FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF MASTER'S THESIS

Title:	Security and Performance of IoT Application Protocols
Student:	Bc. David Mládek
Supervisor:	Ing. Tomáš Pajurek
Study Programme:	Informatics
Study Branch:	Computer Security
Department:	Department of Information Security
Validity:	Until the end of summer semester 2021/22

Instructions

In recent years, the need for automation is still growing in many industries. This automation is frequently achieved by introducing IoT (Internet of Things) solutions. Details of these solutions vary but all of them require secure connection of devices to the cloud or other server-side system. This thesis should describe and compare widely used application protocols, mainly from the perspective of security and its impact on performance.

1. Describe following application-layer protocols: AMQP, MQTT, CoAP in enough detail to support the subsequent tasks.

2. Analyze security aspects of the protocols.

3. Benchmark all three protocols (+ HTTP over TLS as a baseline) on a specific HW/platform provided by the supervisor and analyze the results and discuss tradeoffs.

References

Will be provided by the supervisor.

prof. Ing. Róbert Lórencz, CSc.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague February 1, 2021

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Security and Performance of IoT Application Protocols

Bc. David Mládek

Department of Information Security
Supervisor: Ing. Tomáš Pajurek

January 6, 2022

Acknowledgements

First and foremost I would like to thank my supervisor for all valuable advice and unending patience.

I would also like to thank my partner, family, and friends for making sure I have a thesis to hand in and all the other support I was given.

I would also like to thank all the contributors to the open source project that have been used in this work. Every maintainer or contributor I had a chance to interact with was very helpful and quick to response to any questions or issues. Of all these I would like to mention Ivan Markov (ivmarkov on GitHub) for all the work he put into enabling the community to flash Rust programs including `std` onto ESP32.

Lastly I would like to thank Datamole. This company allowed me to learn more than I could have expected and supported me through both the Bachelor's and now Master's thesis.

Thank you all.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46 (6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity.

In Prague on January 6, 2022 . . . . . .

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Mládek, David. *Security and Performance of IoT Application Protocols.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2022.

Abstrakt

Tato práce představuje tři IoT protokoly—CoAP, MQTT, a AMQP. Všechny tři jsou popsány včetně bezpečnostních vlastností a jejich rozdílů.

Bylo provedeno několik experimentů na dvou vývojových deskách pro IoT. Tyto výsledky byly srovnány s přenosem přes HTTP. Všechny protokoly byly použity s i bez TLS. Ukázalo se několik rozdílů v rychlosti, počtu přenesených bytů i použitelnosti daných protokolů, konkrétně AMQP se ukázalo být prakticky nepoužitelné na ESP32. To vedlo k závěru, že je lepší pro zařízení s omezeným výkonem použít více podporované protkoly jako je například MQTT. Pokud má zařízení dostatečný výkon a je k dispozici vyhovující knihovna, AMQP může být použito, jelikož má srovantelný výkon s ostatními protokoly. CoAP posílaný přes UDP měl nejmenší rychlost přenosu kvůli předcházení přetížení sítě, což bránilo paralelnímu posílání dat. Toto byl jediný protkol a způsob přenosu, který byl pomalejší než HTTP. U MQTT byly jasně zřetelné rozdíly v počtu přenesných bytů i v rychlosti mezi garancemi doručení nejvýše jednou, nejméně jednou a právě jednou. Tyto rozdíly ve výkonu by měly být vždy brány v úvahu, když má být tento protokol použit.

Experimenty také ukázaly, že TLS výrazně nezpomaluje žádný těchto protokolů. Pro většinu protokolů se také jen mírně zvýšil počet přenesených bytů, pouze pro MQTT se tento počet více než zdvojnásobil. Tato práce doporučuje vždy používat TLS nebo DTLS, pokud je to technicky možné.

Klíčová slova IoT, MQTT, CoAP, AMQP, ESP32, Raspberry Pi, vestavěné systémy

Abstract

This work introduces three IoT protocols—CoAP, MQTT, and AMQP. All three protocols are described including security measures they provide and some of the differences are noted.

Several experiments were done on the selected protocols using two distinct IoT platforms. Results of these experiments were compared to HTTP which has been used as a baseline. All protocols were examined both with and without using TLS. There were some differences observered in both speed, bandwidth usage, and usability, specifically AMQP turned out to be unusable on ESP32 in practice. This led to the conclusion that it is better to use other, more supported protocols like MQTT, for messaging in constrained devices. If the device has sufficient computing power and a suitable library is available, AMQP can be used as it has comparable performance to the other protocols. CoAP over UDP did not perform very well under high load due to its congestion control rules which prevented any parallel operations. This has been the only protocol that performed worse than HTTP. For MQTT the differences between at most once, at least once, and exactly once delivery guarantees were clearly visible both in the amount transferred bytes and the speed of the protocol. The performance difference should be always taken into account when choosing to use MQTT.

The experiments also showed that TLS does not considerably increase the time it takes to transfer messages. For most protocols there was also very small increase in the number of transferred bytes with only MQTT more than doubling the used bandwidth. This work advises to always use TLS or DTLS if it is technically possible.

Keywords IoT, MQTT, CoAP, AMQP, ESP32, Raspberry Pi, Embedded, Constrained

Contents

List of Figures

List of Tables

Introduction

This work aims to provide an introduction and comparison of three protocols used in IoT. These are CoAP, MQTT, AMQP.

The protocols will be first examined theoretically. All three are open-sourced and their specification is publicly available. There is some history behind each of the protocols with AMQP specifically changing considerably between versions 0-9-1 and 1.0. MQTT has also seen considerable changes between versions 3.1.1 and 5.0. In both cases, the latest version will be looked at in this work.

CoAP was developed and is being actively extended by IETF and it has been defined in RFC 7252[1]. This specification will be the main focus of the work and other extensions such as block transfer[2] and observe[3] will not be explained, but are mentioned where appropriate. The only extension that will be considered more closely is RFC 8323[4], which makes slight adjustments to the protocol so that it can be run on TCP as well as UDP, the latter being the only option for the base protocol.

MQTT 5.0 will be also described. It is standardized by OASIS[5] This work will not go over changes from the previous versions and only the standalone protocol will be considered. There is also MQTT-SN which is aimed at constrained devices. However, it is not compatible with the MQTT protocol itself and will not be considered in this work.

AMQP 1.0 is the last protocol that will be the focus of this work. It has been designed for business messaging needs, but it has found its way into IoT where for example Microsoft uses it in its IoT hub[6]. The protocol will also be considered only as is and there will be no explanation on the changes between this and prior versions.

Implementations of all three protocols will then be uploaded into two different devices and experiments will be run on them. First, device will be ESP-WROOM-32[7] which is more constrained while still having enough RAM and processing power to run more complicated workloads. One of its main advantages is the integrated WiFi chip.

Second tests will be run from Raspberry Pi model 3 b+[8]. This is a much stronger single board computer which can run even Linux OS.

The focus will be placed on the speed of the protocols when transmitting many small messages and one large message. Next, a qualitative examination of the protocol exchanges on the wire will be done through packet captures with Wireshark. There number of transferred bytes, as well as packets, will be compared between the protocols.

Chapter 1

IoT Security Problems Overview

1.1 Constrained Nodes

There is a very fast growing trend of creating small devices connected to the Internet which do not have processing power, memory, or power that most computers do. These devices are also called constrained nodes or IoT devices.

Internet of Things (IoT) is the interconnection through the Internet of everyday objects from kitchen appliances to ordinary items such as shoes. These small devices can send and receive data to provide insights or better usage to the user. These devices use almost exclusively Machine-to-Machine (M2M) communication with some servers on the Internet without any manual trigger by the user.

IETF defines constrained nodes in RFC 7228[9] thus:

> A node where some of the characteristics that are otherwise pretty much taken for granted for Internet nodes at the time of writing are not attainable, often due to cost constraints and/or physical constraints on characteristics such as size, weight, and available power and energy. The tight limits on power, memory, and processing resources lead to hard upper bounds on state, code space, and processing cycles, making optimization of energy and network bandwidth usage a dominating consideration in all design requirements. Also, some layer-2 services such as full connectivity and broadcast/multicast may be lacking.

Similarly, constrained network is defined as:

> A network where some of the characteristics pretty much taken for granted with link layers in common use in the Internet at the time of writing are not attainable.

And finally a constrained-node network:

Name	data size (e.g., RAM)	code size (e.g., Flash)
Class 0, C0	« 10 KiB	« 100 KiB
Class 1, C1	10 KiB	100 KiB
Class 2, C2	50 KiB	250 KiB

Table 1.1: Classes of constrained nodes based on available computing capabilities.

> A network whose characteristics are influenced by being composed of a significant portion of constrained nodes.

These definitions were created by IETF in 2014 and they are still current.

Constrained networks are also further specified with their constraints, which may include:

- low achievable bit rate and throughput,
- high packet loss and high variability of packet loss,
- highly asymmetric link characteristics,
- severe penalties for using larger packets,
- limits on reachability over time, and
- lack of (or constraints on) advanced services such as IP multicast.[9]

These constraints may be present only on some nodes and they can stem both from constraints of the connected devices as well as from constraints of the network itself.

The document then also differentiates between constrained nodes based on their available computing capabilities to three classes as shown in table 1.1.

This distinction is several years old now and the document acknowledges that the values may change with time, however it expects that most improvements will be directed to lowering power consumption, thus prolonging battery lives of the devices, and making the devices smaller.

Class 0 is envisioned to be simple sensors that will not be able to communicate on the Internet securely on their own. It is expected that these devices will only send raw data and react to the simplest of commands such as on/off and reset. Therefore this class of devices will not interest us in this thesis at all.

Class 1 is expected to be able to use protocols specifically created for their use such as CoAP. However they may not be able to use computationally intensive or bandwidth-heavy protocols such as TLS and HTTP.

Name	Strategy	Ability to communicate
P0	Normally-off	Reattach when required
P1	Low-power	Appears connected, perhaps with high latency
P9	Always-on	Always connected

Table 1.2: Classes of constrained nodes based on power consumption strategy.

Class 2 encompasses devices that have enough processing power to run the same protocols as laptops or servers. They may however still benefit from protocols with lower network footprint and energy consumption.

The same RFC 7228[9] further differentiates classes of devices based on their network usage. This is important because constrained node networks are almost exclusively wireless networks and the active or passive usage of the network modules has a huge impact on the energy usage and battery life. The table 1.2 shows the different strategies for devices, based on their power consumption strategy (hence classes P):

Class P0 is for devices that are usually turned off and do not send or receive any communications for most of the time. These devices always start new sessions when they are woken and do not try to resume older sessions because the assumption of prolonged sleep mostly guarantees that any sessions have already timed out. It is expected that the energy expended on creation of new connections and sessions would be offset by the prolonged sleep time and that this would be more economic than trying to reconnect. Upstream services usually also cannot contact the devices directly and must instead wait for the device to contact the service on its own, or another channel must be used.

Class P1 encompasses devices that communicate frequently but do not need to reply with low latencies. These devices can then also sleep for some periods of time, however only so that sessions and connections can be kept and any keep-alive and ping/pong messages can be sent and replied to in a timely manner. For upstream services these devices always look online but their response times can fluctuate wildly.

Class P9 includes all devices that are continuously connected and reply in a timely manner. This can consume very large amounts of energy and is recommended mostly for devices that do not have any constraints on energy usage, usually when they are mains-powered.

The LoRaWAN specification[10] specifies three different classes marked A, B, and C, which are somewhat similar to the latter distinction in RFC 7228[9]. This distinction is based on the network behavior of the constrained device. These classes are formally requirements for devices to be LoRaWAN certified, but the behvaviour can be generalized and may be of interest for this thesis.

Class A is called bi-directional end-devices. Devices in this class are required to listen for incoming packets only shortly after sending their own data.

Class B, bi-directional end-devices with scheduled receive slots, works in the same way as class A, but additionally also listens for network traffic at scheduled times.

Lastly class C, bi-directional end-devices with maximal receive slots, allow nearly continuously open receive windows.

1.2 Problems

All of the protocols discussed in this work provide some form of authentication. While the mechanisms offered by the protocols will be reviewed, the problem of bootstrapping credentials such as passwords or private keys to the devices will not be considered in this work. This is a complicated problem and its discussion and some ways to handle it can be found for example in [11] and [12].

Constrained devices also by definition do not have much processing power and therefore, any Internet protocol employed for communication should have small code size, not be computationally intensive, and use as little bandwidth as possible. The constraints can also affect the operational aspect of some security protocols as it may be more complicated to harvest entropy to generate sufficiently complex keying material or nonce values.

The communications of IoT devices are often done through an intermediary which can provide several services such as caching responses, batching operations, or translating requests between different protocols. This can often be a single point of failure and in case of compromise the blast radius can be much lowered by having an end-to-end integrity or confidentiality guarantees for communications between the device and the destination service.

Chapter 2

Protocol Descriptions

2.1 Constrained Application Protocol (CoAP)

The Constrained Application Protocol (CoAP) is a specialized web transfer protocol for use with constrained nodes and constrained networks in the Internet of Things. The protocol is designed for Machine-to-Machine (M2M) applications such as smart energy and building automation.[13]

Like HTTP, CoAP has servers make resources available under a URL, and clients access these resources using methods such as GET, PUT, POST, and DELETE.[13] Since then PATCH, FETCH, and iPATCH methods have been added.[14]

CoAP is currently defined in 13 RFCs and there are 14 further active Internet-Drafts. The original specification in RFC 7252[1] was released as an official Proposed Standard in June 2014 and has been worked on since late 2009.[15]

CoAP makes use of GET, PUT, POST, and DELETE methods in a similar manner to HTTP. The detailed semantics of CoAP methods are "almost, but not entirely unlike"[16] those of HTTP methods: intuition taken from HTTP experience generally does apply well, but there are enough differences that make it worthwhile to actually read the specification.[1] There are some differences in response codes and what should be expected as well as general as some calls like large updates to server resources, which may be split into multiple requests.

The protocol was originally designed to be run on UDP and therefore benefits from its lower overhead, but in cases where reliable messaging is required needs to define its own mechanisms for packet resending and timeouts, otherwise handled by TCP or other reliable protocol. Devices in networks that often fail can benefit from not having to do handshakes and in case of non-confirmable messages can simply send the message without much other overhead.

The headers and options are also optimized for size so every message must

contain only a 4-byte header, options such as path to the requested resource or query, and the payload itself prefixed with a single byte with all bits set. Each option is prefixed with one to five bytes specifying the option number. Therefore the smallest CoAP message is just four bytes long and a message with payload and no options would incur only five bytes of overhead. There is guaranteed MTU of at least 1280 in IPv6[1] and which makes CoAP very efficient.

2.1.1 Features

2.1.1.1 Reliability

There are two kinds of messages—Confirmable (CON) and Non-confirmable (NON). The former requires an acknowledgement message to be sent by the recipient back to the sender. If either the original or the acknowledgement is lost, the message is retransmitted until either acknowledgement is received or maximum retrying is reached. On the other hand Non-confirmable messages are sent with no expectation for confirmation of the message from the receiver. For higher chance of successful delivery the sender may send several copies of a non-confirmable message in a time slot. Multiple identical messages, caused either because of retransmissions or packet duplication in the network, are deduplicated based on their message ID which is part of the CoAP header.

Every confirmable request elicits a confirmable response and similarly every non-confirmable request can receive only a non-confirmable response. If Non-confirmable message is used for the request, the client must be prepared for the scenario that the request or the response are lost. It also cannot assume that the server did or did not receive the original request. This is useful mostly for sensors updating periodically their readings of non-critical data where simply waiting for the next value is preferable to potentially causing further congestion with retrying.

Because CoAP is expected to be used in highly constrained networks, default parameters of the protocol are set to be both simple and prevent congestion. With the default settings, each client can have only a single outstanding unacknowledged message for each server it communicates with. This doesn't mean there cannot be multiple parallel requests, but the next request can be sent only after the one before was acknowledged, even if there is no response message yet. Retransmission of confirmable messages uses exponential back-off with total time from sending the original message until giving up on receiving an acknowledgement message being 93 seconds with default parameters.

[1] There may however still be fragmentation on the link layer, for example 6LoWPAN has MTU of 127 bytes but has an adaptation layer to reconstruct any fragmented packets if necessary.

2.1.1.2 Piggybacked Responses

The acknowledgement packets of Confirmable messages are only four bytes in size, but to further save bandwidth they may be combined with a response when appropriate and when it is available early. This is useful for example for errors such as when a resource is not found or when the response is readily available and small. It may be preferable to not use this feature when both the request and response have large size because when a piggybacked response is lost both the original request and response are retransmitted. If normal acknowledgement was used, only four byte reply to the first request message would be sent which could improve the chances of it arriving, saving the original sender from having to retry the request.

2.1.2 Security

CoAP was originally built on top of UDP and relies heavily on Datagram Transport Layer Security (DTLS) to provide confidentiality and integrity guarantees. Since then the protocol was extended to also work on TCP, TCP with TLS, and WebSockets.[4]

It can use three security bindings of DTLS—using pre-shared keys, asymmetric cryptography with raw public key, or a certificate. In any case the keys are expected to be made available to the device beforehand. Implementation of `TLS_ECDHE_ECDSA_WITH_AES_128_CCM_8`[2] is mandated.

The CoAP RFC has its own security section[1] which references HTTP 1.1[17] for breakdown of security issues since CoAP realizes a subset of the features of HTTP 1.1. The section then talks about CoAP-specific security considerations.

If the protocol runs on UDP without DTLS any node can spoof requests and responses, including RST packets which would abort any request. Since there is no packet sequence number like in TCP, this vulnerability can be to some extent leveraged even if the attacker is not in a position for a MITM attack. The attacker can also spoof multicast requests which can drain resources in the target server faster or induce network congestion in the network. This may also be used to send traffic to a constrained network which is protected by firewall rules that allow inbound traffic from a set of machines.

There is also the risk of using CoAP servers in an amplification attack, where small requests with a spoofed source IP address generate potentially very large packets to the victim. This can be to some extent mitigated by the server not providing responses with large amplification factors to unauthenticated nodes, leveraging extensions of the protocol to send large responses in separate packets[1][2].

[2]TLS using elliptic curve Diffie-Helman ephemeral key exchange, elliptic curve Digital Signature Algorithm, and AES with 128 bit key size in counter mode with 8 byte CBC Message Authentication Code

2.2 Message Queuing Telemetry Transport (MQTT)

MQTT is a Client Server publish/subscribe messaging transport protocol. It is lightweight, open, simple, and designed to be easy to implement. These characteristics make it ideal for use in many situations, including constrained environments such as for communication in Machine-to-Machine (M2M) and Internet of Things (IoT) contexts where a small code footprint is required and/or network bandwidth is at a premium.[5]

The MQTT protocol runs on network protocols that provide ordered, lossless, and bi-directional connections. The specification names as examples TCP/IP[18], TLS[19], and WebSocket[20]. It also explicitly states that connectionless protocols such as UDP are not suitable because the data can be lost, duplicated, or reordered.[5]

Features of MQTT according to the specification [5] include:

- Use of the publish/subscribe message pattern which provides one-to-many message distribution and decoupling of applications.
- A messaging transport that is agnostic to the content of the payload.
- Three qualities of service for message delivery:
  - "At most once", where messages are delivered according to the best efforts of the operating environment. Message loss can occur. This level could be used, for example, with ambient sensor data where it does not matter if an individual reading is lost as the next one will be published soon after.
  - "At least once", where messages are assured to arrive but duplicates can occur.
  - "Exactly once", where messages are assured to arrive exactly once. This level could be used, for example, with billing systems where duplicate or lost messages could lead to incorrect charges being applied.
- A small transport overhead and protocol exchanges minimized to reduce network traffic.
- A mechanism to notify interested parties when an abnormal disconnection occurs. [5]

  The MQTT For Sensor Networks (MQTT-SN) specification[21] was developed and released by IBM. It is not an official OASIS standard but version 2 is being worked on to be released by OASIS. This document was developed to make MQTT more reliable on wireless sensor networks

where bandwidth is low and packet loss may be high. MQTT-SN is also optimized for running on low-cost, battery-operated devices with limited processing power and storage.[21] MQTT-SN relaxes the requirements on the network and requires only bi-directional data transfer service, therefore it can run for example on UDP.

2.2.1 History

The first version of the protocol was developed by IBM and Eurotech, Inc. in 1999. The standard was released in 2010 and submitted in 2013 in version 3.1 to Organization for the Advancement of Structured Information Standards (OASIS), a non-profit standards organization. Version 3.1.1 was released by OASIS in 2014. Next version is version 5.0, which was released in 2019. Version 4 was skipped over, because as part of the protocol the connecting party selects which version of the protocol is to be used and while the value `3` was used for version 3.1, version 3.1.1 uses `4`. The MQTT committee decided to mark the next version 5.0 to be consistent with this byte used in the CONNECT packet.

2.2.2 Features

There are two defined actors, client and server. The server is a central system with which multiple clients can communicate. Every message belongs to a so-called topic. Each client can publish messages and subscribe to these topics. When the server receives a message it checks whether there are any clients subscribed to the given topic and then routes the message to each of them. The client publishing a message to a topic acts as sender with the server acting as receiver and when the server then delivers the message to subscribed clients, the server acts as sender and the clients act as receivers. Both of these cases are identical in practice with the same messages, only the roles inverted.

The communication uses two terms - connection and session. Connection is a construct provided by the underlying transport protocol that is being used by MQTT for relaying data between client and server[5]. This can be safely regarded as a TCP connection or its equivalent in another suitable protocol. A session is a stateful interaction between a client and a server. Some sessions last only as long as the network connection, others can span multiple consecutive network connections between a client and a server[5]. The server should keep the session state for as long as the client has requested during connection. However, this is not guaranteed. The client can also keep its own session state to allow resending messages that were not acknowledged by the server possibly due to network failure. Keeping the state can save some resources because the process of resuming a session may be simpler than initiating a new one. Furthermore any messages that were received by the server during the time when the connection was down will be sent to the client

upon reconnection if the client has subscribed to the given topic. This way the client can to some extent also control whether it is interested in messages that were sent during the time when it was disconnected.

2.2.2.1 Publish and Subscribe

The protocol uses a topic-based publish–subscribe pattern where each client communicates only with the central server sometimes called a broker, which facilitates message exchange between all the other clients. Central unit inside the broker is a topic which is a path-like string with sections delimited by forward slashes to which each device can publish messages or it can subscribe to the topic. If a message is published to a given topic it is sent to every device that is at that time subscribed to that topic. This implies that if there is no subscribed client, any published messages are discarded upon delivery to the broker. This does not however mean that the client must be connected, it just needs to have an active session. All messages on the subscribed topic will be delivered upon reconnection, unless the session or the individual messages expire.

The protocol also allows subscriptions to topic filters with wildcards so that a device can subscribe to all topics with common structure, for example with the same first segment in the path. Another new feature in MQTT version 5 is shared subscription, where the published message is not sent to all the subscribed clients, but to only one. This can be useful in cases where all the subscribing clients are peers and messages should be processed in parallel but only once.

2.2.2.2 Will Messages

When a client connects to the server it may provide a so-called will message which should be saved by the server and sent to a given topic in case the client disconnects. There is also a configurable timeout during which the client can reconnect and thus cancel the process of sending the will message. This can notify other clients that the client has disconnected and may not be processing any further messages. Not receiving the will message does not imply that the device is connected though, because it is not normally sent on standard successful disconnect and even on failure there can be an unknown timeout before the message is sent.

2.2.2.3 Retained Messages

Clients can also publish retained messages which will be saved on a given topic and if a new subscribing client chooses so, they may receive the retained message upon subscription. There can be only a single retained message on each topic. If a new one is published there, the old one is deleted.

Sender Action	Receiver Action
PUBLISH QoS 0, DUP=0	
	Deliver Application Message to appropriate onward recipient(s)

Figure 2.1: MQTT actions by the sender and receiver in QoS 0 exchange.

2.2.2.4 Quality of Service (QoS)

There are three Quality of Service (QoS) options:

- QoS 0—at most once,
- QoS 1—at least once,
- QoS 2—exactly once.

Each QoS level has larger overhead than the one before.

These guarantees apply only to a single exchange between client and server. If a client publishes a message to a topic with exactly once guarantee it cannot be sure that every subscribed client will receive the message exactly once in the general case because the subscribing client may not support some of the higher QoS options or it may have chosen to subscribe with lower guarantees for performance reasons. Furthermore there may not be any clients subscribed to the topic and therefore no one to read the message. The sending client may be notified of this, but the server can omit this information.

2.2.2.4.1 QoS 0—At Most Once The lowest Quality of Service possible, useful when lowest overhead is desired and delivery of each message is not critical.

Sender sends the message in a PUBLISH packet and discards it. Receiver does not reply to it in any way. These messages are always delivered unless the receiver disconnects or restarts when the message should be received. In that case the message can be lost.

The whole exchange consists of exactly one message containing the data and no data is saved by the MQTT client or server.

2.2.2.4.2 QoS 1—At Least Once Medium Quality of Service option where delivery of every message is needed and possible duplicates can be either idempotently processed or identified and discarded.

Sender persists the message and sends it to the receiver in a PUBLISH packet with QoS 1. Receiver passes the message to any consuming applications and sends an acknowledgement in a PUBACK packet upon delivery.

Sender Action	Receiver action
Store message	
Send PUBLISH QoS 1, DUP=0, <Packet Identifier>	
	Initiate onward delivery of the Application Message
	Send PUBACK <Packet Identifier>
Discard message	

Figure 2.2: MQTT actions by the sender and receiver in QoS 1 exchange.

Receiver does not save any information about the message and therefore if any duplicates arrive it treats them the same way and does not discard them, therefore the consuming application may process duplicates multiple times. When the PUBACK packet is received by the sender it discards the message.

If the receiver disconnects when it should receive the PUBLISH packet the message is still kept with the sender which will resend it upon reconnection. If the message itself is delivered but the PUBACK message is not sent due to unexpected restart of the receiver's application or the original sender disconnects, the sender will send the original PUBLISH message again upon reconnection. In this case the message may be delivered more than once.

During this exchange at least two messages are sent and the sender is required to keep the message in its state until it is acknowledged.

2.2.2.4.3 QoS 2—Exactly Once The highest Quality of Service level where every message is guaranteed to be delivered exactly once but with a considerable overhead of a two-step acknowledgement process.

This process uses four different packets:

1. PUBLISH,
2. PUBREC—Publish received,
3. PUBREL—Publish release, and
4. PUBCOMP—Publish complete.

Sender persists the message and sends it to the receiver in a PUBLISH packet with QoS set to 2. The receiver stores the message passes it to any processing applications and sends confirmation of the receipt back to the sender in a PUBREC packet. After confirmation is received the sender persists the information that the message has been received and sends a PUBREL packet

signaling release of the message on the sender's side. The sender may now safely discard the contents of the message but it must still keep information about the ongoing exchange and the packet identifier of the original PUBLISH message. When the receiver reads the PUBREL packet it can discard the message and sends PUBCOMP, signalling the end of processing of the message. When this last message is received by the sender it discards all state pertaining to the exchange.

In case the two parties disconnect during the exchange only the packets sent by the sender (PUBLISH and PUBREL) are resent. If the PUBLISH or PUBREL message is lost due to disconnection, the sender resends them and the protocol works the same way as if nothing has happened. If the PUBREC message is lost, the receiver has already consumed the message and saved the identifier of the message. In case of reconnection, the sender will send the original PUBLISH message containing data again but the receiver will ignore the data and reply again with confirmation of receipt of the message. Then the protocol continues as usual again. In case the PUBCOMP message is lost, the receiver has already discarded information about the exchange but may still receive a duplicate message from the sender about the release of the message. Then, assuming everything until now has been correct, the receiver can infer that it has already processed the message in question and can simply send the last message again.

There are always four messages plus any retries during this exchange. The sender must keep the whole message at least until delivery is confirmed by PUBREC and must keep the state of the message along with its identifier until the end of the exchange. The receiver must keep the identifier and state of the message until it completes the exchange by sending PUBCOMP.

2.2.2.5 Flow Control

The client cannot control when it will receive messages and the broker delivers them usually as soon as they are available. The only option to limit these messages is by setting the Receive Maximum option which limits the number of QoS 1 and 2 messages that can be unacknowledged at any given time. Both the client and broker can set this limit independently to signal how many messages they are willing to process concurrently. If the limit is reached the sending party has to wait until some of the pending messages are acknowledged or completed and only then further publish messages may be sent.

2.2.3 Security

The MQTT specification[5] also has a (non-normative) chapter on the security of the protocol. The protocol is designed to run over TCP/IP and it is advised to use TLS.

Sender Action	Receiver Action
Store message	
PUBLISH QoS 2, DUP=0 <Packet Identifier>	
	Store <Packet Identifier> then Initiate onward delivery of the Application Message
	PUBREC <Packet Identifier> <Reason Code>
Discard message, Store PUBREC received <Packet Identifier>	
PUBREL <Packet Identifier>	
	Discard <Packet Identifier>
	Send PUBCOMP <Packet Identifier>
Discard stored state	

Figure 2.3: MQTT actions by the sender and receiver in QoS 2 exchange.

The specification lists a number of threats that solution providers should consider. For example:

- Devices could be compromised;
- Data at rest in Clients and Servers might be accessible;
- Protocol behaviors could have side effects (e.g. “timing attacks”);
- Denial of Service (DoS) attacks;
- Communications could be intercepted, altered, re-routed, or disclosed; and
- Injection of spoofed MQTT Control Packets[5].

It is important to note that the protocol is stateful and messages may be persisted both in client and server and steps should be taken that anyone with access to the hardware isn’t able to obtain the data.

The implementations should also be aware that it may run in a hostile environment and in these cases authentication, authorization, integrity, and confidentiality also have to be taken into account[5].

OASIS has also released a supplemental publication to serve as a guidance to integrate MQTT with the NIST Framework for Improving Critical Infrastructure Cybersecurity[22][23]. Other standards and security profiles must also be met for certain use-cases such as processing credit card payment information.

The CONNECT packets of the protocol support User Name and Password fields which may be used for authentication of clients at the server. These fields can be used for other purposes than basic authentication, they may also contain certificates or other forms of credentials. Another option is to use mutual TLS for authenticating the clients.

The protocol also supports enhanced authentication where the client can declare what authentication flow it would like to use with the server. These authentication mechanisms are not specified so the client and server should know beforehand what protocol they wish to use. Examples that can be used this way include SASL, Kerberos, and SCRAM challenge.

There is no mechanism in the protocol to authenticate the original client that published a message that was received on a topic. Therefore the users should also consider the possibility of a rogue client connected to the server and sending fake data to a topic. This must be prevented by correct authorization rules in the scope of topics.

Authorized clients can also in some cases cause restarts of subscribing clients by including some forbidden features that the server may not check for. This can happen for example when a rogue client publishes to a topic that contains a forbidden character in its name and another client is subscribed to the topic with a wildcard subscription. Then if the broker does not correctly check the topic name and does not reject the message, when it is forwarded to other correctly implemented clients they may disconnect from the broker because of the invalid topic name.

2.3 Advanced Message Queuing Protocol (AMQP)

Advanced Message Queuing Protocol is an OASIS and ISO-IEC standard. The protocol is intended for business messaging and allows all common behaviours and delivery guarantees.

The specification defines the wire-level encoding of all primitives and objects that can be used and transported in the protocol, communication over TCP, messaging layer, and transactional messaging[24].

2.3.1 History

The protocol originated in 2003 and was pioneered by JPMorgan Chase & Co. The company eventually reached out to other firms such as Cisco and Red Hat to create a working group, which later grew to have 23 members including for example Microsoft and VMware. Several versions of the protocol were released with the most widely used being 0-9-1 released in 2008, sometimes also referred to as 0.9.1. The working group became an OASIS section in 2011 and in that year version 1.0 was officially released.

The OASIS version is substantially different from the older ones[25]. This work specifically focuses on version 1.0.

2.3.2 Features

An AMQP network consists of nodes connected via links. Nodes are named entities responsible for the safe storage and/or delivery of messages. Messages can originate from, terminate at, or be relayed by nodes[24].

A node is any entity participating in the exchange of messages using AMQP. Multiple nodes may run on the same computer or even in the same application. Examples are queues in a broker, where each is considered a separate node.

A link is a unidirectional route between two nodes. A link attaches to a node at a terminus. There are two kinds of terminus: sources and targets. A terminus is responsible for tracking the state of a particular stream of incoming or outgoing messages. Sources track outgoing messages and targets track incoming messages. Messages only travel along a link if they meet the entry criteria at the source[24].

Communication between nodes is carried on an AMQP connection, which is a full-duplex reliably ordered sequence of frames. It is assumed connections are transient and can fail for a variety of reasons resulting in the loss of an unknown number of frames, but they are still subject to the aforementioned ordered reliability criteria. This is similar to the guarantee that TCP or SCTP provides for byte streams, and the specification defines a framing system used to parse a byte stream into a sequence of frames for use in establishing an AMQP connection[24].

Each connection may contain several sessions where each provides bidirectional, sequential conversation between two containers, where a container is a machine or a program that contains one or more nodes. Sessions provide a flow control scheme based on the number of transfer frames transmitted. Since frames have a maximum size for a given connection, this provides flow control based on the number of bytes transmitted and can be used to optimize performance[24].

A link is a unidirectional route between two nodes. A link attaches to a node at a terminus. There are two kinds of terminus: sources and targets. A terminus is responsible for tracking the state of a particular stream of incoming or outgoing messages. Sources track outgoing messages and targets track incoming messages[24].

Each connection can have several sessions. A single session can be simultaneously associated with any number of links. Therefore for sending messages in one direction, the applications need one connection, one session, and one link; for sending and receiving messages one connection, at least one session, and two links. There is a link (or a pair of links in case of bi-directional messaging) for each node recipient, which may be a device or a queue on the Internet. Links are named, and the state at the termini can live longer than the connection on which they were established. The retained state at the termini can be used to reestablish the link on a new connection (and session)

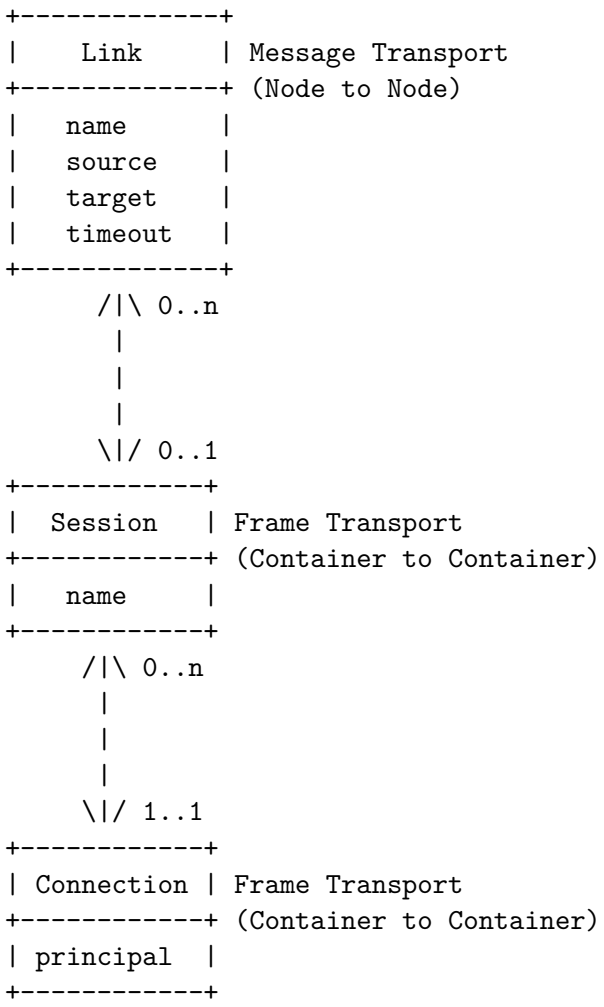

Figure 2.4: Relationship between links, sessions, and connections in AMQP.

with precise control over delivery guarantees (e.g., ensuring "exactly once" delivery)[24].

2.3.2.1 Transferring Messages

Messages are sent in AMQP frames with a transfer performative, which is a term that the AMQP specification uses for an object that dictates what the packet is supposed to do, such as opening a connection or transferring a message. Each message may be split into multiple frames which are individually sent and the original message is then reconstructed at the target. The state of the message is tracked at both ends of the exchange with either of them being able to change it. In normal operations, the receiver processes the mes-

sage and then notifies the sender that the message has been processed. The sender then does any operations needed such as notifying interested parties about message processing, cleans up all information pertaining to the message, and confirms the new state of the message to the receiver, while also settling it, effectively announcing that it has now forgotten the message. Any further messages from the receiver pertaining to the message would then be considered an error.

The sender may also change the state of the message on its own, for example if the message times out. If both the sender and the receiver wish to change the message at the same time, the sender's version is authoritative.

There are four terminal states of the message defined:

1. Accepted—the message was successfully processed,
2. Rejected—the message is malformed and will not be processed,
3. Released—the message was not acted upon and should be delivered again, and
4. Modified—the message has been modified and should be delivered again.

2.3.2.2 Quality of Service

All messages are reliably transferred if links are not detached unexpectedly. If links are detached (for example because the underlying TCP connection is interrupted), the current state of each ongoing message for that link should be kept at the terminus at the node. The link is then reattached, the source and the target compare the information they have about the transfer of the messages and restart, resume, or abort any of them.

The specification does not explicitly provide guidance as to how to implement given QoS level. It only provides hints for some decisions about redelivery on reattached links and if those must be done to guarantee for example the at least once semantics.

2.3.2.3 Flow Control

The protocol can be used for the transport of messages between two nodes where there is very little assumed about them. They can be peers exchanging messages, but usually, publish-subscribe pattern is implemented with distribution nodes serving as the central broker of messages. Operation and state of messages at the distribution nodes is also part of the specification[24].

The number of messages and when they are delivered can be controlled both on the level of a session and on individual links. The controls on links are especially useful when the consuming application wants to for example receive only a single message and no more until it has been processed or it wants to wait until a given number of messages are ready. The flow is controlled by the

sending node advertising number of messages it can send at a given time and the receiving node giving the sending node link-credits, where each credit can be spent to deliver one message.

2.3.2.4 Multiplexing

Since there can be multiple sessions and ptentially many links in a single connection, multiple devices can be multiplexed onto a single TCP connection[26]. This may be advantageous in cases for IoT gateways that communicate with brokers on the Internet on behalf of some devices. These devices can then communicate with the gateway with simpler protocols. For this feature to work an extension of the protocol[27] is needed for authentication of the devices.

Chapter 3

Experiment Design

3.1 Common Design

The experiments were performed on two different devices. One was more constrained where a custom firmware had to be built and flashed on it, the other was running the Linux OS.

3.1.1 Platforms

The first device was ESP-WROOM-32 with integrated 2.4Ghz WiFi module. It uses Xtensa architecture on its two low-power 32-bit 240Mhz LX6 microprocessors and provides 520 kB of RAM[7]. Devices with ESP32 or ESP2866 WiFi chips are cheap and often used by hobbyists for personal projects as well as for industrial applications that need WiFi connectivity. This device type is fairly constrained while still having a wireless network interface and being easily obtainable. There will be no operating system and the code must utilize the hardware directly.

The second device on which experiments were run is Raspberry Pi 3b+[8] running Manjaro OS. These devices are much less constrained, but still widely used for various IoT projects. All standard functions of the Linux operating system will be available. The device has a 1.4Ghz 64-bit quad-core ARMv8 CPU, 1 GB of RAM, and both 2.4Ghz and 5Ghz WiFi capability.

All experiments will be run against brokers run in Docker on a personal computer with no other significant load. The device with the computer will be in a local network with a router with dual-band WiFi. The computer where brokers and servers will be running will be connected through the 5Ghz band with hroughput of 867 Mb/s. ESP32 will connect to the 2.4Ghz WiFi band with 300 Mb/s speed. Raspberry Pi will be connected with an Ethernet cable. As each device will be connected through a different channel they should have minimal influence on each other. Some interference with other wireless networks in the vicinity can still occur though.

3.1.2 Experiments

The experiments will mainly test properties of the protocol under load by:

- Sending a very high amount of small messages—this test should measure the performance of the protocol in a setting where many small messages are sent. This can be the case when there is a lot of data generated by a device such as a sensor and all data are to be sent for processing. Possibly a device can also cache data and then send a large number of messages at once to save power by not being continuously connected to the network.

- Sending large messages—here we test for the other possible mode of operation, that is sending large amounts of data at once that are packaged in one message. This can either be a single message like a picture from a photo trap or it can be an aggregate message with multiple buffered readings.

The first test will be performed by sending 65536 messages with a 2-byte random payload. The second test will be done with a single 65536-byte message.

We will measure the time it takes to finish these tasks. We can assume that most of the inefficiencies in performance come from the constrained devices, not the brokers.

Each test will be run 100 times on the Raspberry Pi or 20 times on ESP32 in similar conditions to diminish any interfering effects or to split them between all protocols as evenly as possible.

Qualitative analysis of some exchanges will also be done:

- Sending several messages, capturing these packets with Wireshark, and examining the contents and sizes.

Bandwidth used by the protocols will be measured as network efficiency is also an issue that constrained nodes must take into account. This will not be done several times but instead, it will be evaluated qualitatively by examination of individual packets through Wireshark. The focus will be placed on two aspects of the protocols, the connection phase and the data transfer itself. Both the number of exchanged packets as well as the size on the wire will be examined.

3.1.3 AMQP

The protocol itself is symmetrical, so the speed should not differ depending on whether the device is client or broker. But logically the device will always be a client connecting to a broker to send messages to some queue. Therefore in the experiments, the device will publish messages to a broker.

The implementation to which it will connect is RabbitMQ 3.9.11 run in a Docker container on a personal computer in the same local network as the tested device. This broker uses by default the AMQP 0-9-1 protocol, but there is also a plugin for version 1.0 which will be used. As Quality of Service is not integrated into the protocol, simple publishing of messages with an acknowledgement from the broker will be done, effectively resulting in an at least once guarantee.

3.1.4 MQTT

In the case of MQTT, there is no reason for a constrained device to serve as a central broker with which all other devices communicate.[3] In all experiments the device will be the one connecting and sending publish messages to the broker. We will measure all three settings of QoS in individual experiments.

The broker will be EMQ X 4.3.10, an open-source implementation of MQTT broker supporting version 5.0. It also contains a plugin for translating CoAP requests to MQTT publish or subscribe messages which will be useful in the last test. The broker is again run in Docker on a personal computer in the same local network as the tested device.

3.1.5 CoAP

We will let the constrained device take the role of the client. In some instances, an IoT device may also be a CoAP server but this configuration generally needs a continuous connection to the network throughout its lifetime. On the other hand, a client can enter long periods of deep sleep to save power otherwise expanded on the network interface. There is a tradeoff between responsiveness and power saving based on the frequency of uploading data, polling for configuration changes for the client, and other communications. Furthermore, CoAP has an Observe option which lets the device get any updates if continuously listening to network traffic is not an issue. Thus we conclude that the device acting as a client in the CoAP protocol is generally more favourable and therefore that is the role in which the device shall be tested.

We will measure both non-confirmable and confirmable messages. For confirmable messages, we will focus only on piggy-backed responses (responses embedded in the ACK packet) because all scenarios are about the device sending data to the server. Therefore it makes no sense for the server to acknowledge the request and send a response in a separate message.

[3] We can think of an exception to this where an edge device like Raspberry Pi runs the broker for all local devices. In this section, we are however considering primarily more constrained devices and Raspberry Pi itself as a device serving as an IoT device and not an edge computer.

A custom server was built using Golang and the Go-CoAP library[28]. It is very simple as it only accepts data on a given path and sends a success indicating result code. It can serve CoAP over UDP, DTLS, TCP, and TLS.

3.2 ESP32

The tests were written in Rust, which is a systems programming language focused on three goals: safety, speed, and concurrency[29]. There is support from the community to use Rust in embdedded systems with documentation being available on the internet[30]. There is also guidance on how to flash Rust programs onto ESP32 boards[31].

In the Rust ecosystem, published libraries executables are generally referred to as crates. This work will follow this naming, so the word crate used in this work in the context of Rust will always mean library.

3.2.1 HTTP

Bindings to the C implementation of HTTP in Espressif IoT Development Framework (ESP-IDF)[32] were used. This feature is very crude and in an early development stage in Rust, but the underlying implementation is stable. As the test is run in a single thread without a support of an operating system, each request is sent sequentially after the previous one has been responded to. Both HTTP and HTTPS were tried.

3.2.2 AMQP

Attempts to find a suitable library to use on the selected device as a library for the AMQP client were largely unsuccessful. There exist a lot of different libraries in many programming languages, however, none were found that would make an effort to run directly on bare metal without basic OS services such as threading, except for one C library.

First, the Rust language was tried, where there are many open-source libraries both for the 0-9-1 and 1.0.0 versions of AMQP, however, all depend on a library called Metal IO (mio)[33] which cannot be currently run on the xtensa architecture[34]. Next, was tried using the C language with a library called uamqp[35]. Although there is a system for components in the official ESP-IDF[32], it was too complicated to repurpose the CMake files needed for a successful build and then also find and fix all transitive dependencies. Then, MicroPython[36] was tried. It is a smaller Python runtime designed to work on embedded devices with constrained resources. Two AMQP libraries were tried—amqp[37] and pika[38]. Both needed some minor adjustments because they relied on functionality not provided in standard MicroPython packages, but eventually, both failed with allocation errors during initialization. After that, Espruino[39], which is marketed as JavaScript for microcontrollers, was

attempted to be used. This ended up the same way as Micropython, because there are no AMQP packages created for microcontrollers and the ecosystem does not support simple usage of standard packages that can be found for example in npm. Lastly, a version of .NET runtime called Nanoframework[40] was found. This runtime should be able to run managed C# code on controllers. This ecosystem also has its own packages created for the purposes of embedded devices. It is in part developed by Mircosoft whose several Azure services leverage the AMQP 1.0 protocol[41] and therefore there are libraries ready for interoperation with some of these services. However, documentation on how to properly use the Nanoframework is somewhat sparse and the flashed managed code was unable to start properly on the ESP32 device.

Therefore this work concludes that using AMQP on severely constrained devices without the ability to run a standard operating system should be discouraged because there is simply no community to support this scenario. That said there does not seem to be a technical problem that would prevent implementation in any language if it was needed.

3.2.3 MQTT

The implementation for MQTT tests uses the Rust programming language and a library called MiniMQ[42].

This library is specifically designed to run on bare metal, in devices with no OS support, and also without the standard library. It does not even depend on mechanisms like heap allocations which introduces a trade-off where the user must declare during compilation time some constants such as sizes of buffers and how many messages can be in-flight. The library supports asynchronous operation by the way of making the user poll for responses and getting messages and ackonwledgements in return. Since all unacknowledged messages must be stored in memory this allows for some, but not many, messages to be in transfer at the same time. The experiments will use at most 64 messages in transit in at the same time. All messages are sent through the same TCP connection.

The library uses an abstraction of the network stack which currently supports only unencrypted UDP and TCP sockets, therefore encrypted transfers were not tested on this device.

A notable implementation in C that could have been used is the MQTT implementation that is present in the ESP-IDF. However this implementation uses only versions 3.1 and 3.1.1 of the MQTT protocol, therefore it could not have been used for this work.

3.2.4 CoAP

Originally some open-source fully functional library was supposed to be used. However, some of them (like the library called coap) also relied on the mio and

tokio libraries for handling the asynchronous nature of the protocol. The only one which has been found that used other mechanisms used a crate for asynchronous operations that uses too much memory on stack resulting in stack overflow in any program that tried to send CoAP requests on a constrained device. Therefore the coap_lite crate without dependencies on any standard library functionality was used for serializing and deserializing the messages and the protocol itself was custom-written.

The CoAP tests were implemented with the use of the coap_lite[43] crate for Rust. This library provides all necessary functions for manipulating CoAP messages and is intended to serve as a building block for more complex libraries which can handle all the messaging concepts and resend rules and thus extract some of this work. Because of the relative simplicity of CoAP when compared to the other protocols this was not an issue and the basic functionality was written in about 150 lines of code.

Because the protocol allows only one unacknowledged request at a time the messages will be sent strictly sequentially. Only when the previous message has been responded to will the next message be sent to the server.

Only the UDP transport has been tried on ESP32.

3.3 Raspberry Pi 3 B+

Tests for all three protocols in question will be run on Raspberry Pi 3 Model B+ which is a single-board computer with 1.4Ghz 64-bit quad-core ARMv8 CPU, 1 GB of RAM, dual-band WiFi, and Gigabit Ethernet over USB 2.0[8]. It also provides four USB ports and forty pins out of which twenty-eight are GPIO with support for SPI, I^2C, etc. This type of board is much less constrained than the ESP32 devices but it is still often used for IoT solutions that need high computational power or a standard Linux environment into which the developer can connect for example through SSH. This comes with much higher power consumption so it is not suited for battery-powered applications. Using the Linux system also introduces considerably longer boot time until the board can execute its intended function. This can be tweaked to some extent by disabling some unneeded services like Bluetooth or WiFi (when connected through Ethernet) but it may never be suitable for some very time-sensitive operations like a camera photo trap. For the tests, Manjaro OS for ARM which is based on Arch Linux will be used.

The Python programming language has been used for all three protocols on this device. This language is very well known and regarded for its simplicity and for that reason is used on several projects, both by hobbyists and IoT companies. The source code in attachments also include `pyproject.toml` and `poetry.lock` files that define dependencies and their versions so that the projects can be easily rebuilt with the same versions.

3.3.1 HTTP

HTTP requests which would be used as a baseline were done with the aiohttp library[44]. This is a library that can send many requests in parallel and asynchronously wait for the responses, no other advanced functionality of HTTP was needed in the experiments. No limit was imposed on the library and all messages were asynchronously enqueued for sending as fast as possible.

3.3.2 AMQP

The Qpid Proton library[45] was used for AMQP implementation. This library is maintained by Apache and it is written in C with bindings to several languages, such as Python. It supports AMQP 1.0 with all its standard functionality.

The library provides very granular support over handling of events in the protocol through callbacks. These events include for example when the other peer accepts the message which is used to count the number of successfully trasfered and accepted messages. The code as written has at least once semantics when publishing. It only sends a predefined number of messages and then counts acknowledgements with callbacks in a separate thread. All messages are sent through the same TCP connection.

3.3.3 MQTT

For MQTT tests, the Eclipse Paho MQTT Python Client library[46] was chosen. This library is written in Python and is not a wrapper around C functions and therefore some operations can be slower because of garbage collection and other runtime overhead. It supports protocol versions 3.1, 3.1.1, and 5.0 out of which only tests with the latest version, which has been discussed in the second chapter, were run. All levels of Quality of Service are supported and all have been measured individually.

This library also provides a way to register callbacks for events such as finishing publishing a message. These callbacks were used to count the number of published messages similarly to the AMQP experiment.

3.3.4 CoAP

Originally, tests that simulate a CoAP client were first created using the aiocoap[47] a Python native library using `asyncio`—a library for running concurrent IO-bound Python code. It implements all major CoAP functionality and several of the extensions that were added to the standard later.

However this library and any other popular Python CoAP did not seem to support TCP and full DTLS. The Python library however does not support TCP. Therefore, all tests were rewritten in Golang with the Go-CoAP library[28].

Tests with UDP and DTLS were limited to only one concurrent request as is the default for CoAP over UDP. Each request was sent only after the previous was settled. CoAP over TCP does not limit the number of concurrent requests as congestion control is not managed by CoAP, but rather by TCP. Therefore, in these tests asynchronous tasks are spawned and executed in parallel where each sends one request. There can be up to 1024 concurrent requests. This number was used because further increase seemed to not have any effect on the speed anymore while some higher configurations ran into issues with memory and thread scheduling. All requests are sent through the same socket.

Chapter 4

Results Evaluation

4.1 Expected Results

The three protocols serve very specific purposes. While CoAP shines in very simple scenarios with the request-response pattern, MQTT and AMQP are designed for messaging patterns with a central broker. MQTT is also better suited for messages that are to be routed directly to the subscribers while AMQP has found uses in more standard queue-oriented setups. Another big difference is AMQP's ability to control the number of incoming messages with credits while MQTT only has a quota set during the connect phase for how many messages can be in-flight at any given moment. On the other hand, the ability to set the desired and supported QoS of MQTT gives more control to the developer where the same behaviour would be largely dependent on the implementation of the broker in AMQP.

Therefore the chosen protocol for a given solution should be based first and foremost on the scenario and guarantees. As we have seen, CoAP is generally very simple to implement and use if there is no need for its advanced features. MQTT also has great support from the IoT community with many implementations aimed at various constrained devices. Usage of AMQP for IoT solutions is harder because there is not much support for it and the protocol itself is fairly complicated. While Microsoft uses in several of its Azure services AMQP 1.0[41] and Red Hat publicly supporting the newest version[48] the community still uses AMQP 0-9-1. This can be illustrated by the fact that although there are more repositories for AMQP 1.0 on GitHub, the most popular repository[49] has 322 stars and only four have more than 100 stars while the most started repository for AMQP 0-9-1 is a Go implementation with 4161 starts[50] followed by implementations for Node.JS and Python, both having around three thousand stars. There is about the same number of repositories that have more than one star. Another example is RabbitMQ which can use both versions, but it uses only the older version by default and the newer version is supported as an opt-in plugin. Therefore there should be

strong reasons to introduce AMQP to a very constrained IoT solution because it carries some risk and complexity.

Of the three protocols, CoAP should have theretically the lowest overhead. Not only can data be sent with only a five-byte protocol overhead (not including options such as URI path) but it can also run on UDP which has a smaller header than TCP and for spurious transmissions also saves bandwidth because there are no handshake or termination messages. Handshake however is present if DTLS is used. Usage of UDP also brings some drawbacks such as shorter NAT timeout times[51][52][53], speed of transfer in good networks[54], and implementation of congestion control in the application instead of the transport protocol. CoAP also supports TCP transmissions which have a minimum of three- to seven-byte overhead depending on the length of the sent data[4].

MQTT should have fairly small overheads for the PUBLISH messages, but compared to CoAP the connection establishment may be somewhat expensive. It is however simple if no advanced features of the protocol are needed. This can be suboptimal if the device should sporadically connect to the broker and send only one or a few small messages.

The most complicated connection establishment is that of AMQP where first a proto-header stating protocol name and version is sent (8 bytes), then each party needs to send an Open packet, Begin packet, and Attach packet, each having 8-byte headers and some data. Each transfer message then needs at least 30 bytes of additional overhead just for AMQP on each transfer packet. All these packets then also need to be acknowledged and in most scenarios settled with further messages in each direction.

4.2 Size on the Wire

For this experiment only 1024 messages were transmitted, each message contained two bytes of data. The exchanges were tracked using Wireshark and the corresponding packet capture files are available in the enclosed memory. Only exchanges not using TLS or DTLS were saved as the captures of encrypted communications provide very little outside of the numbers discussed in this section.

The focus will be placed on the number of bytes transmitted including all network headers. Next to that number of packets will be also examined. This metric is of interest because it indirectly shows the effectiveness of the transfer. Each transferred TCP packet has an overhead of at least 54 bytes of headers[4].

All conversations were captured using Wireshark which has some limitations such as not being able to easily decode TLS communications and not being able to capture all transferred bytes, namely Frame Check Sequence

[4]Ethernet header is 14 bytes, IP header is 20 bytes, TCP header is 20 bytes plus options.

	Bytes transferred (B)	Packets transferred
Theoretical minimum of HTTP	140460	98
HTTP	500304	4107
HTTPS	565768	4113
CoAP over UDP	110464	2048
CoAP over UDP with DTLS	197150	2054
CoAP over TCP	115954	1451
CoAP over TCP with TLS	187277	1765
MQTT QoS 0	13172	34
MQTT QoS 1	61438	764
MQTT QoS 2	114932	1559
MQTT QoS 0 with TLS	39416	66
MQTT QoS 1 with TLS	129902	1166
MQTT QoS 2 with TLS	251094	2369
AMQP	88058	183
AMQP with TLS	90558	181

Table 4.1: Comparison of transmitted bytes and packets for 1024 messages consisting of 2 bytes each with different protocols.

(FCS) of the Ethernet packets. Therefore the figures shown may differ from actual numbers of transmitted bytes on the wire by 4 bytes per packet. This is however consistent across the protocols and mostly negligible and therefore it will not be considered beyond this point.

4.2.1 Protocol Results

4.2.1.1 HTTP

The time it took to transmit HTTP packets was rather long. The packets themselves were fairly large as both the client library as well as the server sent several headers along with the HTTP messages. Furthermore, the client library is optimized to use several concurrent connections which may be beneficial in other scenarios for multiplexing responses but here it considerably added to the overhead of the TCP protocol. For this experiment the library was configured to use only one TCP connection to not pollute the results with multiple three-way handshakes and other overhead.

HTTP is notoriously very verbose with all of its headers and options being sent in ASCII. The client library also always sent only the headers in one packet and after the server acknowledged them it sent the two bytes of payload in a separate message. Therefore each request consisted of two packets, one

with 191 bytes of payload which contained just the headers, HTTP method, and other protocol information, and then another containing two bytes of payload. Each of these incurred 54 bytes of network overhead because of Ethernet, IP, and TCP headers. Furthermore, the packets for acknowledgement from the server took another 54 bytes on the wire. Then the server sent 129-byte response, out of which 75 bytes were the HTTP response. The response itself contained no payload, only the status code, and headers. Then the client acknowledged this message in a separate 54-byte TCP packet.

This behaviour would be highly inefficient in IoT applications. If HTTP should be used in a constrained network, special care should be placed on correctly configuring all relevant options.

If the client used as little bandwidth as possible, it would have needed 25 bytes for declaring HTTP verb, version, and path, 19 bytes for the content-length header, 2-byte CR LF to signal the end of headers and the start of the payload, and finally 2 bytes of user data. In total every HTTP request would need at least 48 bytes plus network headers for a total of 102 bytes per request if every request was sent individually. Each response needs 17 bytes to declare HTTP version and status code, 19 bytes for content length[5], and 2-byte CR LF to signal the end of headers. This is a total of 36 bytes per response plus network headers.

If we assume MTU of 1500 and buffering both requests and responses in as large packets as possible, there would be at least 58 TCP packets containing all the requests and at least 33 TCP packets containing responses. Therefore including the three-way and four-way handshakes, there would be at least 140460 bytes transmitted in 98 packets.

4.2.1.2 CoAP

4.2.1.2.1 CoAP over TCP As TCP is connection-oriented, CoAP adds one message at the beginning of the connection. That is the Capabilities and Settings Messages (CSM), where both the client and server declare options and extensions that they would like to use. Both the client and server sent a CSM message with no options which amounted to 64 bytes in each direction, where 54 bytes were network headers.

Then each transferred message was 73 bytes long, that is 19 bytes without network headers. Out of these 2 were the CoAP header, 8 were for the token that is used for identifying response packets to the correct requests, 6 were options for the message, then 1 byte is used to mark the beginning of the body and the last 2 bytes were the payload itself. Each of these messages then needed a response that contained no data, only a status code. Along with that it also needed a 2-byte header and an 8-byte token to pair the

[5]This header is not actually mandatory, but when it is omitted, the client should consider everything received from the server as part of the response until the connection is closed.

response with the correct request. Each reply was 10 bytes without network headers, 64 bytes with Ethernet, IP, and TCP headers.

It is important here to note that the 8-byte token is the longest that it can be and this length was chosen by the library. Each message can contain as short as 1-byte tokens as long as they are not reused until the requests time out. Therefore in our experiment with 65536 messages tokens with two bytes (or three since the CSM message also needs a valid token) would be sufficient.

We can see that TLS adds some overhead to the bytes transferred but not to the number of packets.

4.2.1.2.2 CoAP over UDP The number of transferred bytes is lower with UDP. This is in part due to smaller UDP headers compared to TCP, but also because the Python implementation used for this used only 2-byte tokens, saving 12 bytes per each request-response pair, totalling at 12288 bytes. The shorter header itself may also not provide that large benefit since, unlike TCP, each UDP packet can contain only a single CoAP message. Therefore exactly 2048 packets were sent, 1024 with a request, 1024 with a response. This means that there are more packets sent, each needing its own headers. This is furthermore illustrated with the use of DTLS where there are 6 more UDP packets for DTLS handshake. In total, the difference between TCP and UDP network headers was less than 3 % because the shorter length of UDP headers was balanced by the higher number of packets.

The messages themselves were mostly the same as with TCP except that there is no deduplication in UDP so each CoAP message contains 2-byte message ID for that purpose. Confirmable and non-comfirmable messages are identical except for a single bit and therefore only one result is shown in the results table.

4.2.1.3 AMQP

The client sent a total of 245 bytes in AMQP proto-header with the protocol and version declaration, Open, Begin, and Attach packets to estabilish a connection, session, and link respectively, all of which were sent in a single TCP packet, therefore with further 54 bytes of network headers. RabbitMQ responded to each of these messages in a separate packet, sending a total of 708 bytes including all headers. In the last TCP packet, there was also an AMQP flow packet which is needed to set up flow control so that the client may start sending data.

Then the client started sending messages with an overhead between 50 and 64 bytes per AMQP message, depending on some numbers sent in the AMQP headers and their variable length encoding. These messages were usually sent in bulk so TCP overhead is negligible in this experiment, although in normal circumstances the messages would be likely not sent at the same time. RabbitMQ then acknowledged the messages dispositions packet, often sending

each individually in a TCP packet. However, a single AMQP disposition message may acknowledge multiple transferred messages at the same time with no further overhead. Each disposition packet has 31 bytes, not including the TCP overhead.

Therefore opening up a connection until the client could send any useful data consisted of sending 245 bytes to the broker and receiving 708 bytes. Then each individual message incurs an overhead of 104 bytes per message in one direction and another 85-byte confirmation, assuming each message is sent individually.

We can see that there is only a slight increase in transferred bytes when using TLS. The number of packets is also very similar between the encrypted and unencrypted versions. This number can however fluctuate and in some experiments got as low as 120. This is because in some circumstances the broker can send several disposition packets in the same TCP packet.

4.2.1.4 MQTT

To initiate a connection the client sent a 73-byte packet and the broker responded with a 75-byte packet, both figures including all headers. Then each PUBLISH message was sent with only 9-byte overhead for QoS 0 and 11-byte overhead for QoS 1 and 2. For QoS 1 there is also a 6-byte (60 including network overhead) response, for QoS 2 there is also 6-byte PUBREC, 4-byte PUBREL, and 6-byte PUBCOMP.

There is a significant increase in transferred bytes when encrypted using TLS. Possible causes for this increase are higher packet fragmentation and padding in TLS, especially for short messages such as PUBACK, but it was not feasible to examine the unencrypted packet contents due to the limitations of the used tooling.

4.2.2 Comparison

We can see that all of the protocols behaved much better than HTTP with default settings. Every unencrypted version of the protocol used smaller bandwidth than is the theoretical minimum that could have been used by HTTP. Only CoAP over TCP used about 3 % more bandwidth than the theoretical minimal HTTP client. This is due to the number of transferred packets. While we have assumed perfect usage of MTU by the HTTP client and server with a total of 98 packets, the CoAP client sent and received 1636 packets. This is an overhead of at least 88344 bytes in network packet headers, which is about 61 % of the bytes transferred in the test.

There was an expectation for CoAP over UDP to have much better performance, but in the setting of many messages sent at the same time, the advantage of a smaller L4 header is offset by the need to send each individual message in its own packet as UDP is packet-oriented, unlike TCP which is

connection-oriented. While the CoAP over UDP performed a bit better than CoAP over TCP, these tests were done with different libraries, and the TCP implementation suffered from larger token size and imperfect buffering of messages into fewer packets resulting in a large number of packets as discussed earlier.

On the other hand, we can see that MQTT with QoS 0 sent the lowest number of packets. This is because it sent all 1024 PUBLISH messages in just 14 TCP packets. Since there is no response from the broker for this QoS most of the other packets are TCP ACK packets and connection establishment.

AMQP used a surprisingly small amount of packets. On closer inspection, it however sent TCP packets as large as 13194 bytes even though MTU was configured to 1500. This means that it ran a great risk of fragmentation on the network layer which may be dangerous in unstable network since losing a single Ethernet frame can mean retransmission of all fragments. While it slightly lowered the results in this experiment, the potential overhead on retransmission is much larger. But even if the AMQP implementation used more packets it seems to have a lower footprint than CoAP. It is also worth noting that if the developer also optimized the broker to wait with the disposition packets to acknowledge data transfer the overhead would be significantly lowered as the broker could potentially send just a single disposition packet in response. This would however increase the risk that a message may be delivered more than once.

MQTT with QoS 1 seems to have performed better than AMQP since even though there were several times more packets in the exchange, the overall bandwidth was lower. However, MQTT has a strange increase with a factor of more than 2 when used with TLS.

MQTT with QoS 2 used much more bandwidth than QoS 1 and this should be taken into account when deciding what QoS is needed for messages. The bandwidth is still lower than both CoAP with TCP and HTTP.

Since AMQP and MQTT with QoS 1 are the most similar protocols, it is worth focusing part of the comparison on these two cases. It is visible that MQTT sent fewer bytes but it sent a lot more packets. This is because although both of the protocols need to acknowledge the messages, AMQP can send a disposition packet that acknowledges multiple packets at the same time. It also sent some packets that were much larger than expected. Furthermore, it seems that the MQTT broker did not buffer the acknowledgement messages to send them in a bulk in a single TCP packet. But even with this inefficiency MQTT with QoS 1 sent fewer bytes on the wire.

MQTT has used the shortest messages with QoS 0 PUBLISH messages being only 11 bytes and higher QoS being 13 bytes long. The length of the CoAP messages is similar to that with the length of 19 in our experiments. As discussed previously this could be improved upon by using tokens shorter than 8 bytes. The token length also influences the length of the response which is 64 bytes for CoAP and 60 bytes for MQTT in our experiments. Therefore

	Mean transfer time and standard deviation (s)
HTTP	281.274 ± 3.19
HTTPS	313.013 ± 2.743
CoAP over UDP	239.182 ± 17.443
CoAP over UDP with DTLS	289.001 ± 24.316
CoAP over TCP	8.916 ± 0.549
CoAP over TCP with TLS	12.778 ± 0.613
MQTT QoS 0	18.293 ± 1.318
MQTT QoS 1	83.2529 ± 1.254
MQTT QoS 2	150.997 ± 1.314
MQTT QoS 0 with TLS	16.614 ± 0.681
MQTT QoS 1 with TLS	82.0632 ± 1.539
MQTT QoS 2 with TLS	152.580 ± 9.777
AMQP	78.187 ± 1.146
AMQP with TLS	78.831 ± 4.286

Table 4.2: Comparison across 100 measurements on Raspberry Pi of transmission times of 65536 messages in seconds with different protocols.

with better settings for tokens MQTT with QoS 1 and CoAP would have used very similar bandwidth and most of the difference would then come from the network stack usage and buffering.

4.3 Protocol Speed

4.3.1 Protocol Results

All discussed measurements were measured on the Raspberry Pi and with the libraries discussed in the previous chapter unless stated otherwise.

4.3.1.1 CoAP

There is an extremely large difference between transmission times when using CoAP with UDP and TCP. This is because CoAP defines very crude congestion control[1]. That is because the protocol should be very simple and therefore the congestion rules were created to be simple to implement. It was

	Mean transfer time and standard deviation (s)
HTTP	0.02487 ± 0.00178
HTTPS	0.04858 ± 0.00759
CoAP over UDP	0.19635802 ± 0.03137
CoAP over UDP with DTLS	0.66193877 ± 0.08757
CoAP over TCP	0.019998632 ± 0.00228
CoAP over TCP with TLS	0.034399883 ± 0.00866
MQTT QoS 0	0.10306 ± 0.00034
MQTT QoS 1	0.10316 ± 0.00055
MQTT QoS 2	0.10316 ± 0.00066
MQTT QoS 0 with TLS	0.103
MQTT QoS 1 with TLS	0.103
MQTT QoS 2 with TLS	0.10305 ± 0.00041
AMQP	0.091985 ± 0.00855
AMQP with TLS	0.110612 ± 0.01363

Table 4.3: Comparison across 100 measurements on Raspberry Pi of transmission times of one message with 65536 bytes in seconds with different protocols.

	Mean transfer time and standard deviation (s)
HTTP	903.667 ± 23.721
HTTPS	1059.431 ± 29.931
CoAP over UDP confirmable	1716.413 ± 123.643
CoAP over UDP nonconfirmable	1732.160 ± 147.458
MQTT QoS 0	16.650 ± 2.075
MQTT QoS 1	645.518 ± 25.807
MQTT QoS 2	985.738 ± 27.363

Table 4.4: Comparison across 20 measurements on ESP32 of transmission times of 65536 messages in seconds with different protocols.

also designed for very constrained networks and therefore the protocol reacts more severely than it could in normal settings.

When a request or response is lost in UDP the protocol waits at least one second to resend the request and then continues with exponential back-off. TCP uses advanced mechanisms to speed up retransmission such as fast retransmit[55]. Furthermore, CoAP in default settings allows only one pending request from a client to a given server. Therefore while the CoAP client can send as many parallel requests over a TCP connection as the link can handle, it waits with each UDP request.

4.3.1.2 MQTT

The results show that the time grows based on the chosen level of QoS. This is expected as each of the guarantee levels has more packets that need to be transmitted over the network. The difference between QoS 0 and QoS 1 is very large because the client must not only send the data but also store it in case that it must be retransmitted later. Therefore based on the implementation, the data may be copied on the disk or in memory and a structure holding IDs of unacknowledged messages must be kept up to date. The difference between QoS 1 and QoS 2 is also considerable as the former needs a single round trip per message while the latter needs two round trips. A probable explanation for such an increase in time is provided by an examination of the traffic with Wireshark, where we can see that the broker sends confirmation messages in individual packets instead of sending multiple in a single packet. This can strain the network as the size of the headers is much larger than the messages themselves.

We see an even bigger performance drop on the ESP32 where the MQTT client must save the message for later potential resend which includes copying of the data in the chosen implementation because there is no memory heap. Due to the constrained nature, there is also a limit to the number of messages that can be saved this way and further messages cannot be published until at least some of the preceding ones were acknowledged. This also prevents the network stack to send as many messages in one packet as it did with QoS 0 because not enough messages can be sent at the same time.

On the other hand, there is no measurable difference between different QoS levels when there is only a single large message.

Surprisingly the experiments show that the protocol is faster with TLS than without it for two of the QoS settings. This is most likely the case because the overhead of TLS is dominated by the work done by the protocol itself and the transmission of data and that other factors causing variance in the data ended up skewing the experiment toward this result. We can however conclude that the overhead of TLS does not significantly slow the transfer time for the protocol.

Another surprising result is the increase in the number of packets when encrypted with TLS. This is hard to explain since the contents of the encrypted packets are unknown. The most plausible reason would be that the TLS library used by the broker is very inefficient with buffered data.

4.3.1.3 AMQP

The times for AMQP transfers show no irregularities. The transfer is a little bit slower with encryption. The time of transfer is similar to MQTT QoS 1 although it is faster.

4.3.2 Comparison

All tested protocols were significantly faster than HTTP except for CoAP over UDP. This is of course because of the default congestion control because of which each request is done sequentially. We can clearly see that when it was sent over TCP it performed much better. The TCP implementation made use of high parallelism with up to 1024 concurrent requests. When the client was configured to have a maximum of only one outstanding request at a time in a TCP connection similarly to the transport over UDP, the transfer of all messages took about 4 minutes. This discrepancy seems to be mostly because of different implementations.

There was a very small increase in processing time when TLS was used with all protocols. Only CoAP showed a considerable relative drop in speed, however, it has still been the fastest and the test included the exchange of certificates which add one-time overhead to the connection establishment.

We can see that all of the protocols seem to perform well when a single large message is sent.

On ESP32 CoAP shows much worse performance than MQTT. There does not seem to be any difference in the speed of transfer between confirmable and non-confirmable CoAP messages. This is mostly because the program always waits for a response before starting another request. Even then it is able to serve about 40 messages per second. The MQTT implementation had at most 16 unacknowledged messages at once and with QoS 1 it achieved a throughput of about 100 messages per second.

Conclusion

All the protocols were introduced to the reader to the extent that they should know how it generally behaves and what kind of messages are needed to transfer data. At that point, several areas where the protocols differ significantly were identified and these differences were confirmed by the experiments. There are some features that do not have an alternative in the other protocols, such as CoAP's block transfer, MQTT's native implementation of Quality of Service, or AMQP's superior flow control and multiplexing.

During the experiments implementation phase, several problems with various code libraries were uncovered. As part of this work, issues were raised in several GitHub repositories and even some contributions were made to a Rust MQTT implementation[42] for constrained devices.

These problems were most visible with the AMQP implementation, where there was very little support to be found. Most implementations focused on business messaging for more complex systems which heavily relied on OS functionality not available on ESP32. The uamqp library[35] was the only library that was optimized for low RAM footprint and portability. Nonetheless, attempts to create a firmware to flash this library on the ESP32 board failed. After that several runtimes and languages were tried to find if any supported an AMQP 1.0 library. Rust[31], C[32], Python[36], C#[40], and JavaScript[39] were all tried with no success.

CoAP and MQTT were both successfully flashed onto the ESP32 chip. The former was in part implemented for the purposes of this thesis. The coap-lite library[43] was used for manipulating the CoAP messages but sending and resending were implemented by hand. For MQTT the MiniMQ library[42] was used. It was extended to support QoS 1 and QoS 2.

All three protocols were then successfully used on the Raspberry Pi model 3 b+. For the experiments Python libraries for HTTP[44], MQTT[46], and AMQP[45]; and a Golang library for CoAP[28] were used.

The results show that all three protocols perform considerably better than simple HTTP. The only exception was the speed of CoAP over UDP which

has very strict default settings for congestion control which makes the protocol perform worse. This can be worked around by using it over TCP assuming higher network load is not an issue. Using TCP should not introduce any issue if there is no expectation to transfer a very large number of small messages in a short amount of time and the network is stable enough to support it.

The results also showed differences in speed and bandwidth usage when different Quality of Service guarantees were used in MQTT. When choosing which guarantee to use these perofrmance measurements should be taken into consideration as this setting has a tremendous influence on the speed and possibly on congestion.

All three protocols can be used with TLS or DTLS. This work showed that there is not a significant performance hit when these are used. Only MQTT used considerably more bandwidth, this however didn't significantly slow it down. Therefore, this work concludes that these secure protocols should be used whenever technically possible.

Bibliography

[1] Shelby, Z.; Hartke, K.; et al. The Constrained Application Protocol (CoAP). RFC 7252, RFC Editor, June 2014. Available from: http://www.rfc-editor.org/rfc/rfc7252.txt

[2] Bormann, C.; Shelby, Z. Block-Wise Transfers in the Constrained Application Protocol (CoAP). RFC 7959, RFC Editor, August 2016.

[3] Hartke, K. Observing Resources in the Constrained Application Protocol (CoAP). RFC 7641, RFC Editor, September 2015.

[4] Bormann, C.; Lemay, S.; et al. CoAP (Constrained Application Protocol) over TCP, TLS, and WebSockets. RFC 8323, RFC Editor, February 2018. Available from: http://www.rfc-editor.org/rfc/rfc8323.txt

[5] OASIS. MQTT Version 5.0. Technical report, OASIS, March 2019. Available from: https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html

[6] Microsoft. Communicate with your IoT hub by using the AMQP Protocol. 2021. Available from: https://docs.microsoft.com/en-us/azure/iot-hub/iot-hub-amqp-support

[7] Systems, E. ESP32-WROOM-32 Datasheet. 2021. Available from: https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32_datasheet_en.pdf

[8] Foundation, R. P. Raspberry Pi 3 Model B+ Product Brief. Available from: https://static.raspberrypi.org/files/product-briefs/Raspberry-Pi-Model-Bplus-Product-Brief.pdf

[9] Bormann, C.; Ersue, M.; et al. Terminology for Constrained-Node Networks. RFC 7228, RFC Editor, May 2014, http://www.rfc-editor.org/rfc/rfc7228.txt. Available from: http://www.rfc-editor.org/rfc/rfc7228.txt

[10] Committee, L. A. T. LoRaWAN Link Layer Specification v1.0.4. Technical report, LoRa Alliance, inc., 2020.

[11] Sethi, M.; Sarikaya, B.; et al. Secure IoT Bootstrapping: A Survey. Internet-Draft draft-irtf-t2trg-secure-bootstrapping-00, IETF Secretariat, April 2021, https://www.ietf.org/archive/id/draft-irtf-t2trg-secure-bootstrapping-00.txt. Available from: https://www.ietf.org/archive/id/draft-irtf-t2trg-secure-bootstrapping-00.txt

[12] Heer, T.; Garcia-Morchon, O.; et al. Security Challenges in the IP-Based Internet of Things. *Wireless Personal Communications*, volume 61, 12 2011: pp. 527–542, doi:10.1007/s11277-011-0385-5.

[13] Bormann, C. CoAP. 2016. Available from: http://coap.technology/

[14] van der Stok, P.; Bormann, C.; et al. PATCH and FETCH Methods for the Constrained Application Protocol (CoAP). RFC 8132, RFC Editor, April 2017. Available from: http://www.rfc-editor.org/rfc/rfc8132.txt

[15] IETF. RFC 7252 - The Constrained Application Protocol (CoAP). Available from: https://datatracker.ietf.org/doc/rfc7252/

[16] Adams, D. *The Hitchhiker's Guide to the Galaxy*. Pan Books, 1979, ISBN 3320258648.

[17] Fielding, R. T.; Gettys, J.; et al. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, RFC Editor, June 1999. Available from: http://www.rfc-editor.org/rfc/rfc2616.txt

[18] Postel, J. Transmission Control Protocol. STD 7, RFC Editor, September 1981, http://www.rfc-editor.org/rfc/rfc793.txt. Available from: http://www.rfc-editor.org/rfc/rfc793.txt

[19] Dierks, T.; Rescorla, E. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, RFC Editor, August 2008, http://www.rfc-editor.org/rfc/rfc5246.txt. Available from: http://www.rfc-editor.org/rfc/rfc5246.txt

[20] Fette, I.; Melnikov, A. The WebSocket Protocol. RFC 6455, RFC Editor, December 2011, http://www.rfc-editor.org/rfc/rfc6455.txt. Available from: http://www.rfc-editor.org/rfc/rfc6455.txt

[21] Stanford-Clark, A.; Truong, H. L. MQTT For Sensor Networks (MQTT-SN) Protocol Specification Version 1.2. Technical report, IBM, 2013. Available from: https://www.oasis-open.org/committees/download.php/66091/MQTT-SN_spec_v1.2.pdf

[22] Barrett, M. Framework for Improving Critical Infrastructure Cybersecurity. Technical report, Apr. 2018, doi:10.6028/nist.cswp.04162018. Available from: https://doi.org/10.6028/nist.cswp.04162018

[23] OASIS. MQTT and the NIST Cybersecurity Framework Version 1.0. Technical report, OASIS, May 2014. Available from: http://docs.oasis-open.org/mqtt/mqtt-nist-cybersecurity/v1.0/cn01/mqtt-nist-cybersecurity-v1.0-cn01.html

[24] OASIS. OASIS Advanced Message Queuing Protocol (AMQP) Version 1.0. Technical report, OASIS, October 2012. Available from: http://docs.oasis-open.org/amqp/core/v1.0/os/amqp-core-complete-v1.0-os.pdf

[25] Inc., V. Which protocols does RabbitMQ support? 2007-2021. Available from: https://www.rabbitmq.com/protocols.html#amqp-10

[26] Microsoft. Choose a device communication protocol. Available from: https://docs.microsoft.com/en-us/azure/iot-hub/iot-hub-devguide-protocols

[27] OASIS. AMQP Claims-based Security. March 2021. Available from: https://docs.oasis-open.org/amqp/amqp-cbs/v1.0/amqp-cbs-v1.0.html

[28] plgd. Go-CoAP. Available from: https://github.com/plgd-dev/go-coap

[29] Klabnik, S.; Nichols, C. *The Rust Programming Language*. San Francisco, CA: No Starch Press, June 2018.

[30] Munns, J. The Embdedded Rust Book. Available from: https://docs.rust-embedded.org/book/

[31] Markov, I. Rust on ESP32 STD demo app. Available from: https://github.com/ivmarkov/rust-esp32-std-demo

[32] Systems, E. Espressif IoT Development Framework. Available from: https://github.com/espressif/esp-idf

[33] Lerche, C.; et al. Mio – Metal IO. Available from: https://github.com/tokio-rs/mio

[34] Scheff, P.; Markov, I. error when compiling mio. Available from: https://github.com/ivmarkov/rust-esp32-std-demo/issues/9

[35] Microsoft. uAMQP. Available from: https://github.com/Azure/azure-uamqp-c

[36] George, D. P.; et al. MicroPython documentation. Available from: https://docs.micropython.org/en/latest/index.html

[37] Pederson, B.; Solem, A.; et al. Python AMQP 0.9.1 client library. Available from: https://github.com/celery/py-amqp

[38] Garnock-Jones, T.; Roy, G. M.; et al. Pika. Available from: https://github.com/pika/pika

[39] Ltd, P. Espruino on ESP32. Available from: https://www.espruino.com/ESP32

[40] Foundation, N. .NET nanoFramework. Available from: https://docs.nanoframework.net/index.html

[41] Microsoft. Advanced Message Queueing Protocol (AMQP) 1.0 support in Service Bus. June 2021. Available from: https://docs.microsoft.com/en-us/azure/service-bus-messaging/service-bus-amqp-overview

[42] QUARTIQ. MiniMQ. Available from: https://github.com/quartiq/minimq

[43] Disch, M.; et al. coap-lite. Available from: https://github.com/martindisch/coap-lite

[44] aiohttp maintainers. aiohttp. Available from: https://docs.aiohttp.org/en/v3.8.1/

[45] Contributors, A. Q. Qpid Proton Python API Documentation. 2019. Available from: https://qpid.apache.org/releases/qpid-proton-0.35.0/proton/python/docs/index.html

[46] Foundation, E. Eclipse Paho Python Client. Available from: https://www.eclipse.org/paho/clients/python/

[47] chrysn; et al. aiocoap – The Python CoAP library. Available from: https://github.com/chrysn/aiocoap

[48] Ingham, D. AMQP 1.0 becomes an International Standard. May 2014. Available from: https://www.redhat.com/en/blog/amqp-10-becomes-international-standard

[49] Chen, X.; et al. AMQP.Net Lite. Available from: https://github.com/Azure/amqpnetlite

[50] Treadway, S.; et al. Go RabbitMQ Client Library. Available from: https://github.com/streadway/amqp

[51] Hätönen, S.; Nyrhinen, A.; et al. An experimental study of home gateway characteristics. In *Proceedings of the 10th annual conference on Internet measurement - IMC '10*, ACM Press, 2010, doi:10.1145/1879141.1879174. Available from: `https://doi.org/10.1145/1879141.1879174`

[52] Guha, S.; Biswas, K.; et al. NAT Behavioral Requirements for TCP. BCP 142, RFC Editor, October 2008.

[53] Audet, F.; Jennings, C. Network Address Translation (NAT) Behavioral Requirements for Unicast UDP. BCP 127, RFC Editor, January 2007, `http://www.rfc-editor.org/rfc/rfc4787.txt`. Available from: `http://www.rfc-editor.org/rfc/rfc4787.txt`

[54] Al-Dhief, T.; Sabri, N.; et al. Performance comparison betweenTCP and UDP protocols in different simulation scenarios. *International Journal of Engineering & Technology*, volume 7, no. 4.36, 2018: pp. 172–176.

[55] Allman, M.; Paxson, V.; et al. TCP Congestion Control. RFC 5681, RFC Editor, September 2009, `http://www.rfc-editor.org/rfc/rfc5681.txt`. Available from: `http://www.rfc-editor.org/rfc/rfc5681.txt`

Appendix A

Acronyms

6LoWPAN IPv6 over Low-Power Wireless Personal Area Networks.

AES Advanced Encryption Standard.

AMQP Advanced Message Queuing Protocol.

CBC Cipher Block Chaining.

CoAP Constrained Application Protocol.

CPU Central Processing Unit.

CSM Capabilities and Settings Messages.

DSA Digital Signature Algorithm.

DTLS Datagram Transport Layer Security.

ESP-IDF Espressif IoT Development Framework.

FCS Frame Check Sequence.

GPIO General-Purpose Input/Output.

HTTP Hypertext Transfer Protocol.

HTTPS Hypertext Transfer Protocol Secure.

I^2C Inter-Integrated Circuit.

IBM International Business Machines Corporation.

IEC International Electrotechnical Commission.

IETF Internet Engineering Task Force.

IoT Internet of Things.

IP Internet Protocol.

IPv6 Internet Protocol version 6.

ISO International Organization for Standardization.

LoRaWAN Long Range Wide Area Network.

M2M Machine-to-Machine.

MAC Message Authentication Code.

mio Metal IO.

MITM Man in the Middle.

MQTT Message Queuing Telemetry Transport.

MQTT-SN MQTT For Sensor Networks.

MTU Maximum Transmission Unit.

NAT Network Address Translation.

NIST National Institute of Standards and Technology.

OASIS Organization for the Advancement of Structured Information Standards.

OS Operating System.

QoS Quality of Service.

RAM Random Access Memory.

RFC Request for Comments.

SASL Simple Authentication and Security Layer.

SCRAM Salted Challenge Response Authentication Mechanism.

SCTP Stream Control Transmission Protocol.

SPI Serial Peripheral Intrface.

SSH Secure Shell.

TCP Transmission Control Protocol.

TLS Transport Layer Security.

UDP User Datagram Protocol.

URI Unique Resource Identifier.

URL Unique Resource Location.

USB Universal Serial Bus.

Appendix B

Contents of Enclosed Flash Drive

- `text` ..... The thesis text directory
  - `thesis.pdf` ..... The thesis text in PDF format
- `src` ..... The directory of source codes
  - `thesis` ..... The directory of LaTeX source codes of the thesis
  - `code` ..... The directory source codes for the experiments
    - `server` ..... Go source code to run a CoAP and HTTP servers
    - `esp` ..... Source codes for the ESP32 experiments
    - `rpi` ..... Source codes for the Raspberry Pi experiments
- `wireshark` ..... Wiresharek capture files
- `results` ..... Files containing measured times of experiments
  - `esp` ..... Experiments with ESP32
  - `rpi3` ..... Experiments with Raspberry Pi
    - `small` ..... Experiments with many small messages
    - `large` ..... Experiments with a single large message