Thesis Dissertation

PERFORMANCE EVALUATION OF MQTT OVER QUIC

Viktoras Milios

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

May 2023

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

Performance Evaluation of MQTT over QUIC

Viktoras Milios

Supervisor
Dr. Andreas Pitsillides

The Diploma Thesis was submitted for partial fulfilment of the requirements for obtaining the degree of Computer Science of the Department of Computer Science of the University of Cyprus

May 2023

Acknowledgements

I would like to express my sincere appreciation to my supervisor, Professor Andreas Pitsillides, for their guidance and introduction to the topic of my thesis.

I am also incredibly grateful to my family and friends for their support and understanding throughout my studies.

Abstract

The Internet of Things (IoT) has seen exponential growth in the previous years as more and more applications benefit from the usage of IoT devices. From home automation systems, smart cities, smart homes to industrial automation and healthcare systems, the IoT ecosystem continues to expand, enabling more and more innovative solutions to everyday challenges.

One of the key challenges in IoT is the need of reliable and efficient communication between devices, since these devices are often resource-constrained, bandwidth limited and many times battery powered. MQTT is a lightweight protocol well suited for IoT applications due to its small code footprint and its publish-subscribe messaging architecture, enabling devices to efficiently transmit and receive data.

However, as IoT deployments grow in scale and complexity, there is a need to address the limitations of the underlying transport protocol used by MQTT. Transmission Control Protocol (TCP), which MQTT relies on, comes with numerous limitations such as Protocol Entrenchment, Implementation Entrenchment, Handshake Delay and Head-of-Line (HOL) Blocking Delay.

QUIC, a relatively new transport layer protocol designed by Google aims to address these issues. QUIC is a transport layer protocol built on top of the User Datagram Protocol (UDP) and it was initially designed to replace TCP in HTTP/3.0. It incorporated features similar to TCP such as congestion control, connection multiplexing, stream prioritization and built-in encryption.

In this thesis, the usage of QUIC as the underlying transport layer protocol of MQTT is studied. After a performance evaluation is performed to compare MQTT over QUIC and MQTT over TCP, the findings show that MQTT over QUIC outperformed MQTT over TCP+TLS in terms of connection establishment delay, traffic delay and overall CPU utilization. Furthermore, QUIC’s support of connection migration allows seamless roaming of mobile IoT devices.

Table of Contents

Chapter 1

Introduction

1.1 Motivation

The Internet of Things (IoT) has seen a rapid growth in the previous years, as more and more devices becoming connected. Automation systems, smart cities, smart homes, health monitoring, smart grids and so on benefit from the usage of IoT devices such as sensors and actuators. IoT devices are used to sense, actuate, process and communicate via the Internet, either directly through a cellular network or through a gateway, most commonly using radio technologies. When designing protocols for the IoT several factors should be taken into consideration such as device and traffic characteristics, since IoT devices are often constrained in terms of power, memory and bandwidth. In the meantime, the connected devices communications protocol must fulfill multiple design requirements such as resource control, energy awareness (battery powered devices), quality of service (QoS), interoperability, interference management, security and privacy, low latency communication, small code footprint, high throughput and scalability [4, 5]. These requirements led to the design and implementation of several communication protocols for IoT devices.

One of most well known and widely used communication protocol for the IoT is MQTT. MQTT is widely used due to its small code footprint, low overhead, easy integration, security, good performance and it’s publish-subscribe architecture which is ideal for most IoT applications, thanks to its ease of deployment and increased scalability & reliability. However, it is traditionally used over TCP (Transport Control Protocol), which comes with numerous limitations such as Protocol Entrenchment, Implementation Entrenchment, Handshake Delay and Head-of-Line Blocking Delay (HOL)[3]. Replacing TCP as the underlying transport protocol can allow MQTT to overcome the limitations imposed by TCP.

The main motivation of this thesis is to provide an in-depth analysis of a promising alternative replacement for the transport protocol used by MQTT and conduct several benchmarks to compare it to the current state of the protocol.

1.2 Objective and Contribution

To address the issues and limitations of TCP, a newly established protocol developed by Google, QUIC, can be used as the underlying transport layer protocol of MQTT. QUIC is a transport layer protocol that is designed to replace TCP in HTTP/3 and uses the User Datagram Protocol (UDP). It incorporates on top of it features such as connection multiplexing, stream prioritization, and built-in encryption which can potentially improve the performance and security of MQTT.

The purpose of this thesis is to analyze and compare the performance of MQTT over QUIC in order to examine the potential advantages of adopting QUIC as a transport protocol for MQTT. To evaluate the performance of MQTT over QUIC several benchmarks have been established to collect metrics like latency, throughput, packet loss, CPU usage, memory usage, network utilization, handshake delay etc.

1.3 Document Organization

This thesis consists of 7 Chapters. Chapter 2 will go through some prerequisite knowledge of protocols and concepts, as well as a literature survey. Precisely, there is an introduction to the Internet of Things and the Publish-Subscribe which is commonly used in the IoT. Following that, there is a review of UDP and TCP, followed by a look into TLS and a literature survey. Chapter 3 reviews technologies used in the test environment. In Chapter 4, an in-depth analysis of QUIC is presented. In Chapter 5, we will go through a detailed overview of MQTT. The experimental setup, metrics examined and results are discussed in Chapter 6. A comprehensive overview, potential directions for future work and key conclusions are provided in the last chapter of this thesis.

Chapter 2

Background

2.1 Internet of Things

The Internet of Things (IoT) is a term used to describe the growing network of physical objects, from appliances and vehicles to buildings and entire cities, that are connected to the internet and can collect, exchange, and act on data. The term was introduced by the computer scientist Kevin Ashton in 1999, which at the time described RFID-tagged objects. The following years, as more and more connected devices came to market, the public interest in IoT began to take off. Now, the IoT is often seen as the next major thing of the internet, enabling a wide range of new applications and services and it is projected that the total number of IoT connected devices is expected to nearly double by 2030[2].

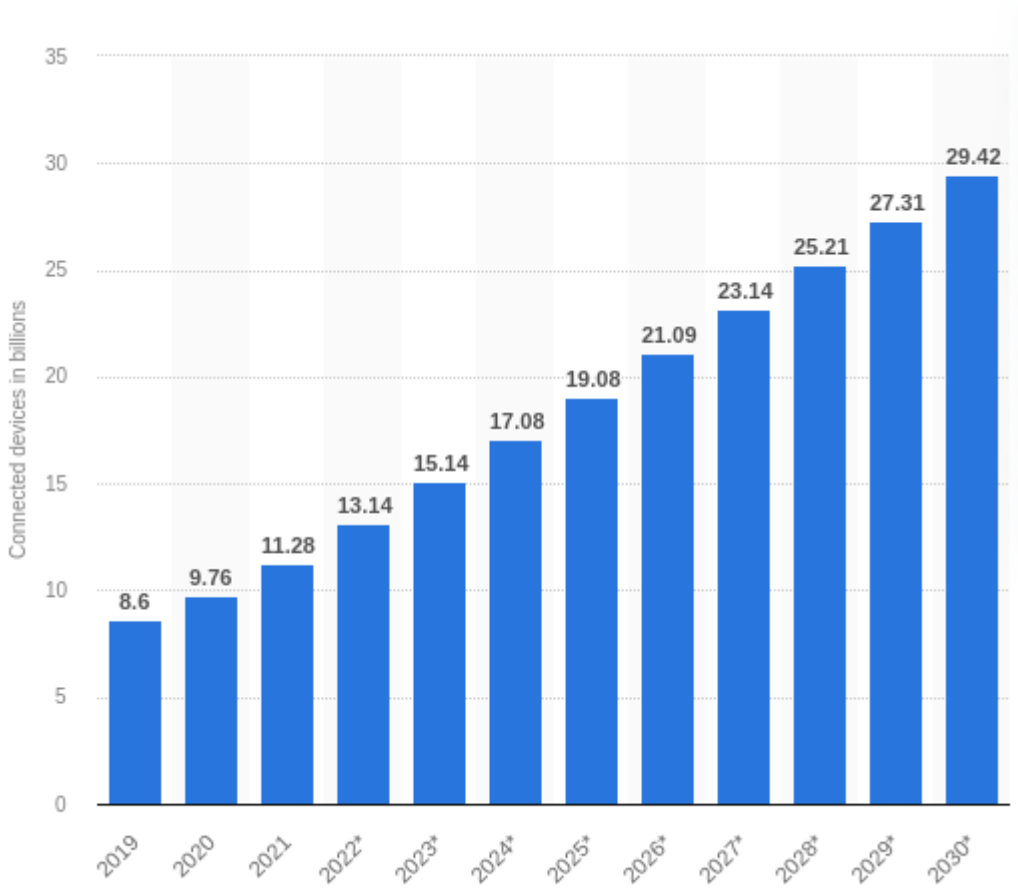


Figure 1: Statista. (2022) Number of IoT connected devices worldwide 2019-2021, with forecasts to 2030 (statista.com)

An Internet of Things device generally refers to an object or “thing” embedded with electronics, sensors, actuators and so on. Some of these devices include smart home devices (e.g. smart thermostats, smart lights), wearables (e.g. smartwatches, fitness trackers), smart appliances (e.g. smart refrigerators, smart washing machines), industrial sensors, connected vehicles and many more. These devices are mainly used to sense, actuate, process and exchange information with other “things” via the Internet, either directly through a cellular network or through a gateway, most commonly using radio technologies. Therefore, these devices can be considered as part of Low-Power and Lossy Networks (LLNs) [32], which are characterized by low power consumption, limited processing capabilities, and lossy communication links[31].

Due to the restrictions LLNs impose, traditional implementations of RESTful architectures like HTTP client/server model are unsuitable and an adaptation is needed. While LLNs must carefully consider several of these features because the services provided by lower layers are significantly more constrained, REST architectures make assumptions on effective reliable

transport and are not strictly constrained by payload size because expensive fragmentation is dealt with at lower layers. For instance, 6LoWPAN allows for the pricey IPv6 packet fragmentation into 127-byte packets, but abusing this feature renders the network unusable. Limiting packet extension is thus a requirement for application layer.

Numerous application layer protocols have been developed, each with their own advantages and disadvantages based on each application requirements. Table 1 provides a brief comparison of the most common protocols.

Table 1: Comparison of IoT application protocols

	CoAP	**MQTT**	**XMPP**	**AMQP**	**DDS**	**REST**
Transport	UDP	TCP/ QUIC	TCP	TCP	TCP/UDP	TCP
Publisher/subscriber	Yes	Yes	Yes	Yes	Yes	No
Request/response	Yes	No	Yes	No	No	Yes
Security	DTLS	SSL	SSL	SSL	SSL/DTLS	SSL
QoS	Yes	Yes	No	Yes	Yes	No
Low power and lossy network	Exc.	Fair	Fair	Fair	Poor	Fair
Dynamic discovery	Yes	No	No	No	Yes	No
Binary encoding	Yes	Yes	Yes	Yes	Yes	No
Real-time	No	No	No	No	Yes	No
Open source	Yes	Yes	Yes	Yes	Yes	No
Architecture style	P2P	Broker	P2P	P2P Broker	Data Space	P2P
Organization	IETF	OASIS	IETF	OASIS	OMG	IETF

2.2 Publish-Subscribe Communication Pattern

Publish-Subscribe communication systems describe a communication pattern where the *publisher* is the service responsible for publishing the message (e.g. events) and the *subscriber* is the one that receives them. These two services are operating independently of each other, meaning that both of them can act without being aware of the precense of the other. A central service, usually called a broker (or server), is responsible for receiving the messages sent by the publisher and forwards them to subscribers who listen to a specific topic. However, in some cases, it is desireable to use publish-subscribe for peer-to-peer communication but it is unfeasible or undesirable to include a seperate node to operate as a

broker. An example of this is the brokerless publish-subscribe implementation is CoAP resource/observe operation[6].

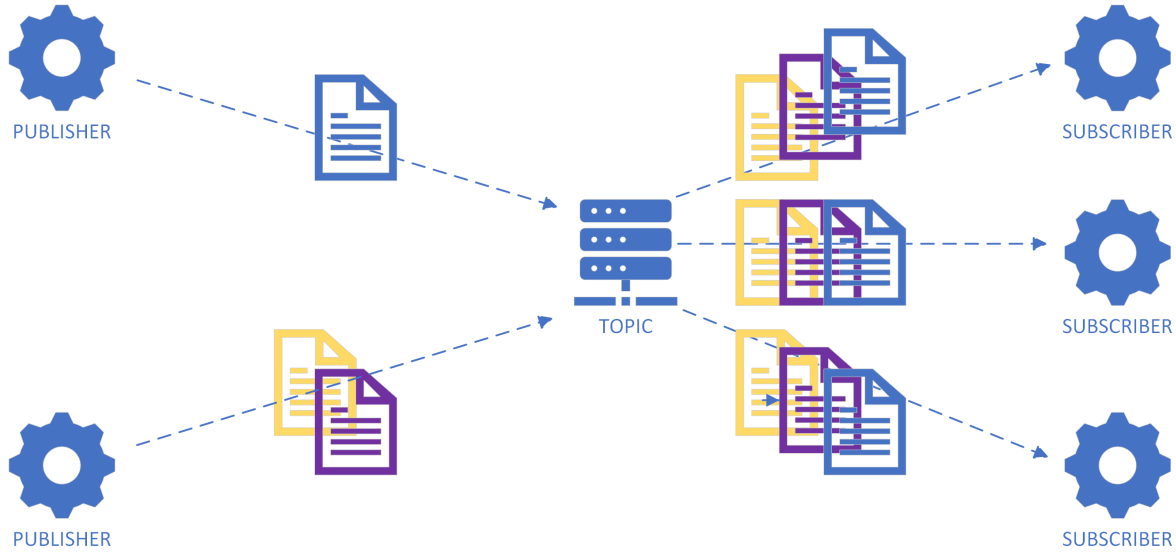


Figure 2: Publish-Subscribe Messaging

The broker node operates as a message queuing system and besides its store and forward operation, implements additional functionality that varies based on it's implementation requirements. Some of these optional services and operations of a queuing system may be scalability (i.e. multiple queues working in parallel), error-tolerance (i.e. queues hold a message copy), persistent storage (i.e. store messages on the disk), asynchronous or synchronous communication (i.e. messages enter the queues at any time or waiting for the previous message to be acknowledged before processing next message)[7].

The publish-subscribe pattern makes an ideal choice for the development of IoT communication protocols, as it shares many of the requirements of publish-subscribe systems as discussed above.

2.3 Transport Layer

The transport layer is a crucial component of the TCP/IP stack that is responsible for exchanging segments between two communicationg applications. It operates above the network layer (IP) and protocols of this layer are used to provide error correction, flow control, congestion control, multiplexing/demultiplexing and end-to-end connection between applications.

2.3.1 User Datagram Protocol (UDP)

The User Datagram Protocol is a connectionless transport layer protocol. It offers a simple and unreliable service and does not establish a connection before sending any data. It is described as a “best-effort” protocol as it operates in a fire-and-forget manner, meaning that after it sends a packet, it does not provide any guarantees about the transmission. UDP is widely used in applications that prioritize low latency and high throughput, where occcasional out of order delivery or lost packets are acceptable.

One of the distinguishing features of UDP is its stateless nature, where each datagram is handled as an independent entity without any guarantee of succesful delivery or packet ordering. Due to its low latency and overhead, UDP is well-suited for real-time applications like VoIP, multimedia streaming and online gaming.

Another notable aspect of UDP is its lightweight header, which only takes up 8 bytes. Due to its improved overhead and bandwidth efficiency, UDP is suitable for use in resource-constrained environments, such as IoT devices, where reducing overhead is essential for energy conservation and reducing network utilization.

However, the lack of reliability and lack of flow control, leads applications using UDP to, based on their needs, manage error detection, correction and retransmission at the application layer. Additionally, due to its connectionless nature, UDP also lacks congestion control mechanisms, leading to network congestion and packet loss under high network utilization. These reasons, led some application layer communication protocols to be built on top of TCP even though UDP is more appealing for IoT devices.

2.3.2 Transmission Control Protocol (TCP)

The Transmission Control Protocol is a connection-oriented transport layer protocol which unlike UDP, comes with a reliable and stream-oriented end-to-end communication that provides error-checked, ordered and reliable delivery of data. In addition to that, TCP implements congestion control mechanisms and relies on TLS for encryption. TCP is widely used in applications that require reliable data stream such as HTTP, SMTP, FTP and etc.

TCP follows a client-server model, where the client initiates a connection request to the server in order to a stablish a reliable communication channel. Once the connection is established it is guaranteed that data will be delivered without loss or duplications to both ends of the connection.

However, all these features that TCP provides, introduce numerous limitations. The implementation of its mechanisms require acknowledgements, retransmissions and error detection which increase latency. Furthemore, the 3-way handshake needed to initiate a connection introduces startup latency (1 RTT), where this becomes 2 or 3 RTTs when Transport Layer Security (TLS) is needed to encrypt the conversation.

Using TCP to ensure reliable and secure communication exerts high overhead and deems it unsuitable for use for IoT applications. As the authors in [8-10] point out, some of the drawbacks of using TCP with IoT are as follows:

- Devices frequently switch between sleep and awake mode in order to save power. This causes the TCP connection to be short-lived.
- Since communication in the IoT is usually small in data size, high overhead of connection is unsuitable.
- TCP handshake introduces high latency which is undesirable, especially for time-sensitive applications.
- Considering the inherent LLN nature of the IoT, TCP may frequently cause HOL blocking.
- Half-open connections that can be exploited for SYN flooding attack[11].

2.4.1 Transport Layer Security (TLS)

The Transport Layer Security is a security protocol created to enable communications over the Internet privacy and data security. TLS is the most common connection-based and stateful client-server security protocol and is layered on tome of some reliable transport protocol. It’s primary use case is to encrypt the communication between a client and a server, such as web browsers loading a website (HTTPS). It can also be used to provide encryption to other services such as email, messaging and VoIP.

TLS is often used interchangeably with the term SSL which stands for Secure Sockets Layer. The reason for this is that TLS evolved from SSL, which is a previous encryption protocol

developed by Netscape its name was changed in order to indicate that it is no longer associated with Netscape.

Apart from the encryption, TLS also ensures the integrity of the messages to detect any tampering or modification of the data during transmission. In addition to that it includes mechanisms for authentication, to verify the indentity of the server and, optionally, the client, to ensure that communication is initiated between the intended hosts.

To start a communication that uses TLS, a TLS handshake must be performed in order for the hosts to acknowledge and verify each other, establish the cryptographic method they will use and exchange session keys. A typical TLS over TCP handshake is seen below.

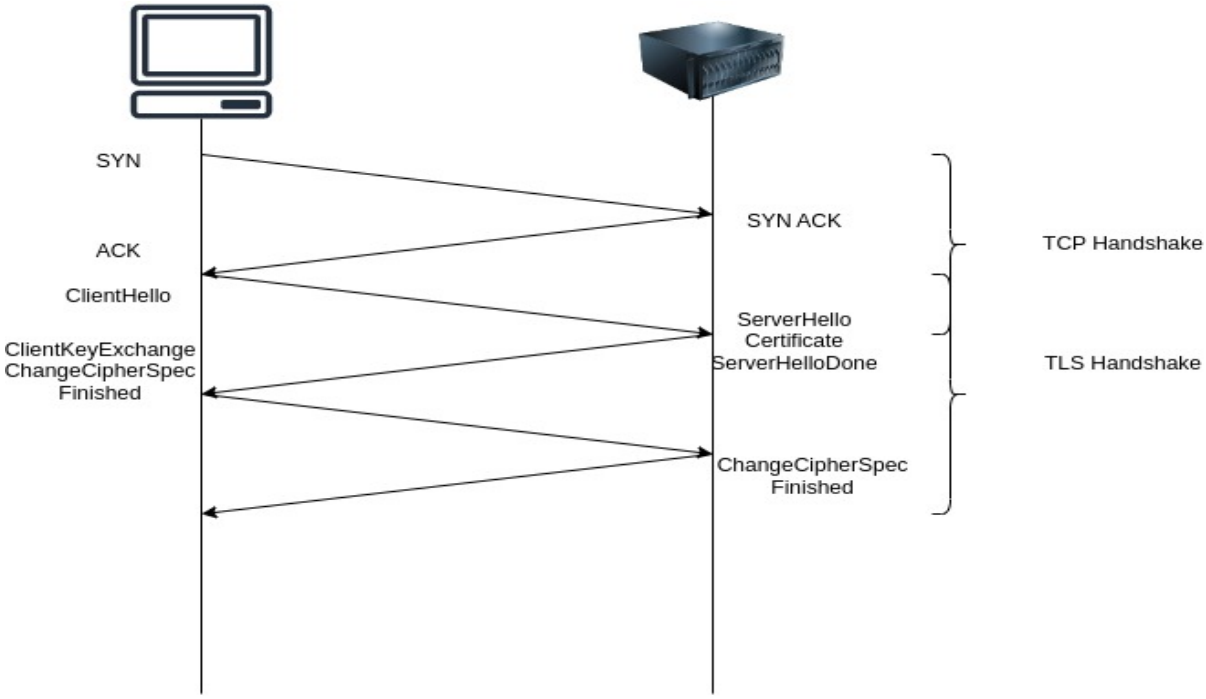


Figure 3: TLS over TCP handshake

2.4.2 Datagram Transport Layer Security (DTLS)

DTLS is an adaptation of TLS for use over unreliable datagram transport. As discussed in [13], using DTLS instead of TLS for the IoT gives three high-level design decisions:

- Implementation of a standards based design
- Focus on application layer end-to-end security
- Support for unreliable transport protocols

2.5 Related Work

Authors in [15] have noticed that the use of TCP comes with numerous limitations and is not suitable for IoT applications due to the size of its packet header, sensitive congestion control algorithms and its slow start strategy. They evaluated several UDP based transport protocols to find the most suitable protocol for the three IoT applications they have tested. These protocols include RUBDP, UDTP and PA-UDP. For the evaluation they compared metrics such as throughput, delay, CPU utilization and energy usage on wireless networks. Based on their findings they concluded that PA-UDP has the best throughput performance but CPU usage on the receiver side during the start and end of the transmission. UDT protocol as the most efficient in terms of cpu usage but has lower throughput. RBUDP performed consistently in wireless networks but the CPU usage as very high at the sender side. They conclude by acknowledging that the UDT protocol is the most suitable of the three for IoT applications due to its consistent performance and CPU efficiency.

The authors in [14], have used the Go programming language to integrate AMQP with QUIC in order to reduce latency and reduce energy usage for IoT devices. They point out that one of the main sources of delay in AMQP is the use of TCP as the underlying transport layer protocol. They used dockers to containerize the AMQP broker, sender and receiver implementations. Their experimental results indicate that RTT as 71% higher using QUIC, but the Startup Latency and Total Communication Time is improved by 62% and 22% respectively. The proposed scheme, AMQP over QUIC, transported 3.5 times more data than the existing scheme, AMQP over TCP, but QUIC showed a throughput 7 times higher, that in turn shorten the communication time and reduces energy by 31%. Additionally, their results outperformed consistently the existing scheme in terms of packet loss, delay and channel bandwidth with the exception of low channel bandwidth scenarios.

The author in [16] analyzed the use of CoAP over QUIC, instead of the standardized TCP and UDP implementations. His findings show that QUIC is less lossy than TCP and UDP under the same conditions, given that the jitter buffer size and packet transmission period are within certain limits. The experimental method he used, estimates an application message loss probability for UDP, TCP and QUIC which is 1.67%, 65.06% and 79.42% respectively. TCP and QUIC likely appears to be less accurate due to the fact that RTT is assumed to be constant over the length of the experiment. The experiment also confirms that QUIC provides an application message loss of 5.33% and 30.76% on average, when compared to UDP and

TCP respectively. In conclusion, UDP and TCP appears to be superseded by QUIC in the context of CoAP IoT sessions.

The authors in [23], have developed a common middleware for MQTT and CoAP, to provide a common programming interface supported by both. The middleware is designed to be extensible to support future protocols. Using the common middleware they designed and implemented, they proceed to conduct several experiments to measure the performance of the two protocols. The evaluated metrics include end-to-end delay and bandwidth consumption under simulated packet loss environments using WANEM. The experimental results showed that CoAP messages have higher delay than MQTT messages at lower packet loss rates and lower delay than MQTT messages at higher packet loss rates. In addition, they point out that if the message size is small and the packet loss is equal or less than 25%, CoAP generates lower additional traffic than MQTT to ensure message reliability.

The authors in [9], pointed out the several limitation that TCP/TLS and UDP/DTLS impose for IoT applications proposes a new transport protocol for MQTT, QUIC. . High connection overhead, latency and connection migration are some of the limitations of TCP/TLS and UDP/DTLS. The paper begins by discussing the challenges of using MQTT over the Internet and the benefits of using QUIC as its underlying transport protocol. Then they show implementation details of their MQTT over QUIC implementation, before the proceed to test it over wired, wireless and long-distance testbeds. Theirs findings show that MQTT over QUIC significantly lowers the connection overhead based on the number of exchanged packets up to 56%. They also show that QUIC eliminates TCP half open connections and reduces the cpu usage by 83% and memory usage by 50%, from the broker’s point of view. Latency is also reduced up to 55%, due to removing HOL blocking. Lastly, thanks to connection migration feature of QUIC, during connection migration the throughput of MQTT over QUIC is significantly lower compared to MQTT over TCP.

Chapter 3

Technologies

3.1 Raspberry Pi

The Raspberry Pi[24] is a series of single-board computers designed by the UK-based charity, Raspberry Pi Foundation, with a mission to promote computing education and access to technology. First released in 2012, the Raspberry Pi has undergone multiple iterations and variations, with the latest model featuring a 64-bit ARM-based quad-core CPU running at 1.8GHz (which can also be overclocked) and up to 8GB RAM. It has 4 USB ports, 40 GPIO pins, 2x micro-HDMI ports, a DSI display ports, CSI camera port, a gigabit ethernet port and an IEEE 802.11ac wireless card which supports bluetooth 5.0 and Wi-Fi. Additionally, it provides a Micro-SD card slot for loading the operating system and data storage. The Raspberry Pi supports different operating systems such as Raspbian, Ubuntu, Windows 10 etc. Notably, the price point for Raspberry Pi has always remained under $100, with the most affordable model, the Pi Zero, priced at just $5.

The Raspberry Pi has gained worldwide popularity for its affordability and versatility, with individuals and organizations utilizing it for a range of applications, including learning programming, building hardware projects, home automation, and industrial applications. Additionally, the supports the running of Linux and GPIO (general purpose input/output) pins, along with its low cost and size, makes it possible and ideal for use in the IoT.

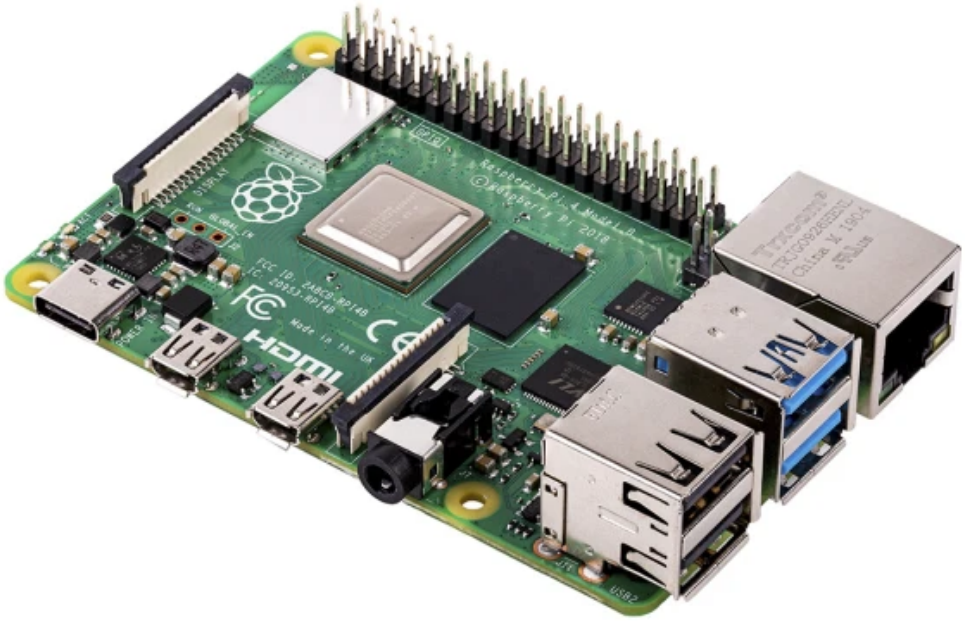


Figure 4: Raspberry Pi 4B

3.2 Docker and Containers

Docker[26] is a software tool that provides a simple and efficient method for developers, system administrators, and others that use OS-level virtualization to to deploy applications in a sandbox, named containers. The primary advantage of Docker is that it enables users to bundle an application and its dependencies into a standardized software unit for development. In contrast to virtual machines, containers are more resource-efficient, resulting in reduced overhead and improved utilization of system resources.

Docker makes use of resources in the Linux Kernel such as kernel namespaces and cgroups, to allow the usage of multiple containers to operate concurrently on the same OS. This allows containers to avoid the need for additional computer resources that would require the use of a Virtual Machine.

Containers provide a convenient way to package applications independently of their execution environment. This enables easy and consistent deployment across various

environments, such as public clouds, private data centers, or personal computers, in a “plug and play” manner. By separating application logic from the execution environment, developers can create predictable and isolated environments that are easy to manage.
Docker containers are created from image file called docker file. The Docker image contains all the necessary executables, code, libraries and configuration needed to run a container without requiring any changes.

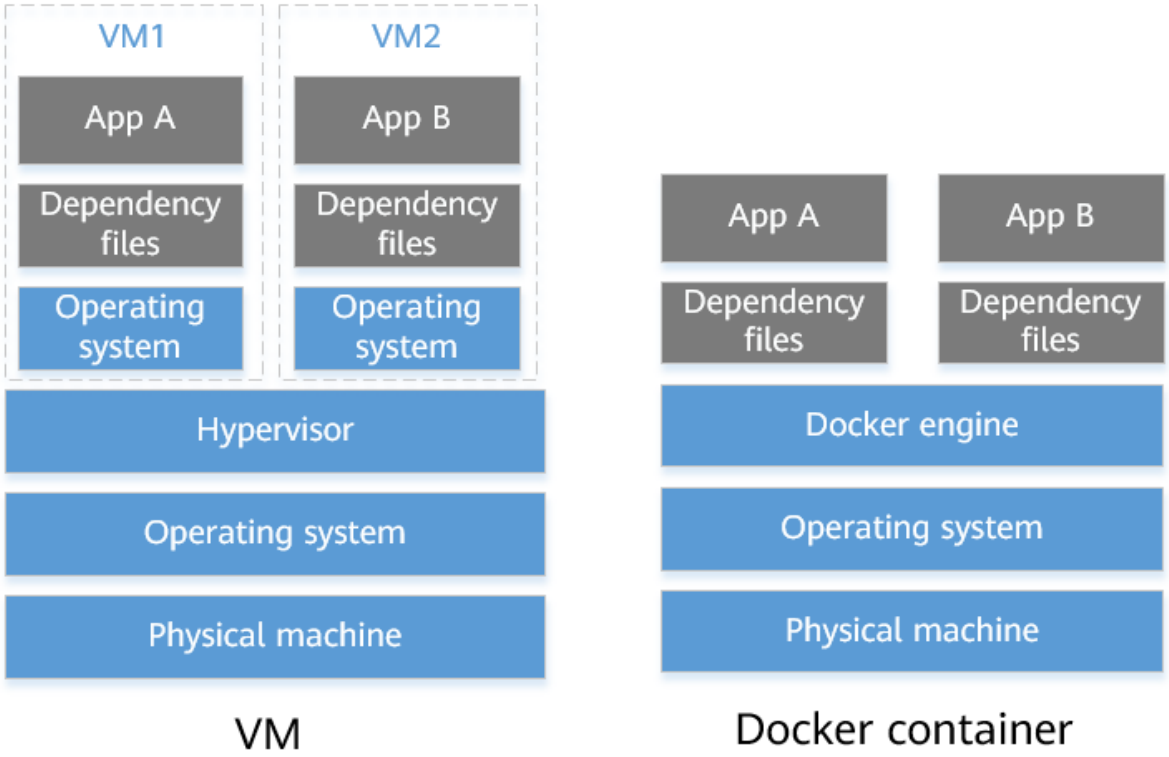


Figure 5: VM compared to Docker Container[26]

3.3 Linux Traffic Control

The Linux Traffic Control (TC) [27] subsystem enables the management of network traffic by providing functionalities such as policing, classification, shaping, and scheduling. This is achieved through the use of queuing disciplines (qdiscs), which form a crucial component of the TC architecture. During classification, packet content can be altered through the application of filters and actions. The scheduling mechanism involves organizing packets before they enter or exit queues, with the FIFO scheduler being the most commonly used. qdisc operations can be carried out temporarily using the tc utility or permanently using NetworkManager.

TC can be used to simulate different network conditions and test how applications and services perform under different scenarios. For example, the queuing disciplines in TC allows

us to manipulate traffic and simulate different network conditions, such as packet loss, latency, jitter and so on, to simulate a lossy network.

3.4 Wireshark

Wireshark is a free and open-source software used for analyzing and profiling network traffic. It can be referred to as a network analyzer, sniffer, or network protocol analyzer. Wireshark allows network administrators to examine network traffic at various levels, from connection-level information to individual packet bits. Packet capture provides network administrators with information about individual packets, such as source, destination, protocol type, header data, and transmit time. This information can be useful for evaluating security events and troubleshooting network security device issues.

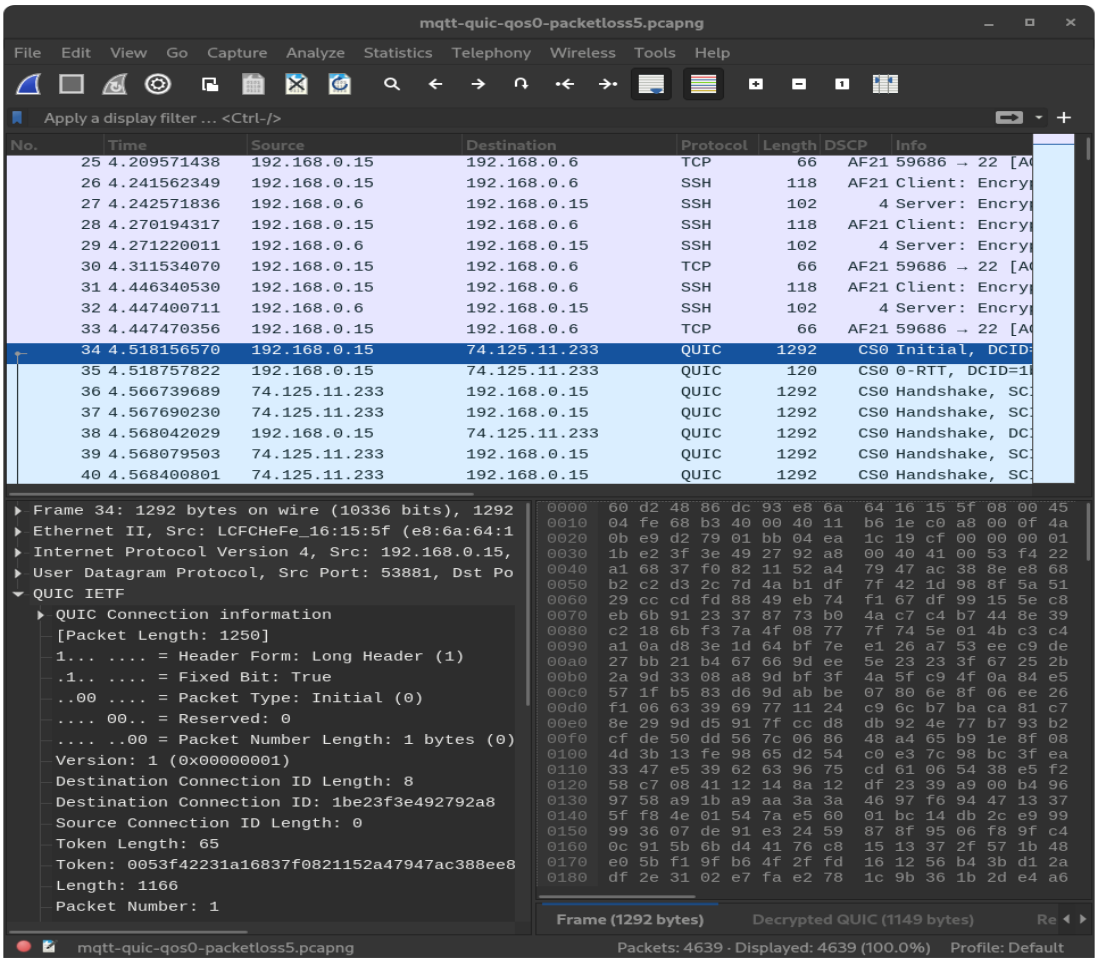


Figure 6: Viewing packet capture in WIreshark

As shown in figure 5, Wireshark typically displays information in three panels. The top panel lists frames individually, with key data on a single line. The bottom-left panel explains any single frame selected in the top panel, showing packet details that belong to the data link layer, network layer, transport layer, or application layer. Finally, Wireshark's bottom-right

panel displays the raw frame with a hexadecimal rendition on the left and the corresponding ASCII values on the right.

3.5 Erlang

Erlang [28] is a programming language developed by Joe Armstrong, Robert Virding, and Mike Williams in 1986 to provide soft real-time, fault-tolerance, distribution and hot code-swapping in Telecom applications. It is a functional programming language that enables the creation of distributed systems through the use of pure functions and immutable state. It comes with a library of tooles called OTP, designed to help build reliable systems. Erlang's base unit of abstraction is a process, and a normal Erlang application is built out of hundreds of small Erlang processes.

The authors in [30] through an experimental study, conclude that Erlang programming language for the development of IoT applications mainly due to it's low latency and low memory usage, in addition to its built-in support for concurrency, distribution and fault tolerance.

Chapter 4

QUIC

4.1 Introduction

QUIC is a relatively new connection-oriented transport protocol originally designed by Google[17] in 2013, and standarized by Internet Engineering Task Force (IETF)[18]. The design of QUIC originally started to replace TCP+TLS+HTTP/2.0 in order to improve user experience, having page load times in mind. It's main objective is to provide the reliability of TCP, while aiming to overcome it's limitations by providing a more efficient and secure protocol, utilizing UDP. In this section, QUIC's main mechanisms are discussed below, as presented in [3] and [22].

4.2 Connection Establishment

QUIC minimizes the number of RTT's needed to setup a secure connection by combining the cryptographic and transport handshake. The cryptographic handshake is done on a dedicated reliable stream. If the handshake is successful, information regarding the origin is cached by the client, allowing the client to establish an encrypted connection without an additional RTT after the client handshake, for every subsequent connection to the same origin. Explained below, are QUIC's cryptographic handshake and the 0-RTT connection setup.

- **Initial Handshake:** Since the client has no information about the server, before the handshake is done, an incomplete client hello (CHLO) message is sent to the server in order to a reject (REJ) message as a reply. The REJ message contains: (i) a server configuation that contains the server long-term Diffie-Hellman public key, (ii) a certificate chain that authenticates the server, (iii) signature of the server configuration and (iv) a source-address token, an encrypted block that consists of the client's public IP and server's timestamp. This token is used by the client for subsequent connections as proof of ownership of their IP. In other words, its a security mechanism to guard against IP spoofing. After the client receives the server configuration, the configuration is authenticated by verifying the received certificate chain and signature. Following that, the client sends a complete CHLO message that includes it's ephemeral Diffie-Hellman public key.
- **Final (and repeat) handshake:** At this point, the client can start sending encrypted application data to the server since it can calculate the shared value from the Diffie-Hellman public key sent by the server in the initial handshake and it's own ephemeral Diffie-Hellman private key, generating its initial keys. If the client starts sending data encrypted with its initial keys without waiting for a server reply, it can achieve 0-RTT data latency. On a successful handshake, server replies with server hello (SHLO) message. As soon as the server sends the SHLO message it switches to the forward-secure keys for encryption. This is also the case for the client, after SHLO is received. If at any time, the source address token or the server config expires, the server changed certificates and cause handshake failure, even if the client sends a complete CHLO. The server handles these cases, the same as if it received an incomplete CHLO and replies with a REJ message and the handshake continues.

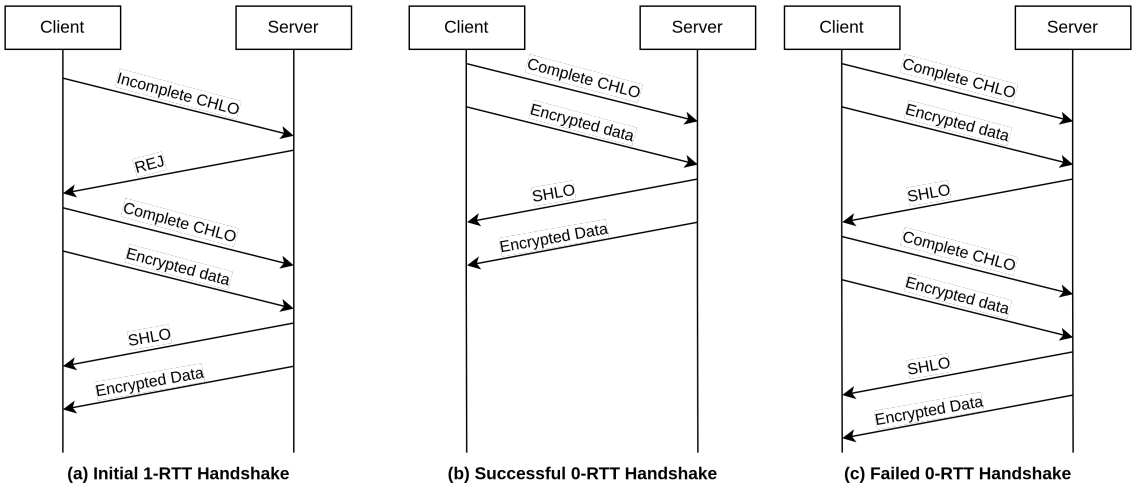


Figure 7: QUIC handshake under difference scenarios

4.3 Stream Multiplexing

TCP's sequential delivery causes a performance-degrading phenomenon to occur, where queued packets are held up in the queue because the first packet in line is taking too long to be processed and leave the queue. This problem is typically refered to as Head-of-Line (HOL) blocking, because, as the name suggests, the head of the queue is blocking the traffic. To overcome this, QUIC allows numerous streams over a single connectin.

A stream is an bidirectional channel of a bytestream, that is used to send data from the client to the server and vice versa. Each stream is assigned a stream id, to which the value is odd if it is initiated by the client, and even otherwise in order to avoid collisions. The usage of streams, ensures that a lost packet does not affect streams that their data was not included in the packet, allowing applications to continue receiving subsequent data from other streams.

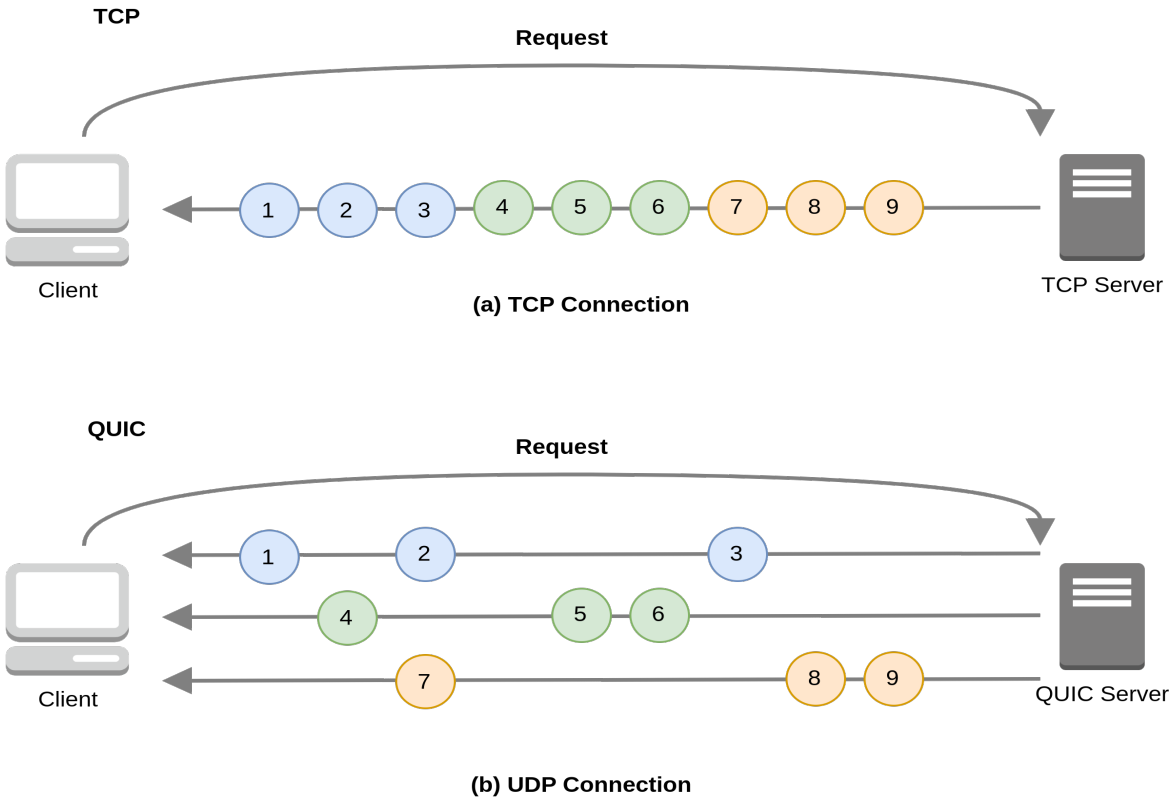


Figure 8: (a) shows TCP in-order deliver and (b) shows QUIC stream multiplexing

4.4 Security

QUIC packets are almost fully encrypted and authenticated except for some early handshake and reset packets. Some parts of the header remain unecrypted since they are necessary for routing and decryption. QUIC relies TLS 1.3[20] for all of its security aspects.

4.5 Connection Migration

QUIC allows connections to persist even when endpoint addresses (IP address and port) change, for example, due to NAT rebinding or an endpoint joining another network. This is achieved with the introduction of 64-bit connection ID, which is used to identify endpoints.
As soon as a network change occurs, an endpoint can initiate a connection migration process using its new addresses by sending a packet to earlier established connection ID's. During connection migration, the endpoints must verify whether or not the other peer is reachable to their new address (IP and port) through a process named path validation.

Connection migration can only be initiated from the client, but there are some restrictions to it. A client may not initiate connection migration before a handshake is confirmed because the handshake requires a stable address during its duration.

4.6 Flow Control

QUIC utilizes flow control mechanisms to prevent receivers from being overwhelmed by excessive data, thus ensuring optimal performance. The protocol employs two types of flow control, connection-level and stream-level flow control to limit the amount of data that a sender can send across all streams or an individual stream, respectively. Additionally, QUIC implements credit-based flow control similar to HTTP/2, whereby a receiver advertises the absolute byte offset that it is willing to receive for each stream. The sender can then send data up to this limit, and the receiver periodically sends window update frames that increase the advertised offset limit for the stream, enabling the sender to send more data. Connection-level flow control operates similarly to stream-level flow control, but it aggregates the bytes delivered and the highest received offset across all streams.

4.7 Loss Detection

QUIC uses acknowledgments and a Probe Timeout (PTO) algorithm to detect and recover from lost packets. The transport protocol implements various acknowledgment-based loss detection algorithms, such as TCP's Fast Retransmit, Early Retransmit, and Forward Acknowledgment. If a packet meets certain criteria, it is declared lost, and QUIC takes actions such as retransmitting data or discarding the frame. Spurious loss detection can lead

to unnecessary retransmissions, so implementations can adjust reordering thresholds to minimize recovery latency.

The PTO algorithm triggers the sending of one or two probe datagrams when ack-eliciting packets are not acknowledged within the expected time or the server may not have validated the client's address. PTO operates per packet number space and does not indicate packet loss or cause prior unacknowledged packets to be marked as lost. The algorithm uses Tail Loss Probe, RTO, and F-RTO algorithms from TCP, and the timeout computation is based on TCP's RTO period. Loss detection is separate from RTT measurement and congestion control because it relies on key availability.

4.8 Congestion Control

QUIC implements congestion control and reliability control separately to improve network performance. Congestion control is achieved through a window-based scheme that limits the maximum number of bytes a sender can have in transit at any time. QUIC incorporates existing congestion control algorithms, but does not aim to develop its own. Instead, it provides generic signals for congestion control, allowing senders to implement their own mechanisms. To avoid unnecessary congestion window reduction, QUIC only collapses the congestion window if it detects persistent congestion.

QUIC also employs pacing to prevent short-term congestion by regulating the interval between packets based on factors such as the average round trip time, congestion window size, and packet size. Congestion control is based on packet numbers, while reliability control uses stream frame offsets. The QUIC standard [18] includes a congestion control algorithm in the appendix, but senders are free to implement their own mechanisms.

Chapter 5

MQTT

5.1 Introduction

MQTT [31] is a publish/subscribe messaging protocol designed for use in machine-to-machine communication (M2M) and IoT applications. It is designed to run over a reliable transport layer protocol, so for this reason TCP was traditionally used, however, some recent implementations are built on top of QUIC, such as EMQX broker and EMQX client written in Erlang[21, 22]. MQTT is ideal for use in IoT applications and M2M communications where small code footprint is required and network bandwidth is limited. Additionally, its low power use makes it suitable for battery powered devices such as smartphones or sensors. The contents of the following sections are derived from [31, 34].

5.2 History

MQTT was invented in 1999 by Dr. Andy Stanford-Clark of IBM and Arlen Nipper of Arcom Control Systems (now at Cirrus Link). At the time, it was designed for monitoring oil

pipelines over satellite connections. Over the years, its lightweight and efficient nature drawn the attention of the IoT community and became an OASIS standard in 2014.

The original requirements [34] specified by the two inventors:

- Simple Implementation
- Quality of Service for data delivery
- Lightweight and low bandwidth usage
- Continuous session awareness
- Data agnostic

Although MQTT was originally designed for proprietary embedded systems, its core goals of lightweight and efficient messaging still remain relevant in the open IoT use cases. As a result, the protocol has shifted its focus towards these use cases, leading to confusion about its acronym.

MQTT is no longer considered an acronym, but rather the name of the protocol itself. The former acronym stood for MQ Telemetry Transport, with "MQ" referring to the MQ Series, an IBM product that supported MQ telemetry transport. Despite being incorrectly labeled as a message queue protocol by some sources, MQTT is not a traditional message queuing solution, although it is possible to queue messages in certain cases. IBM used the protocol internally for ten years until they released MQTT 3.1 as a royalty-free version in 2010, allowing everyone to implement and use the protocol. Since then, several applications were built using it (e.g. Facebook Messenger adopted MQTT in an effort to increase battery life on smartphones).

5.3 Architecture

MQTT follows the publish/subscribe pattern which presents an alternative to the conventional client-server architecture. As discussed previously, unlike the client-server model, where a client communicates directly with an endpoint, pub/sub decouples the client that sends a message, or publisher, from the client or clients that receive the messages, or subscribers. This allows MQTT to:

- Provide spatial decoupling between publishers and subscribers, where they only need to know the hostname/IP and port of the broker to publish or receive messages.

- Provide decoupling by time, where the broker can store messages for offline clients, provided the client had connected with a persistent session and subscribed to a topic with QoS > 0.
- Work asynchronously, so a client is not blocked while waiting or publishing a message, however synchronization is possible if desirable.

In addition to that, MQTT is very lightweight on the client side, since the pub/sub architecture allows most of the logic to be on the broker-side.

5.3.1 MQTT Client

In MQTT, an MQTT client refers to any device, ranging from microcontrollers to full-fledged servers, that is connected to an MQTT broker through a network and runs an MQTT library. This client can be a resource-constrained device that connects over a wireless network and uses a minimal library or a standard computer with a graphical MQTT client for testing. Therefore, any device that speaks MQTT over a TCP/IP stack can be considered an MQTT client. The MQTT protocol's client implementation is straightforward and streamlined, making it an ideal choice for small devices. The ease of implementation is one of the reasons MQTT is suitable for resource-constrained devices.

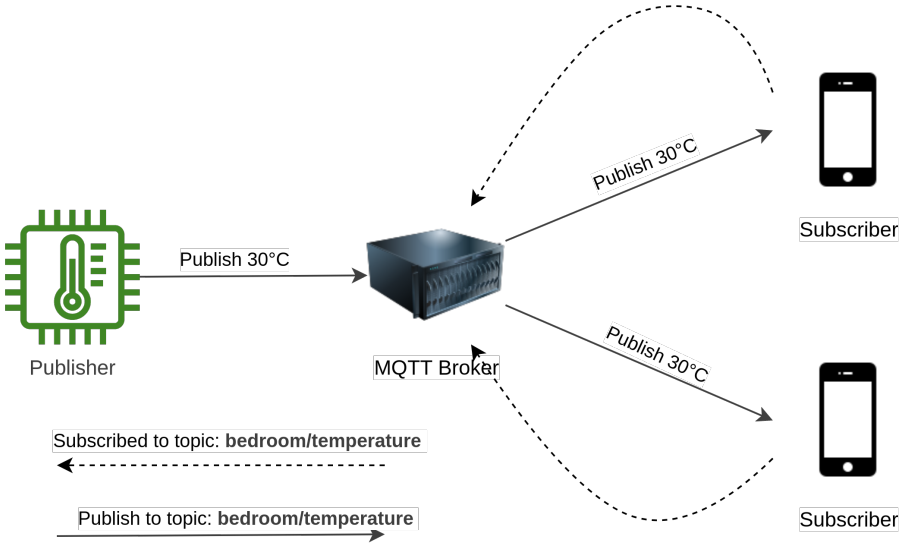
5.3.2 MQTT Broker

In MQTT, the MQTT broker is the central component that performs the key functions of the publish/subscribe protocol. It is capable of handling a large number of MQTT clients simultaneously, depending on the implementation.

The MQTT broker is responsible for receiving and filtering all incoming messages, determining which clients are subscribed to each topic, and delivering the message to those clients. In addition, the broker stores session data for clients with persistent sessions, including subscriptions and missed messages. The broker is also responsible for client authentication and authorization. It can be extended to support custom authentication, authorization, and integration with backend systems. Integration is crucial for the broker, as it is often directly exposed on the internet, handles numerous clients, and needs to transfer messages to downstream processing and analysis systems.

5.4 MQTT Topics

MQTT brokers use topics to to decide which clients receive which message. A topic refers to a string that is used by the broker to identify to which clients each published message should be forwarded. A topic may consist of multiple topic levels, where each topic is seperated by a slash. Topics are lightweight and a broker accepts any valid topic, meaning that the client can publish or subscribe to a topic without creating it.



For example, in Figure 6 we can see the two subscribers are subscribed to the topic bedroom/temperature. When the publisher sends the temperature to the broker, the topic to be published is specifically noted in the publish packet, so the broker forwards the message to all the subscribers subscribed to the corresponding topic.

Clients may also use wildcards to subscribe to multiple topics simultaneously, but not to publish a message. There are two types of wildcards:

- **Single Level:** Replaces one topic level by using + as a topic level. For example a subscription to topic **home/firstfloor/+/temperature** will subscribe to:
  - home/firstfloor/masterbedroom/temperature
  - home/firstfloor/guestbedroom/temperature

  but not:
  - home/firstfloor/masterbedroom/brightness
  - home/firstfloor/masterbedroom/bathroom/temperature
  - home/groundfloor/masterbedroom/temperature

- **Multi Level:** Replaces many topic levels by using # as a topic level. A # may only be used as the last topic level for a topic.For example a subscription to topic **home/firstfloor/#** will subscribe to:
    - home/firstfloor/masterbedroom/temperature
    - home/firstfloor/masterbedroom/brightness
    - home/firstfloor/guestbedroom/temperature
    - home/firstfloor/guestbedroom/brightness

    but not:
    - home/groundfloor/livingroom/temperature

5.5 MQTT Control Packets

MQTT uses control packets to establish and manage communication between clients and brokers. There are several types of control packets in MQTT:

- **CONNECT:** This packet is sent by a client to initiate a connection to a broker. It contains information such as the client identifier, username and password (if required), and the clean session flag.
- **CONNACK:** This packet is sent by the broker in response to a CONNECT packet. It contains a status code indicating whether the connection was successful or not.
- **PUBLISH:** This packet is sent by a client to publish a message to a specific topic on the broker. It contains the topic name, message payload, and QoS (Quality of Service) level.
- **PUBACK:** This packet is sent by the broker to the client to acknowledge receipt of a PUBLISH packet with a QoS level of 1.
- **PUBREC:** This packet is sent by the broker to the client to acknowledge receipt of a PUBLISH packet with a QoS level of 2.
- **PUBREL:** This packet is sent by the client to the broker to release a PUBREC packet with a QoS level of 2.
- **PUBCOMP:** This packet is sent by the broker to the client to acknowledge receipt of a PUBREL packet with a QoS level of 2.
- **SUBSCRIBE:** This packet is sent by a client to subscribe to one or more topics on the broker. It contains the topic name and QoS level for each subscription.
- **SUBACK:** This packet is sent by the broker in response to a SUBSCRIBE packet. It contains a list of QoS levels for each subscription requested by the client.
- **UNSUBSCRIBE:** This packet is sent by a client to unsubscribe from one or more topics on the broker. It contains the topic name for each subscription.

- **UNSUBACK:** This packet is sent by the broker in response to an UNSUBSCRIBE packet to acknowledge the unsubscribe request.
- **PINGREQ:** This packet is sent by the client to the broker to keep the connection alive.
- **PINGRESP:** This packet is sent by the broker to the client to acknowledge receipt of a PINGREQ packet.
- **AUTH:** This packet is used for authentication and is sent by a client to the broker. It contains authentication data such as a username and password.
- **DISCONNECT:** This packet is sent by either the client or the broker to terminate the connection.

5.6 Packet Format

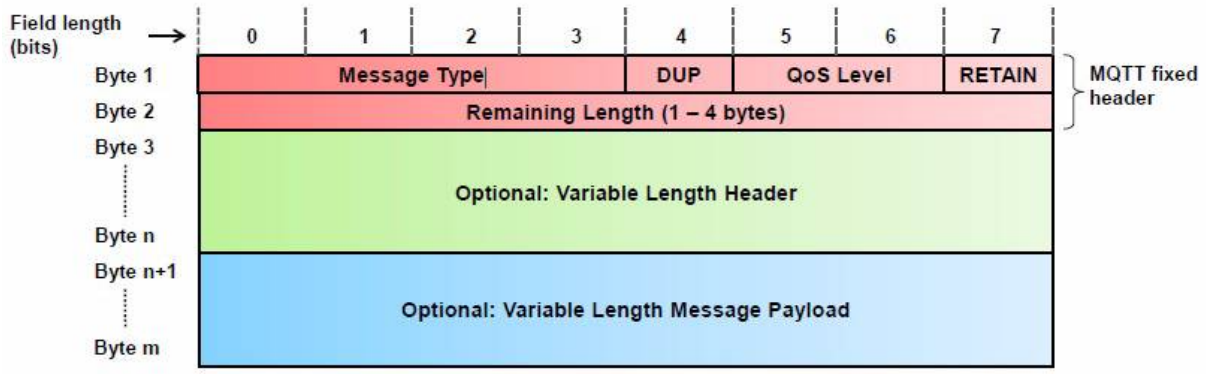


Figure 10: MQTT Packet Format[33]

One of the main reasons MQTT is considered ideal for IoT communication is due to the low overhead of its packets. Unlike, for example, HTTP, that has a large header, MQTT keeps a concise header. This leads to efficient communication in resource-constrained environments and keeps bandwidth usage low. The packet format is shown in Figure 10.

The fixed header takes 2 bytes for every packet, and a variable header is used for some control packets to set fields like topic name, QoS level, DUP flag etc.

5.5 Quality of Service (QoS)

Quality of Service (QoS) in MQTT is a mutually agreed upon level of delivery guarantee between the message sender and receiver. When discussing QoS in MQTT, it is important to

consider both sides of message delivery, i.e., message delivery from the publishing client to the broker and message delivery from the broker to the subscribing client. The publishing client determines the QoS level of the message and the broker transmits it to the subscribing client using the QoS level specified by the subscriber during the subscription process.

QoS is a crucial feature of MQTT as it enables clients to choose a service level that suits their network reliability and application logic. MQTT ensures message re-transmission and delivery even when the underlying transport is unreliable, making communication in unreliable networks much easier.

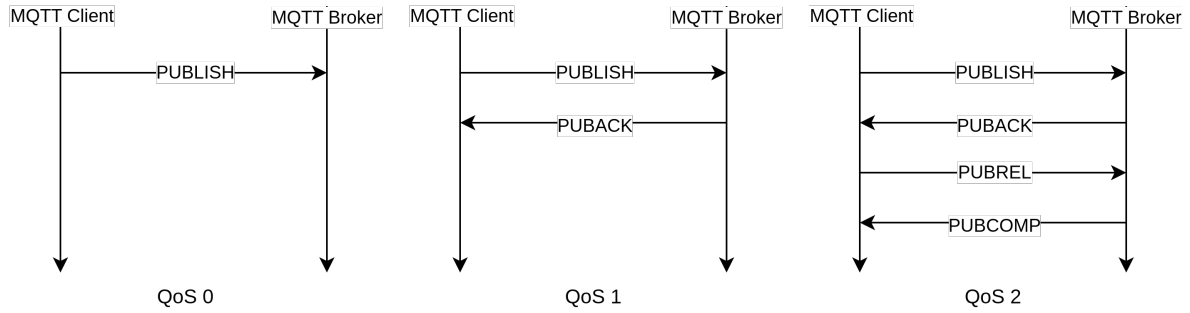


Figure 11: MQTT Publish Message Flow based on QoS level

MQTT provides three levels of QoS:

- **0 - At most once:** QoS level 0 in MQTT is considered the minimum level of service and provides a best-effort delivery guarantee. This means that there is no assurance of message delivery, and the recipient does not confirm receipt of the message. Additionally, the sender does not store or retransmit the message. QoS level 0 is often referred to as "fire and forget" and has similar reliability guarantees to the underlying transport protocol.
- **1 - At least once:** QoS level 1 provides assurance that a message will be delivered at least once to the recipient, and the sender will store the message until it receives a PUBACK packet from the receiver, which confirms that the message has been received. However, it is possible for the message to be delivered multiple times. To match the PUBLISH packet to the corresponding PUBACK packet, the sender uses the packet identifier in each packet. If the sender does not receive a PUBACK packet within a reasonable time frame, it resends the PUBLISH packet. Upon receiving a message with QoS 1, the recipient can process it immediately. For instance, if the recipient is a broker, the broker broadcasts the message to all subscribing clients and then responds with a PUBACK packet. In the event that the publishing client resends

the message, it sets a duplicate (DUP) flag. The DUP flag is used for internal purposes only in QoS 1 and is not processed by either the broker or the client. The recipient sends a PUBACK packet regardless of the DUP flag.

- **2 - Exactly once:** In MQTT, QoS level 2 provides the highest level of service, ensuring that each message is delivered exactly once to its intended recipients. This level is considered the safest but also the slowest. The guarantee of delivery is accomplished through a four-part handshake, consisting of at least two request/response flows between the sender and receiver. During this process, both parties use the packet identifier of the original PUBLISH message to coordinate the delivery of the message. When a receiver receives a QoS 2 PUBLISH packet from a sender, it processes the message and responds with a PUBREC packet as an acknowledgement. If the sender does not receive a PUBREC packet, it sends the PUBLISH packet again with a duplicate (DUP) flag until it receives an acknowledgement. Once the sender receives a PUBREC packet, it can safely discard the initial PUBLISH packet and respond with a PUBREL packet. After the receiver receives the PUBREL packet, it can discard all stored states and reply with a PUBCOMP packet. The same is true when the sender receives the PUBCOMP packet. Until the receiver sends the PUBCOMP packet, it stores a reference to the packet identifier of the original PUBLISH packet to avoid processing the message twice. After the sender receives the PUBCOMP packet, the packet identifier of the published message becomes available for reuse. Upon completion of the QoS 2 flow, both the sender and receiver are certain that the message is delivered, and the sender receives confirmation of delivery. In case of packet loss, the sender is responsible for re-transmitting the message within a reasonable amount of time, and the recipient must respond to each command message appropriately.

5.6 Security

MQTT protocol employs several layers of security to prevent different types of attacks. As MQTT aims to be a lightweight and user-friendly communication protocol for the IoT, it specifies only a few security mechanisms. Therefore, most MQTT implementations use other state-of-the-art security standards, such as TLS, to provide transport security. Building on accepted standards is a sensible approach to ensuring security, as it can be a complex and challenging task. In short, the security mechanisms that MQTT provides are:

- Authentication using username and password to connect to the broker.

- Authorization in order to manage which clients can access certain topics, which control packets one can use and the options a client can set.
- Encryption is optional when TCP is used. MQTT provides the option of encryption through TLS+TCP on port 8883, instead of port 1883 used for not-encrypted TCP connections. In case of MQTT over QUIC, the communication is always encrypted using TLS1.3 on default port 14567(for emqx).

Chapter 6

Evaluation Setup and Results

6.1 Hardware Setup

The hardware setup used for the experimental evaluation, as shown in Figure 7, consists of a laptop, a Raspberry Pi 4 and a switch. The laptop acts as a broker, and it was also used to run Wireshark [24] for all packet captures, as all traffic passes through the broker. The Raspberry Pi played the role of both the publisher and the subscriber and was connected to a switch through Ethernet.

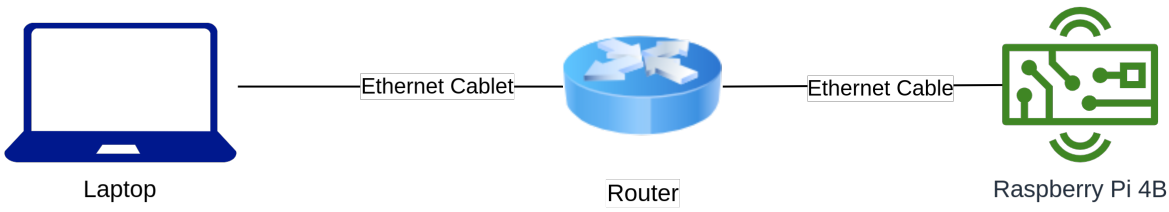


Figure 12: Evaluation Setup

The MQTT broker was run inside a docker container, on a laptop with 8GB RAM, which was also connected to the router through Ethernet.

To ensure time synchronization for calculating delay, the publisher and subscriber both ran on the Raspberry Pi as in [23]. Therefore, every message published was transmitted through the broker and back to the Raspberry Pi.

6.2 Software Setup

An open source implementation of MQTT broker and client, named emqx [22] and emqtt_bench [21] were used for conducting the experiments. For the broker, a docker image was used to run into a container for ease of deployment, whereas the benchmarking tool, emqtt_bench, used as a client was built from source. In order to emulate a lossy network for several experiment scenarios, the linux tc utility was used, which allows as to configure the kernel packet scheduler and control the egress packet loss rate. Wireshark was used to record all the traffic between the pulisher and the subscriber. For gathering system information such as cpu and memory usage, the linux utility, htop was used.

Note than EMQX broker and client use the MSQUIC library for QUIC, which implements congestion control using CUBIC.

6.3 Results

This section presents the performance evaluation conducted on the previously described testbed. We examine the performance of MQTT over TCP and QUIC in terms of traffic delay under different packet loss scenarios. It's important to note that for all configurations, TCP is utilized alongside TLS to ensure consistent functionality and enable fair comparisons, unless stated otherwise.

As mentioned previously, the linux TC utility was used, to replicate a lossy network environment. By conducting these experiments, we aim to gain insights into the behavior of MQTT over QUIC in comparison to the current standard of MQTT and assess the impact the underlying transport layer protocol has on delay.

6.3.1 Connection Establishment

To test the connection establishment latency, we deployed 1000 client subscribe requests from emqtt_bench. For the QUIC 0-RTT scenario, the client needs to firstly connect with a 1-

RTT handshake to obtain the session ticket needed for the 0-RTT handshake using the flag –load-qst in emqtt_bench. Before going through the results, the connection establishment process of MQTT with TCP+TLS, MQTT with QUIC 1-RTT and MQTT with QUIC 0-RTT is demonstrated.

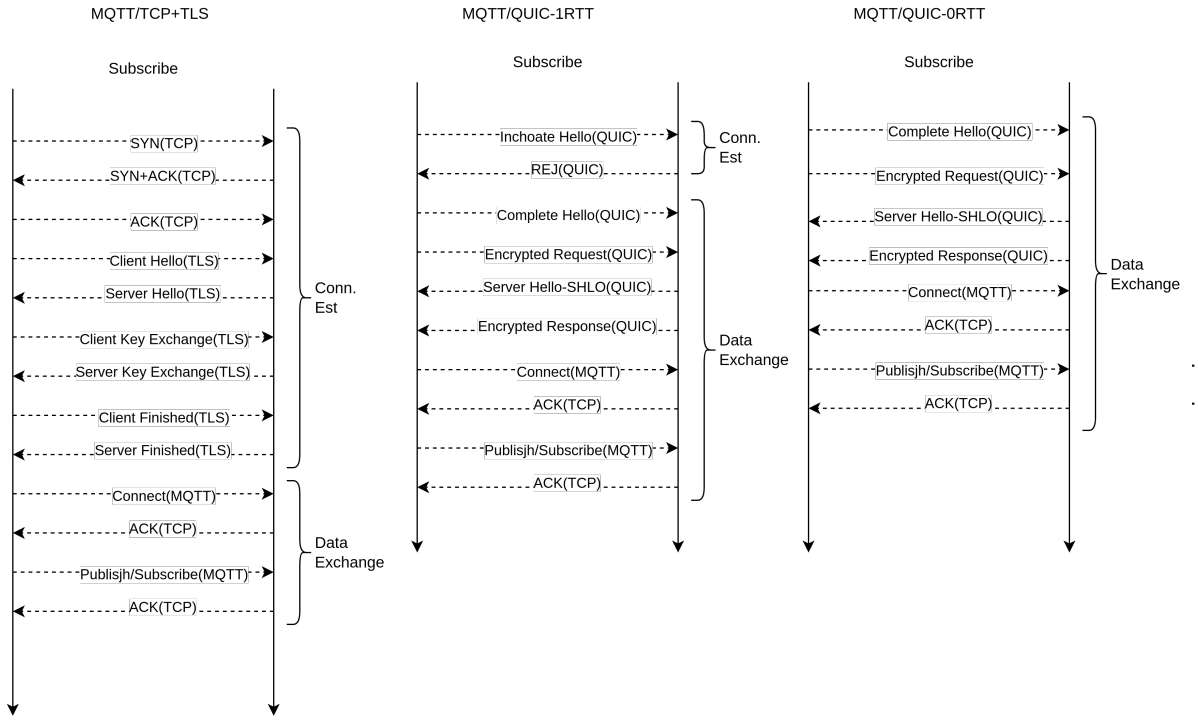
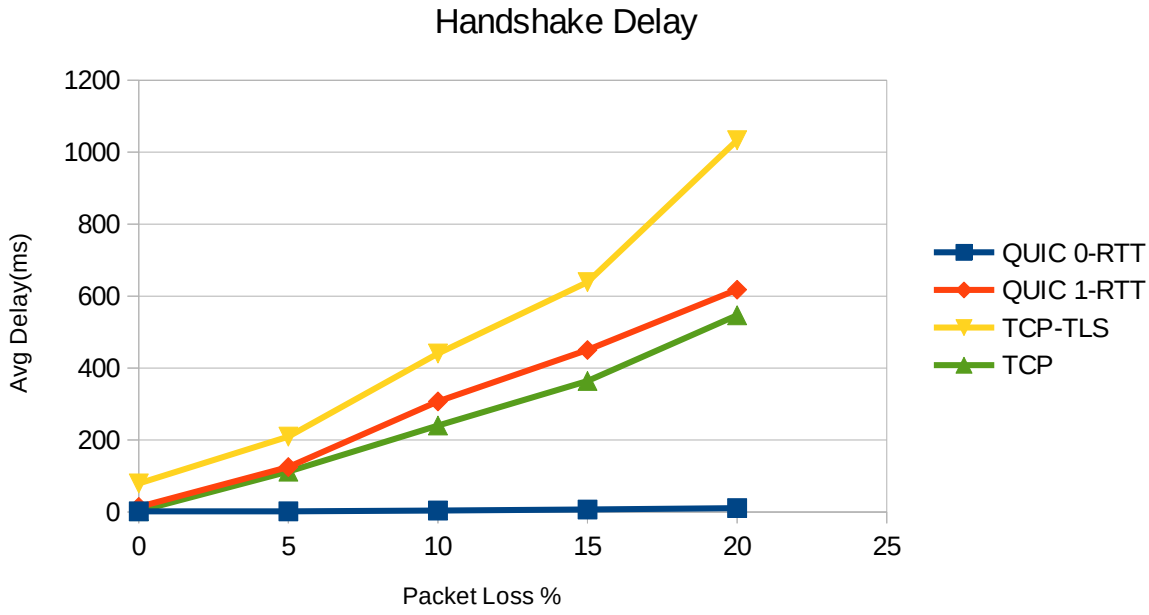
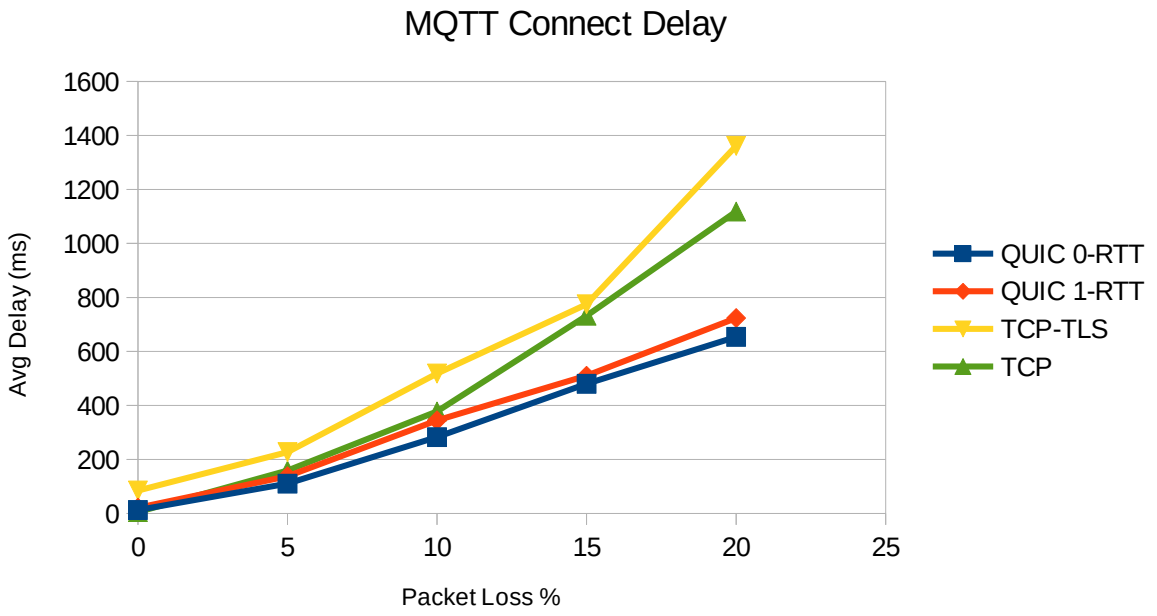


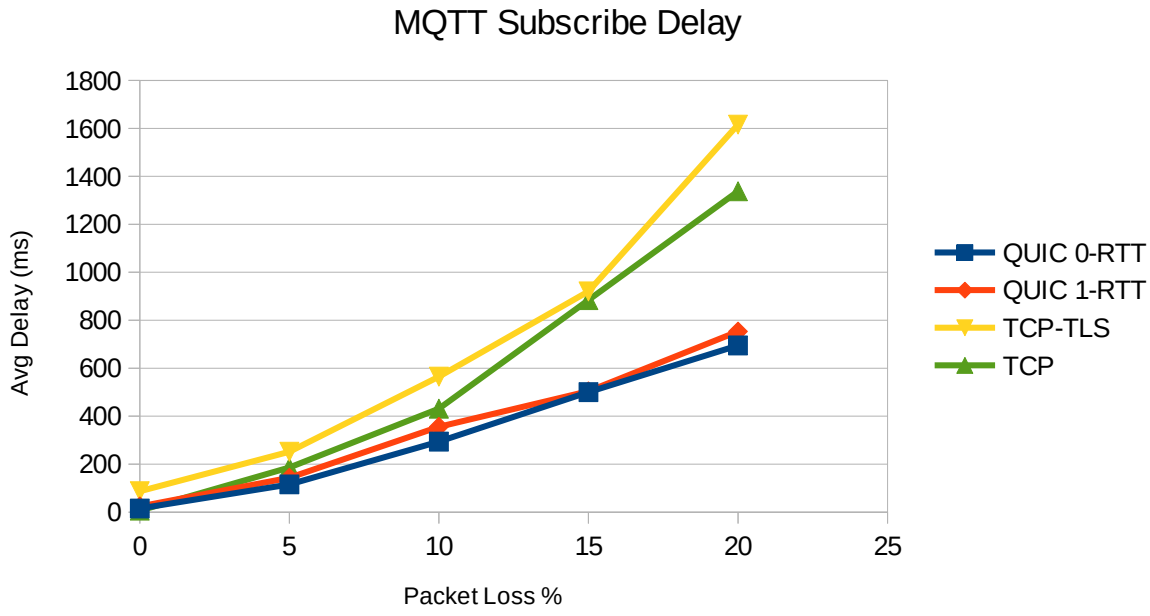
Figure 13: Connection Establishment process. MQTT QoS=0

Shown below is the handshake performance of QUIC vs TCP+TLS vs TCP under various packet loss configurations.



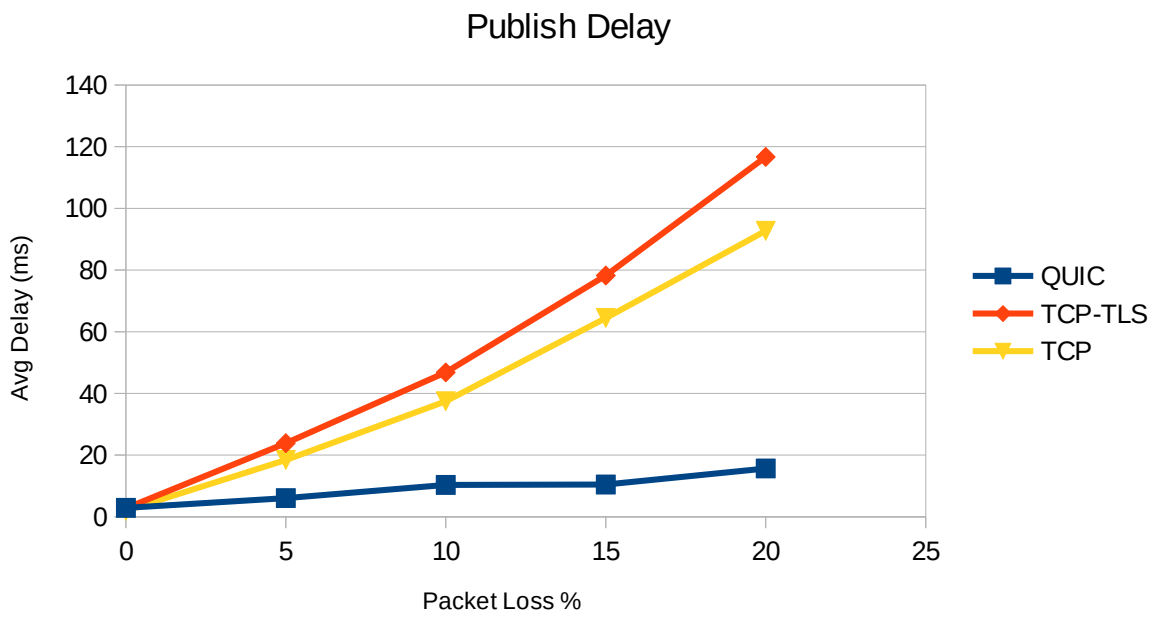
With low packet loss rates, QUIC 1-RTT, TCP+TLS and TCP do not show any major differences in the avg handshake delay, however as the packet loss grows, TCP+TLS gets outperformed by QUIC with almost half the delay. The handshake delay of QUIC 1-RTT is comparable to TCP's, however TCP does not provide us with any encryption on its own. The delay of QUIC 0-RTT(reconnection) handshake is negligible, and even though it effectively reduces the handshake overhead and latency, emqx does not support early data by default, which means encrypted application data cannot be sent with the first complete hello during QUIC 0-RTT handshake. The reason is, early data are vulnerable to replay attacks and QUIC recommends not carrying data on 0-RTT that may possibly change the application state. Therefore, the connect and subscribe delay results of QUIC 1-RTT and QUIC 0-RTT will be identical.





6.3.2 Publish Delay

For the message publishing test, a single publisher and a single subscriber were used, so both QUIC and TCP achieved 100% delivery ratio irrespective of the packet loss rate observed in all conducted experiments. Nonetheless, messages transferred over TCP suffered considerably more retransmissions than QUIC, which can be possibly attributed to better flow control, error correction and retransmission mechanisms. The publisher was set to send 1000 messages with an interval of 1s inbetween, in order to minimize congestion.

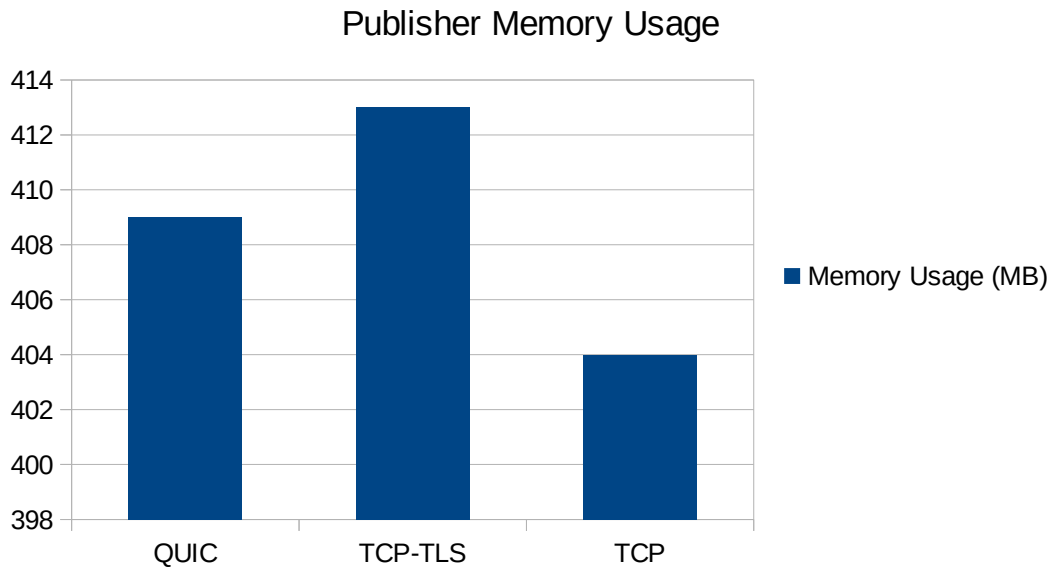


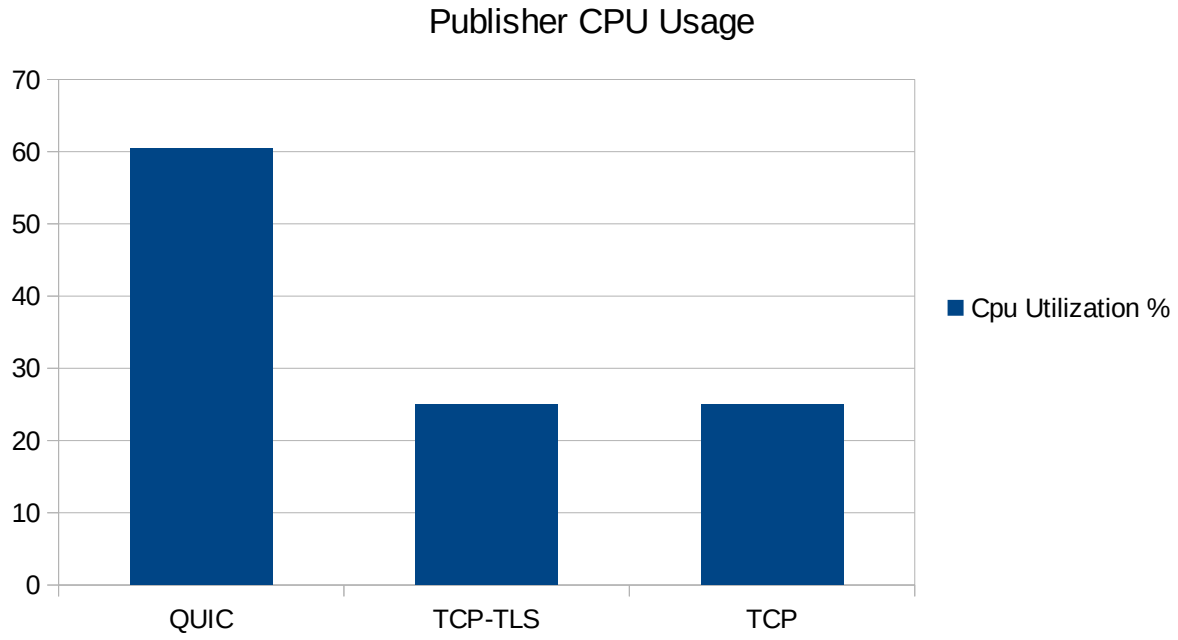
QUIC outperforms TCP carried messages for all values of packet loss. The results show that TCP and TCP+TLS suffer from high delay which grows linearly as packet loss increases. On the other hand, messages transported with QUIC have stable delay for all values of packet loss. The high delay noticed for the TCP streams, is most likely attributed to HOL blocking, which in turn caused noticeable amount of packet loss compared to QUIC stream. This effect, would have even more negative impact in applications which rely on bursty traffic such as motion sensors where congested links are more common.

6.3.3 CPU and Memory Usage

The publisher running over QUIC, also showed higher peak CPU usage than its counterparts. Precisely, 61%, 25% and 25%, max CPU utilization was shown by the QUIC, TCP+TLS and TCP implementations of MQTT respectively. In terms of memory usage TCP had a slight edge, followed by QUIC and last TCP+TLS. However, the overall CPU and Memory usage

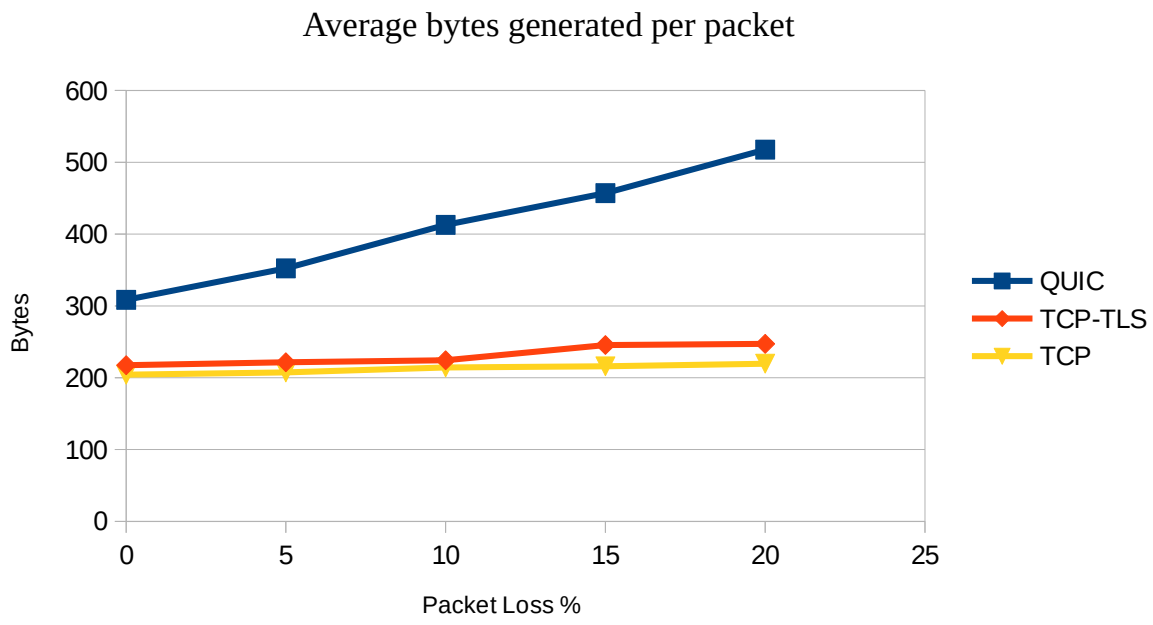
of QUIC appears to be much lower if take into consideration the total communication delay. The publish delay for QUIC, TCP and TCP+TLS is 18ms, 92ms and 118ms respectively. At worst case scenario for each we had a a peak of 60%, 25% and 25% CPU utilization. We can use the *CPU utilization x Communication Delay* product to calculate the total amount of CPU recources used over the given duration. Therefore, the CPU time for QUIC, TCP and TCP+TLS is 10.8ms, 30ms and 30ms respectively, concluding that QUIC has a lower CPU usage that translates to lower energy usage.





6.3.4 Bytes generated per packet

In terms of bandwidth usage, QUIC imposes more overhead, and as the packet loss ratio increases, the traffic generated by QUIC is over 100% higher than TCP(+TLS). This is caused due to the larger header and additional features of QUIC such as Forward Error Correction(FEC). As the packet loss rate gets higher, the FEC redundancy added to the original packet also gets higher, thus leading to higher average size per packet, if QUIC is used.



Chapter 7

Conclusion

7.1 Summary

In this thesis, an in-depth analysis and comparison of MQTT over TCP with MQTT over QUIC is made. We analyzed the theoretical improvements that QUIC offers as a transport layer protocol over TCP and why it can also benefit the MQTT, and the IoT in general. Using an open-source implementation of an MQTT client and broker, both by EMQX, we studied the performance of MQTT over QUIC in comparison with MQTT over TCP. Experimental results showed that MQTT w/QUIC outperforms MQTT w/TCP in every scenario in terms of delay of handshake, connection establishment and MQTT traffic. MQTT over QUIC peak CPU usage appears to be higher than TCP(+TLS) but the overall CPU used is lower. However, using QUIC generates more traffic, as the traffic generated is up to double compared to TCP, for high packet loss rate, due to FEC.

7.2 Future Work

MQTT over QUIC showed higher traffic generated than the current state of the protocol using TCP. This can be reduced by modifying the FEC mechanism to be less aggressive or completely removing it. In general, several optimizations can be done to reduce the code footprint of QUIC and reduce the computational complexity of the encryption mechanisms. Also, the open-source EMQX client and broker used, has not utilized all of QUIC features such as flow control. Flow control may not seem that useful for IoT applications, but there are some applications that depend on bursty traffic such as motion sensing apps. The above can be addressed in future work.

7.3 Conclusions

QUIC shows very promising results for usage with MQTT in terms of traffic delay, CPU usage and the connection establishment phase. In addition, MQTT over QUIC suffered a noticeably less amount of packet loss than MQTT over TCP. Moreover, the client migration mechanism it provides makes it suitable for the IoT and mobile devices as it can seamlessly migrate from one network to another and survive ip or port changes (e.g. mobile phone switching from Wi-Fi to cellular network). Also, the fact that QUIC is not affected from HOL blocking, makes MQTT even more capable to support bursty traffic,for use in applications that rely on it such as applications relying on motion sensors.

Bibliography

[1] S. Madakam, R. Ramaswamy, and S. Tripathi, “Internet of Things (IoT): A Literature Review,” Journal of Computer and Communications, vol. 03, no. 05, pp. 164–173, 2015, doi: https://doi.org/10.4236/jcc.2015.35021.

[2] “IoT connected devices worldwide 2019-2030,” Statista. https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/#:~:text=The%20number%20of%20Internet%20of.

[3] A. Langley et al., “The QUIC Transport Protocol,” Proceedings of the Conference of the ACM Special Interest Group on Data Communication - SIGCOMM ’17, 2017, doi: https://doi.org/10.1145/3098822.3098842.

[4] I. Yaqoob et al., “Internet of Things Architecture: Recent Advances, Taxonomy, Requirements, and Open Challenges,” IEEE Wireless Communications, vol. 24, no. 3, pp. 10–16, Jun. 2017, doi: https://doi.org/10.1109/mwc.2017.1600421.

[5] G. A. Akpakwu, B. J. Silva, G. P. Hancke, and A. M. Abu-Mahfouz, “A Survey on 5G Networks for the Internet of Things: Communication Technologies and Challenges,” IEEE Access, vol. 6, pp. 3619–3647, 2018, doi: https://doi.org/10.1109/access.2017.2779844.

[6] M. Koster, A. Keränen, and J. Jimenez, “Publish-Subscribe Broker for the Constrained Application Protocol (CoAP),” IETF Datatracker, May 04, 2022. Accessed: Apr. 22, 2023. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-core-coap-pubsub-10

[7] A. Lazidis, K. Tsakos, and E. G. M. Petrakis, “Publish-Subscribe approaches for the IoT and the cloud: Functional and performance evaluation of open-source systems,” Internet of Things, p. 100538, May 2022, doi: https://doi.org/10.1016/j.iot.2022.100538.

[8] W. Shang, Y. Yu, R. Droms, and L. Zhang, “Challenges in IoT Networking via TCP/IP Architecture,” 2016. Available: https://named-data.net/wp-content/uploads/2016/02/ndn-0038-1-challenges-iot.pdf

[9] P. Kumar and B. Dezfouli, “Implementation and analysis of QUIC for MQTT,” Computer Networks, vol. 150, pp. 28–45, Feb. 2019, doi: https://doi.org/10.1016/j.comnet.2018.12.012.

[10] K. Lakshminadh, N. Siva, Rao, and A. Radha, “ANALYSIS OF TCP ISSUES IN INTERNET OF THINGS.” Accessed: Nov. 28, 2022. [Online]. Available: https://acadpubl.eu/jsi/2018-118-14-15/articles/14/24.pdf

[11] S. Saini and A. Fehnker, “Evaluating the Stream Control Transmission Protocol Using Uppaal,” Electronic Proceedings in Theoretical Computer Science, vol. 244, pp. 1–13, Mar. 2017, doi: https://doi.org/10.4204/eptcs.244.1.

[12] T. Dierks and E. Rescorla, “The Transport Layer Security (TLS) Protocol Version 1.2,” Aug. 2008, doi: https://doi.org/10.17487/rfc5246.

[13] T. Kothmayr, C. Schmitt, W. Hu, M. Brünig, and G. Carle, “DTLS based security and two-way authentication for the Internet of Things,” Ad Hoc Networks, vol. 11, no. 8, pp. 2710–2723, Nov. 2013, doi: https://doi.org/10.1016/j.adhoc.2013.05.003.

[14] F. Iqbal, M. Gohar, H. Karamti, W. Karamti, S.-J. Koh, and J.-G. Choi, “Use of QUIC for AMQP in IoT networks,” Computer Networks, vol. 225, p. 109640, Apr. 2023, doi: https://doi.org/10.1016/j.comnet.2023.109640.

[15] M. Masirap, M. H. Amaran, Y. M. Yussoff, R. A. Rahman, and H. Hashim, “Evaluation of reliable UDP-based transport protocols for Internet of Things (IoT),” IEEE Xplore, May 01, 2016. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7575063.

[16] R. Herrero, “Analysis of the constrained application protocol over quick UDP internet connection transport,” Internet of Things, vol. 12, p. 100328, Dec. 2020, doi: https://doi.org/10.1016/j.iot.2020.100328.

[17] “QUIC, a multiplexed transport over UDP,” www.chromium.org. https://www.chromium.org/quic/

[18] M. Thomson and J. Iyengar, “RFC 9000 QUIC: A UDP-Based Multiplexed and Secure Transport” 2021. Available: https://www.rfc-editor.org/rfc/rfc9000.pdf

[19] A. Langley and W.-T. Chang, “QUIC Crypto,” Google Docs. https://docs.google.com/document/d/1g5nIXAIkN_Y-7XJW5K45IblHd_L2f5LTaDUDwvZ5L6g/edit.

[20] M. Thomson and S. Turner, “RFC FT-IETF-quic-TLS: Using TLS to secure quic,” IETF Datatracker, 27-May-2021. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc9001.

[21] “emqtt-bench,”, https://github.com/emqx/emqtt-bench.

[22] “EMQX,”, https://github.com/emqx/emqx.

[23] D. Thangavel, X. Ma, A. Valera, H.-X. Tan, and C. K.-Y. Tan, “Performance evaluation of MQTT and CoAP via a common middleware,” IEEE Xplore, Apr. 01, 2014. https://ieeexplore.ieee.org/abstract/document/6827678

[24] Wireshark Foundation, “Wireshark,” Wireshark.org, 2016. https://www.wireshark.org/

[25] R. P. Ltd, “Raspberry Pi,” Raspberry Pi. https://www.raspberrypi.com/

[26] Docker. https://www.docker.com/

[27] C. Yunlong, “What Is Docker Container? How Does It Work? - Huawei,” info.support.huawei.com. https://info.support.huawei.com/info-finder/encyclopedia/en/Docker+Container.html

[28] EMQX. https://www.emqx.io/

[29] “tc(8) - Linux manual page,” man7.org. https://man7.org/linux/man-pages/man8/tc.8.html

[30] G. Fedrecheski, L. C. P. Costa, and M. K. Zuffo, “Elixir programming language evaluation for IoT,” IEEE Xplore, Sep. 01, 2016. https://ieeexplore.ieee.org/abstract/document/7797392

[31] MQTT Version 5.0. Edited by Andrew Banks, Ed Briggs, Ken Borgendale, and Rahul Gupta. 07 March 2019. OASIS Standard. https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html.

[32] J. P. Vasseur, “Terms Used in Routing for Low-Power and Lossy Networks,” IETF, Jan. 01, 2014. https://datatracker.ietf.org/doc/rfc7102/.

[33] “MQTT Tutorial | MQTT architecture, MQTT protocol use cases,” *www.rfwireless-world.com*. https://www.rfwireless-world.com/Tutorials/MQTT-tutorial.html

[34] HiveMQ, “MQTT & MQTT 5 Essentials A comprehensive overview of MQTT facts and features for beginners and experts alike”