SAJA AHMED ALI BAWI

ATILIM UNIVERSITY

PERFORMANCE EVALUATION OF IOT APPLICATION LAYER PROTOCOLS

THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

OF

ATILIM UNIVERSITY

SAJA AHMED ALI BAWI

A MASTER OF SCIENCE THESIS

IN

INFORMATION TECHNOLOGY

NOVEMBER 2020

PERFORMANCE EVALUATION OF IOT APPLICATION LAYER PROTOCOLS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
ATILIM UNIVERSITY

BY

SAJA AHMED ALI BAWI

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
INFORMATION TECHNOLOGY

NOVEMBER 2020

Approval of the Graduate School of Natural and Applied Sciences, Atılım University.

____

Prof. Dr. Ali KARA

Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science in Information Technology, Atilim University.

____

Assoc. Prof. Dr. Korhan Levent ERTÜRK

Head of Department

This is to certify that we have read the thesis "PERFORMANCE EVALUATION OF IOT APPLICATION LAYER PROTOCOLS" submitted by SAJA AHMED ALI BAWI and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

____

Prof. Dr. Murat KOYUNCU

Supervisor

Asst. Prof. Dr. Çiğdem TURHAN

Software Eng. Department, Atilim University ________________________

Prof. Dr. Murat KOYUNCU

Information Sys. Eng. Department, Atilim University ________________________

Asst. Prof. Dr. Tolga PUSATLI

Mathematics Department, Çankaya University ________________________

Date: 05/11/2020

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

SAJA AHMED ALI BAWI

Signature :

ABSTRACT

PERFORMANCE EVALUATION OF IOT APPLICATION LAYER PROTOCOLS

ALI BAWI, SAJA AHMED

M.Sc., Information Technology

Supervisor: Prof. Dr. Murat KOYUNCU

November 2020, 86 pages

The Internet of Things (IoT) is the newest technology in the information world, where each device can now be configured as a smart device to communicate with others over the Internet. However, these devices are often connected by means of either low bandwidth or unreliable and intermittent wireless communication links. IoT supports communications among machines, individuals, and the two in just the same way. As IoT continuously becomes more widely used in the course of time, many protocols have been put in place to support communications among these devices. These protocols are of various standards in the application layer, network layer, data link and the physical layer. Henceforth, a question worthy of interest in the IoT world is which application layer protocol better suits a given purpose.

Against this backdrop, the objective of the present thesis is to shed light on the common application layer IoT protocols in terms of their characteristics, architecture and transmission structure to compare their performances and to identify the most suitable ones for implementation in IoT environments depending on different user application requirements. The comparison is done theoretically by analysing the characteristics of protocols at the theoretical level, and by using the simulation platform IoTIFY providing a series of ready-to-run IoT modules at the pratctical level. The parameters used in simulation results are the connection time, the latency of generating message, and the latency of sending packets.

Keywords: IoT, MQTT, HTPP, CoAP, XMPP, IoTIFY simulator.

ÖZ

IOT UYGULAMA KATMANI PROTOKOLLERİNİN PERFORMANS DEĞERLENDİRMESİ

ALI BAWI, SAJA AHMED

Yüksek Lisans, Bilişim Teknolojileri

Tez Yöneticisi: Prof. Dr. Murat KOYUNCU

KASIM 2020, 86 sayfa

Nesnelerin İnterneti (IoT), bilgi dünyasındaki en yeni teknolojilerden birisidir ve her cihaz artık İnternet üzerinden başkalarıyla iletişim kurmak için akıllı bir cihaz olarak yapılandırılabilir. Bununla birlikte, bu cihazlar genellikle ya düşük bant genişliği ya da güvenilmez ve kesintili kablosuz iletişim protokolleri aracılığıyla bağlanır. IoT, makineler, bireyler ve ikisi arasındaki iletişimi aynı şekilde destekler. IoT uygulamaları yaygınlaştıkça, bu cihazlar arasındaki iletişimi desteklemek için birçok protokol uygulamaya konmuştur. Bu protokoller, uygulama katmanı, ağ katmanı, veri bağlantısı ve fiziksel katmanda çeşitli standartları ortaya koymaktadır. Bundan böyle, IoT dünyasında ilgi çekici bir soru, hangi uygulama katmanı protokolünün hangi amaca daha uygun olduğudur.

Bu açıklamalar çerçevesinde, mevcut tezin amacı, özellikleri, mimarisi ve iletim yapısı açısından uygulama katmanı IoT protokollerini inceleyerek performanslarını karşılaştırmak ve kullanıcı uygulama gereksinimlerine bağlı olarak farklı IoT uygulamaları için uygun olanları belirleme konusuna ışık tutmaktır. Karşılaştırma teorik olarak protokollerin özelliklerini teorik düzeyde analiz ederek ve pratik düzeyde bir dizi çalışmaya hazır IoT modülü sağlayan simülasyon platformu IoTIFY kullanılarak yapılmıştır. Simülasyon sonuçlarında kullanılan parametreler bağlantı süresi, mesaj üretme gecikmesi ve paket gönderme gecikmesidir.

Anahtar Kelimeler: IoT, MQTT, HTPP, CoAP, XMPP, IoTIFY simülatörü.

To my father, my first and last love, my ideal and supporter;
To my first teacher, my mother, for her pure prayers and love;
To my sisters, Sura and Sally, the support factors in my life;
To my friends who always encourage me;
To my country, Iraq;
And to those who paved our way of science and knowledge.

ACKNOWLEDGMENTS

In the Name of Allah, the Knower of All, Who has always blessed us with potential knowledge and success.

I would like to express my deepest appreciation to my supervisor, Prof. Dr. Murat Koyuncu, for his support, guidance, encouragement, useful critiques of this research work, and his patience in the course of its preparation. I owe great thanks to him. Much appreciation goes to all the academic staff in the Department of Information Systems Engineering, Computer Engineering, and Software Engineering at Atilim University for their efforts and assistance. I shall also thank the staff of the Graduate School of Natural and Applied Sciences, especially Mr. Cagatay Ozturk, for their support and helps.

I wish to thank my parents, Ahmed and Madeeha, for their love, sacrifice, encouragement, and believing in me. Without them, I could not continue.

Last, but certainly not the least, thanks to Sura and Sally, my sisters who have been supporting and illuminating my path with their love, kindness, and sacrifices for the last three years, and who have given me the extra strength and motivation to get things done.

This thesis is dedicated to them; without them this dissertation would not be possible.

TABLE OF CONTENTS

ABSTRACT iii
ÖZ iv
ACKNOWLEDGMENTS vi
LIST OF TABLES x
LIST OF FIGURES xi
LIST OF ABBREVIATIONS xiii
CHAPTER 1 1
INTRODUCTION 1
1.1 Internet of Things (IoT) 1
1.2 Related Studies 3
1.3 Research Topic 5
1.4 Methodology of Research 5
1.5 Contribution of Research 6
1.6 Organization of Thesis 6
CHAPTER 2 7
BACKGROUND 7
2.1 Application Layer Protocols for IoT 7
2.1.1 Message Queue Telemetry Transport Protocol (MQTT) 8
2.1.1.1 MQTT Interaction Architecture 9
2.1.1.2 MQTT Features 10
2.1.1.3 MQTT Quality of Services Levels 11
2.1.1.4 MQTT Control Packet Format 12
2.1.2 Hypertext Transfer Protocol Version 2 (HTTP 2) 16
2.1.2.1 HTTP2 Features 17
2.1.2.2 HTTP2 Interaction Architecture 17
2.1.2.3 HTTP2 Frame Format 20
2.1.2.4 Frame Types 21
2.1.3 Constrained of Application Protocol (CoAP) 22
2.1.3.1 CoAP Features 23
2.1.3.2 CoAP Interaction Architecture 23
2.1.3.3 CoAP Message Types 24

2.1.3.4 CoAP Message Format ... 25
2.1.4 Extensible Messaging and Presence Protocol (XMPP) ... 26
2.1.4.1 XMPP Features ... 27
2.1.4.2 XMPP Interaction Architecture ... 27
CHAPTER 3 ... 29
QUALITATIVE COMPARISON ... 29
3.1 Comparison of IoT Application Layer Protocols ... 29
3.2 Qualitative Comparison ... 30
3.2.1 Communication Pattern ... 30
3.2.2 Security ... 31
3.2.3 Header and Payload Size ... 32
3.2.4 Fragmentation ... 33
3.2.5 Protocols Implementation ... 33
3.2.6 Organizational Support ... 33
3.3 Appropriate Protocols for the Different Requirements ... 35
CHAPTER 4 ... 37
SIMULATION TOOLS AND MODELLING ... 37
4.1 Network Simulator ... 37
4.2 IoTIFY Network Simulator ... 37
4.3 Template ... 39
4.4 Simulation Study ... 40
4.5 Measurement ... 40
4.5.1 Connection Duration ... 40
4.5.2 Packet Sending Latency ... 40
4.5.3 Message Generation Latency ... 41
4.6 Software Environment ... 41
4.7 MQTT Parameters with IoTIFY ... 42
4.8 HTTP Parameters with IoTIFY ... 43
4.9 Basic Network Simulation Parameters ... 44
4.10 Outcomes of IoTIFY ... 45
CHAPTER 5 ... 47
SIMULATION RESULTS ... 47
5.1 IoTIFY Modeler ... 47
5.2 Connection Duration Performances ... 49

5.2.1 Effects of Client Numbers and Repeating Messages on HTTP Connection Duration 50
5.2.2 Effects of Client Numbers, Repeating Messages, and QoS on MQTT Connection Duration 52
5.3 Message Generation Latency Analysis 57
5.3.1 Message Generation Latency Analysis for HTTP 57
5.3.2 Message Generation Latency Analysis for MQTT 61
5.4 Packet Sending Latency Analysis 65
5.4.1 Analysis of Packet Sending Latency for HTTP Protocol 66
5.4.2 Analysis of Packet Sending Latency for MQTT Protocol 70
5.5 Comparison of HTTP and MQTT 76
CHAPTER 6 80
CONCLUSIONS AND FUTURE WORK 80
6.1 Conclusions 80
6.2 Limitations 81
6.3 Future Work 82
REFERENCES 83

LIST OF TABLES

Table 2.1 MQTT Control Packet Types 13
Table 2.2 MQTT Flag Bit Values 14
Table 2.3 Control Packet with Packet Identifier List 15
Table 2.4 MQTT Payload List 16
Table 3. 1 Comparison of Application Layer Protocol 34
Table 3. 2 Proper Protocol for Different Requirements 35
Table 5.1 The Connection Duration of HTTP 51
Table 5.2 MQTT Connection Duration of Repeating Message 5 Times 53
Table 5.3 MQTT Connection Duration of Repeating Message 25 Times 54
Table 5.4 MQTT Connection Duration of Repeating Message 50 Times 55
Table 5.5 MQTT Connection Duration of Repeating Message 100 Times 56
Table 5. 6 Message Generation Latencies of HTTP Protocol 58
Table 5.7 MQTT Message Generation Latencies with 5 Repeats 62
Table 5.8 MQTT Message Generation Latencies with 25 Repeats 63
Table 5.9 MQTT Message Generation Latencies with 50 Repeats 64
Table 5.10 MQTT Message Generation Latencies with 100 Repeats 65
Table 5.11 The HTTP Average Packet Sending Latency 69
Table 5.12 The MQTT Average Packet Sending Latency 71
Table 5.13 Connection Time of HTTP and MQTT 77
Table 5.14 Average Message Generation Latency of MQTT and HTTP 78
Table 5.15 Average Packet Sending Latency of MQTT and HTTP 79

LIST OF FIGURES

Figure 1.1 IoT Environments 2
Figure 2.1 IoT protocols 8
Figure 2.2 MQTT Architecture 10
Figure 2.3 MQTT Control Packet Format 12
Figure 2.4 HTTP Frame Format 20
Figure 2.5 CoAP Architecture 24
Figure 2.6 CoAP Message Format 25
Figure 2.7 XMPP Interaction Architecture 28
Figure 4.1 IoTIFY Templates 38
Figure 4.2 The IoTIFY Simulation Steps 41
Figure 4.3 The Formulation Step 42
Figure 4.4 MQTT Parameters 43
Figure 4.5 HTTP Parameters 44
Figure 4.6 Network Simulation Parameters 45
Figure 4.7 Summary Result 46
Figure 4.8 State of IoTIFY 46
Figure 5.1 MQTT Scenarios Chart 48
Figure 5.2 HTTP Scenarios Chart 49
Figure 5.3 HTTP Duration of 10 Clients and 5 Repeat 50
Figure 5.4 HTTP Duration of 100 Clients and 100 Repeat 50
Figure 5.5 MQTT Duration of 25 Repeat and 25 Clients with 1 QoS 52
Figure 5.6 MQTT Duration of 50 Repeat and 50 Clients with 2 QoS 52
Figure 5.7 HTTP Message Generation Latency of 10 Client and 5 Repeat 59
Figure 5.8 HTTP Message Generation Latency of 25 Client and 25 Repeat 59
Figure 5.9 HTTP Message Generation Latency of 50 Client and 50 Repeat 60
Figure 5.10 HTTP Message Generation Latency of 100 Client and 100 Repeat 60
Figure 5.11 HTTP Packet Sending Latency of 10 Client and 5 Repeat 66
Figure 5.12 HTTP Packet Sending Latency of 25 Client and 25 Repeat 67

Figure 5.13 HTTP Packet Sending Latency of 50 Client and 50 Repeat.................. 67
Figure 5.14 HTTP Packet Sending Latency of 100 Client and 100 Repeat............... 68
Figure 5.15 Hightet Packet Sending Latency Show in Iteration 5 of Repeating Message 50 Times and Client Number 10 69
Figure 5.16 MQTT Packet Sending Latency of 5 Repeat 10 Client 0 QoS 73
Figure 5.17 MQTT Packet Sending Latency of 5 Repeat 50 Client 0 QoS 73
Figure 5.18 MQTT Packet Sending Latency of 25 Repeat 25 Client 1 QoS 74
Figure 5.19 MQTT Packet Sending Latency of 25 Repeat 100 Client 1 QoS 74
Figure 5.20 MQTT Packet Sending Latency of 50 Repeat 50 Client 2 QoS 75
Figure 5.21 MQTT Packet Sending Latency of 50 Repeat 100 Client 2 QoS 75
Figure 5.22 MQTT Packet Sending Latency of 100 Repeat 100 Client 2 QoS 76

LIST OF ABBREVIATIONS

CoAP	Constrained Application Protocol
CoRE	Constrained RESTful Environments
DDS	Data Distribution Service
DTLS	Datagram Transport Layer Security
ETSI	European Telecommunications Standards Institute
HOL	Head of Line
HTTP	Hyper Text Transport Protocol
IBM	International Business Machines
IEEE	Institute of Electrical and Electronics Engineers
IM	Instant Messaging
IoT	Internet of Things
IPSec	IP security
IPSec-ESP	Encapsulating Security Payload Protocol
ITEF	Internet Engineering Task Force
ITU	International Telecommunication Union
JID	Jabber Identifier
LAN	Local Area Network
M2M	machine to machine
MQTT	Message Queue Telemetry Transport
OASIS	Organization for the Advancement of Structured Information Standards

PAN	Personal Area Network
QoS	Qualities-of-Service
REST	Representational State Transfer
RFID	Radio-Frequency Identification
SASL	Simple Authentication and Security Layers
TLS/SSL	Transport Layer Security/Secure Sockets Layer
URI	Uniform Resource Identification
URL	Uniform Resource Locator
WAN	Wide Area Network
WSNs	Wireless Sensor Networks
XMPP	Extensible Messaging and Presence Protocol

CHAPTER 1

INTRODUCTION

1.1 Internet of Things (IoT)

IoT is a new technology that is rapidly becoming a part of modern wireless telecommunications. The main idea is to propagate devices that are able to communicate with each other over the Internet, such as Radio-Frequency Identification (RFID) tags, sensors, devices actuators, mobile phones, buildings, and more. IoT is a network with a large number of physical things that can connect to each other through the Internet and are capable of exchanging data in accordance with the networking standards and protocols. By 2025, IoT may appear in our everyday-living items; those that are not necessarily computers but still have some form of IoT in them so as to operate smartly and communicate with other things, people, or machines. The ultimate goal of IoT is to enable connection at anytime and anyplace with anything and anyone using any path/network and any service over the Internet. The main advantage of IoT is its high impact in everyday-life, this impact will be in all environments and fields, as shown in Figure 1.1, namely automation, industrial, management, logistics, intelligent transportation, smart cities, studies, healthcare, business processing, and goods and services [1].

IoT is not an individual system; rather, it comprises many applications and services; some are personal applications, others are citywide, and some are worldwide. Such diversity of applications makes the IoT system to comprise many infrastructures. The term IoT was first used in 1998/1999 by Kevin Ashton, who presented the power of RFID in supply chain system firms that count and track goods without human involvement. The prediction shows that the numbers of objects, or things, connected to the Internet surpass that of people and devices, having approached 50 billion by

2020 in different domains [2]. This increase in the number of connected devices means an increase in data, which in turn requires certain technologies such as real-time computing, machine learning, big data, security, privacy, and signal processing. As a result, this leads to the need for computer sciences, computer engineering, and electrical engineering to interact and generate a smart world with smart devices, smartphones, smart cars, smart homes, smart cities and smart industry.

Figure 1.1 IoT Environments

"Things", as the term implies in IoT, have important characteristics; they are intelligent and able to make decisions, which makes them dynamic and self-adapting. They are now components within complex services, are able to send information about themselves, and can access information released by other things from different platforms (web servers, cloud, or smartphones to name a few). Therefore, IoT things are heterogeneous as they are interconnected with different platforms and networks. The capability of having a unique IP-address makes them self-configuring and gives them the advantage to reduce human interaction. Also, the connection over the Internet makes IoT cost-effective [3].

To better understand this technology, one has to first learn the IoT architecture, which comprises three layers: The first layer is the sensor layer which confronts the physical layer in the network stack. This layer is made of smart objects integrated with sensors to collect and process physical data and to convert them into digital signals understood

by upper layers. There are many sensors, namely environmental sensors, body sensors, home appliance sensors, vehicle telematics sensors, etc. The second layer is the gateway layer, or data link layer, where sensors are connected to a gateway to exchange their data by means of a Local Area Network (LAN) such as Ethernet and Wi-Fi, or through a Personal Area Network (PAN) such as ZigBee, Bluetooth, and Ultra-Wideband. The third layer is the network layer where the vast data collected from the lower layers need to be transferred across the networks using global protocols and standards. This layer includes IPv6/IPv4 as a network protocol, TCP and UDP as the transport protocol and the application layer protocol, which are the focus of this thesis. Each IoT device must have a management service layer as a supportive layer to provide services such as analysis of information, security, modeling process, management of devices, and supporting rule engines by formulating the decision logic. The management layer is necessary because data transfer across different infrastructures may require filtering, capturing, or immediate actions [4].

Each IoT implementation must use an application protocol, deciding upon which is a critical issue. When the IoT technology appeared, it immediately became a hot topic for both the academic and the industrial sectors, leading to many application protocols created to support IoT data transmission. The IoT application layer protocol performance is a critical factor for the entire IoT system. On the other hand, conformance to the global platforms and network organizations is another requirement. There are many different alternatives for application protocols, such as Message Queue Telemetry Transport (MQTT), Constrained Application Protocol (CoAP), Data Distribution Service (DDS), Extensible Messaging and Presence Protocol (XMPP), Hyper Text Transport Protocol 2.0 (HTTP). Yet, because HTTP is has been accepted by all infrastructures and somehow all protocols depend on it, there have been attemtps likewise to make it compatible with IoT devices [5].

1.2 Related Studies

The papers [6], [7], and [8] provided a survey on the application layer protocols and a comparison among the most common application protocols. They explain their characteristics and structures by giving a vision of how these protocols work and where

they can be implemented. According to the comparisons, MQTT can be considered as a better protocol for applications with overhead and power-sensitivity; if the application is constrained, though, CoAP is a good choice; the best option for applications built with XML is XMPP, while the HTTP is suitable for all kinds of applications.

In [9], a real testbed is done to collect the related environment data and to analyze the performance of application protocols under the same conditions. A quantitative comparison is carried out in terms of packet creation time and packet transmission time for the MQTT, CoAP, and XMPP protocols. The results show that MQTT performs faster than CoAP, while XMPP is the lowest according to the XML usage that adds extra latency.

The performance of application protocols in the medical sector is presented in [10], where a quantitative comparison among MQTT, DDS, and CoAP is made. The protocols performances were evaluated using a network emulator according to bandwidth, latency, and packet loss. The outcomes show protocols that use TCP (MQTT, DDS) resulted in no packet loss as opposed to protocols that use UDP (CoAP). For reliability and low latency, DDS is proposed as the best option for IoT medical applications.

Another comparison is made between MQTT and HTTP in [11], the testbed was a temperature data request from a client app to a server app that is, from node ESP8266. The outcomes show that MQTT performance is better from the point of low power usage and low latency.

Although there have been several studies investigating application level protocols for IoT systems, they are very limited in terms of quantity and content. Therefore, more detailed studies are needed to investigate IoT protocols to develop more efficient systems for this purpose. This is a gap in the literature which the present thesis aims to fill for IoT practitioners.

1.3 Research Topic

The existing studies and models have mostly used MQTT and CoAP as their application layer protocol for IoT and related comparisons. Henceforth, there is a clear shortage of applying HTTP as an application layer protocol in IoT, in providing a systematic comparison with other protocols, and in performance evaluation related to HTTP [12]. Although there are many studies that compare HTTP with other protocols in terms of the main characteristics and structures, there are very limited studies to compare HTTP with other IoT application protocols in practical terms and actual settings. In this vein, the present thesis will focus on shedding light on the performance of HTTP and MQTT as IoT application protocols for the same device and with different parameters.

1.4 Methodology of Research

In this research, an explanation is provided pertaining to IoT application layer protocols (MQTT, HTTP, CoAP, and XMPP), whose features, architecture, message types and format are elaborated upon. Also, a theoretical comparison is done among these application layer protocols to understand the differences and similarities among them. In practical terms, the performances of MQTT and HTTP are investigated and since IoT applications are short on HTTP usage, the performance of these two protocols are further explored according to a designed simulation model of a connected-car. The simulation is conducted in terms of connection duration and latency. More specifically, the study aims is to answer two questions in this respect:

- Is it possible to use the HTTP protocol as the application layer protocol of IoT?
- If positive, what is the performance of HTTP compared to MQTT?

64 different simulation scnerios are designed and tested for comparison of HTTP and MQTT protocols. These scnerios are basically obtained by changing number of clients (10, 25, 50, and 100) and, number of repeating-messages (5, 25, 50, and 100). Next, the comparison between protocols is done by checking the connection duration, message generation latency, and packet sending latency of the same client number and repeat times.

1.5 Contribution of Research

Previous attempts in this field have focused on comparing the evaluation of MQTT with CoAP or XMPP, excluding HTTP as such. In addition, there are few studies that evaluate HTTP behavior in the IoT devices. Thus, this study provides a detailed analysis of performance in terms of connection duration, message generation latency, and packet sending latency. More specically, this thesis contributes to the domain by:

- Providing a theoretical comparison for the common IoT application protocols (MQTT, HTPP, CoAP, and XMPP); and
- Providing practical performance analysis for MQTT and HTTP.

1.6 Organization of Thesis

This thesis is structured within six main chapters. Chapter 1 introduces the topic with a discussion of IoT application protocols along with the problem statement. Chapter 2 provides the basic information and background about MQTT, HTTP, CoAP, and XMPP as application layer protocols in IoT. A qualitative comparison of IoT protocols is provided in Chapter 3. Chapter 4 explains our method and models along with the simulation tool IoTIFY for a practical comparison of IoT application layer protocols. The obtained test results and their analysis of two focused protocols (MQTT, HTTP) are presented in Chapter 5. Finally, conclusions and future work are given in Chapter 6.

CHAPTER 2

BACKGROUND

The present chapter provides an overview of the most common Internet of Things (IoT) Application Layer Protocols. It includes the architectures and message formats with the important features of each protocol.

2.1 Application Layer Protocols for IoT

IoT achieves its significance by achieving collaboration among several technologies, devices, and internet protocols; therefore, there is a need for special standards and communication protocols for IoT. Based on this need, there have been protocols offered by the Institute of Electrical and Electronics Engineers (IEEE), International Telecommunication Union (ITU), Internet Engineering Task Force (IETF), the European Telecommunications Standards Institute (ETSI) and other standards organizations. A large amount of data is generated and needs to be transferred to the data centers in IoT applications. These protocols are applied in many smart application domains such as homes, cities, agriculture, water supply, transportation, healthcare monitoring, utilities, animal farming, security, emergencies, industrial control, environment monitoring, etc.

Application protocols allow messaging among several of IoT communication parties and end-users over the Internet. Generally, IoT applications and other applications use the IP protocol used to transport the TCP/UDP segments. These application protocols are a challenge in IoT system, where none of them is able to pass all their message types, thereby making it necessary to understand the pros and cons of application protocols in further detail [8]. Deciding the usage of a protocol is a tradeoff of a measurable factor of performance vs. features vs. compatibility depending on the design or the platform support [13].

Figure 2.1 shows the generally used protocols by the IoT applications. Since this thesis aims to analyze application layer protocols, some basic information is provided about them in the following sections.

Network stack	**protocols**
Application layer	MQTT, CoAP, XMPP,DDS,AMQP,HTPP
Transport layer	UDP,TCP
Network layer	6LowPAN, 6TiSCH, 6Lo, IPV6, RPL, CoRPL, CARP, E-CARP
Data link and physical layer	WiFi, IEEE 802.15.4, Bluetooth low energy, Z-Wave, 3G/LTE, NFC, HomePlug, LoRaWAN, Wireless HART, ZigBee smaet energy, DASH7, G.9959, wirghtless, DECT/ULE, EnOcean

Figure 2.1 IoT protocols

2.1.1 Message Queue Telemetry Transport Protocol (MQTT)

This is a lightweight publish/subscribe message protocol designed for machine to machine (M2M) communication, constrained devices with low power requirement, and rare for untypical network connection conditions such as low bandwidth and high latency. It was introduced by the International Business Machines (IBM) Company in 1999, the latest version being MQTT v3.1.1. Because of the simplicity and small message header, MQTT was accepted as an IoT application protocol by the Organization for the Advancement of Structured Information Standards (OASIS) in 2013 [13]. MQTT runs over TCP on port number 1883. Security in MQTT includes username and password, and additional mechanisms that depend on the MQTT broker. Each broker may have a specific mechanism based on Transport Layer Security (TLS) which runs on the port number 8883 [14].

2.1.1.1 MQTT Interaction Architecture

The MQTT publish/subscribe architecture is composed of three elements:

1. **Clients:** any IoT devices or applications that are able to publish messages, subscribe to message, or do both can be a client. The MQTT client libraries must be installed in all clients.
2. **Topic**: a message type in MQTT is called topic. The client may subscribe to many topics, while different clients may as well subscribe to the same topic. The topic format is topic-name followed by /. For example, the temperature is a topic name that client wants to subscribe (temperature/).
3. **Broker**: this is the central piece of MQTT architecture which contains a list of published topics. Client can receive the corresponding message from the broker for topic name that client subscribes. For devices that have the role of a broker, it is important to install broker library. There are various open-source MQTT brokers, for example, Mosquitto broker, RabbitMQ, Emqttd, ActiveMQ, VerneMQ, IBM MessageSight, and JoramMQ. Broker is positioned in the middle of communication with two or more clients (publisher and subscriber). Figure 2.2 shows the MQTT architecture, a message is published from client to broker under a specific topic name. This message will be temporarily stored in the broker; then, subscribers will send a subscribe-message to the broker with topic name in order to receive the message published under the same topic name [6].

   There are important expressions in the MQTT architecture [15]:

   - **Topic name:** The label indicating name of information that is sent over MQTT message. Clients subscribe to a topic name that they want to get their data from.
   - **Topic filter:** An expression in client subscription which refers to one or more topics.
   - **Session:** It is the effective connection among clients and server. Some sessions last as long as the network connection remains active, whereas others are extended from one connection to another.
   - **MQTT Control Packet:** There are 14 various types of control packets sent across MQTT connection. These packets contain information about how the application messages are transferred.

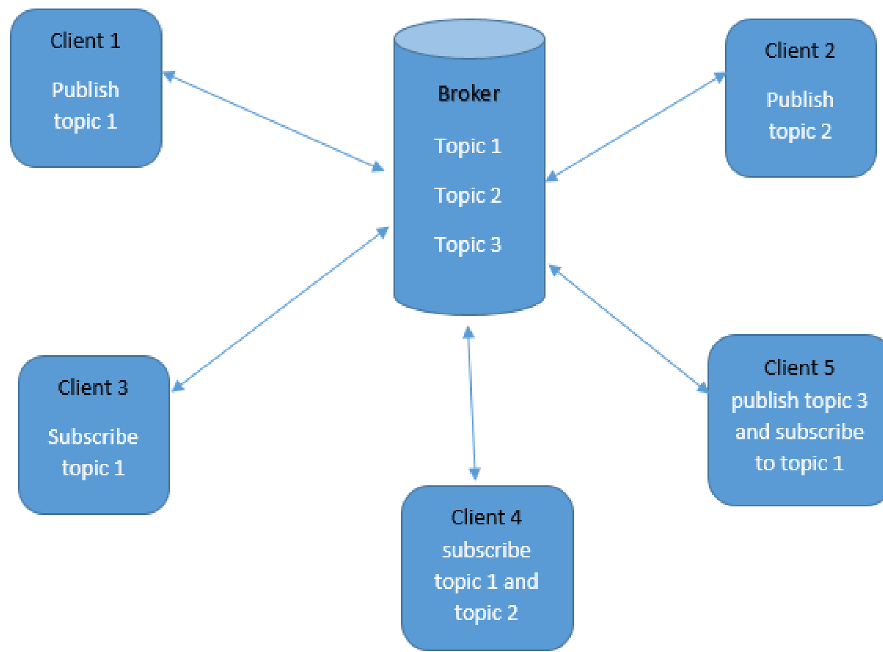


Figure 2.2 MQTT Architecture

2.1.1.2 MQTT Features

The important features in MQTT are [15]:

1. The network connection is built over TCP/IP.
2. Publish-subscribe structure of MQTT is one-to-many message distribution.
3. The header in MQTT is 2 bytes, which is small and helps in decreasing overhead and network traffic.
4. The Last Will and Testament features are used to inform the clients of any unusual disconnections by any other client.
5. Three Qualities-of-Service (QoS) types to deliver a message: “at most once”, “at least once” and “exactly once”. These will be detailed in the following section.
6. It is able to store the message by setting the “retain” flag value to “true” in order to store the message for new subscribers.
7. It is a lightweight protocol and does not provide message encryption. Thus, messages exchanges are done as plain-text.
8. It provides authentication by setting CONNECT message, which defines a username and a password requested by a broker to validate the connection.

2.1.1.3 MQTT Quality of Services Levels

As stated above, the three types of QoS to deliver an MQTT message are [16]:

1. **At most once** (QoS= 0): There is no guarantee that message will arrive. This type is suitable for sensor data, in which message loss is not a problem since the next data will arrive soon. The client publishes a message to broker, who acts toward publishing the message to subscribers.

2. **At least once** (QoS= 1): In this type, duplications can occur, but it definitely delivers the message at least once. The process is done with two steps (1) The client publishes a message to broker and stores it. (2) The broker stores message first, publishes it to subscribers, deletes it, and finally sends a Puback message. Puback is a server acknowledgment to receive a message. If client does not receive it, then the first message will be resent; otherwise, client will discard message.

3. **Exactly once** (QoS =2): This type is ideal for important systems where message must arrive exactly once. The process of sending message follows these steps: (1) The client publishes message to the broker and stores it. (2) The broker first stores message or message ID, then publishes it, and finally sends a Pubrec as an acknowledgment. (3) The client acknowledges with a Pubrel message. (4) The broker, then, publishes message, deletes it or deletes ID, and sends Pubcomp. If a client does not get this Pubcomp, the missing part of the message after Pubrel has arrived will be sent again.

2.1.1.4 MQTT Control Packet Format

MQTT has three control packet formats: Fixed header, Variable header, Payload.

1. **Fixed header:** it is presented in all MQTT control packets, where the header size is 2 byte as shown in Figure 2.3. The first byte contains control packet type and flags type, while the second byte represents the remaining length [15]. The control packet type is represented in bits 4-7 of first byte, as shown in Table 2.1. The flags are represented in bits 0-3 for each MQTT control packet. These four bits are reserved in all control packet types for future use, except PUBLISH as shown in Table 2.2. Where DUP flag refers to the duplicate delivery of PUBLISH. QoS refers to PUBLISH quality of service, and RETAIN refers to PUBLISH retain. The remaining length is four bytes which represents remaining bytes for control packet, data, and payload.

<table>
<tr><td rowspan="2">Byte 1</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="4">MQTT Control Packet type</td><td colspan="4">Flags specific to each MQTT Control Packet type</td></tr>
<tr><td>Byte 2</td><td colspan="8">Remaining Length</td></tr>
</table>

Figure 2.3 MQTT Control Packet Format

Table 2.1 MQTT Control Packet Types

Name	value	Direction of flow	Description
Reserved	0	Forbidden	Reserved
CONNECT	1	Client to Server	Client request to connect to Server
CONNACK	2	Server to Client	Connect acknowledgment
PUBLISH	3	Client to Server Or Server to Client	Publish message
PUBACK	4	Client to Server or Server to Client	Publish acknowledgment
PUBREC	5	Client to Server or Server to Client	Publish received (assured delivery part 1)
PUBREL	6	Client to Server or Server to Client	Publish release (assured delivery part 2)
PUBCOMP	7	Client to Server or Server to Client	Publish complete (assured delivery part 3)
SUBSCRI	8	Client to Server	Client subscribe request
SUBACK	9	Server to Client	Subscribe acknowledgment
UNSUBSCRIBE	10	Client to Server	Unsubscribe request
UNSUBACK	11	Server to Client	Unsubscribe acknowledgment
PINGREQ	12	Client to Server	PING request
PINGRESP	13	Server to Client	PING response
DISCONNECT	14	Client to Server	Client is disconnecting
Reserved	15	Forbidden	Reserved

Table 2.2 MQTT Flag Bit Values

Control Packet	**Fixed header flags**	**Bit 3**	**Bit 2**	**Bit 1**	**Bit 0**
CONNECT	Reserved	0	0	0	0
CONNACK	Reserved	0	0	0	0
PUBLISH	Used in MQTT 3.1.1	DUP	QoS	QoS	RETAIN
PUBACK	Reserved	0	0	0	0
PUBREC	Reserved	0	0	0	0
PUBREL	Reserved	0	0	1	0
PUBCOMP	Reserved	0	0	0	0
SUBSCRIBE	Reserved	0	0	1	0
SUBACK	Reserved	0	0	0	0
UNSUBSCRIBE	Reserved	0	0	1	0
UNSUBACK	Reserved	0	0	0	0
PINGREQ	Reserved	0	0	0	0
PINGRESP	Reserved	0	0	0	0
DISCONNECT	Reserved	0	0	0	0

2. **Variable header:** Located between fixed header and payload in some type of control packets, it depends on 2 bytes of packet identifier field, as shown in Table 2.3. Client and server assign packet identifier independently, but they can exchange messages synchronously using the same packet identifier. Packet and corresponding packet must have the same packet identifier. In PUBLISH packet with QoS =1, the corresponding packet is PUBACK, and in QoS=2, it is PUBCOMP. While in SUBSCRIBE or UNSUBSCRIBE, the corresponding packet is SUBACK or UNSUBACK [15].

Table 2.3 Control Packet with Packet Identifier List

Control packet	**Packet identifier field**
CONNECT	NO
CONNACK	NO
PUBLISH	YES (If QoS > 0)
PUBACK	YES
PUBREC	YES
PUBREL	YES
PUBCOMP	YES
SUBSCRIBE	YES
SUBACK	YES
UNSUBSCRIBE	YES
UNSUBACK	YES
PINGREQ	NO
PINGRESP	NO
DISCONNECT	NO

3. Payload: Some of MQTT control packets have a payload at the end of control packet, as listed in Table 2.4.

GCHS

Table 2.4 MQTT Payload List

Control packet	**Payload**
CONNECT	Required
CONNACK	None
PUBLISH	Optional
PUBACK	None
PUBREC	None
PUBREL	None
PUBCOMP	None
SUBSCRIBE	Required
SUBACK	Required
UNSUBSCRIBE	Required
UNSUBACK	None
PINGREQ	None

2.1.2 Hypertext Transfer Protocol Version 2 (HTTP 2)

As stated before, HTTP is the most common web protocol used by everyone, especially developers, because of its harmonious capability with all networking infrastructures. HTTP is essentially an example of a client-server model which uses a Uniform Resource Locator (URL) [17]. It is emerged by IETF as a request-response RESTful Web architecture that runs over TCP. HTTP uses Transport Layer Security/Secure Sockets Layer (TLS/SSL) for security. The older HTTP versions have a long and repeated header. Such issues affect network performance and make it unfit for sending small data or when using constrained devices, hence inapplicable to IoT. All these drawbacks paved the way toward developing HTTP version 2.0 in 2015. This version has a header field compression for optimized transportation, applies the methodology of server as push and priority, and reduces latency to the levels applicable for IoT systems [18].

2.1.2.1 HTTP2 Features

HTTP2 has the following important features:

1. It is an application layer protocol run over TCP.
2. The basic transmitted unit is a frame.
3. There is no QoS; instead, ensuring the delivery of data is done with TCP.
4. It uses the binary message framing that can process messages efficiently.
5. It decreases the number of TCP messages to improve network capacity.
6. It has a port number similar to HTTP1.1 (443 for https and 80 for http).
7. It improves network performance by allowing an overlap of request and response in the same connection, coding the HTTP header, as well prioritization to process important requests faster.
8. To discover if servers support HTTP2 or not, the HTTP2 bears two identifications when sending URL [19].
9. When it runs over TLS, the string “h2” identifier is used in the TLS application layer negotiation field.
10. The string “h2c” is the second identifier used when HTTP2 is run over a clear text of TCP.
11. Choosing between “h2” and “h2c” depends on transport, framing, message semantics, and security.

2.1.2.2 HTTP2 Interaction Architecture

HTTP follows a client-server model where communication between the two is carried out with a request-response message. Client sends a URL request to server, who sends back a response as URL content requested by client. HTTP is linked with Representational State Transfer (REST), an architectural design for large-scale networked software across the World Wide Web protocols and technologies. The REST architecture functions are: defining and addressing resources, and responding to any resources requested by HTTP client. REST is adopted by many cloud platforms and supported by both XML and JSON [20].

The standardized method to create, read, update, and delete data (CRUD operations) in REST corresponds with that in HTTP2 (POST, GET, PUT and DELETE). Thus, the integration between HTTP2 and REST is typical for IoT [6].

With HTTP 1.1, the GET request for the same server is performed by consecutive requests that waste network resources and increase latency. For this, a pipelining method was introduced to allow multiple requests to be sent simultaneously without having to wait for the first answer. However, with large and extended responses, this solution can face a problem called Head of Line (HOL). If the response is big, the next responses get delayed until the previous response arrives. To solve HOL, several TCP connections were set up to the same server, but these TCP connections are still limited (6 in Chrome, 15 in Firefox). Another solution is distributing the content on multiple servers with a different IP address, and then by sending multiple-parallel TCP requests to these servers, client can get requested content at the same time. Though, this solution proved to be ineffective, too; therefore, HTTP2 was developed to solve the problems as a whole [21]. In HTTP2 there is no change in the core concepts and functionality of previous HTTP versions, yet this version helps with problems that affect network performance and latency to make it compatible with the latest technologies.

The followings are the new mechanisms applied in HTTP2 to improve its performance:

1. **Binary framing layer:** The major enhancement in HTTP2 is its new binary framing layer. It is an encoding mechanism providing message encapsulation and transmission between client and server. The HTTP2 message is encoded in a binary format and divided into smaller frames and messages on a bidirectional stream. Both client and server must follow the binary framing to successfully communicate. A frame is the smallest unit of HTTP2 communication, either a header or a payload; while a request or response message is a sequence of these frames and each message belongs to a bidirectional stream that may carry one or more messages [22].
2. **Full request and response multiplexing:** To solve the HOL problem and reduce multiple parallel TCP connections for the same endpoint, the binary framing layer enables full request and response multiplexing. By allowing breaking down of the HTTP messages into separate frames, it becomes possible to interleave them and

collect them at the endpoint. This means many frames can be transmitted on a single TCP connection in parallel with different streams and different message type without any blocking [22].

3. **Stream prioritization:** While a TCP connection is split into frames, stream multiplexing calls for a new mechanism to decide which stream must be processed first. This stream must dominate the usage of CPU, memory, and bandwidth to guarantee high-performance for the client response. The prioritization mechanism in HTTP2 is developed by providing a prioritization tree on server side. HTTP2 assigns each stream an integer weight (1-256) and an explicit dependency on another stream in the prioritizing tree. Then, server prioritizes a streams by checking the tree, starting at its highest point (explicit dependency) and with less weight [22] .
4. **Server push:** It is an interesting functionality of HTTP2 that decreases time for loading pages and reduces TCP requests. In addition to the resource requested by a client, server can predict the next requests that a client may need in future related to the first request; it pushes for resources without waiting explicit requests from the client. These predictive contents are useful and, hence, stored in the browser cache. Client decides whether to accept or reject these resources [22].
5. **Header compression:** Each HTTP1.x request or response is carrying metadata for transferred resources. This metadata is sent as plain text, which adds 500-800 bytes overhead to each HTTP request or response [22]. Against this, HTTP2 comes with a compression format named HPACK to reduce such overhead and improve the performance of HTTP by using two techniques: encoding by Huffman code, and inclusion in the static or dynamic tables. Huffman code is an encoded header field and provides an indexed list for previous headers seen by both client and server. The static table consists of the most common HTTP headers field, whereas the dynamic table is initiated as empty and updated with exchange values of a particular connection. Thus, the size of each HTTP request reduces because the header requested is either stored in the index list of Huffman code or already exits in the static or dynamic list [21].

2.1.2.3 HTTP2 Frame Format

The frame is the transmitter unit for HTTP2, and it starts with a 9-octal fixed header and variable length payload, as shown in Figure 2.4.

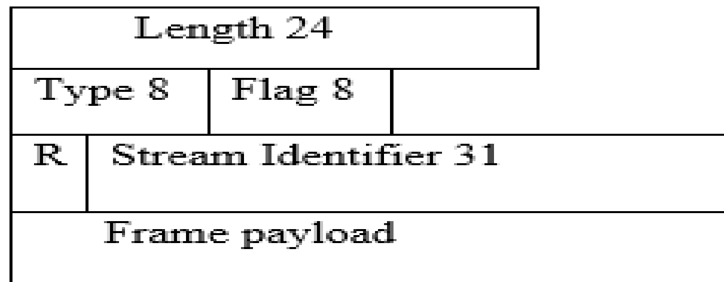


Figure 2.4 HTTP Frame Format

The fields of frame header are [19]:

- **Length:** 24-bit explaining payload length, whose value must not be greater than (2^14).
- **Type**: 8-bit to define frame format and semantics.
- **Flag:** 8-bit reserved for the Boolean value of the frame type. The flag that does not have semantics must be unset to (0x0).
- **R:** 1-bit unset to (0x0) for sending, and ignored for receiving because the semantics of R-bit is not defined.
- **Stream identifier:** 31-bit unsigned integer. The stream identifier of a client to the initial stream must be an odd number, while server uses an even number. The 0x0 value is reserved for connection control messages.
- **Payload:** The content of payload and its structure depends on frame type. The minimum size of payload is up to (2^14) octal with a 9-octal header. The maximum payload is defined on receiver side and between (2^14) to (2^24 -1) octal in the SETTINGS_MAX_FRAME_SIZE.

2.1.2.4 Frame Types

HTTP2 has 10 frame types [19]:

1. **Frame type (0x0):** It is the DATA frame with an alternative of octet variable length sequence indicating HTTP request or response. Data frame may have a padding field to hide the size of a message.
2. **Frame type (0x1):** Or the HEADER frame, as it is often called, is used for opening streams. This frame has a header block fragment and may have padding with length if padding is used. A header can only have a 31-bit stream dependency, 8 bit for weight, and a single-bit flag E to signify that stream dependency is exclusive only when the PRIORITY flag is on.
3. **Frame type (0x2):** It is a PRIORITY frame that identifies the sender-advised priority of any stream. It must have a stream dependency, weight, and E.
4. **Frame type (0x3):** It is the RST_STREAM frame that cuts off stream, cancels it, or highlights an error. The 0x3 frame has only 32-bits to identify error codes.
5. **Frame type (0x4):** or the SETTINGS frame, transfers the configuration parameters of how client and server communicate. To adjust settings frame for connection, both sides must send settings frame at the beginning or at any time during connection. The 0x4 frame contains a 16-bit setting identifier and a 32-bit value.
6. **Frame type (0x5):** It is the PUSH_PROMISE frame that notifies the peer of sender who plan to initiate a PUSH_PROMISE frame of an open or half-closed stream state. It may have a Pad length and Padding, a header block fragment, reserved bit (R), and also a 31-bit promised stream ID to indicate that the stream is received by push promise.
7. **Frame type (0x6**): Or the PING frame, is a mechanism to scale minimal round-trip time from the sender, and to check if the idle connection is still active. The PING frame must have opaque data (8 octal) in payload addition to the frame header. These opaque data can include any value and can be used in any fashion.
8. **Frame type (0x7):** Or GOAWAY frame is used to start shutting down a connection by stopping receiving a new stream. However, the previous stream will continue to process. GOAWAY may also be used to signalize a serious error.

It contains an R bit, a 31-bit last stream ID, a 32-bit error code, and additional debug data.

9. **Frame type (0x8):** It is the WINDOW_UPDATE frame used to perform flow control of each peer and of the whole connection. This frame contains an R bit and a 31-bit window size increment (1- 2^31-1) that defines the limit of octal number the sender sends with the existing flow control.
10. **Frame type (0x9):** It is the CONTINUATION frame used for maintaining a sequence of header block fragments so it includes header block fragment for any designated number.

2.1.3 Constrained of Application Protocol (CoAP)

It is an M2M protocol designed in 2010 for constrained nodes with low CPU and RAM, and for constrained networks with low power and low bandwidth as a web transfer protocol. CoAP is designed by IETF and Constrained RESTful environments (CoRE) Working Group to meet the IoT requirements. The CoAP protocol follows REST architecture, similar to HTTP, and can be a client-server model to exchange the request-response of applications that support unicast communications, or serve as a publish-subscribe model that supports multicast communications. It is one of the lightweight protocols that run over UDP to reduce overhead on port number 5683. CoAP is applicable in different domains, such as smart homes, mobile IoT deployments, cloud services, healthcare, smart cities, and industrial applications [23]. CoAP applies the Datagram Transport Layer Security (DTLS), which is based on TLS but with some changes to fit the UPD connection. DTLS adds other verification queries to the handshaking message between the client and server, and makes sure that client sends “hello” message from the original address. This mechanism is used to prevent denial of service attacks and support data exchange protection [6].

2.1.3.1 CoAP Features

Important CoAP features can be summarized as follows:

1. It is a lightweight application protocol that runs over UDP.
2. It supports the unicast and multicast transport methods.
3. It is a synchronous and asynchronous request-response application layer protocol.
4. It specifies 2 bits in the header to represent the message type and quality of services.
5. The client-server architecture in CoAP uses RESTful-HTTP to provide resource-oriented interactions.
6. The CoAP packet has 2 bytes for message ID in its header which is unique and used to detect duplicates [7].

2.1.3.2 CoAP Interaction Architecture

The CoAP protocol has two logically different layers: (1) the request-response layer which applies RESTFul mode by using the CRUD operations of HTTP; the only difference is that response message is not sent over the same request connection, making it an asynchronous message [24]. In a client-server architecture, the client uses GET method with Uniform Resource Identification (URI) to request sensor data as shown in Figure 2.5. Later, the server sends back response. The client also can use POST method to update sensor data. (2) Even if CoAP uses UDP which does not ensure reliability, the message layer provides such reliability by defining various message types of CoAP to retransmit the lost data packets [6]. CoAP supports publish-subscribe model that is used to support a large number of IoT users better than client-server model. In this way, publisher sends a multicast message, which can then be received by many subscribers [13].

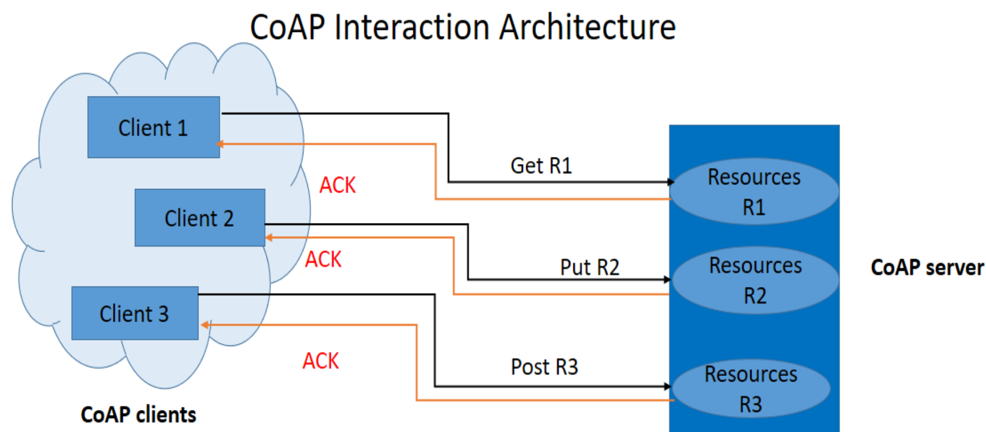


Figure 2.5 CoAP Architecture

2.1.3.3 CoAP Message Types

There are 7 message types in CoAP; the first four are the main types, while others depend on them [25]:

1. **Conformable Message (CON):** The message requested requires an acknowledgment reply indicating that message has arrived correctly.
2. **Non-Conformable Message (NON):** The message requested does not require an acknowledgment reply.
3. **Acknowledgment Message (ACK):** A message to confirm that the Conformable Message has arrived at the receiver side.
4. **Reset Message:** It is a message used to reboot the missing context process for both Conformable and Non-Conformable at the receiver side by sending an empty Conformable message.
5. **Piggybacked Response:** A response message including an acknowledgment that confirms the sender-request is answered.
6. **Separate Response:** It is a Conformable message with an empty Acknowledgment; a server sends a response in many separate response messages when it does not have a complete response yet.
7. **Empty Message**: A 4-byte header with a 0.00 code that is neither a request message nor a response message.

2.1.3.4 CoAP Message Format

The message format of CoAP is shown in Figure 2.6, and it consists of a header, a token value, an option, and a payload [25].

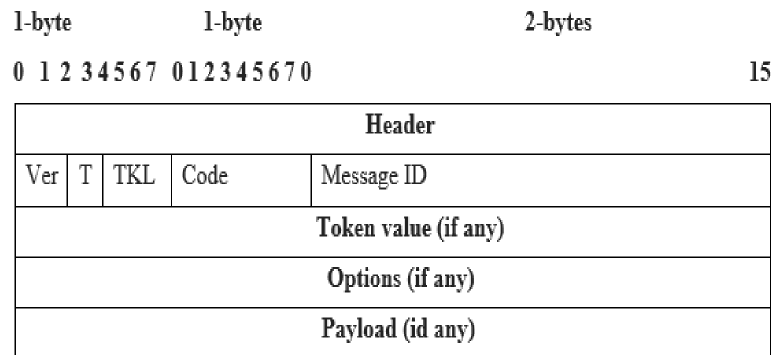


Figure 2.6 CoAP Message Format

The header of a CoAP message is a fixed 4 bytes consisting:

- **Version (Ver):** It refers to the current version of the CoAP in 2 bits.
- **Type of CoAP message (T):** It refers to the message type in 2 bits, where the Conformable T is (0), the Non-conformable T is (1), the Acknowledgment T is (2), and Reset T is (3).
- **Token Length (TKL):** The length of variable-length-token is in 4 bits. The length values (9-15) are reserved and must not be sent or processed.
- **Code:** The 8 bits are divided to two parts: the most significant 3 bits refer to class, with a one-digit value from 0-7. When the class has a value of 0, it refers to the request; 2 refers to the success response; 4 refers to a client error response; and 5 refers to a server error response, while the other values are reserved. The second part, least significant 5-bits, refers to details with two-digit value 00-31. Special code 0.00 refers to an empty message.

- **Message-ID:** The 16 bit that is used to detect duplicated messages and to match message types (Acknowledgement/Reset to message type Conformable/Non-conformable).
- **Token value:** It is defined in TKL from 0 to 8 bytes; it is used to link the request and response.
- **Options:** CoAP messages may have many options or none.
- **Payload:** If any, it comes after the options field with a prefix as a payload marker (0xff) to indicate the end of options and the start of the payload. In turn, Payload size is calculated from marker until the end of UDP datagram.

2.1.4 Extensible Messaging and Presence Protocol (XMPP)

XMPP is one of the application layer protocols which was developed in 1999 by the open-source Jabber community. The protocol is designed for real-time applications like chats and message applications, voice and video calling, and telepresence. XMPP is an open standard of IETF as an Instant Messaging (IM), based on XML and it runs over TCP. It is compatible with different protocols and operating systems, containing many mechanisms including end-to-end and hop-to-hop encryptions, privacy measurement, authentication, and access control for IM applications [26]. XMPP is high-scalable because of its decentralized architecture as shown in Figure 2.7. It supports both request-response and publish-subscribe message exchange models, where devices need to talk with servers in two-way communications. It incorporates TLS mechanism for security, which provides reliability, confidentiality, and data integrity. A native security mechanism with XMPP client is Simple Authentication and Security Layers (SASL) that guarantees server validation [2].

2.1.4.1 XMPP Features

1. It does not guarantee QoS.
2. It is designed for real-time applications to exchange small messages with low latency.
3. It provides an asynchronous and synchronous message because of using publish-subscribe and request-response model to exchange the message.
4. It is not suitable for lossy, low power wireless networks because of the persistent TCP connection that makes it also less suitable for IoT.
5. With publish-subscribe model that makes XMPP a lightweight protocol, it offers optimization for IoT devices [6].

2.1.4.2 XMPP Interaction Architecture

XMPP is a protocol running over a variety of Internet-based platforms in a decentralized architecture. It is a client-server-server-client model , where clients do not connect directly to each other and, instead, they use servers as a mediator. To do so, a client connects with the server through TCP port number 5222, while servers communicate with each other on port number 5269. The most common XMPP server is Openfire. Each client is defined by a unique address called Jabber Identifier (JID), which resembles an email address with three parts: user, server, and resource. The ability to run over various internet platforms solves the problem of blocking XMPP connections. A client can send a message with XMPP and the server can listen with a normal HTTP port [2].

XMPP makes use of publish-subscribe model by allowing for the creation of topics and broadcasting information for subscribers, thereby making it more suitable for constraining devices. XMPP communication makes use of stream of XML stanzas (semantic data units to send information among XML entities) as shown in Figure 2.7. There are three types of stanza in an XMPP message: <message>, <presence>, and <iq> (info/query). The message stanza is responsible for sending messages among XMPP entities, and it defines source and destination address, type, and ID of XMPP

entities. Presence stanza is responsible for notifying clients subscribed to the present JID about the availability of any update.

The <iq> stanza function is similar to the HTTP GET and POST; <iq> is used to get information from server by both the sender and receiver, to apply some settings on server, and to learn more about server and registered clients [6].

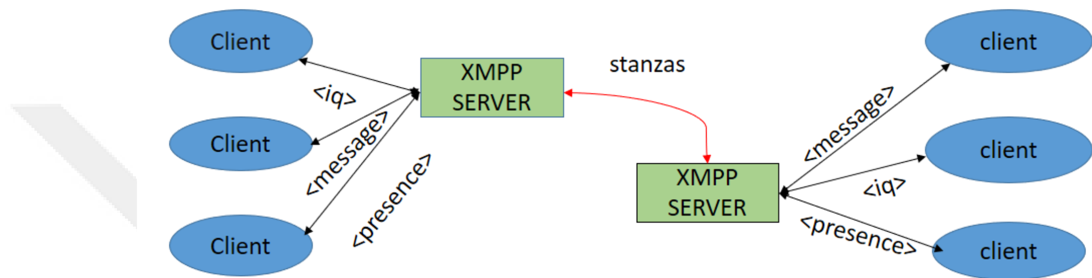


Figure 2.7 XMPP Interaction Architecture

CHAPTER 3

QUALITATIVE COMPARISON

3.1 Comparison of IoT Application Layer Protocols

There is no IoT protocol guaranting all IoT requirements. Therefore, there is a need to compare existing IoT application layer protocols to be able to determine the appropriate ones for different types of user requirements. This is a qualitative comparison based on the literature, which can help the stakeholders who are interested in the IoT protocols. This comparison would also help engineers to select an appropriate protocol depending on their needs by analyzing network performance, battery life prediction, latency requirement, bandwidth, and device capability along with security, reliability, interoperability, and scalability [27].

There are different application protocols and they vary from each other. These protocols are used in different IoT environments and for different needs. They have large potentials for IoT developments. Therefore, it is important to analyze their corresponding limitations and the strengths against one another. The relative comparison of IoT application protocols depends on many criteria such as IoT system, devices, resources, applications, conditions, and specific requirements for system and network. These comparisons are based on the static features of the protocols and some experimental evidence from the literature, and they do not consider the dynamic behaviors of network conditions or costs of a retransmitted message, factors that can change the comparison results extensively [17].

The major questions to be raised in this respect are: What are the important characteristics of these protocols (common or different)? What are their differences? How can one decide which IoT protocol is suitable for a specific application? Which message exchange architecture is used by these protocols? Which transport layer

protocol is used by these protocols? What are their header and message sizes, their QoS support, security support, latency, and other features?

3.2 Qualitative Comparison

It is important to present a qualitative comparison for the application layer protocols (MQTT, HTTP, CoAP, and XMPP) by discussing their characteristics and the differences among them. The protocols are compared considering different factors in this section, while a summary of the comparison is given in Table 3.1.

3.2.1 Communication Pattern

The communication in any application protocol contains numerous categories that vary from one to another. The communication paradigm for MQTT protocol is publish-subscribe where there are clients and a broker, with data put in topics. Clients can publish topics with the help of a broker, and other clients can subscribe to topics as well. This communication paradigm depends on the broker, where broker controls the communication and delivered topic data to subscribers [28]. The HTTP protocol uses a client-server communication paradigm as a request-response architecture where client requests data from server that sends back requested data. On the other hand, the communication of CoAP and XMPP can follow both request-response paradigm and publish-subscribe paradigm [26]. The MQTT, CoAP and XMPP protocols support multicast communication, which means that server or broker can send data to one or more clients at the same time.

According to the network architecture, all application-layer protocols run on transport layer protocols, that is either UDP or TCP. CoAP protocol uses a UDP/IP stack which is good to reduce overhead size. Since UDP is a connectionless protocol, the need to achieve reliability is satisfied by applying CoAP reliability mechanism itself, which is based on the message type (conformable, acknowledgment, reset message) to ensure delivery of the message. Since HTTP, MQTT and XMPP adopt TCP/IP, these protocols can benefit from reliability of TCP directly to deliver messages without loss.

In addition to the TCP services, MQTT has its three-level quality of service to ensure delivery of messages [24]. Establishing a connection between communication parties is the first step before sending packets. In the client-brokers paradigm, broker applies a registration mechanism of username and password, which adds complexity to the communication; whereas, the client-server paradigm connection is established immediately. However, complexity lies under layer's protocol, and the communication complexity can also be affected by the traffic, which is not related to users but to the protocol itself [27].

3.2.2 Security

Providing security in IoT is a big challenge because an IoT system is heterogeneous and consists of many devices that process many sensor data with its computing ability with less human interference. These operations need to apply security mechanisms at different levels of the system. XMPP is one of the most secure application protocols which enforces security from client-to-client and client-to-server through a TLS mechanism that is incorporated in the protocol specifications to provide reliability, confidentiality and data integrity [29]. Additional security services included in XMPP are authentication, privacy, and access control. XMPP is also a native supporter of authentication via SASL [6]. The security in HTTP2 protocol relies on TLS version 1.2 or higher. However, applying TLS 1.2 must meet two requirements, otherwise the connection will be terminated. To deploy HTTP2 over TLS 1.2, the generic compression must be disabled because the protocol already provides a header compression so there is no need for more compression mechanisms. Disabling the renegotiation mechanism is required too, but if the endpoint uses renegotiation to provide confidentiality, this must be done before sending the connection preface. Take into consideration the deployment of HTTP2 over TLS 1.2 which should not use any cipher suites from the blacklist [19]. The MQTT security relies on TLS encryption, which supports the encryption of username and password flags of MQTT variable header, and it simplifies user authentication [30].

The CoAP protocol achieves integrity, authentication, confidentiality, non-repudiation, and automatic key management by applying DTLS, which is an extension

of TLS with an 8-byte addition, but developed to run over UDP [31]. Sometimes CoAP applies an IP security (IPSec), which is an independent layer-3 protocol for IP to enable a secure application and transport layers. IPSec is used with CoAP by applying the method of Encapsulating Security Payload Protocol (IPSec-ESP) [32].

There are four ways to apply security in CoAP and MQTT [33]:

a. **No security:** It means there is no security applied to the message in the CoAP transport layer, while in MQTT it means that the message does not contain a username and password. Yet, there is a possibility to apply other security mechanisms as the designer needs, namely anonymization, application data encryption, and secure communication channels.

b. **Pre-shard key:** This mode enforces the transport layer to use TLS and DTLS. It works with a shared symmetric encryption key, and data is encrypted using a shared secret key between two sides. The receiver side applies the same key to decrypt data. An advance security level is to use different keys for each side.

c. **Certificates:** It is the basic security technique for the Internet and it is applied in some IoT systems, too, even if it is costly and heavy for IoT constrained devices. Certificates enforce using public key authentication and certificate items that communication parties sign with security companies.

d. **Raw Public Key:** This mode allows to generate a new key for each connection session, and if the key discovered, communication pairs will not be able to encrypt message. Pairs-key is used as a certificate to authenticate the public key, but without an actual certificate. The authentication of a public key is implemented either in a pre-provisioning way, where each pair has a list of IP addresses and domain names of the destinations, or with DNS-Based Authentication of the entities' names.

3.2.3 Header and Payload Size

MQTT has a very small header size of 2 bytes, and the maximum size for payload is 4 bytes. HTTP2 has a fixed header of 9-octal with a 9-byte variable-length payload, but the payload is undefined. The header size of CoAP is 4 bytes, while payload size of CoAP is calculated to the end of the UDP datagram. XMPP protocol header and payload size are undefined.

3.2.4 Fragmentation

It is an IP process to fragment the datagram into small chunks to fit in the MTU of the network layer. In addition to that, CoAP protocol has a fragmentation mechanism called Block-Wise. When message is too large to carry on a single CoAP datagram message, a block can be used to handle a message in both request and response. Number 1 refers to (Block 1, size 1) of the request message, and number 2 refer to (Block 2, size 2) of response message [34]. While MQTT, HTTP, and XMPP protocols do not have a built-in fragmentation mechanism, only under layer fragmentation of IP or data-link layer can be used.

3.2.5 Protocols Implementation

Nowadays, the most common application which uses MQTT protocol is Facebook Messenger. The main ability of MQTT is to deliver messages in milliseconds on any type of Internet connection with low power usage; a feature that is important for smartphones batteries [35]. XMPP protocol is used in back-end servers of applications, like Google Hangouts, WhatsApp, Gtalk, Facebook Chat, and with other IM applications like LJ Talk, Nimbuzz, and HipChat [2]. CoAP protocol is applied in all constrained devices of different domains, such as factory instrumentation, aircraft equipment, defense equipment, building automation, and smart gird [36]; while the HTTP is applied in almost all internet transmissions, from browser to server and vice versa.

3.2.6 Organizational Support

An application protocol is nothing if no organization adopts it. For MQTT, the supporting organizations are IBM, Facebook, Eurotech, Cisco, Red Hat, Software AG, Tibco, ITSO, M2Mi, Amazon Web Services (AWS), InduSoft, and Fiorano. CoAP is used by large web community support establishments, including Cisco, Contiki, Erika, and IoTivity. HTTP is already a global web standard, which means that it is supported by all organizations. XMPP, as well, is used by certain standard foundations known as the XSF, JIV software, IBM, Google, Apple, Ignite Realtime, and LiVEJOURNAL [17].

Table 3. 1 Comparison of Application Layer Protocol

Communication Categories	**MQTT**	**CoAP**	**XMPP**	**HTTP2**
The organization that develops protocol	IBM, OASIS	IETF, CoRE Working Group	IETF with Jabber community	IETF
Licensing Model	Open source	Open source	Open source	Free
Published year	1999	2010	1999	2015
Port number	1883 8883 with TLS	5683	Client-server 5222 Server-server 5269	80 without security 443 with security
Architecture	Client-Broker	Client-Server Client-Broker	Client-Server Client-Broker	Client-Server
Communication paradigm	Publish-subscribe	Request-response and publish-subscribe	Request-response and publish-subscribe	Request-response
Transport layer	TCP	UDP	TCP	TCP
Header size	2 byte	4 byte	Undefined	9 octal
Message size	Small (up to 256 MB)	small to fit in single IP datagram	Large and Undefined	Large and Undefined
Security	TLS	DTLS, IPsec	TLS, SASL	TLS/SSL
Support multicast	Yes	Yes	yes	No
QoS (Reliability)	Through three-level QoS and TCP	Through CoAP message type (CON, NON)	Through TCP	Through TCP

Authentication	Username and password	Only server	SASL	401 status code and the WWW-Authenticate res ponse header

3.3 Appropriate Protocols for the Different Requirements

There are important requirements for IoT applications which affect the use of respective protocols [17]. Table 3.2 shows the convenient protocol for specific requirements.

Table 3. 2 Proper Protocol for Different Requirements

Application Requirements	**The most priority protocol**	**Explaintion**
Power Consumption	1: MQTT or CoAP	MQTT and CoAP requires lowest power with the same settings (MQTT QoS 0 vs CoAP NON) and (MQTT QoS 1 or 2 vs CoAP CON). While HTTP requires maximum power.
Resource Requirement	1: MQTT 2: CoAP	MQTT and CoAP are designed for resource constrained devices. HTTP requires maximum resources.
Bandwidth	1: CoAP 2:MQTT	The bandwidth used by MQTT is higher than CoAP when (MQTT QoS 0 vs CoAP NON), while MQTT bandwidth is twice that of CoAP when (MQTT QoS 1 or 2 vs CoAP CON). HTTP does not achieve the full utilization of bandwidth with TCP because the congestion window opens gradually and doubles the number of packets in each roundtrip time. However, the

		bandwidth of HTTP is the highest among other protocols.
QoS	1: MQTT	MQTT has the highest level of QoS (at most once, at least once, and exactly once). In CoAP, there is no clear QoS mechanism, but it offers mechanisms for resource discovery and retransmission through NON and CON in a similar way to MQTT QoS 0 and 1. HTTP has no built-in QoS mechanism.
Reliability	1: HTTP	HTTP reliability is not a core service, and only achieved by TCP. The reliability of MQTT is obtained through QoS. While CoAP reliability is achieved through NON and CON.
Interoperability	1: HTTP 2: MQTT	HTTP has the highest interoperability because of the RESTful client-server architecture. MQTT publish-subscribe architecture supports almost all IoT variations. CoAP interoperability is limited in devices that only use UDP.
Security	1:HTTP and MQTT	Security in HTTP and MQTT protocols relies on TLS/SSL; whereas, In CoAP, security is done with DTLS/IPSec.

CHAPTER 4

SIMULATION TOOLS AND MODELLING

4.1 Network Simulator

Creating an IoT environment can face many obstacles due to the device elements and software. It can be expensive to integrate different environments for exchanging real data and performing emergency actions to help people who need it. In this vein, choosing an application layer protocol and simulating the network performance are important issues in the IoT environment; for that reason, the present thesis focuses on the application layer protocol performance. For this, IoTFY is a network simulator that provides a complete IoT environment with the ability to manage and analyze the application protocol performances. Therefore, we preferred it as the simulator to be used in this thesis study.

IoTIFY is selected here because of its user-friendly nature to accommodate many smart-object prototypes, and its ability to apply them with different IoT application protocols to examine and analyse their performances. Creating a simulation is rather simple. It starts by selecting the protocol and giving a specific name to the prototype. Then, we specify the important features for each protocol with the basic network parameters. After all these definitions, the simulation starts and the results will be ready after a few minutes.

4.2 IoTIFY Network Simulator

IoTIFY is an intelligent simulation platform on Cloud for IoT systems that simulate large-scale deployment of IoT devices on the cloud. It is designed exclusively for IoT to consider all possibilities and to manage IoT applications flexibly and easily.

The features of IoTIFY can be defined as follows:

1. **Scalability:** This is the major feature for the IoTIFY platform to guarantee the best performance with minimal resource consumption and solving the problems of IoT environment that occur when millions of sensors and thousands of gateways could be connected to the IoT platform at any time with different connection media, all causing added latency, packets loss, repetition, and out-of-order deliveries.
2. **Supporting Multiple Protocols:** The basic protocols that IoTIFY supports is MQTT and HTTP, with TCP and UDP as transport layer protocols. For advanced and enterprise usage, IoTIFY supports the basic protocols with CoAP, AMQP, and LWM2M.
3. **Integration with IoT cloud platform:** The IoTIFY has been integrated with AWS IoT and Azure IoT hub for advanced and enterprise use.
4. **Complete template:** There are 18 templates in IoTIFY for different IoT simulations. Figure 4.1 shows some IoT devices, such as connected vehicles, smart homes, smart trucks, coffee makers, Arduino, Raspberry Pi, and more. Each simulation must be given a unique name and the protocol to be used must be determined with initial creation steps to start the network simulation.

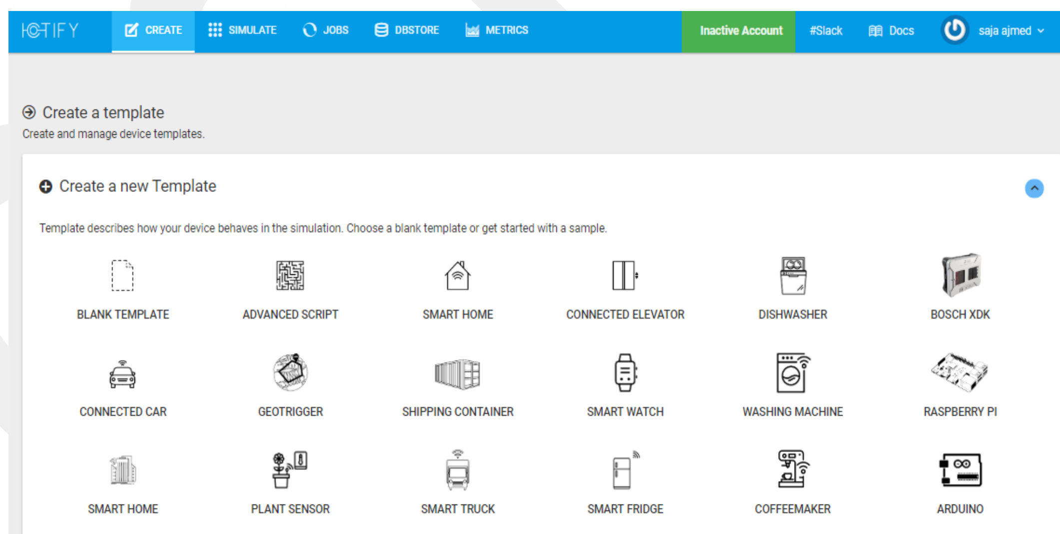


Figure 4.1 IoTIFY Templates

4.3 Template

Before addressing the simulation, the template structure has to be determined; the template is a JavaScript document describing every IoT device feature and its behaviour during the simulation. Under the device model, we find the basic device information, connection protocol and the security parameters of each template. The template contains information regarding the message to be sent, connection, and credentials. Each template defines the JavaScript functions that must return payload sent to the IoT Cloud platform; the function object is called a state, which holds device-specific-parameters.

Functions types:

- Setup function: The first function before the connection establishment is the setup function, which is used to restore values from persistent storage, initialize one-time variables, and assign some random names. The setup function does not return any value.
- State object: a JSON object is used to save the device status that includes the IoT device parameters to be reported to the Cloud platform.
- Message function: The core function of simulation is the message function that is recalled with all state objects and for all the specified number of iterations in the template run.
- Finish function: This function is invoked after all messages have been sent. It may help to clean up the server resources or to summarize test results.

Once the simulation starts, the engine analyzes the template and creates a virtual IoT device, each with its own memory and state, and connects to the IoT cloud platform.

4.4 Simulation Study

The protocols used in this simulation are MQTT and HTTP. The goal is to simulate 16 different HTTP scenarios. The parameters changed in HTTP simulations are 4 different client numbers, and 4 different number of message repeats. The number of MQTT scenarios intended for observation are 48, with the same HTTP parameters to be changed. In addition, the same scenarios are simulated with different QoS parameters which affect the MQTT simulation. The simulation is done for a connected car and the aim is to collect some information related to the car, such as license number, location, and odometer, with the timestamps and connection duration for each car. After setting up the protocol parameters and starting a simulation, the connection duration and latency performances of these two protocols are compared.

4.5 Measurement

At this point, we will measure and discuss the parameters which are essential for any protocol performance test. There are three main performance measures provided by IoTIFY, namely connection duration, packet sending latency and message generation latency.

4.5.1 Connection Duration

The default parameters provided by IoTIFY simulator are 10 clients and 5 repeats for 10-second intervals. These parameters have an obvious effect on the duration of each simulation. IoTIFY clarifies the start and end time of any scenario in second. In general, decreasing client numbers and repeating messages lead to faster performance when compared to added clients and repeating times at the same interval times.

4.5.2 Packet Sending Latency

Latency is the time required to transmit a packet across a network for each iteration, measured by millisecond. The latency of sending a packet to the back-end depends on

the application protocol and transport protocol. The MQTT and HTTP protocols use TCP, which is directly affected by latency and which ensures packet delivery correctly with the acknowledgment mechanism. IoTIFY provides the average, minimum and maximum packet latency values for the whole connection period.

4.5.3 Message Generation Latency

Message generation latency is the time elapsed between generation of the message at its source node and delivery of the message at its destination node in millisecond. It is directly affected by the processor idle time and memory access time on the server side. Within IoTIFY, three values are shown for each iteration: the average, minimum and maximum time to run message. If message generation latency increases, it will slow down the entire performance test.

4.6 Software Environment

An IoTIFY simulation can be divided into four main steps as in Figure 4.2. The first step is formulation step of choosing the template and protocol, and giving simulation a unique name as shown in Figure 4.3. The second step is protocol specification; each protocol has its own parameters. The third step is simulation specification, when the network parameters are determined. The fourth step is to view and analyze the results.

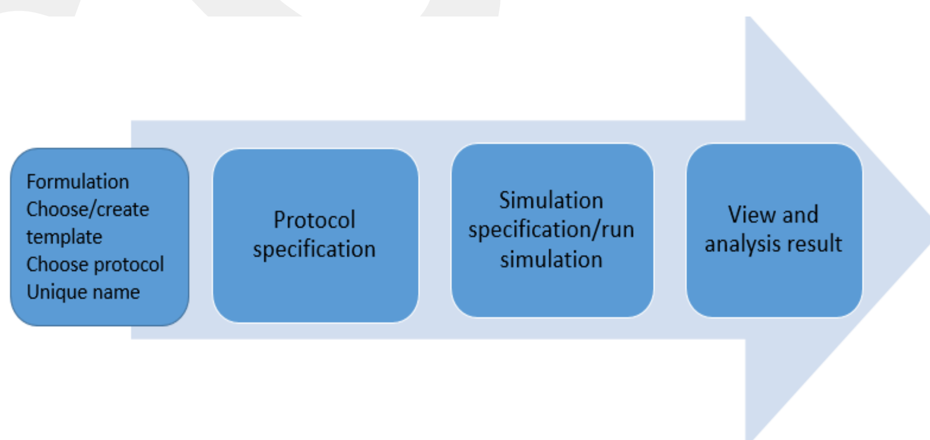


Figure 4.2 The IoTIFY Simulation Steps

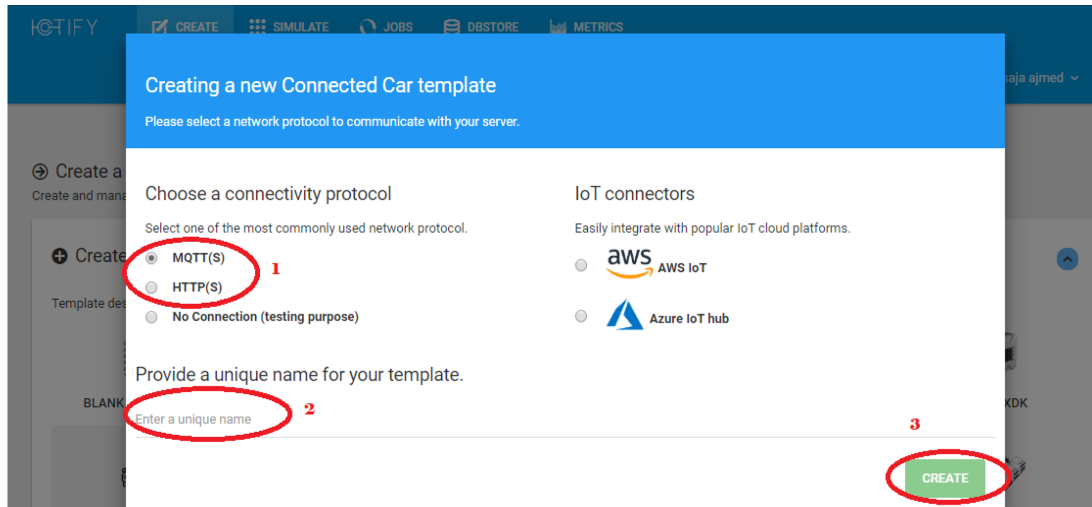


Figure 4.3 The Formulation Step

4.7 MQTT Parameters with IoTIFY

We choose a connected car template to analyze the MQTT protocol performance and obtain the data published to the broker HiveMQ on the port number 1883 under the topic name /hello/ {{client ()}}.

There are important parameters that must be set before the simulation starts:

- Subscribe: The subscription handler can be enabled by clicking on the 'checkbox' to allow the publishing of new data every time it runs.
- Keep-alive: The time in seconds to keep the connection alive.
- QoS: Choose among the three QoSs provided by MQTT protocol.
- Retain: By clicking on the checkbox the message will be stored.
- Time out: The time in seconds to establish a new connection.

The parameters which are shown in Figure 4.4 must be set and saved. Then, the simulation parameters can be accessed by clicking on the 'simulate' button.

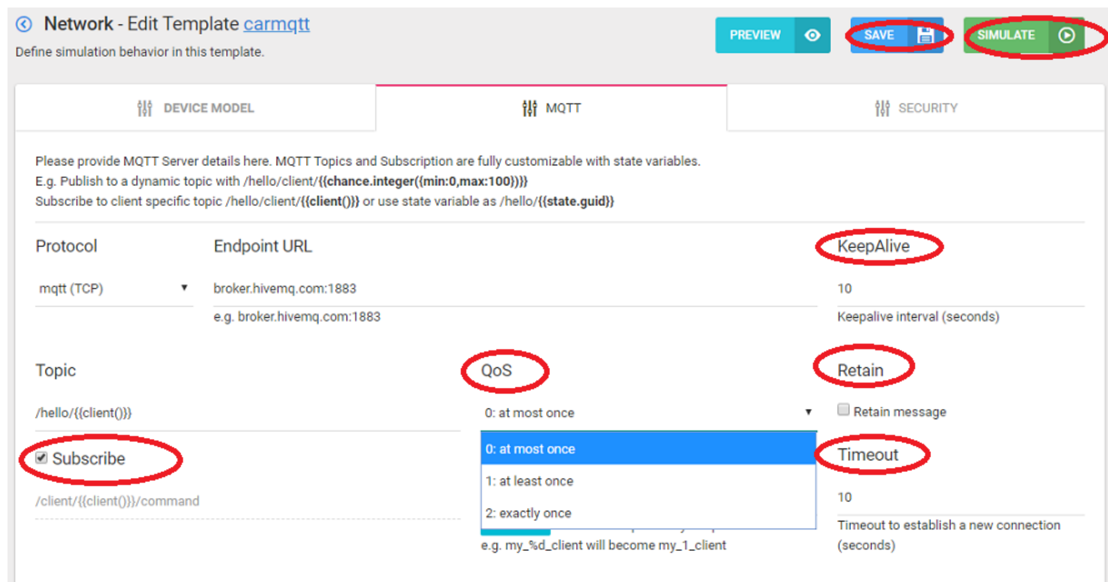


Figure 4.4 MQTT Parameters

4.8 HTTP Parameters with IoTIFY

The HTTP parameters, as illustrated in Figure 4.5, are determined after choosing the HTTP as the protocol and giving the connected car template a unique name. The HTTP host is provided by httpbin.org as a simple HTTP request and response service with HTTP method, GET.

The parameters that must be set before starting the simulation are:

- Timeout: The time in seconds to establish a new connection.
- Invoke response handler be set by clicking on the 'checkbox'.
- The URL is encoded with a payload in JSON.

After parameter selection, the 'save' button is clicked on to save the values of the parameters. Then, pressing on the 'simulation' button will lead to the network parameters setting page.

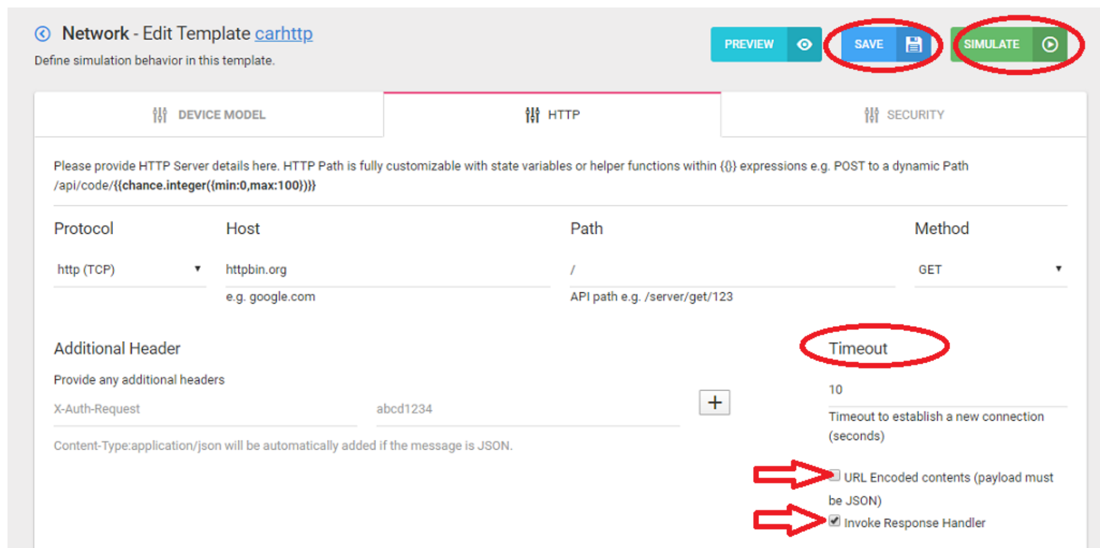


Figure 4.5 HTTP Parameters

4.9 Basic Network Simulation Parameters

The basic network simulation parameters shown in Figure 4.6 must be set before starting the simulation. These parameters may affect protocol performance according to its important role in the network. After specifying network parameters, the simulation can be started, and the result will be ready to analyze and view. The simulation parameters to be set are:

- Clients number out of 100.
- The iteration number to repeat the message.
- The gap time between each iteration in seconds.

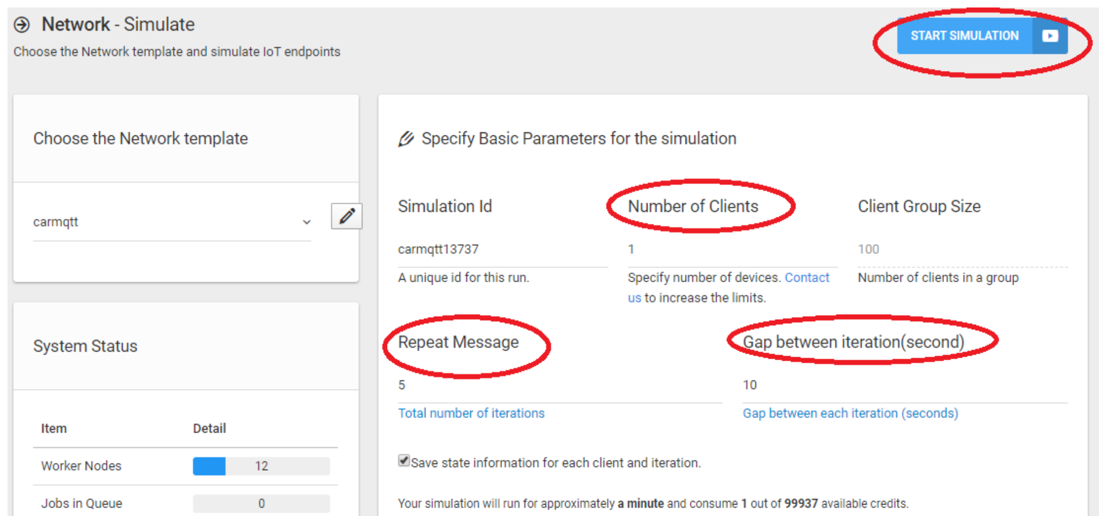


Figure 4.6 Network Simulation Parameters

4.10 Outcomes of IoTIFY

1. Result Summary: It contains details of the connection, including start time, end time, template name, protocol name, host, client number, repeat, interval, and other features as shown in Figure 4.7.

2. Network Activity Timestamps: IoTIFY provides an overview in a precision graph describing the number of connections established and the number of messages sent for each iteration.

3. Packet Sending Latency: The related chart depicts the minimum, maximum, and average time in a millisecond to send packet in each iteration.

4. Message Generation Latency: This chart depicts the minimum, maximum, and average time in a millisecond to run a message in each iteration.

5. State: The state information tab captures the value of state and any logs produced through every client and iteration. The information is captured only if the save state information flag is checked while running the simulation.

The table has multiple views by showing all iterations for a single client, all clients for a single iteration, or all clients and all iterations simoltaneously. The state tab is shown in Figure 4.8.

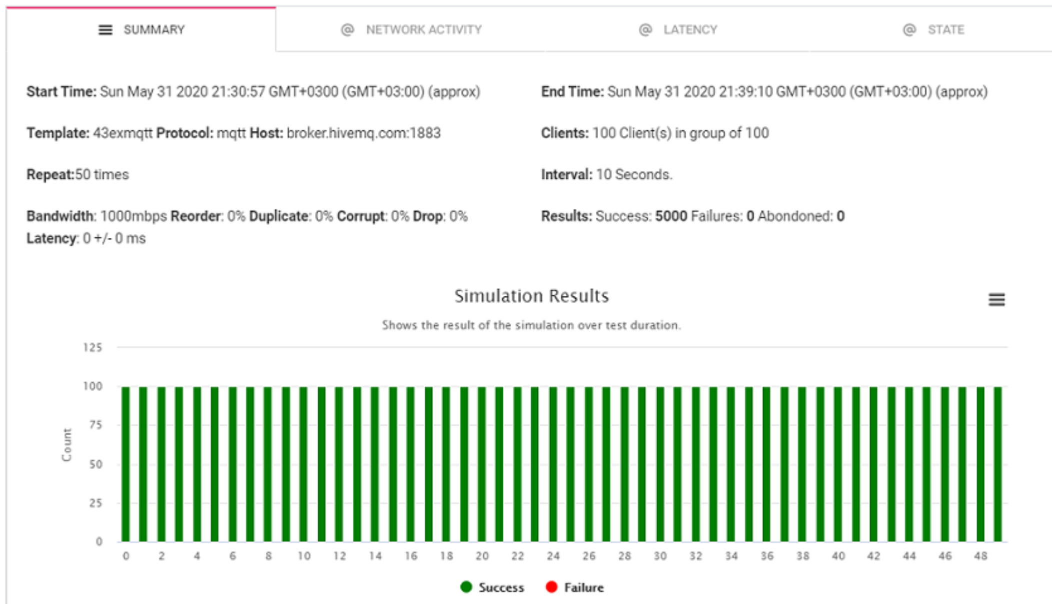


Figure 4.7 Summary Result

SUMMARY NETWORK ACTIVITY LATENCY STATE

The following table describes the state object across each client and iteration.

Choose Result Type: Client / Iteration / All

Choose Index: 1

Action: RELOAD DOWNLOAD CSV

license	odometer	VIN	remoteDisabled	location_latitude	location_longitude	location_speed	lo
CA IE06N1	209904	9EQOJ8NPN8QOW5Y	false	33.80546	-117.92688	4	0
CA IE06N1	211504	9EQOJ8NPN8QOW5Y	false	33.80641	-117.92398	7	0

Figure 4.8 State of IoTIFY

CHAPTER 5

SIMULATION RESULTS

5.1 IoTIFY Modeler

The simulation of IoT application layer protocol over the Internet is provided by the IoTIFY modeler as used in this research by means of creating a free account on the IoTIFY official website. IoTIFY supports five IoT application layer protocols: MQTT, HTTP, CoAP, AMQP, and LWM2M.

In this thesis, the performance of two application layer protocols in particular (MQTT and HTTP) is investigated by using 64 simulation scenarios, shown in Figure 5.1 and 5.2. These scenarios consist of 48 MQTT simulations (3 different QoS x 4 different groups of client number x 4 different values of message repeating) and 16 different HTTP simulations (4 different groups of client number x 4 different values of message repeating), all amounting to 64 (48 + 16) simulation scenarios. The MQTT simulations are based on three parameters: QoS, client number, and repeating-message; while HTTP simulations are based on two parameters: client number and repeating-message. The connection duration, message generation latency, and packet sending latency (minimum, maximum, and average) are the measured parameters to evaluate the performance of the protocol.

To analyze and compare the application layer protocols performance, the differences are observed pertaining to connection duration, message generation latency, and packet sending latency. HTTP protocol is investigated with 16 performance scenarios; with four groups of client numbers (10, 25, 50, and 100); and with four different values to repeat messages (5, 25, 50, 100). In turn, MQTT protocol is simulated with 48 performances scenarios; with 3 different QoS values (0, 1, 2), four groups of client numbers (10, 25, 50, 100), and four different values to repeat message (5, 25, 50, 100).

The efficiency of MQTT and HTTP protocols is examined by comparing the usage of a variable number of clients, a variable message repeating values, and a variable QoS, if any. The outcomes are analyzed using graphs to see which protocol is more effective than the other. The scenarios were designed and run by means of IoTIFY. In these scenarios, the simulation environment is modeled using the HTTP and MQTT protocols. We consider connection duration, message generation latency, and packet sending latency as the parameters used to measure the efficiency of the protocols. Scaling actual latencies in generating messages and sending the packets is key to any performance test, and the average time to run message function is addressed as a message generation latency, while the packet sending latency is the average time it takes to send a message to the back-end.

Each simulation is run for a certain period of time depending on the simulation parameters. The connection duration of each simulation is shown in the result summary window along with respective information. IoTIFY provides statistical analysis diagrams for timestamps and latencies which are very beneficial to test protocol performance. The obtained figures are saved for the statistical analysis as images with tables containing details for each simulation iteration. These figures will be discussed in the coming sections.

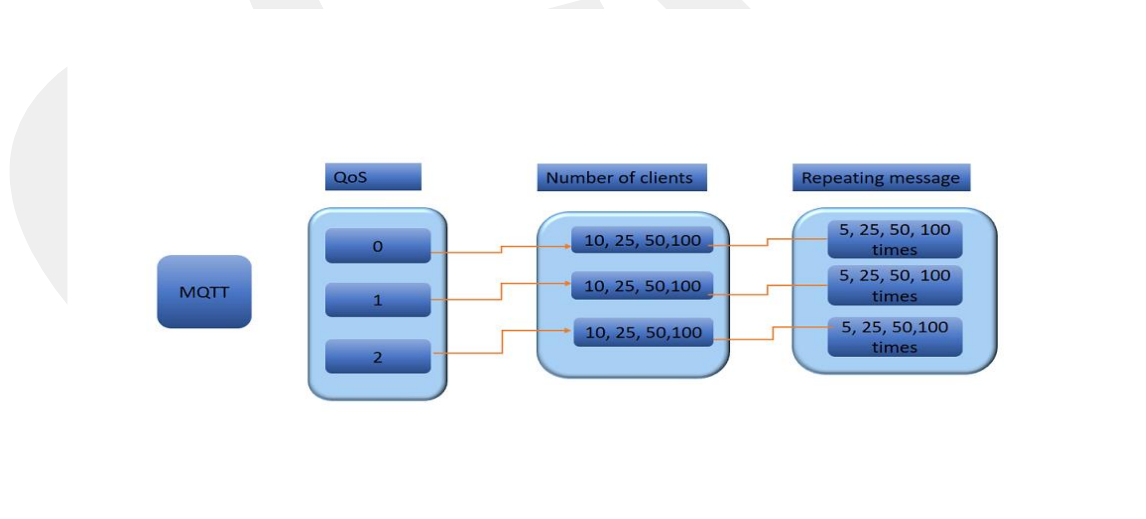


Figure 5.1 MQTT Scenarios Chart

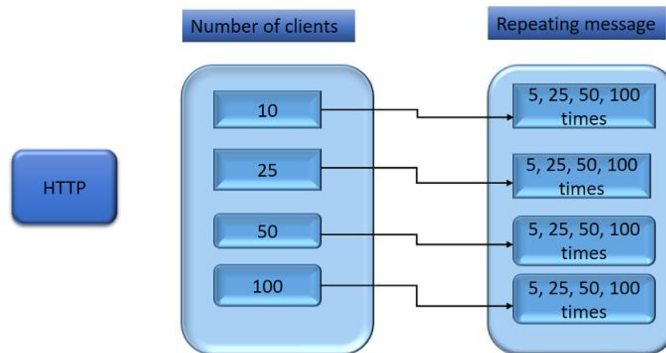


Figure 5.2 HTTP Scenarios Chart

5.2 Connection Duration Performances

To better understand the effect of simulation time on protocol efficiency, the client number starts from 10 clients, with 5 times repeating of 10-second intervals as default parameters provided by IoTIFY simulator, up to 100 clients and 100 repeating times for the same intervals. These parameters have an important influence on the protocols performance time. In general, decreasing client numbers and repeating message lead to a faster performance than increasing clients and repeating times at the same interval time.

The goal is to obtain the duration for a particular number of clients and repeating message. In the first set of scenarios HTTP protocol is used and in the second set of scenarios, it is MQTT protocol. The client number and repeating message parameters are set with QoS, if any. Then, the connection duration is measured to obtain a better understanding of each protocol performance based on these parameter values.

5.2.1 Effects of Client Numbers and Repeating Messages on HTTP Connection Duration

The relation of repeating message with connection duration is a direct one, and whenever repeating message increases, duration becomes longer. A change in the client number does not show an effect on connection duration. For HTTP, connection duration is long because of the three-way handshaking according to the protocol architecture. We examine the duration for 4 different client numbers (10, 25, 50, 100) and 4 different numbers of repeating message (5, 25, 50, 100) of HTTP connection. Figure 5.3 and Figure 5.4 show the duration for 10 clients and 5 repeats, and 100 clients and 100 repeats, respectively. In the first simulation, all packets are delivered successfully. However, 7 out of 10,000 messages could not be delivered with increasing client number and repeating time.

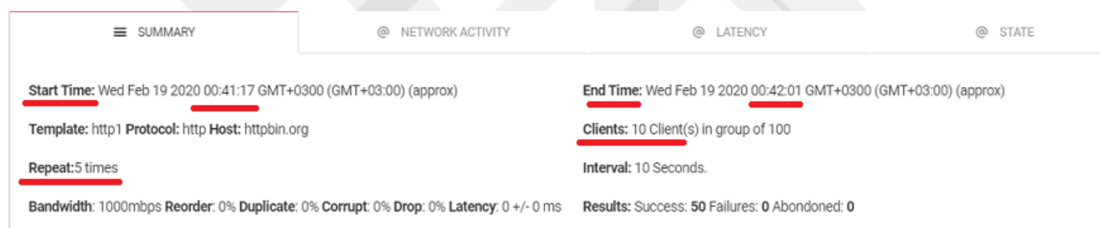


Figure 5.3 HTTP Duration of 10 Clients and 5 Repeat

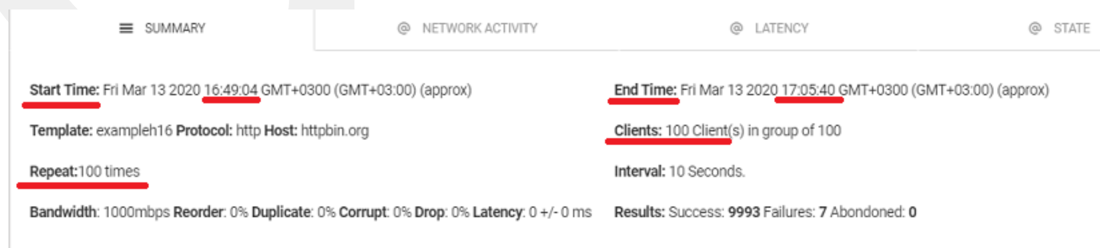


Figure 5.4 HTTP Duration of 100 Clients and 100 Repeat

The results of the simulation duration for each scenario are shown in Table 5.1, where one can see that the duration increases depending on the message repetitions. The number of clients does not have any effect on duration. The average durations are 45, 256, 494.25 and 994.5 second for 5 times, 25 times, 50 times and 100 times repeat of the messaging, respectively.

Table 5.1 The Connection Duration of HTTP

Client number	Repeating message	Timestamps	Duration (sec)
10	5 times	00:41:17 - 00:42:01	44
	25 times	18:01:29 – 18:05:31	242
	50 times	22:14:23 – 22:22:41	498
	100 times	21:19:15 – 21:35:50	995
25	5	17:09:37 – 17:10:21	44
	25	00:24:31 – 00:28:41	250
	50	22:36:50 – 22:45:01	491
	100	22:57:53 - 23:14:30	997
50	5	17:31:56 - 17:32:41	45
	25	01:10:23 - 01:14:31	284
	50	00:44:57 - 00:53:11	491
	100	16:27:30 - 16:44:00	990
100	5	17:46:04 - 17:46:51	47
	25	18:37:33 - 18:41:41	248
	50	16:07:54 - 16:16:11	497
	100	16:49:04 - 17:05:40	996

5.2.2 Effects of Client Numbers, Repeating Messages, and QoS on MQTT Connection Duration

In MQTT protocol, one more important parameter is added to the simulation parameters in addition to the client number and repeating-message parameters. It is related to QoS to allow for sending the message with three different values (at-most-once, at-least-once, and exactly one). Figure 5.5 and Figure 5.6 show the scenarios with different client numbers, repeating times, and QoS. In all, 48 scenarios are examined by applying each QoS value (0, 1, 2) with 10 clients and 5 repeats, 25 clients and 25 repeats, 50 clients and 50 repeats, and 100 clients and 100 repeats.

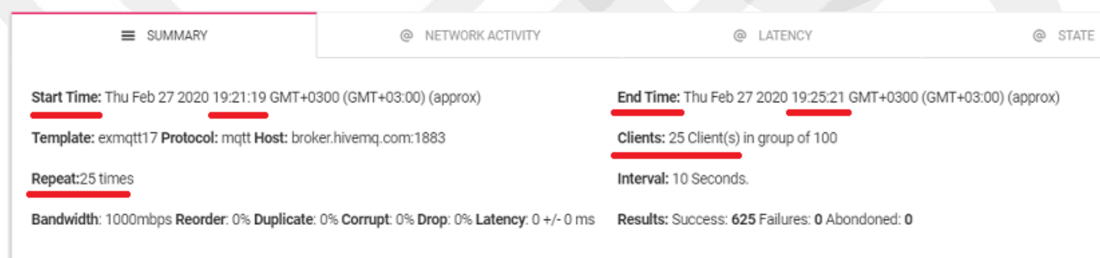


Figure 5.5 MQTT Duration of 25 Repeat and 25 Clients with 1 QoS

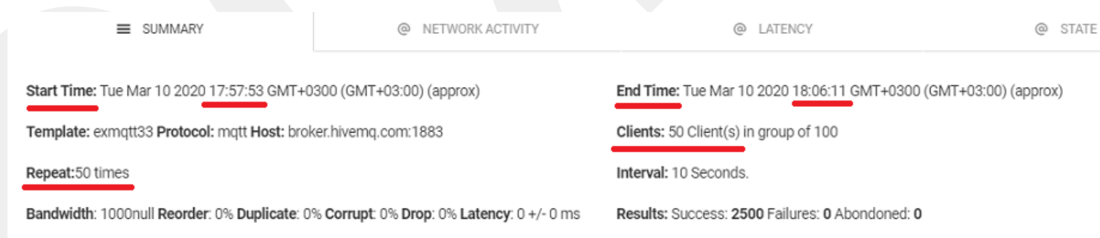


Figure 5.6 MQTT Duration of 50 Repeat and 50 Clients with 2 QoS

From the simulation results, it is evident that repeating-message affects the connection duration as their relationship is a direct one; that is, if the repeating-message increases, the duration will increase, too, regardless of the client numbers and QoS values.

Table 5.2 shows the connection duration of MQTT for 12 scenarios pertaining to the average duration of sending message and repeating 5 times with 4 groups of client

numbers and all values of QoS as 45.58 sec. However, as shown in Table 5.3, the average connection duration is 240.83 sec for the repeating message value of 25; whereas, this duration stands at 496.66 sec and 996.66 sec for the repeat values of 50 times and 100 times, respectively, as depicted in Table 5.4 and Table 5.5.

Table 5.2 MQTT Connection Duration of Repeating Message 5 Times

Repeating-message	Client number	QoS	Timestamps	Duration (sec)	Average Duration
5	10	0	19:14:35 - 19:15:21	46	45.58
		1	20:01:29 - 20:02:11	42	
		2	15:54:48 - 15:55:30	42	
	25	0	18:44:51 - 18:45:41	50	
		1	19:01:11 - 19:02:01	50	
		2	18:52:46 - 18:53:31	45	
	50	0	17:39:03 - 17:39:51	48	
		1	19:51:20 - 19:52:01	41	
		2	22:56:31 - 22:57:21	50	
	100	0	00:47:34 - 00:48:21	47	
		1	01:57:08 - 01:57:50	42	
		2	02:14:27 - 02:15:11	44	

Table 5.3 MQTT Connection Duration of Repeating Message 25 Times

Repeating-message	Client number	QoS	Timestamps	Duration /sec	Average Duration
25	10	0	22:11:53 - 22:16:01	248	240.83
		1	22: 32:02 - 22:36:11	249	
		2	22:57:02 - 23:01:11	189	
	25	0	19:14:40 - 19:18:41	241	
		1	19:21:19 - 19:25:21	242	
		2	19:27:00 - 19:31:00	242	
	50	0	23:46:26 - 23:50:31	245	
		1	01:30:34 - 01:34:41	247	
		2	02:34:24 - 02:38: 31	247	
	100	0	18:21:53 - 18:26:01	248	
		1	18:15:42 - 18:19:50	248	
		2	16:49:56 - 16:54:00	244	

Table 5.4 MQTT Connection Duration of Repeating Message 50 Times

Repeating-message	**Client number**	**QoS**	**Timestamps**	**Duration /sec**	**Average Duration**
50	10	0	23:19:04 - 23:27:21	497	496.66
		1	23:41:52 - 23:50:11	499	
		2	16:30:01 - 16:38:21	500	
	25	0	17:06:08 - 17:14:21	493	
		1	17:55:04 - 18:03:21	497	
		2	18:08:59 - 18:17:11	490	
	50	0	17:07:34 - 17:15:51	497	
		1	17:26:43 - 17:35:01	498	
		2	17:57:53 - 18:06:11	498	
	100	0	19:19:48 - 19:28:01	493	
		1	18:15:50 - 18: 24:11	501	
		2	19:50:04 - 19:58:21	497	

Table 5.5 MQTT Connection Duration of Repeating Message 100 Times

Repeating-message	Client number	QoS	Timestamps	Duration /sec	Average Duration
100	10	0	17:30:22 - 17:47:01	999	996.66
		1	17:59:03 - 18:15:41	998	
		2	18:18:01 - 18:34:41	1000	
	25	0	17:18:58 - 17:35:31	993	
		1	18:42:45 - 18:59:21	996	
		2	19:15:07 - 19:31:41	994	
	50	0	17:11:34 - 17:28:10	996	
		1	17:37:52 - 17:54:31	999	
		2	17:56:37 - 18:13:11	994	
	100	0	21:05:01 - 21:21:40	999	
		1	20:20:25 - 20:37:01	996	
		2	20:39:34 - 20:56:10	996	

5.3 Message Generation Latency Analysis

IoTIFY provides an x-axis accurate chart for presenting the minimum, average, and maximum times taken to send a message, also known as message generation latency. These three latency values are shown for each iteration in milliseconds. From the charts, it can be seen which iteration takes more or less time to send a message, along with the average time a message needs to be send and, finally, how that affects the connection.

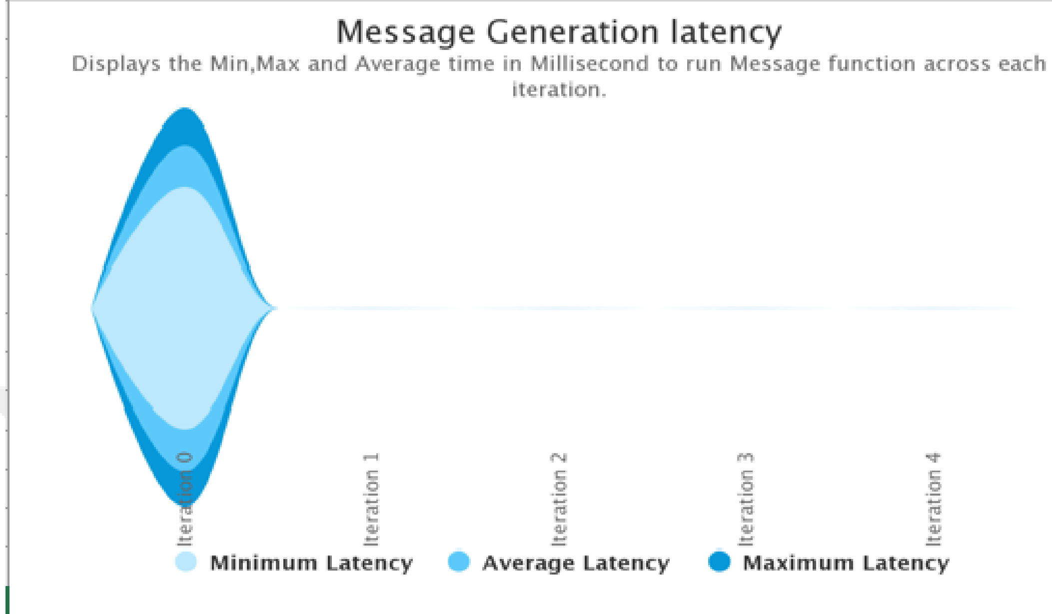
5.3.1 Message Generation Latency Analysis for HTTP

The message generation latency is measured for HTTP with different client numbers and different repeating times. For each iteration the minimum, average and maximum latencies are calculated, where the first iteration presents the highest latencies, then the latencies for other iterations settle down in the range of (1ms) for minimum, average, and maximum latency. The results of message generation latency for 16 HTTP scenarios are shown in Table 5.6.

In these scenarios, GET calls were made with a message function to an external server, which appears to take a long time to accept GET request at the first iteration, only to become stable in the upcoming iteration latencies. Observations are made as to the effect of the increase in client number, which leads to an increase in the message generation latency. The resulting table is prepared by making the repeating message constant and changing the client number where the average latency shows an increase with all scenarios. The HTTP message generation latencies for 10 clients and 5 repeats, 25 clients and 25 repeats, 50 clients and 50 repeats, and 100 clients and 100 repeats, as displayed in Figure 5.7, Figure 5.8, Figure 5.9, and Figure 5.10, respectively.

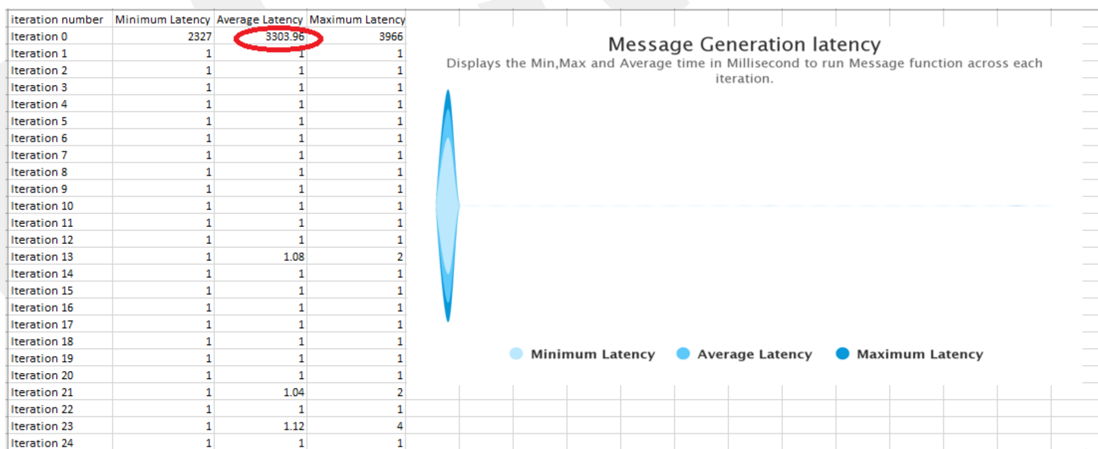
Table 5. 6 Message Generation Latencies of HTTP Protocol

Repeating message	Client number	Minimum latency	Average latency	Maximum latency
5	10	1377	1855.8	2265
	25	1475	2323.32	3108
	50	1294	2681.84	3674
	100	1092	3592.03	5597
25	10	1136	1570	1969
	25	2327	3303.96	3966
	50	1415	3035.28	4310
	100	337	3967.75	5812
50	10	1553	1983	2291
	25	1468	2370.92	2942
	50	1534	3104.82	4112
	100	2387	4724.24	6482
100	10	837	1278	1637
	25	1474	2378.6	3051
	50	1507	2917.38	3721
	100	692	3604.12	5304



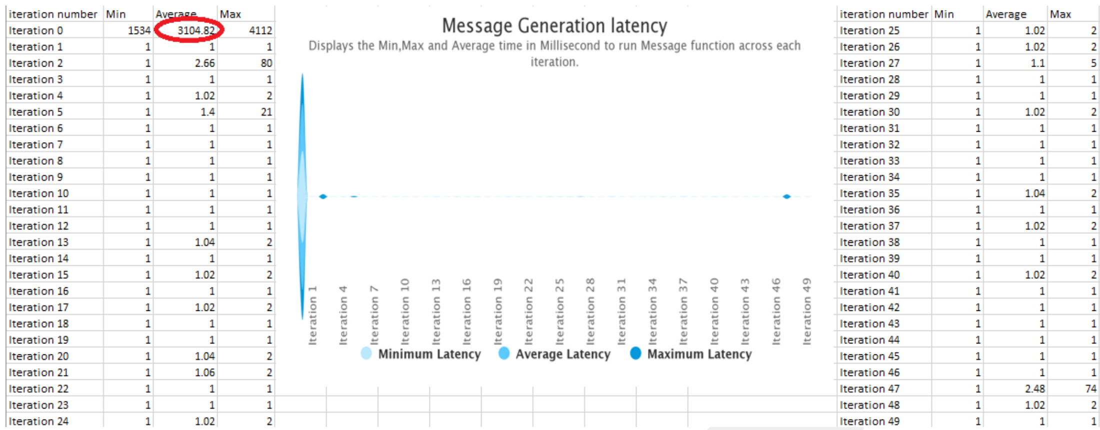
iteration number	Minimum Latency	Average Latency	Maximum Latency
Iteration 0	1377	1855.8	2265
Iteration 1	1	1	1
Iteration 2	1	1	1
Iteration 3	1	1	1
Iteration 4	1	1	1

Figure 5.7 HTTP Message Generation Latency of 10 Client and 5 Repeat



iteration number	Minimum Latency	Average Latency	Maximum Latency
Iteration 0	2327	3303.96	3966
Iteration 1	1	1	1
Iteration 2	1	1	1
Iteration 3	1	1	1
Iteration 4	1	1	1
Iteration 5	1	1	1
Iteration 6	1	1	1
Iteration 7	1	1	1
Iteration 8	1	1	1
Iteration 9	1	1	1
Iteration 10	1	1	1
Iteration 11	1	1	1
Iteration 12	1	1	1
Iteration 13	1	1.08	2
Iteration 14	1	1	1
Iteration 15	1	1	1
Iteration 16	1	1	1
Iteration 17	1	1	1
Iteration 18	1	1	1
Iteration 19	1	1	1
Iteration 20	1	1	1
Iteration 21	1	1.04	2
Iteration 22	1	1	1
Iteration 23	1	1.12	4
Iteration 24	1	1	1

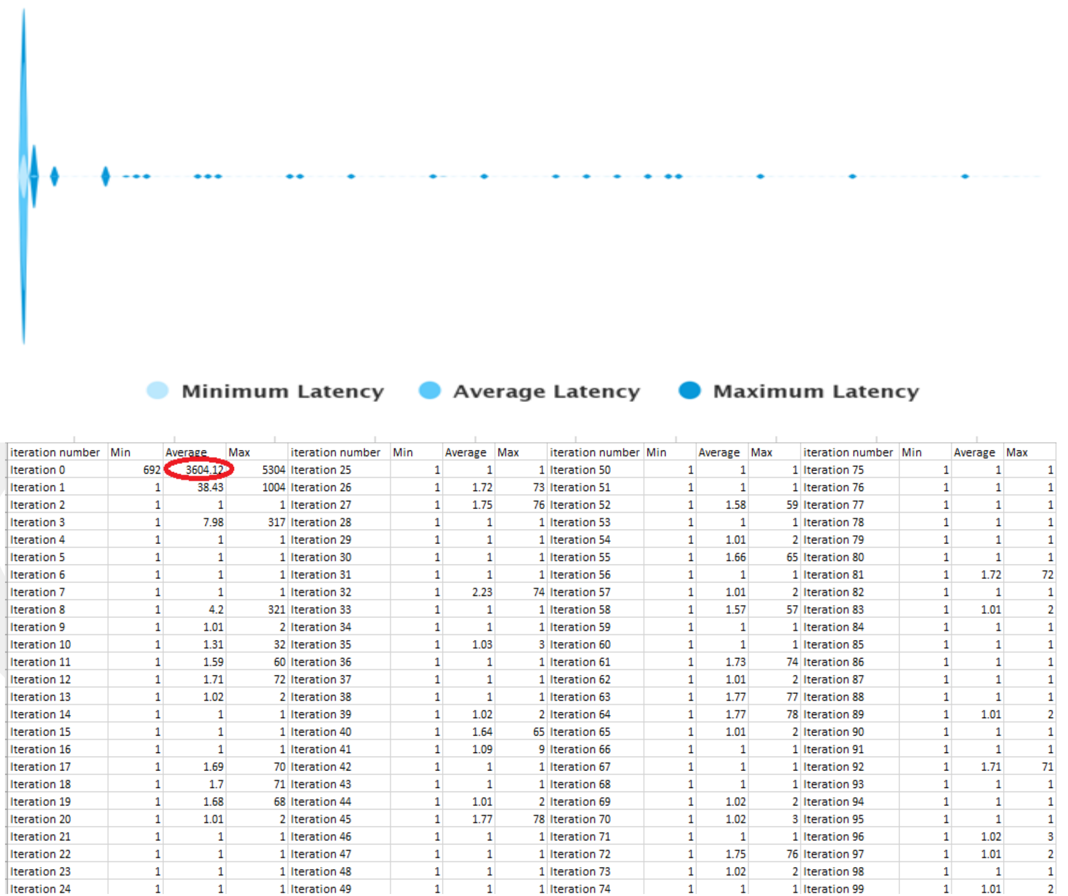
Figure 5.8 HTTP Message Generation Latency of 25 Client and 25 Repeat



iteration number	Min	Average	Max
Iteration 0	1534	3104.82	4112
Iteration 1	1	1	1
Iteration 2	1	2.66	80
Iteration 3	1	1	1
Iteration 4	1	1.02	2
Iteration 5	1	1.4	21
Iteration 6	1	1	1
Iteration 7	1	1	1
Iteration 8	1	1	1
Iteration 9	1	1	1
Iteration 10	1	1	1
Iteration 11	1	1	1
Iteration 12	1	1	1
Iteration 13	1	1.04	2
Iteration 14	1	1	1
Iteration 15	1	1.02	2
Iteration 16	1	1	1
Iteration 17	1	1.02	2
Iteration 18	1	1	1
Iteration 19	1	1	1
Iteration 20	1	1.04	2
Iteration 21	1	1.06	2
Iteration 22	1	1	1
Iteration 23	1	1	1
Iteration 24	1	1.02	2

iteration number	Min	Average	Max
Iteration 25	1	1.02	2
Iteration 26	1	1.02	2
Iteration 27	1	1.1	5
Iteration 28	1	1	1
Iteration 29	1	1	1
Iteration 30	1	1.02	2
Iteration 31	1	1	1
Iteration 32	1	1	1
Iteration 33	1	1	1
Iteration 34	1	1	1
Iteration 35	1	1.04	2
Iteration 36	1	1	1
Iteration 37	1	1.02	2
Iteration 38	1	1	1
Iteration 39	1	1	1
Iteration 40	1	1.02	2
Iteration 41	1	1	1
Iteration 42	1	1	1
Iteration 43	1	1	1
Iteration 44	1	1	1
Iteration 45	1	1	1
Iteration 46	1	1	1
Iteration 47	1	2.48	74
Iteration 48	1	1.02	2
Iteration 49	1	1	1

Figure 5.9 HTTP Message Generation Latency of 50 Client and 50 Repeat



iteration number	Min	Average	Max	iteration number	Min	Average	Max	iteration number	Min	Average	Max	iteration number	Min	Average	Max
Iteration 0	692	3604.12	5304	Iteration 25	1	1	1	Iteration 50	1	1	1	Iteration 75	1	1	1
Iteration 1	1	38.43	1004	Iteration 26	1	1.72	73	Iteration 51	1	1	1	Iteration 76	1	1	1
Iteration 2	1	1	1	Iteration 27	1	1.75	76	Iteration 52	1	1.58	59	Iteration 77	1	1	1
Iteration 3	1	7.98	317	Iteration 28	1	1	1	Iteration 53	1	1	1	Iteration 78	1	1	1
Iteration 4	1	1	1	Iteration 29	1	1	1	Iteration 54	1	1.01	2	Iteration 79	1	1	1
Iteration 5	1	1	1	Iteration 30	1	1	1	Iteration 55	1	1.66	65	Iteration 80	1	1	1
Iteration 6	1	1	1	Iteration 31	1	1	1	Iteration 56	1	1	1	Iteration 81	1	1.72	72
Iteration 7	1	1	1	Iteration 32	1	2.23	74	Iteration 57	1	1.01	2	Iteration 82	1	1	1
Iteration 8	1	4.2	321	Iteration 33	1	1	1	Iteration 58	1	1.57	57	Iteration 83	1	1.01	2
Iteration 9	1	1.01	2	Iteration 34	1	1	1	Iteration 59	1	1	1	Iteration 84	1	1	1
Iteration 10	1	1.31	32	Iteration 35	1	1.03	3	Iteration 60	1	1	1	Iteration 85	1	1	1
Iteration 11	1	1.59	60	Iteration 36	1	1	1	Iteration 61	1	1.73	74	Iteration 86	1	1	1
Iteration 12	1	1.71	72	Iteration 37	1	1	1	Iteration 62	1	1.01	2	Iteration 87	1	1	1
Iteration 13	1	1.02	2	Iteration 38	1	1	1	Iteration 63	1	1.77	77	Iteration 88	1	1	1
Iteration 14	1	1	1	Iteration 39	1	1.02	2	Iteration 64	1	1.77	78	Iteration 89	1	1.01	2
Iteration 15	1	1	1	Iteration 40	1	1.64	65	Iteration 65	1	1.01	2	Iteration 90	1	1	1
Iteration 16	1	1	1	Iteration 41	1	1.09	9	Iteration 66	1	1	1	Iteration 91	1	1	1
Iteration 17	1	1.69	70	Iteration 42	1	1	1	Iteration 67	1	1	1	Iteration 92	1	1.71	71
Iteration 18	1	1.7	71	Iteration 43	1	1	1	Iteration 68	1	1	1	Iteration 93	1	1	1
Iteration 19	1	1.68	68	Iteration 44	1	1.01	2	Iteration 69	1	1.02	2	Iteration 94	1	1	1
Iteration 20	1	1.01	2	Iteration 45	1	1.77	78	Iteration 70	1	1.02	3	Iteration 95	1	1	1
Iteration 21	1	1	1	Iteration 46	1	1	1	Iteration 71	1	1	1	Iteration 96	1	1.02	3
Iteration 22	1	1	1	Iteration 47	1	1	1	Iteration 72	1	1.75	76	Iteration 97	1	1.01	2
Iteration 23	1	1	1	Iteration 48	1	1	1	Iteration 73	1	1.02	2	Iteration 98	1	1	1
Iteration 24	1	1	1	Iteration 49	1	1	1	Iteration 74	1	1	1	Iteration 99	1	1.01	2

Figure 5.10 HTTP Message Generation Latency of 100 Client and 100 Repeat

5.3.2 Message Generation Latency Analysis for MQTT

The client number, repeating message, and three QoS values are tested for the message generation latency in MQTT protocol. We observed the latency of 48 different scenarios in total and, from the eye charts produced by IoTIFY, it can be seen that the highest latency is in the first iteration of each scenario. Then, it settles down and continues in the range of 1 ms. The outcomes of minimum, average, and maximum latency for repeating messages of 5, 25, 50, and 100 times, with client numbers of 10, 25, 50, and 100 and QoS values of 0, 1, and 2, as shown in Table 5.7, Table 5.8, Table 5.9, and Table 5.10. The collected data are grouped and divided into tables considering the repeat times; for example, the first table shows the data for 5 repeat times. Next come the measured values for different client numbers and different QoS values. The other tables are also orginezed in the same manner. It is evident that the average latency increases whenever the client number increases while QoS and repeating message remain constant. The highest latency is seen in the first iteration with 100 as the client number for all scenarios. There may be some scenarios where average latency for a higher client number is less than the others, but the reason may lie in the connection quality with server when the test was done.

Table 5.7 MQTT Message Generation Latencies with 5 Repeats

Repeating message	Client number	QoS	Minimum latency	Average latency	Maximum latency
5	10	0	1900	2240.1	2502
		1	1575	1969.5	2269
		2	1264	1623.3	1871
	25	0	1495	2380.96	3231
		1	1069	1882.72	2667
		2	400	1391.08	2234
	50	0	1604	3622.98	5302
		1	1891	3308.14	4413
		2	1012	3016.9	4090
	100	0	1352	3837.66	5803
		1	771	3165.15	4803
		2	1895	4798.63	7503

Table 5.8 MQTT Message Generation Latencies with 25 Repeats

Repeating-message	**Client number**	**QoS**	**Minimum latency**	**Average latency**	**Maximum latency**
25	10	0	708	1105.5	1455
		1	1532	1905	2298
		2	1388	1882.1	2189
	25	0	534	1520.8	2179
		1	1142	2047.24	2891
		2	644	1454.28	2205
	50	0	2193	3449.5	4388
		1	1796	3256.3	4117
		2	1619	3211.68	4507
	100	0	1486	3552.68	5004
		1	2599	4857.67	6696
		2	1016	3900.2	6412

Table 5.9 MQTT Message Generation Latencies with 50 Repeats

Repeating-message	Client number	QoS	Minimum latency	Average latency	Maximum latency
50	10	0	1658	2211.8	2644
		1	1406	1960.1	2300
		2	1060	1417.9	1714
	25	0	1471	2480	3285
		1	1271	2216.08	3041
		2	693	1683.48	2365
	50	0	1585	2513.82	3494
		1	1311	2708.76	3606
		2	1400	2663.96	3508
	100	0	376	4110.93	6491
		1	1195	4875.81	7567
		2	301	3719.39	5622

Table 5.10 MQTT Message Generation Latencies with 100 Repeats

Repeating-message	**Client number**	**QoS**	**Minimum latency**	**Average latency**	**Maximum latency**
100	10	0	756	1000.9	1304
		1	1452	1828.2	2153
		2	1441	1874	2218
	25	0	805	1822.4	2701
		1	1291	2299.28	2994
		2	1265	2519.24	3330
	50	0	1304	2492.62	3395
		1	870	2645.82	3788
		2	1387	2766.24	3894
	100	0	513	4272.6	7358
		1	522	3914.19	6102
		2	1883	4183.02	6375

5.4 Packet Sending Latency Analysis

The time needed to send messages to the server is measured as packet sending latency. The minimum, average, and maximum latency values are presented in the x-axis of charts for each iteration in milliseconds as a real performance of the server. The packet sending latency is highly dependent on the protocols used. In simulations, we observed the packet sending latency for 16 HTTP scenarios and for 48 MQTT scenarios.

5.4.1 Analysis of Packet Sending Latency for HTTP Protocol

The analysis of packet sending latency for HTTP protocol is done by measuring the effect of client numbers (10, 25, 50, and 100) and repeating message times (5, 25, 50, and 100). The minimum, average and maximum latencies are shown for each iteration, where the highest latency mostly appears in the first iteration, as shown in Figure 5.11, Figure 5.12, Figure 5.13, and Figure 5.14. However, there are also some scenarios that show the highest latency at later iterations during the connection, as shown in Figure 5.15.

iteration number	Min	Average	MAX
Iteration 0	178	527.9	971
Iteration 1	175	179.1	183
Iteration 2	174	180.9	185
Iteration 3	175	181.2	188
Iteration 4	177	179.4	185

Packet sending latency

Displays the Min,Max and Average time in Millisecond to send packet across each iteration.

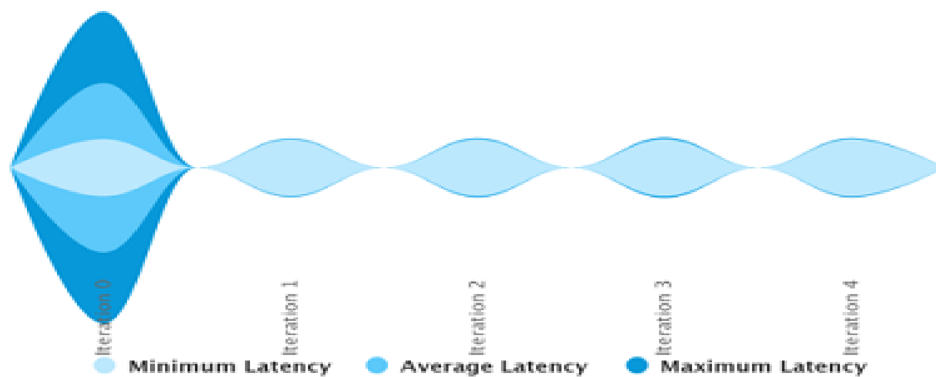
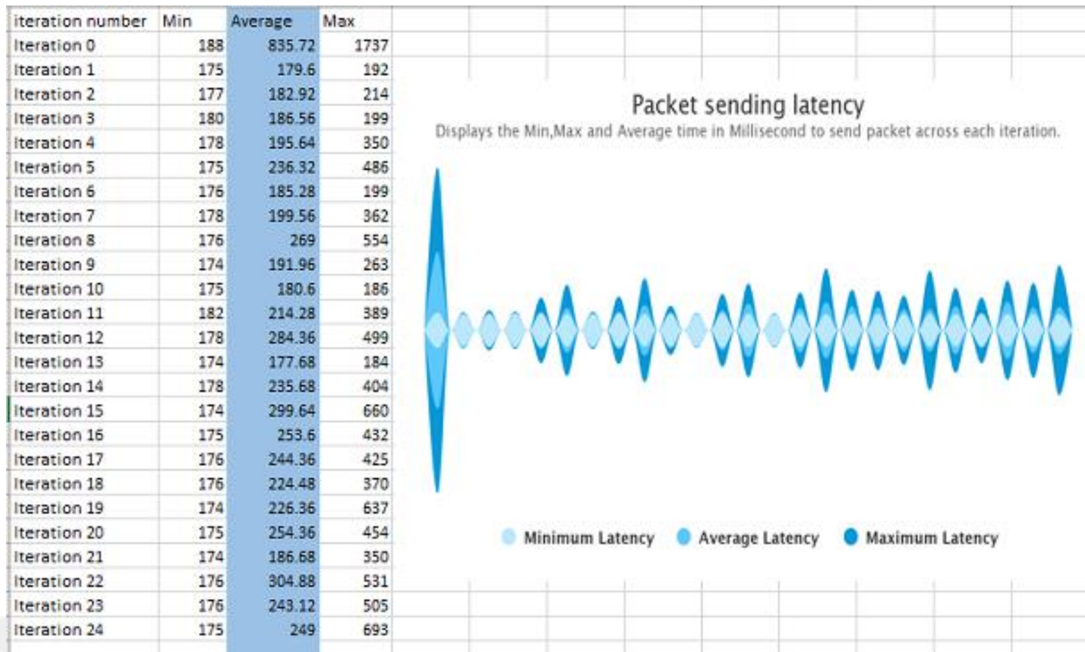
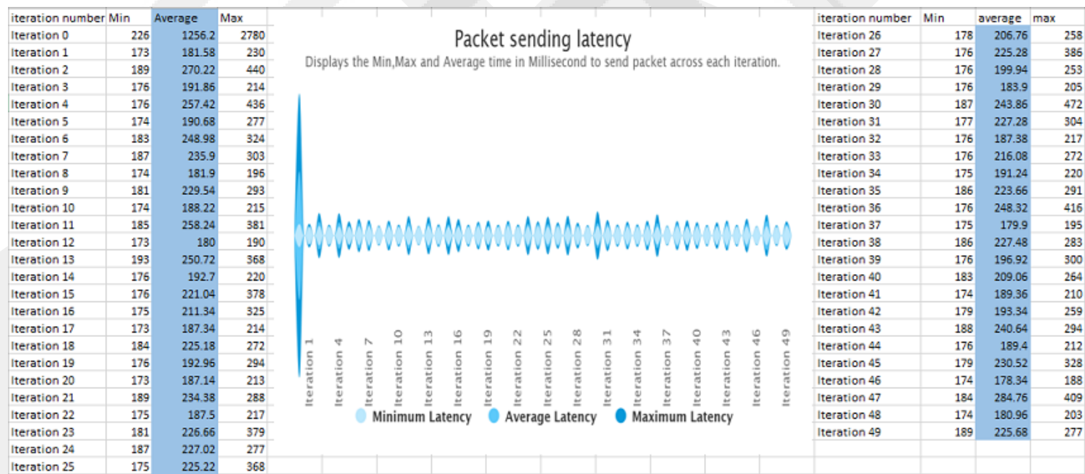


Figure 5.11 HTTP Packet Sending Latency of 10 Client and 5 Repeat



iteration number	Min	Average	Max
Iteration 0	188	835.72	1737
Iteration 1	175	179.6	192
Iteration 2	177	182.92	214
Iteration 3	180	186.56	199
Iteration 4	178	195.64	350
Iteration 5	175	236.32	486
Iteration 6	176	185.28	199
Iteration 7	178	199.56	362
Iteration 8	176	269	554
Iteration 9	174	191.96	263
Iteration 10	175	180.6	186
Iteration 11	182	214.28	389
Iteration 12	178	284.36	499
Iteration 13	174	177.68	184
Iteration 14	178	235.68	404
Iteration 15	174	299.64	660
Iteration 16	175	253.6	432
Iteration 17	176	244.36	425
Iteration 18	176	224.48	370
Iteration 19	174	226.36	637
Iteration 20	175	254.36	454
Iteration 21	174	186.68	350
Iteration 22	176	304.88	531
Iteration 23	176	243.12	505
Iteration 24	175	249	693

Figure 5.12 HTTP Packet Sending Latency of 25 Client and 25 Repeat



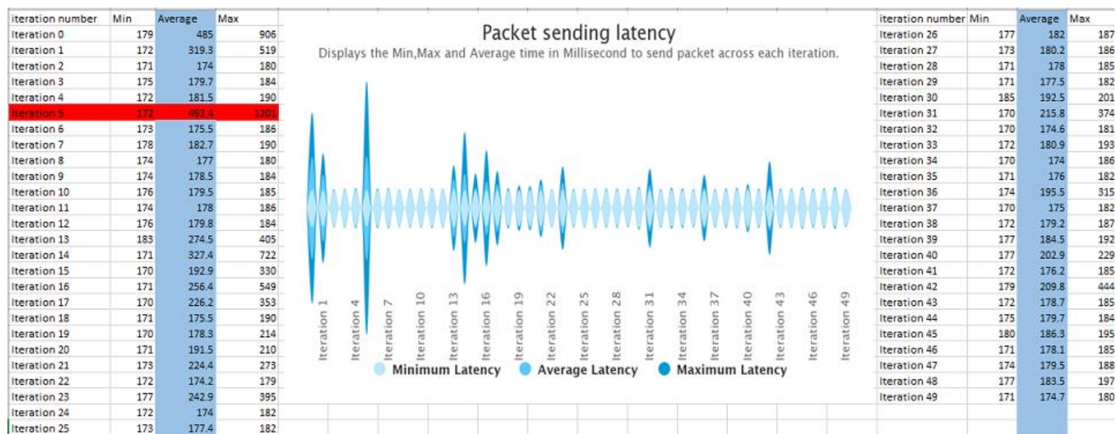
iteration number	Min	Average	Max
Iteration 0	226	1256.2	2780
Iteration 1	173	181.58	230
Iteration 2	189	270.22	440
Iteration 3	176	191.86	214
Iteration 4	176	257.42	436
Iteration 5	174	190.68	277
Iteration 6	183	248.98	324
Iteration 7	187	235.9	303
Iteration 8	174	181.9	196
Iteration 9	181	229.54	293
Iteration 10	174	188.22	215
Iteration 11	185	258.24	381
Iteration 12	173	180	190
Iteration 13	193	250.72	368
Iteration 14	176	192.7	220
Iteration 15	176	221.04	378
Iteration 16	175	211.34	325
Iteration 17	173	187.34	214
Iteration 18	184	225.18	272
Iteration 19	176	192.96	294
Iteration 20	173	187.14	213
Iteration 21	189	234.38	288
Iteration 22	175	187.5	217
Iteration 23	181	226.66	379
Iteration 24	187	227.02	277
Iteration 25	175	225.22	368

iteration number	Min	average	max
Iteration 26	178	206.76	258
Iteration 27	176	225.28	386
Iteration 28	176	199.94	253
Iteration 29	176	183.9	205
Iteration 30	187	243.86	472
Iteration 31	177	227.28	304
Iteration 32	176	187.38	217
Iteration 33	176	216.08	272
Iteration 34	175	191.24	220
Iteration 35	186	223.66	291
Iteration 36	176	248.32	416
Iteration 37	175	179.9	195
Iteration 38	186	227.48	283
Iteration 39	176	196.92	300
Iteration 40	183	209.06	264
Iteration 41	174	189.36	210
Iteration 42	179	193.34	259
Iteration 43	188	240.64	294
Iteration 44	176	189.4	212
Iteration 45	179	230.52	328
Iteration 46	174	178.34	188
Iteration 47	184	284.76	409
Iteration 48	174	180.96	203
Iteration 49	189	225.68	277

Figure 5.13 HTTP Packet Sending Latency of 50 Client and 50 Repeat

iteration number	Min	Average	Max	iteration number	Min	Average	Max	iteration number	Min	Average	Max	iteration number	Min	Average	Max
Iteration 0	1	1803.25	3979	Iteration 26	180	238.1	461	Iteration 52	185	234.26	298	Iteration 78	182	241.12	298
Iteration 1	177	340.08	487	Iteration 27	183	285.19	405	Iteration 53	188	255.73	298	Iteration 79	178	255.53	298
Iteration 2	175	231.65	297	Iteration 28	188	283.41	625	Iteration 54	180	244.61	298	Iteration 80	172	230.03	298
Iteration 3	1	260.69	372	Iteration 29	176	243.33	356	Iteration 55	197	289.04	298	Iteration 81	174	240.43	298
Iteration 4	177	273.36	420	Iteration 30	179	298.46	479	Iteration 56	186	217.85	298	Iteration 82	174	212.46	298
Iteration 5	175	307.01	575	Iteration 31	173	198.12	278	Iteration 57	174	222.62	298	Iteration 83	180	257.13	298
Iteration 6	172	250.2	357	Iteration 32	182	247.93	291	Iteration 58	185	256.85	298	Iteration 84	175	218.41	298
Iteration 7	178	268.5	606	Iteration 33	181	275.03	552	Iteration 59	197	325.13	298	Iteration 85	172	228.13	298
Iteration 8	1	243.47	470	Iteration 34	172	208.17	269	Iteration 60	175	210.24	298	Iteration 86	189	231.28	298
Iteration 9	175	286.89	505	Iteration 35	186	286.72	524	Iteration 61	183	263.1	298	Iteration 87	184	291.59	298
Iteration 10	183	234.97	391	Iteration 36	174	241.5	307	Iteration 62	181	227	298	Iteration 88	178	228.71	298
Iteration 11	181	273.08	307	Iteration 37	180	243.32	304	Iteration 63	181	243.38	298	Iteration 89	176	256.95	298
Iteration 12	191	241.9	286	Iteration 38	191	258.49	313	Iteration 64	197	262.73	298	Iteration 90	175	195.19	298
Iteration 13	187	238.96	430	Iteration 39	181	249.08	389	Iteration 65	185	288.28	298	Iteration 91	175	230.99	298
Iteration 14	191	263.58	326	Iteration 40	182	240.39	305	Iteration 66	182	231.19	298	Iteration 92	195	333.18	298
Iteration 15	183	275.02	395	Iteration 41	181	271.52	397	Iteration 67	194	277.24	298	Iteration 93	172	213.83	298
Iteration 16	181	237.62	286	Iteration 42	183	236.56	388	Iteration 68	181	214.61	298	Iteration 94	175	227.74	298
Iteration 17	185	245.52	387	Iteration 43	182	248.42	386	Iteration 69	183	255.69	298	Iteration 95	175	243.81	298
Iteration 18	185	257.49	301	Iteration 44	188	230.88	278	Iteration 70	180	230.77	298	Iteration 96	183	229.63	298
Iteration 19	184	234.09	371	Iteration 45	183	258.13	303	Iteration 71	177	229.7	298	Iteration 97	182	249.19	298
Iteration 20	184	233.78	290	Iteration 46	192	223.05	282	Iteration 72	186	234.79	298	Iteration 98	176	244.98	298
Iteration 21	191	282.46	406	Iteration 47	188	242.71	333	Iteration 73	183	249.76	298	Iteration 99	186	280.82	298
Iteration 22	190	231.12	299	Iteration 48	187	242.74	295	Iteration 74	190	237.19	298				
Iteration 23	188	283.5	567	Iteration 49	193	257.42	483	Iteration 75	189	259.68	298				
Iteration 24	180	236.38	423	Iteration 50	181	220.81	273	Iteration 76	175	226.72	298				
Iteration 25	178	236.78	379	Iteration 51	181	227.67	298	Iteration 77	172	229.08	298				

Figure 5.14 HTTP Packet Sending Latency of 100 Client and 100 Repeat



iteration number	Min	Average	Max
Iteration 0	179	485	906
Iteration 1	172	319.3	519
Iteration 2	171	174	180
Iteration 3	175	179.7	184
Iteration 4	172	181.5	190
Iteration 5	172	492.4	1201
Iteration 6	173	175.5	186
Iteration 7	178	182.7	190
Iteration 8	174	177	180
Iteration 9	174	178.5	184
Iteration 10	176	179.5	185
Iteration 11	174	178	186
Iteration 12	176	179.8	184
Iteration 13	183	274.5	405
Iteration 14	171	327.4	722
Iteration 15	170	192.9	330
Iteration 16	171	256.4	549
Iteration 17	170	226.2	353
Iteration 18	171	175.5	190
Iteration 19	170	178.3	214
Iteration 20	171	191.5	210
Iteration 21	173	224.4	273
Iteration 22	172	174.2	179
Iteration 23	177	242.9	395
Iteration 24	172	174	182
Iteration 25	173	177.4	182

iteration number	Min	Average	Max
Iteration 26	177	182	187
Iteration 27	173	180.2	186
Iteration 28	171	178	185
Iteration 29	171	177.5	182
Iteration 30	185	192.5	201
Iteration 31	170	215.8	374
Iteration 32	170	174.6	181
Iteration 33	172	180.9	193
Iteration 34	170	174	186
Iteration 35	171	176	182
Iteration 36	174	195.5	315
Iteration 37	170	175	182
Iteration 38	172	179.2	187
Iteration 39	177	184.5	192
Iteration 40	177	202.9	229
Iteration 41	172	176.2	185
Iteration 42	179	209.8	444
Iteration 43	172	178.7	185
Iteration 44	175	179.7	184
Iteration 45	180	186.3	195
Iteration 46	171	178.1	185
Iteration 47	174	179.5	188
Iteration 48	177	183.5	197
Iteration 49	171	174.7	180

Figure 5.15 Hightet Packet Sending Latency Show in Iteration 5 of Repeating Message 50 Times and Client Number 10

The table is organized, first, according to the repeating times and, then according to the number of clients. In the table, one can observe an increase depending on the client numbers; once this number increases, so does the average latency. The number of repeats does not have an evident effect on the packet sending latency.

Table 5.11 The HTTP Average Packet Sending Latency

Repeating message	Client number	Minimum Latency	Average latency (highest)	Maximum Latency
5	10	178	527.9	971
	25	187	937.84	1716
	50	272	1159.02	2401
	100	308	2232.52	4694
25	10	182	542.7	924
	25	188	835.72	1737
	50	219	1466	2989
	100	1	1803.72	3968

50	10	172	492.4	1201
	25	458	949.52	1754
	50	226	1256.2	2780
	100	204	1992.42	4202
100	10	211	557.15	1065
	25	201	918.08	1870
	50	272	1044.84	2395
	100	1	1803.25	3979

5.4.2 Analysis of Packet Sending Latency for MQTT Protocol

The packet sending latency (minimum, average, and maximum) for each iteration in MQTT protocol is measured by changing the number of clients, QoS values, and repeating times. In total, 48 different scenarios were tested, and in Table 5.12 the average packet sending latencies of each scnerio in MQTT are presented depending on repeating message times, client number and QoS values. Accordingly, an increase can be observed in the average latency when client numbers for QoS values of 1 and 2 increased compared to QoS 0 of the same client numbers. An increase in the client numbers with QoS (1and 2) also increases the packet sending latency because each sent message should be acknowledged to guarantee delivery, thereby adding to its latency considerably.

Nonetheless, once the QoS value is 0, the latency values do not show a significant increase. The latency is around 1 ms for all iterations. The measured packet sending latencies for different scnerios are shown in Figure 5.16, Figure 5.17, Figure 5.18, Figure 5.19, Figure 5.20, Figure 5.21, and Figure 5.22.

Table 5.12 The MQTT Average Packet Sending Latency

Repeating-message	Client number	QoS	Average latency
5	10	0	1
		1	228.4
		2	276.2
	25	0	1
		1	699.96
		2	891.4
	50	0	1.12
		1	1057.54
		2	1027.26
	100	0	1.64
		1	1758.57
		2	2626.48
25	10	0	1.1
		1	359.4
		2	326.5
	25	0	1.04
		1	890.68
		2	773.5
	50	0	1.04
		1	914.46
		2	1351.56
	100	0	1.03
		1	2774.24
		2	2498.05
50	10	0	1.1
		1	339
		2	384.1

	25	0	1.04
		1	822.56
		2	834.28
	50	0	4.26
		1	890.72
		2	883.28
	100	0	1.01
		1	2577.65
		2	1446.67
100	10	0	1.1
		1	273.9
		2	416.9
	25	0	1.04
		1	721.64
		2	900.6
	50	0	1.04
		1	938.16
		2	1017.88
	100	0	1.64
		1	1976.43
		2	2114.69

itration number	Minimum Latency	Average Latency	Maximum Latency
Iteration 0	1	1	1
Iteration 1	1	1	1
Iteration 2	1	1	1
Iteration 3	1	1	1
Iteration 4	1	1	1

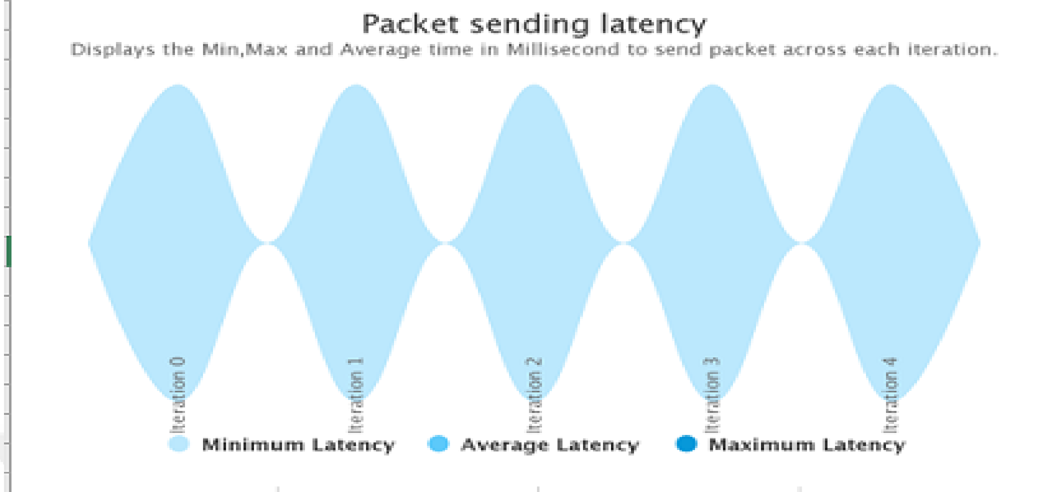


Figure 5.16 MQTT Packet Sending Latency of 5 Repeat 10 Client 0 QoS

iteration number	Minimum Latency	Average Latency	Maximum Latency
Iteration 0	1	1	1
Iteration 1	1	1.12	7
Iteration 2	1	1	1
Iteration 3	1	1	1
Iteration 4	1	1	1

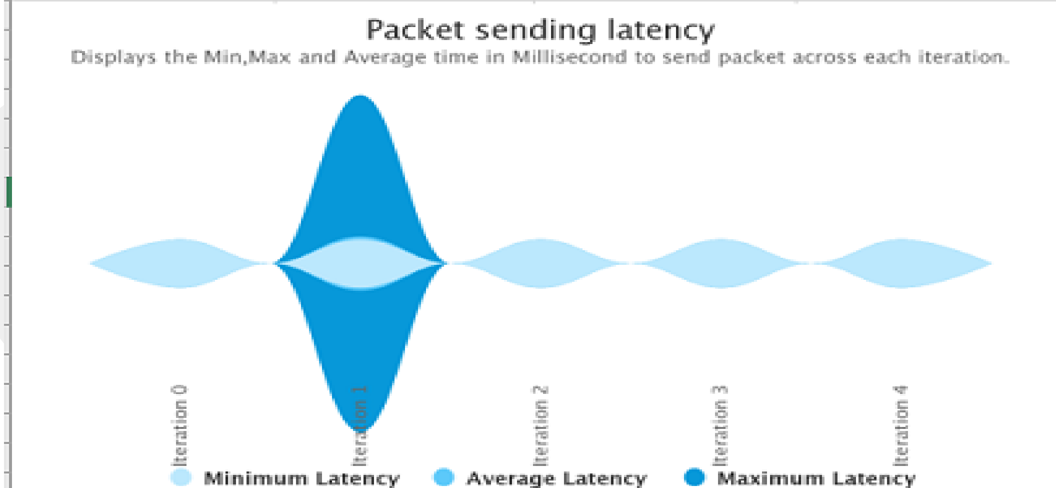
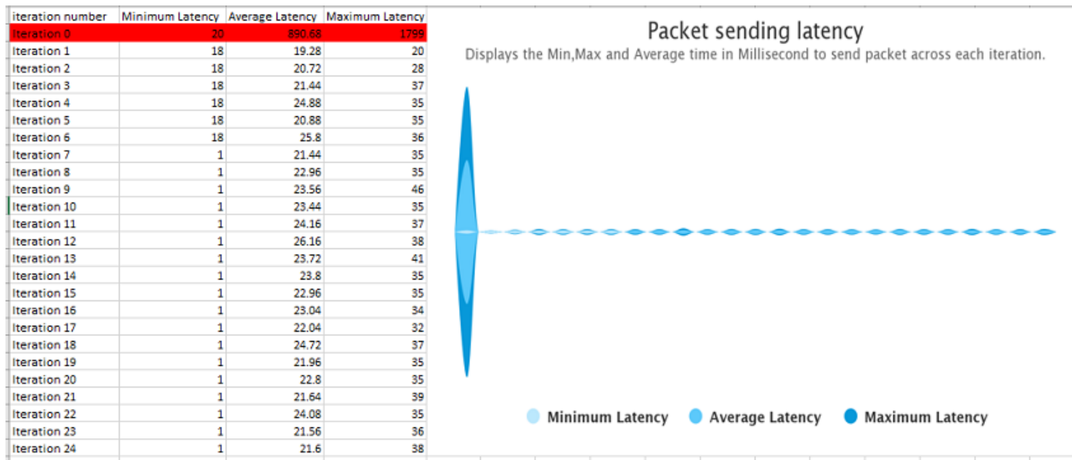
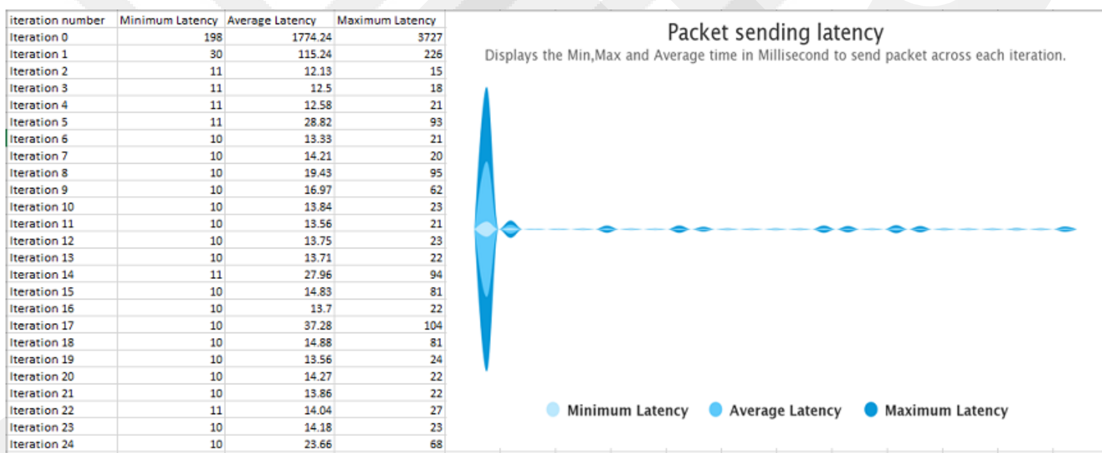


Figure 5.17 MQTT Packet Sending Latency of 5 Repeat 50 Client 0 QoS



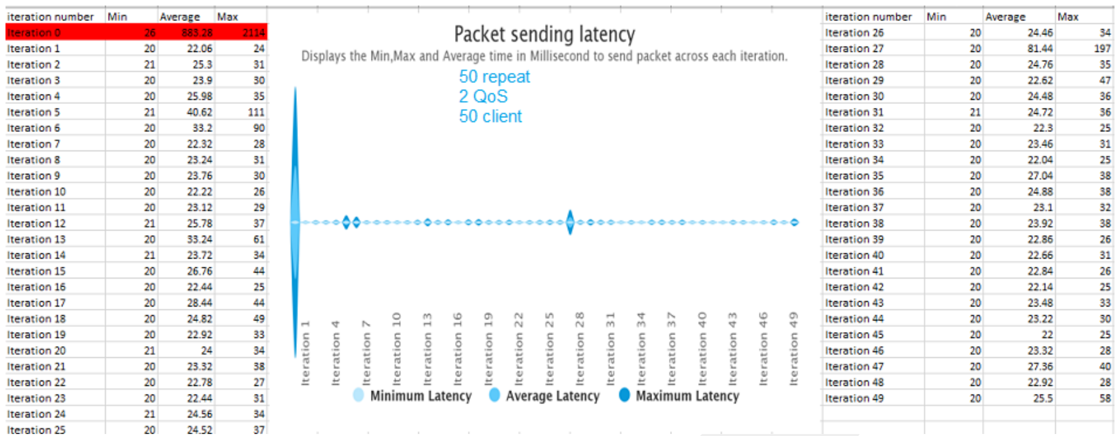
iteration number	Minimum Latency	Average Latency	Maximum Latency
Iteration 0	20	890.68	1799
Iteration 1	18	19.28	20
Iteration 2	18	20.72	28
Iteration 3	18	21.44	37
Iteration 4	18	24.88	35
Iteration 5	18	20.88	35
Iteration 6	18	25.8	36
Iteration 7	1	21.44	35
Iteration 8	1	22.96	35
Iteration 9	1	23.56	46
Iteration 10	1	23.44	35
Iteration 11	1	24.16	37
Iteration 12	1	26.16	38
Iteration 13	1	23.72	41
Iteration 14	1	23.8	35
Iteration 15	1	22.96	35
Iteration 16	1	23.04	34
Iteration 17	1	22.04	32
Iteration 18	1	24.72	37
Iteration 19	1	21.96	35
Iteration 20	1	22.8	35
Iteration 21	1	21.64	39
Iteration 22	1	24.08	35
Iteration 23	1	21.56	36
Iteration 24	1	21.6	38

Figure 5.18 MQTT Packet Sending Latency of 25 Repeat 25 Client 1 QoS



iteration number	Minimum Latency	Average Latency	Maximum Latency
Iteration 0	198	1774.24	3727
Iteration 1	30	115.24	226
Iteration 2	11	12.13	15
Iteration 3	11	12.5	18
Iteration 4	11	12.58	21
Iteration 5	11	28.82	93
Iteration 6	10	13.33	21
Iteration 7	10	14.21	20
Iteration 8	10	19.43	95
Iteration 9	10	16.97	62
Iteration 10	10	13.84	23
Iteration 11	10	13.56	21
Iteration 12	10	13.75	23
Iteration 13	10	13.71	22
Iteration 14	11	27.96	94
Iteration 15	10	14.83	81
Iteration 16	10	13.7	22
Iteration 17	10	37.28	104
Iteration 18	10	14.88	81
Iteration 19	10	13.56	24
Iteration 20	10	14.27	22
Iteration 21	10	13.86	22
Iteration 22	11	14.04	27
Iteration 23	10	14.18	23
Iteration 24	10	23.66	68

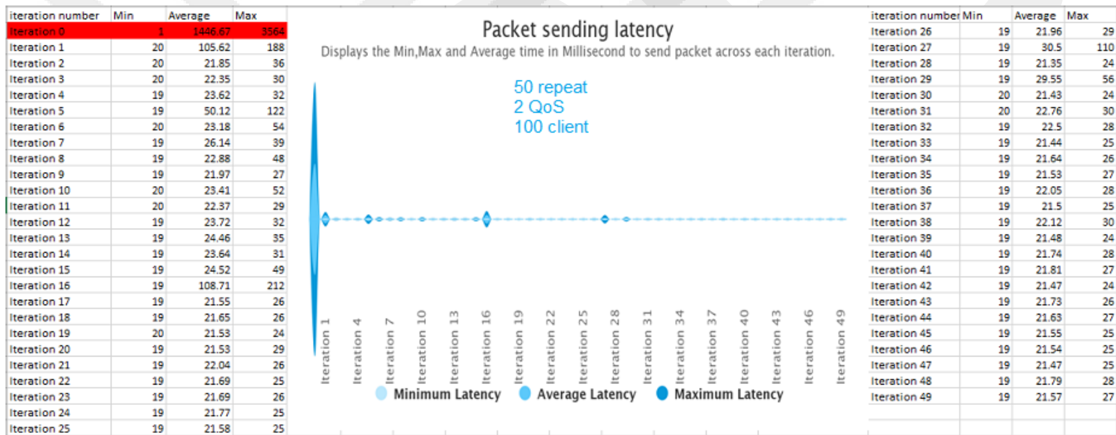
Figure 5.19 MQTT Packet Sending Latency of 25 Repeat 100 Client 1 QoS



iteration number	Min	Average	Max
Iteration 0	26	883.28	2114
Iteration 1	20	22.06	24
Iteration 2	21	25.3	31
Iteration 3	20	23.9	30
Iteration 4	20	25.98	35
Iteration 5	21	40.62	111
Iteration 6	20	33.2	90
Iteration 7	20	22.32	28
Iteration 8	20	23.24	31
Iteration 9	20	23.76	30
Iteration 10	20	22.22	26
Iteration 11	20	23.12	29
Iteration 12	21	25.78	37
Iteration 13	20	33.24	61
Iteration 14	21	23.72	34
Iteration 15	20	26.76	44
Iteration 16	20	22.44	25
Iteration 17	20	28.44	44
Iteration 18	20	24.82	49
Iteration 19	20	22.92	33
Iteration 20	21	24	34
Iteration 21	20	23.32	38
Iteration 22	20	22.78	27
Iteration 23	20	22.44	31
Iteration 24	21	24.56	34
Iteration 25	20	24.52	37

iteration number	Min	Average	Max
Iteration 26	20	24.46	34
Iteration 27	20	81.44	197
Iteration 28	20	24.76	35
Iteration 29	20	22.62	47
Iteration 30	20	24.48	36
Iteration 31	21	24.72	36
Iteration 32	20	22.3	25
Iteration 33	20	23.46	31
Iteration 34	20	22.04	25
Iteration 35	20	27.04	38
Iteration 36	20	24.88	38
Iteration 37	20	23.1	32
Iteration 38	20	23.92	38
Iteration 39	20	22.86	26
Iteration 40	20	22.66	31
Iteration 41	20	22.84	26
Iteration 42	20	22.14	25
Iteration 43	20	23.48	33
Iteration 44	20	23.22	30
Iteration 45	20	22	25
Iteration 46	20	23.32	28
Iteration 47	20	27.36	40
Iteration 48	20	22.92	28
Iteration 49	20	25.5	58

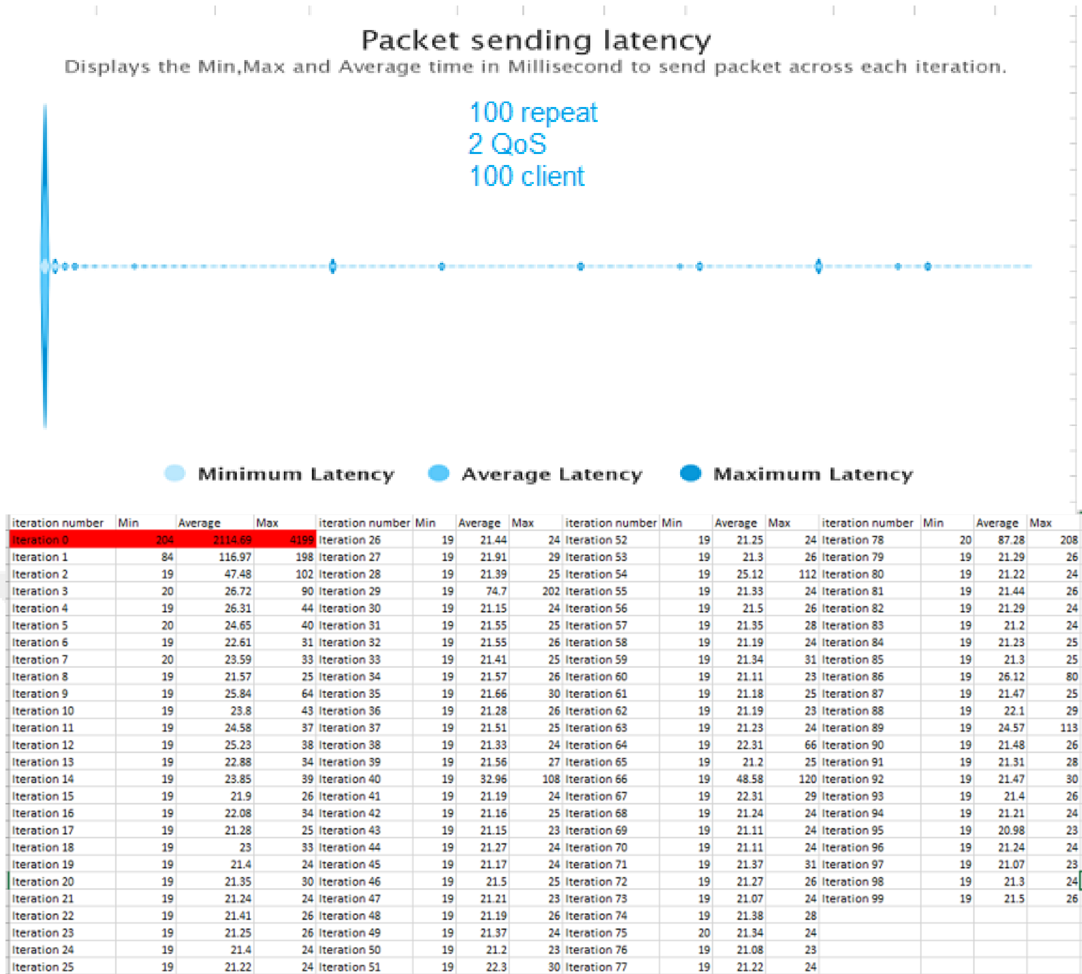
Figure 5.20 MQTT Packet Sending Latency of 50 Repeat 50 Client 2 QoS



iteration number	Min	Average	Max
Iteration 0	1	1446.67	3564
Iteration 1	20	105.62	188
Iteration 2	20	21.85	36
Iteration 3	20	22.35	30
Iteration 4	19	23.62	32
Iteration 5	19	50.12	122
Iteration 6	20	23.18	54
Iteration 7	19	26.14	39
Iteration 8	19	22.88	48
Iteration 9	19	21.97	27
Iteration 10	20	23.41	52
Iteration 11	20	22.37	29
Iteration 12	19	23.72	32
Iteration 13	19	24.46	35
Iteration 14	19	23.64	31
Iteration 15	19	24.52	49
Iteration 16	19	108.71	212
Iteration 17	19	21.55	26
Iteration 18	19	21.65	26
Iteration 19	20	21.53	24
Iteration 20	19	21.53	29
Iteration 21	19	22.04	26
Iteration 22	19	21.69	25
Iteration 23	19	21.69	26
Iteration 24	19	21.77	25
Iteration 25	19	21.58	25

iteration number	Min	Average	Max
Iteration 26	19	21.96	29
Iteration 27	19	30.5	110
Iteration 28	19	21.35	24
Iteration 29	19	29.55	56
Iteration 30	20	21.43	24
Iteration 31	20	22.76	30
Iteration 32	19	22.5	28
Iteration 33	19	21.44	25
Iteration 34	19	21.64	26
Iteration 35	19	21.53	27
Iteration 36	19	22.05	28
Iteration 37	19	21.5	25
Iteration 38	19	22.12	30
Iteration 39	19	21.48	24
Iteration 40	19	21.74	28
Iteration 41	19	21.81	27
Iteration 42	19	21.47	24
Iteration 43	19	21.73	26
Iteration 44	19	21.63	27
Iteration 45	19	21.55	25
Iteration 46	19	21.54	25
Iteration 47	19	21.47	25
Iteration 48	19	21.79	28
Iteration 49	19	21.57	27

Figure 5.21 MQTT Packet Sending Latency of 50 Repeat 100 Client 2 QoS



iteration number	Min	Average	Max	iteration number	Min	Average	Max	iteration number	Min	Average	Max	iteration number	Min	Average	Max
Iteration 0	204	2114.69	4199	Iteration 26	19	21.44	24	Iteration 52	19	21.25	24	Iteration 78	20	87.28	208
Iteration 1	84	116.97	198	Iteration 27	19	21.91	29	Iteration 53	19	21.3	26	Iteration 79	19	21.29	26
Iteration 2	19	47.48	102	Iteration 28	19	21.39	25	Iteration 54	19	25.12	112	Iteration 80	19	21.22	24
Iteration 3	20	26.72	90	Iteration 29	19	74.7	202	Iteration 55	19	21.33	24	Iteration 81	19	21.44	26
Iteration 4	19	26.31	44	Iteration 30	19	21.15	24	Iteration 56	19	21.5	26	Iteration 82	19	21.29	24
Iteration 5	20	24.65	40	Iteration 31	19	21.55	25	Iteration 57	19	21.35	28	Iteration 83	19	21.2	24
Iteration 6	19	22.61	31	Iteration 32	19	21.55	26	Iteration 58	19	21.19	24	Iteration 84	19	21.23	25
Iteration 7	20	23.59	33	Iteration 33	19	21.41	25	Iteration 59	19	21.34	31	Iteration 85	19	21.3	25
Iteration 8	19	21.57	25	Iteration 34	19	21.57	26	Iteration 60	19	21.11	23	Iteration 86	19	26.12	80
Iteration 9	19	25.84	64	Iteration 35	19	21.66	30	Iteration 61	19	21.18	25	Iteration 87	19	21.47	25
Iteration 10	19	23.8	43	Iteration 36	19	21.28	26	Iteration 62	19	21.19	23	Iteration 88	19	22.1	29
Iteration 11	19	24.58	37	Iteration 37	19	21.51	25	Iteration 63	19	21.23	24	Iteration 89	19	24.57	113
Iteration 12	19	25.23	38	Iteration 38	19	21.33	24	Iteration 64	19	22.31	66	Iteration 90	19	21.48	26
Iteration 13	19	22.88	34	Iteration 39	19	21.56	27	Iteration 65	19	21.2	25	Iteration 91	19	21.31	28
Iteration 14	19	23.85	39	Iteration 40	19	32.96	108	Iteration 66	19	48.58	120	Iteration 92	19	21.47	30
Iteration 15	19	21.9	26	Iteration 41	19	21.19	24	Iteration 67	19	22.31	29	Iteration 93	19	21.4	26
Iteration 16	19	22.08	34	Iteration 42	19	21.16	25	Iteration 68	19	21.24	24	Iteration 94	19	21.21	24
Iteration 17	19	21.28	25	Iteration 43	19	21.15	23	Iteration 69	19	21.11	24	Iteration 95	19	20.98	23
Iteration 18	19	23	33	Iteration 44	19	21.27	24	Iteration 70	19	21.11	24	Iteration 96	19	21.24	24
Iteration 19	19	21.4	24	Iteration 45	19	21.17	24	Iteration 71	19	21.37	31	Iteration 97	19	21.07	23
Iteration 20	19	21.35	30	Iteration 46	19	21.5	25	Iteration 72	19	21.27	26	Iteration 98	19	21.3	24
Iteration 21	19	21.24	24	Iteration 47	19	21.21	23	Iteration 73	19	21.07	24	Iteration 99	19	21.5	26
Iteration 22	19	21.41	26	Iteration 48	19	21.19	26	Iteration 74	19	21.38	28				
Iteration 23	19	21.25	26	Iteration 49	19	21.37	24	Iteration 75	20	21.34	24				
Iteration 24	19	21.4	24	Iteration 50	19	21.2	23	Iteration 76	19	21.08	23				
Iteration 25	19	21.22	24	Iteration 51	19	22.3	30	Iteration 77	19	21.22	24				

Figure 5.22 MQTT Packet Sending Latency of 100 Repeat 100 Client 2 QoS

5.5 Comparison of HTTP and MQTT

As stated before, in this research there are three parameters observed for simulation: connection duration, message generation latency, and packet sending latency, all with different client numbers, different repeat times, and with values of QoS, if any. The ultimate goal is to see the efficiency of HTTP and MQTT based on these three parameters. The protocols are simulated utilizing IoTIFY, providing 64 various results. The connection duration for the two protocols is increased by increasing repeating times, where the average connection time of 4 probability of repeating message show the same outcomes in both protocols despite the changes in client number and QoS, if any. Table 5.13 presents the average connection time of these protocols.

Table 5.13 Connection Time of HTTP and MQTT

Repeat Message	Average Duration of HTTP/ sec	Average Duration of MQTT/ sec
5	45	45.58
25	256	240.83
50	494.25	496.66
100	994.5	996.66

It was observed that the message generation latency of both protocols increased by increasing client numbers, for the same repeating times of both protocols. According to Table 5.6, one can see the average message generation latency of HTTP, followed by Table 5.7, Table 5.8, Table 5.9, and Table 5.10 presenting the message generation latency of MQTT. The comparisons are made for HTTP message generation latency of 10 clients and 5 repeats, 25 clients and 25 repeats, 50 clients and 50 repeats, and 100 clients and 100 repeats. The MQTT message generation latency calculated by the same groups of client numbers and repeat times with three QoS values.

HTTP depends only on the reliability mechanism of TCP to ensure delivering messages, while MQTT has three QoS values to ensure delivering messages that extend the duration. The latency of MQTT with QoS 1 is generally higher than the latency of MQTT with QoS 2. The reason of having such a conclusion is because of the behavior of QoS 1(at least once), where a message can be generated and sent more than once leads to an increase in the latency comparing to QoS 2 (exactly once). When we compare the MQTT and HTTP protocols, it can be noted that the message generation latency of the two protocols is found to be roughly in the same range regardless of the repeat times or QoS as shown in Table 5.14.

Table 5.14 Average Message Generation Latency of MQTT and HTTP

Client Number	QoS	Average Message Generation Latency of MQTT of four repeat groups	Average Message Generation Latency of HTTP
10	0	1639.575	1671.7
	1	1915.7	
	2	1699.325	
25	0	2051.04	2594.2
	1	2111.33	
	2	1762.02	
50	0	3019.73	2934.83
	1	2979.755	
	2	2914.695	
100	0	3943.4675	3972.035
	1	4203.205	
	2	4150.31	

Apart from this, the packet sending latency of both protocols is seen to increase by client numbers. Table 5.11 shows packet sending latency of HTTP. According to Table 5.12, the QoS 1 and 2 in MQTT shows an increase in latency compared to QoS 0. The results in the two Tables are compared and listed in Table 5.15.

The packet sending latency in almost all scenarios of HTTP is higher than MQTT for the same client group and the same repeat times with different QoS values. From these outcomes, it can be concluded that MQTT performs better than HTTP in terms of packet sending latency.

Table 5.15 Average Packet Sending Latency of MQTT and HTTP

Client number	QoS	Average packet sending latency of MQTT	Average packet sending latency of HTTP
10	0	1.075	888.855
	1	300.175	
	2	350.925	
25	0	1.03	910.29
	1	783.71	
	2	849.945	
50	0	1.865	1231.515
	1	950.22	
	2	1069.995	
100	0	1.33	1956.9775
	1	2271.7225	
	2	2445.8	

The HTTP and MQTT results are approximately the same from the perspective of connection duration and message generation latency. Nonetheless, HTTP shows higher packet sending latency in almost all scenarios when compared to MQTT. Therefore, deciding which protocol is better for IoT will depend on the usage environment, and even though MQTT is designed specifically for IoT, the HTTP options in different environments and platforms give it an advantage.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

The aim of this work was to compare IoT application layer protocols (MQTT, CoAP, HTTP, and XMPP) theoretically, and to practically evaluate the performance of two of them (HTTP and MQTT), in term of connection duration, message generation latency, and packet sending latency. First, the four IoT protocols are compared through the differences presented in their communication patterns, security, header and message size, their fragmentation and implementation, and organizations. Taking into consideration the differences among these protocols and their major impact on IoT applications, it is not possible to select an IoT application layer protocol that can meet all the IoT requirements. We conclude that an IoT application layer protocol should be selected according to specific requirements of an IoT application. In this scope, the recommendations that we deduce are as follows:

- For minimum power consumption, the MQTT and CoAP protocols are proper protocols.
- For low capacity IoT hardware, again the MQTT and CoAP protocols seem correct solutions for application layer.
- CoAP is the best choice if the IoT application requires minimum bandwidth on the connected network.
- If QoS is a requirement for the IoT application, then MQTT can be preferred as the best choice.
- If reliability and interoperability are the primary requirement of an IoT application, HTTP seems as the best protocol for such an application.

Based on the results of the simulations, a comparative analysis was done between MQTT and HTTP protocols and results were documented. The performance was evaluated based on client numbers, repeat times, and QoS to figure out the effects of these parameters on protocols. The conclusions that we reach from these results are as follows:

- In terms of connection duration, both protocols are in the same range with identical results.
- In terms of message generation latency, MQTT and HTTP latencies are approximately in the same range; therefore, this parameter can not be used to decide which protocol is better.
- Whereas packet sending latency of HTTP protocol is higher than in MQTT, which leads to performing MQTT better than HTTP in terms of packet sending latency.
- As a general conclusion of simulations, MQTT seems better than HTTP2 in terms of network performances.

6.2 Limitations

During this study, two important limitations that we encountered are as follows:

- Although there are various high quality network simulators that can be used for simulation of standard network protocols, this is not the case for the IoT network simulations yet. It is difficult to find find a high quality IoT simulator. Another problem is that high prices are requested for certain simulators, which is difficult for a student to afford such a price.
- For simulations, IoT devices are simulated on a laptop PC connected to the Internet using IoTIFY simulator and the cloud side is simulated by a server on the Internet. Since the Internet traffic is not controllable by us, some test results are not consistent because of Internet traffic. We have repeated simulations many times to reach a conclusion.

6.3 Future Work

The application layer protocols of IoT have been an attractive topic in computer network technologies, impliying that much research is going on and many issues need to be solved. Due to the difficulty of simulating all IoT network layer protocols, the present attempt was limited to application protocols. However, there are many issues that could be subject to further studies, where the simulator environment could be improved by:

- Supporting more application layer protocols;
- Supporting protocols of other layers, namely datalink;
- Developing more complex scenarios with added clients; and
- Including more network specifications such as packet loss, transmission rate, and throughput.

GCRIS

REFERENCES

[1] L. Atzori, A. Iera and G. Morabito, "The Internet of Things: A survey," *Computer Networks,* vol. 54, no. 15, pp. 2787-2805, 2010.

[2] M. Malik, I. McAteer, P. Hannay, S. Firdous and Z. Baig, "XMPP architecture and security challenges in an IoT ecosystem," in *proceedings of the 16th Australian Information Security Management Conference*, Perth, Australia, 2018.

[3] P. P. Ray, "A survey on Internet of Things architectures," *Journal of King Saud University - Computer and Information Sciences,* vol. 30, no. 3, pp. 291-319, 2018.

[4] K. Patel and S. Patel, "Internet of Things-IOT: Definition, Characteristics, Architecture, Enabling Technologies, Application & Future Challenges," *International Journal of Engineering Science and Computing,* vol. 6, no. 5, 2016.

[5] W. Bziuk, C. V. Phung, J. Dizdarevi and A. Jukan, "On HTTP Performance in IoT Applications: An Analysis of Latency and Throughput," in *1st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, Opatija, Croatia, 2018.

[6] J. DIZDAREVIĆ, R. CARPIO, A. JUKAN and X. MASIP-BRUIN, "A Survey of Communication Protocols for Internet of Things and Related Challenges of Fog and Cloud Computing Integration," *ACM Journals,* vol. 51, no. 6, 2019.

[7] V. Karagiannis, P. Chatzimisios, F. Vazquez-Gallego and J. Alonso-Zarate, "A Survey on Application Layer Protocols for the Internet of Things," *Transaction on IoT and Cloud Computing,* 2015.

[8] T. Salman and R. Jain, "A Survey of Protocols and Standards for Internet of Things," *Advanced Computing and Communications,,* vol. 1, no. 1, 2017.

[9] B. H. Çorak, F. Y. Okay, M. Güzel, Ş. Murt and S. Ozdemir, "Comparative Analysis of IoT Communication Protocols," in *International Symposium on Networks, Computers and Communications (ISNCC)*, Rome, Italy, 2018.

[10] Y. Chen and T. Kunz, "Performance Evaluation of IoT Protocols under a Constrained Wireless Access Network," in *International Conference on Selected Topics in Mobile & Wireless Networking (MoWNeT)*, Cairo, Egypt, 2016.

[11] B. Wukkadada, K. Wankhede, R. Nambiar and A. Nair, "Comparison with HTTP and MQTT In Internet of Things (IoT)," in *International Conference on Inventive Research in Computing Applications (ICIRCA)*, Coimbatore, India, 2018.

[12] T. Yokotani and Y. Sasaki, "Comparison with HTTP and MQTT on Required Network Resources for IoT," in *International Conference on Control, Electronics, Renewable Energy and Communications (ICCEREC)*, Bandung, Indonesia, 2016.

[13] M. B. Yassein, M. Q. Shatnawi and D. Al-zoubi, "Application layer protocols for the Internet of Things: A survey," in *International Conference on Engineering & MIS (ICEMIS)*, Agadir, Morocco, 2016.

[14] D. Soni and A. Makwana, "A survey on MQTT: a protocol of internet of things (iot)," in *International Conference on Telecommunication, Power Analysis and Computing Techniques*, India, Chennai, 2017.

[15] A. Banks and R. Gupta, "OASIS MQTT Version 3.1.1 Plus Errata 01," OASIS Standard Incorporating Approved Errata 01, 29 10 2014. [Online]. Available: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html.. [Accessed 0 12 2015].

[16] B. Aziz, "A Formal Model and Analysis of an IoT Protocol," *Ad Hoc Networks,* vol. 36, pp. 49-57, 2016.

[17] N. Naik, "Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP," in *IEEE International Systems Engineering Symposium (ISSE)*, Vienna, Austria, 2017.

[18] N. Naik and P. Jenkins, "Web Protocols and Challenges of Web Latency in the Web of Things," in *Eighth International Conference on Ubiquitous and Future Networks (ICUFN)*, Vienna, Austria, 2016.

[19] M. Belshe, R. Peon and M. Thomson, "Hypertext Transfer Protocol Version 2 (HTTP/2)," *Internet RFCs,* vol. RFC 7540, no. 2070-1721, 2015.

[20] L. NĂSTASE, I. E. SANDU and N. POPESCU, "An Experimental Evaluation of Application Layer Protocols for the Internet of Things," *Studies in Informatics and Control,* pp. 403-412, 2017.

[21] H. d. Saxce, I. Oprescu and Y. Chen, "Is HTTP/2 really faster than HTTP/1.1?," in *IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, Hong Kong, China, 2015.

[22] I. Grigorik and Surma, "Introduction to HTTP/2," in *High Performance Browser Networking*, O'Reilly Media, 2013, p. 571.

[23] K. Fysarakis, I. Askoxylakis, O. Soultatos, I. Papaefstathiou, C. Manifavas and V. Katos, "Which IoT Protocol? Comparing Standardized Approaches over a Common M2M Application," in *IEEE Global Communications Conference (GLOBECOM)*, Washington, DC, USA, 2016.

[24] T. Dimčić, S. Krčo and N. Gligorić, "CoAP (Constrained Application Protocol) implementation in M2M Environmental Monitoring System," *E-society Journal Research and applications,* vol. 3, 2012.

[25] Z. Shelby, K. Hartke and C. Bormann, "The Constrained Application Protocol (CoAP)," *Internet RFCs,* vol. RFC 7252, no. 2070-1721, 2014.

[26] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari and M. Ayyash, "Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications," *IEEE Communications Surveys & Tutorials ,* vol. 17, no. 4, 2015.

[27] S. Kakakhel, T. Westerlund, M. Daneshtalab, Z. Zou, J. Plosila and H. Tenhunen, "A Qualitative Comparison Model for Application Layer IoT Protocols," in *Fourth International Conference on Fog and Mobile Edge Computing (FMEC)*, Rome, Italy, Italy, 2019.

[28] U. Hunkeler, H. L. Truong and A. Stanford-Clark, "MQTT-S — A publish/subscribe protocol for Wireless Sensor Networks," in *3rd International Conference on Communication Systems Software and Middleware and Workshops (COMSWARE '08)*, Bangalore, India, 2008.

[29] X. Wang, X. Zha, W. Ni, R. P. Liu, Y. J. Guo, X. Niu and K. Zheng, "Survey on blockchain for Internet of Things," *Computer Communications,* vol. 136, pp. 10-29, 2019.

[30] M. Singh, R. MA, S. VL and B. P, "Secure MQTT for Internet of Things (IoT)," in *Fifth International Conference on Communication Systems and Network Technologies*, Gwalior, India, 2015.

[31] R. A. Rahman and B. Shah, "Security analysis of IoT protocols: A focus in CoAP," in *3rd MEC International Conference on Big Data and Smart City (ICBDSC)*, Muscat, Oman, 2016.

[32] T. A. Alghamdi, A. Lasebae and M. Aiash, "Security analysis of the constrained application protocol in the Internet of Things," in *Second International Conference on Future Generation Communication Technologies (FGCT 2013)*, London, UK, 2014.

[33] S. Zamfir, T. Balan, I. Iliescu and F. Sandu, "A security analysis on standard IoT protocols," in *International Conference on Applied and Theoretical Electricity (ICATE)*, Craiova, Romania, 2016.

[34] N. D. Caro, W. Colitti, K. Steenhaut, G. Mangino and G. Reali, "Comparison of two lightweight protocols for smartphone-based sensing," in *IEEE 20th Symposium on Communications and Vehicular Technology in the Benelux (SCVT)*, Namur, Belgium, 2013.

[35] L. N□stase, "Security in the Internet of Things: A Survey on Application Layer Protocols," in *21st International Conference on Control Systems and Computer Science (CSCS)*, Bucharest, Romania, 2017.

[36] M. Joshi and B. P. Kaur, "CoAP Protocol for Constrained Networks," *I.J. Wireless and Microwave Technologies,,* pp. 1-10, 2015.