016. [Online]. Available: <https://www.digikey.com/en/articles/techzone/2016/apr/options-for-powering-your-wireless-iot-device>.
- [10] Raspberry Pi, "SSH (Secure Shell)," [Online]. Available: <https://www.raspberrypi.org/documentation/remote-access/ssh/README.md>
- [11] Raspberry Pi, "FAQS," [Online]. Available: <https://www.raspberrypi.org/help/faqs/>
- [12] C. Alcaraz, P. Najera, J. Lopez, R. Roman, "Wireless Sensor Networks and the Internet of Things: Do We Need a Complete Integration?," in *1st International Workshop on the Security of the Internet of Things*, 2010.
- [13] Raspberry Pi, "Power Supply," [Online]. Available: <https://www.raspberrypi.org/documentation/hardware/raspberrypi/power/README.md>

- [14] D. Thangavel, X. Ma, A. Valera, H. Tan, C. Tan, "Performance Evaluation of MQTT and CoAP via a Common Middleware," *Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP)*, 2014.
- [15] CoRE Working Group, "Constrained Application Protocol (CoAP)," 2013. [Online]. Available: <https://tools.ietf.org/html/draft-ietf-core-coap-17#section-1>.
- [16] M. H. Elgazzar, "Perspectives on M2M Protocols," in *IEEE Seventh International Conference on Intelligent Computing and Information Systems*, 2015.
- [17] HiveMQ, "MQTT Essentials Part 6: Quality of Service 0, 1 & 2," 2015. [Online]. Available: <http://www.hivemq.com/blog/mqtt-essentials-part-6-mqtt-quality-of-service-levels>.
- [18] R. Webb, "A Brief, but Practical Introduction to the MQTT Protocol and its Application to IoT," 2016. [Online]. Available: <https://zoetrope.io/tech-blog/brief-practical-introduction-mqtt-protocol-and-its-application-iot>.
- [19] "Why is the keep-alive needed? MQTT," 2014. [Online]. Available: <https://groups.google.com/forum/#!topic/mqtt/zRqd8JbY4oM>.
- [20] HiveMQ, "MQTT Essentials Part 4: MQTT Publish, Subscribe & Unsubscribe," 2015. [Online]. Available: <http://www.hivemq.com/blog/mqtt-essentials-part-4-mqtt-publish-subscribe-unsubscribe>.
- [21] HiveMQ, "MQTT Security Fundamentals: TLS / SSL," 2015. [Online]. Available: <http://www.hivemq.com/blog/mqtt-security-fundamentals-tls-ssl>.
- [22] IANA, "Service Name and Transport Protocol Port Number Registry," [Online]. Available: <https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml?search=8883#OASIS>.
- [23] T. Maddox, "Wearables open new avenues for security and privacy invasions," 2015. [Online]. Available: <http://www.zdnet.com/article/wearables-open-new-avenues-for-security-and-privacy-invasions/>.
- [24] T. Maddox, "The scary truth about data security with wearables," 2014. [Online]. Available: <http://www.techrepublic.com/article/the-scary-truth-about-data-security-with-wearables/>.
- [25] M. t. Napel, "Wearables and Quantified Self Demand Security-First Design," 2014. [Online]. Available: <https://www.wired.com/insights/2014/10/wearables-security-first-design/>.
- [26] "MQTT," [Online]. Available: www.mqtt.org.
- [27] Y. S. Tetsuya Yokotani, "Comparison with HTTP and MQTT on Required Network Resources for IoT," *The 2016 International Conference on Control, Electronics, Renewable Energy and Communications (ICCEREC)*, 2016.
- [28] D. Thomas, R. McPherson, G. Paul, J. Irvine, "Optimizing Power Consumption of Wi-Fi for IoT Devices," *IEEE Consumer Electronics Magazine*, 2016.

- [29] C. Gray, R. Ayre, K. Hinton, R. Tucker, "Power consumption of IoT access network technologies," in *IEEE International Conference on Communication Workshop (ICCW)*, 2015.
- [30] S. D. Nicholas, "Power Profiling: HTTPS Long Polling vs. MQTT with SSL, on Android," [Online]. Available: <http://stephendnicholas.com/posts/power-profiling-mqtt-vs-https>.
- [31] R. Cohn, "A Comparison of AMQP and MQTT," 2012. [Online]. Available: https://lists.oasis-open.org/archives/amqp/201202/msg00086/StormMQ_WhitePaper_-_A_Comparison_of_AMQP_and_MQTT.pdf.
- [32] J. Speed, "REST is for sleeping. MQTT is for Mobile," 2013. [Online]. Available: <https://mobilebit.wordpress.com/2013/05/03/rest-is-for-sleeping-mqtt-is-for-mobile/>.
- [33] Alten Calsoft Labs, "Analyzing MQTT vs CoAP," 2016. [Online]. Available: <http://www.altencalsoftlabs.com/blogs/2016/08/08/analyzing-mqtt-vs-coap/>.
- [34] A. Piper, "Choosing Your Messaging Protocol: AMQP, MQTT, or STOMP," 2013. [Online]. Available: <https://blogs.vmware.com/vfabric/2013/02/choosing-your-messaging-protocol-amqp-mqtt-or-stomp.html>.
- [35] A. Dutta, "Why HTTP is not enough for the Internet of Things," 2013. [Online]. Available: https://www.ibm.com/developerworks/community/blogs/mobileblog/entry/why_http_is_not_enough_for_the_internet_of_things?lang=en.
- [36] M. D. Prieto, B. Martinez, M. Monton, I. V. Guillen, X. V. Guillen, J. A. Moreno, "Balancing Power Consumption in IoT Devices by Using Variable Packet Size," *Complex, Intelligent and Software Intensive Systems (CISIS)*, 2014.

APPENDIX

Subscriber's code in Python for different QoS levels – subscriber.py

```
import paho.mqtt.client as mqtt

def on_connect(mqttc, obj, flags, rc):

    print("rc: "+str(rc))

def on_message(mqttc, obj, msg):

    print(msg.topic+" "+str(msg.qos)+" "+str(msg.payload))

def on_publish(mqttc, obj, mid):

    print("mid: "+str(mid))

def on_subscribe(mqttc, obj, mid, granted_qos):

    print("Subscribed: "+str(mid)+" "+str(granted_qos))

def on_log(mqttc, obj, level, string):

    print(string)

mqttc = mqtt.Client()

mqttc.on_message = on_message

mqttc.on_connect = on_connect

mqttc.on_publish = on_publish

mqttc.on_subscribe = on_subscribe

mqttc.on_log = on_log
```

```
mqttc.connect("192.168.0.104", 1883, 60)
```

```
mqttc.subscribe("topic", 0) #0 denotes the QoS level. We change it to 1 and 2 for
```

```
subsequent parts of the experiment
```

```
mqttc.loop_forever()
```

Publisher's code in Python for different QoS levels– publisher.py

```
import paho.mqtt.publish as publish
```

```
for i in range(500000):
```

```
    publish.single("topic", "Hello World-"+str(i), hostname="192.168.0.104")
```

Publisher's code in Python for different file sizes – publisher.py

```
import paho.mqtt.publish as publish
```

```
f=open("test1MB")
```

```
imagestring=f.read()
```

```
byteArray = bytes(imagestring)
```

```
for i in range(50000):
```

```
    publish.single("topic", byteArray, hostname = "192.168.0.104", qos=2)
```

Publisher's code in Python, amended to use authentication – publisher.py

```
import paho.mqtt.client as mqtt
```

```
f=open("test1MB")
```

```
imagestring=f.read()
```

```
byteArray = bytes(imagestring)
```

```
mqttc1=mqtt.Client()
```

```

def on_connect(mqttc1, obj, flags, rc):

    print("rc: "+str(rc))

def on_message(mqttc1, obj, msg):

    print(msg.topic+" "+str(msg.qos)+" "+str(msg.payload))

def on_publish(mqttc1, obj, mid):

    print("mid: "+str(mid))

def on_subscribe(mqttc1, obj, mid, granted_qos):

    print("Subscribed: "+str(mid)+" "+str(granted_qos))

def on_log(mqttc1, obj, level, string):

    print(string)

mqttc1.on_message = on_message

mqttc1.on_connect = on_connect

mqttc1.on_publish = on_publish

mqttc1.on_subscribe = on_subscribe

mqttc1.username_pw_set('abhishek','abhishek')

mqttc1.on_log = on_log

mqttc1.connect("192.168.0.104", 1883, 60)

for i in range(50000):

    mqttc1.publish("topic", bytearray, qos=1)

mqttc1.loop_forever()

```

Subscriber's code in Python, amended to use authentication – publisher.py

```

import paho.mqtt.client as mqtt

```

```

def on_connect(mqttd, obj, flags, rc):

    print("rc: "+str(rc))

def on_message(mqttd, obj, msg):

    print(msg.topic+" "+str(msg.qos)+" "+str(msg.payload))

def on_publish(mqttd, obj, mid):

    print("mid: "+str(mid))

def on_subscribe(mqttd, obj, mid, granted_qos):

    print("Subscribed: "+str(mid)+" "+str(granted_qos))

def on_log(mqttd, obj, level, string):

    print(string)

mqttd = mqtt.Client()

mqttd.on_message = on_message

mqttd.on_connect = on_connect

mqttd.on_publish = on_publish

mqttd.on_subscribe = on_subscribe

mqttd.username_pw_set('abhishek','abhishek')

mqttd.on_log = on_log

mqttd.connect("192.168.0.104", 1883, 60)

mqttd.subscribe("topic", 1)

mqttd.loop_forever()

```

Mosquitto's configuration file, amended to use authentication – mosquitto.conf

```

allow_anonymous false

```

password_file /etc/mosquitto/pwfile

pid_file /var/run/mosquitto.pid

persistence true

persistence_location /var/lib/mosquitto/

log_dest topic

log_dest error

log_type warning

log_type notice

log_type information

connection_messages true

log_timestamp true

include_dir /etc/mosquitto/conf.d