BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

ASYNCHRONOUS MQTT CLIENT FOR EMBEDDED DEVICES RUNNING ON DROGUE IOT FIRMWARE

ASYNCHRONNÍ KLIENTSKÁ KNIHOVNA PRO VESTAVNÁ ZAŘÍZENÍ PROVOZUJÍCÍ DROGUE-IOT FIRMWARE

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **ONDŘEJ BABEC**
AUTOR PRÁCE

SUPERVISOR **Ing. JAN PLUSKAL**
VEDOUCÍ PRÁCE

BRNO 2022

Brno University of Technology
Faculty of Information Technology

Department of Information Systems (DIFS) Academic year 2021/2022

Bachelor's Thesis Specification

Student: **Babec Ondřej**
Programme: Information Technology
Title: **Asynchronous MQTT Client Library for Embedded Devices Running on Drogue-IoT Firmware**
Category: Embedded Systems

Assignment:

1. Study the basic principles of the Drogue-IoT, MQTT protocol, Rust programming language and Embassy asynchronous library for embedded devices.
2. Design Embassy based MQTT asynchronous client library (fully supporting MQTT v5) in Rust programming language for embedded devices. Respect all constraints introduced by intended usage on these devices enforced by the Drogue-IoT.
3. Implement the MQTT library designed in point 2.
4. Design, implement and evaluate functional and performance tests of your solution.
5. In conclusion, evaluate the solution on the embedded device chosen by the supervisor, propose possible improvements for the created implementation, and describe usability and implementation difficulty for such improvement.

Recommended literature:

- Drogue IoT [online]. 2020. Available at: https://book.drogue.io/drogue-book/index.html
- Banks, A., Briggs, E., Borgendale, K. and Gupta, R. MQTT Version 5.0 [online]. March 2019. Available at: https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html.
- Rust RFC 2394 [online]. 2018. Available at: https://rust-lang.github.io/rfcs/2394-async_await.html.
- Rust RFC 2592 [online]. 2018. Available at: https://rust-lang.github.io/rfcs/2592-futures.html.
- Troutwine, B.Hands-On Concurrency with Rust: Confidently Build Memory-safe,Parallel, and Efficient Software in Rust. Packt Publishing, 2018. ISBN9781788399975

Requirements for the first semester:

- Completed points 1 and 2.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor: **Pluskal Jan, Ing.**
Head of Department: Kolář Dušan, doc. Dr. Ing.
Beginning of work: November 1, 2021
Submission deadline: May 11, 2022
Approval date: October 19, 2021

Abstract

IoT is a branch of informatics that is massively expanding in the last few years. In today's world, IoT is all around us. Smart bulbs, household accessories, or also thousands of devices in industrial buildings all are part of the IoT. There are many projects that allow the integration of IoT devices and cloud processing of their messages. One of these projects is Drogue-IoT. This open-source project allows creating the enterprise cloud solutions but also firmware for the embedded devices. One limitation of this project is the unavailability of the client library for the MQTT messaging protocol. Exactly this library is the main theme of this paper. Work designs and implements Rust native MQTT client for embedded devices, which currently does not exist. The solution is shown with the built device. The final report evaluates the implementation of the client and includes possible improvements in implementation.

Abstrakt

IoT je odvětví informatiky, které v posledních letech masivně expanduje. V dnešním světě je IoT všude kolem nás. Jsou to chytré žárovky a doplňky do domácnosti, ale také tisíce zařízení v průmyslových objektech. Nedílnou součástí IoT jsou protokoly pro zasílání zpráv, které umožňují komunikaci komunikaci s těmito zařízeními. Dnes již existuje mnoho projektů, které umoňují integraci IoT zařízení a následné cloudové zpracování jejich zpráv. Jedním z těchto projektů je Drogue-IoT. Tento open-source projekt umožňuje vytvářet firemní cloudové řešení, ale také firmware pro vestavné zařízení. Jednou z limitací tohoto projektu je nepřítomnost klientské aplikace podporující zasílání zpráv pomocí protokolu MQTT. Právě tato klientská aplikace je tématem této práce. Práce zahrnuje návrh a implementaci klienta protokolu MQTT pro vestavěná zařízení v jazyce Rust, který doposud neexistuje. Řešení je demonstrováno pomocí sestrojeného zařízení. Výsledná práce vyhodnocuje implementaci klienta a obsahuje návrhy na budoucí vylepšení práce.

Keywords

MQTT, Embedded, Rust, Async, IoT

Klíčová slova

MQTT, Embedded, Rust, Async, IoT

Reference

BABEC, Ondřej. *Asynchronous MQTT client for embedded devices running on Drogue IoT firmware.* Brno, 2022. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Jan Pluskal

Rozšířený abstrakt

Internet věcí (IoT) je velké aktuální téma, slyšíme to všude okolo nás. U televizí, ledniček dokonce i sporáků v dnešní době najdeme nějaké klíčové slovo, které nás odkazuje na informaci, že zařízení je možné zapojit do chytré domácnosti. Mimo tyto domácí spotřebiče jsou to, ale i průmysloví roboti a jiné senzory. Všechna tato zařízení potřebují ke svému fungování jeden nejvíce podstatný mechanismus. Tímto mechanismem jsou komunikační protokoly, bez kterých by nemohlo žádné z IoT zařízení komunikovat s okolním světem. Mezi nejpoužívanější protokoly v tomto odvětví patří HTTP, CoAP a MQTT. Každý z protokolů má svá specifika, jako je zabezpečení, náročnost zpracování a další. Protokol MQTT, který je jedním z hlavních prvků této práce je navržen jako velmi efektivní a zároveň nenáročný na hardwarové vybavení.

Mimo komunikační protokoly jsou jedním z velice důležitých prvků IoT zařízení také programovací jazyky, pomocí kterých jsou aplikace pro vestavná IoT zařízení vytvořeny. Každý jazyk má svoje výhody, či nevýhody. Ze starších jazyků se v tomto světě setkáme především s jazyky jako jsou C a C++. Proč zrovna tyto jazyky má poměrně jasné opodstatnění. Jsou to jedny z mála jazyků, které mají možnost přistupovat přímo k hardwarové výbavě počítače. Většina vysoko úrovňových jazyků jako *Java* takovou funkcionalitu nenabízejí, a proto je takřka nemožné v nich vestavné aplikace vytvářet. Jedním z nových hráčů na tomto poli je programovací jazyk *Rust*. Tento programovací jazyk je tvořen obrovskou open-source komunitou a nabízí přímý přístup k hardware, což ve spojení s jeho rychlostí a spolehlivostí umožňuje vývoj velmi efektivních aplikací pro (nejen) vestavná zařízení. Právě tento jazyk je hlavním implementačním jazykem této bakalářské práce.

Cílem této práce je tedy kombinace obou dříve zmíněných technologií. Přesněji jde o vytvoření asynchronní MQTT klientské knihovny pro vestavná zařízení provozující Drogue-device firmware v jazyce Rust. Klient by měl podporovat MQTT verze 5 a umožňovat případné rozšíření pro starší standard MQTT verze 3. Proč MQTT klient v Rustu? Odpověď na tuto otázku je velice jednoduchá, protože žádný takový klient zatím neexistuje. Jeden z hlavních důvodů, proč zatím žádná implementace neexistuje by mohl být samotný standard MQTT verze 5. Tento standard udává, že většina kontrolních paketů tohoto protokolu obsahuje části, jejichž délka je variabilní. Variabilita je poměrně velký problém v Rustu pro vestavěná zařízení, jelikož na vestavných zařízeních není žádný operační systém, nemůžeme využít dynamické alokování paměti. Tato překážka vede k nutnosti využití pokročilých možností jazyka Rust, jako jsou konstantní generické argumenty, či explicitní anotace délky života jednotlivých proměnných.

Vývoj klientské knihovny byl rozdělen do třech základních částí, a to návrh, implementace a testování/evaluace. Návrh takové klientské knihovny je velmi problematický, poněvadž každé vestavné zařízení je vybaveno zcela rozdílným hardwarovým vybavením, bylo nutné návrh vytvořit tak, aby umožňoval co největší konfigurovatelnost chování klienta a jeho využití hardwaru. Zároveň bylo nutné udržet rovnováhu mezi flexibilitou konfigurace a její složitostí. Součástí návrhu vznikla také omezení, která bude nutné pří výsledném použití knihovny respektovat. Tato omezení jsou z pravidla určena zamýšleným použitím dané knihovny. Implementace zcela respektuje návrh klienta a jejím výsledkem je tedy zcela funkční MQTT klient podporující standard MQTT verze 5. Poslední částí vývoje je testování a vyhodnocení. V rámci tohoto kroku vzniklo několik automatizovaných testovacích sad, testy pro zajištění správné funkce klienta při zpracování velkého množství zpráv, aplikace sloužící pro sběr výkonnostních metrik a nakonec demonstrační aplikace pro zařízení micro:bit V2 a Wi-Fi modul ESP8266 realizující reálný případ použití vytvořeného klienta.

Výsledkem této práce je tedy kompletně funkční klientská knihovna. Tato klientská knihovna je schopna pracovat pod velkým zatížením bez ztráty výkonu, či náznaku chybného chování. Demonstrační aplikace zároveň ukázala, že využití knihovny je velice pohodlné a nebude tedy přinášet budoucím uživatelům žádnou negativní zkušenost. Díky implementaci síťového adaptéru pro Drogue device firmware je zároveň dosaženo velké podpory pro různé vestavné zařízení a periferie, jako například STM32, Raspberry Pico, či senzory jako DHT22 a další. Drobným rozšíření knihovny oproti zadání práce je taktéž podpora standardních zařízení pomocí asynchronní knihovny *Tokio*, která umožňuje použití knihovny na standardních zařízeních.

Asynchronous MQTT client for embedded devices running on Drogue IoT firmware

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Ing. Jan Pluskal. The supplementary information was provided by Ing. Jakub Stejskal (Redhat) and Siv. Ing. Ulf Lilleengen (Redhat). I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Ondřej Babec
May 8, 2022

Acknowledgements

I would like to thank my supervisors, Ing. Jan Pluskal from VUT FIT, and Ing. Jakub Stejskal from Red Hat Czech s.r.o for guidance and providing valuable feedback. Also, I would like to thank my technical consultant Siv. Ing. Ulf Lilleengen (Redhat) for his time during the introduction and explanation of the Drogue-IoT project.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

IoT (Internet of Things) is a big topic in the world of information technologies [7]. If we speak about the IoT, we have to mention messaging as the fundamental base of every IoT system. Messaging is the most crucial part of each IoT framework. The reason for it is simple – devices need a way how to talk to each other and how they can accept commands or different kinds of inputs. And for this, we have specific messaging protocols such as *MQTT, CoAP or AMQP* [10]. These messaging protocols have various behaviors in areas like security, speed, and reliability. *MQTT* is one of the oldest protocols, but it is still one of the most used protocols because it is simple and builds on well known *observer* design pattern [1, 6]. It means that MQTT can be used on devices with really less-powerful processors, which are ideal for the IoT world. With *Rust* [12] entering the IoT playground as a relatively new competitor, there are few messaging clients for embedded devices, and the MQTT client is one of them. Drogue-device framework is also missing (at the time of writing of this thesis) the implementation of the MQTT client written in Rust, which the users could widely use. To remedy this inability is the primary motivation for this thesis.

Drogue IoT [4] is an open-source project that provides tools for creating complete enterprise IoT solutions. Drogue comprises two main parts — Drogue-device and Drogue-cloud. Drogue-cloud is a cloud solution that allows users to connect devices and applications. The cloud side has been built on the popular cloud technologies like *Kafka, Keycloak, Cloud events, and Kubernetes*. Although until now we have spoken only about embedded devices but Drogue-cloud is designed to allow users to integrate also larger machines running on standard OS like Linux or Windows. Drogue-device operates directly on embedded devices. It provides firmware with the most important drivers and tools, which are necessary for the development of the Rust applications.

Implementing such a complex ecosystem as Drogue is not a simple job, especially in a new programming language like Rust. The language is under active development, so something perfectly right today does not have to be right tomorrow. Balancing on this edge is difficult, and it is also one of the biggest challenges for development. This challenge is not one-sided, and waiting for feature implementation in Rust can be even more challenging.

The main goal of this work is to create an open-source, asynchronous Rust native MQTT client for embedded devices that do not exist yet. Yes, there are a couple of async clients written in Rust, so the logical question would be: *Why not use this existing client?* These clients use standard Rust libraries which are not supported in *no-std Rust* for embedded devices. Everything is turning around Drogue IoT, but the client should not be just some solution that can be used together with Drogue-device. It should be standalone and serve a purpose for the whole Rust IoT community.

The thesis deals with several topics. In the first two chapters, you find a deeper description of Drogue fundamentals and designs. The following two chapters outline the implementation and design of the MQTT client and evaluation device. The last part of the thesis is focused mainly on the evaluation. Evaluation includes the realization of the device designed in the third chapter, and also a conclusion. The conclusion contains all the positives and negatives of the current implementation together with implementation improvements.

Chapter 2

Fundamentals of Drogue-IoT

This chapter goes through the fundamentals of the toolset Drogue-IoT, which is a base use-case for MQTT client designed in Chapter 4. The Drogue-IoT project provides a set of tools for building an enterprise IoT solution. The project is separated into two main parts, Drogue-cloud and Drogue-device.

Drogue-cloud is a toolset that allows the creation Drogue-IoT cloud that connects (embedded) devices with enterprise applications. Cloud schema is described in two layers, the Data plane and the Control plane described later in the chapter.

Drogue-device is an async open-source framework that allows building embedded applications in Rust. The framework is built on top of the actor model for concurrent applications. Drogue-device supports several microcontrollers specified at the end of the chapter and also drivers for commonly used peripherals.

2.1 Drogue-IoT

Drogue-IoT project separates devices, such as sensors or actuators, from the cloud side. Cloud side called *Drogue-cloud* combines connectivity layer and device management services. Device side is called *Drogue-device*.

As you can see in Figure 2.1 these two sides are connected by protocol endpoints. These endpoints provide mapping between device messaging protocol and *Cloud Events*[1]. Then asynchronous processing starts on the cloud side and when the processing is done, customer's cloud applications can consume results.

The cloud side is designed *as a service* so user should have everything prepared on demand. If you imagine a potential user of this project, he does not care about what is going on somewhere on the cloud. He wants to plug-in an application into the cloud, specify device endpoint and that should be it. That's exactly what Drogue allows you. Drogue is built using Rust programming language. Rust allows developers to avoid memory corruption and undefined behaviors with only a little effort. These, together with its speed, are basically the most important attributes for software that is running on embedded devices.

[1]Specification for describing data in one common way for all the formats. More can be found: https://cloudevents.io/

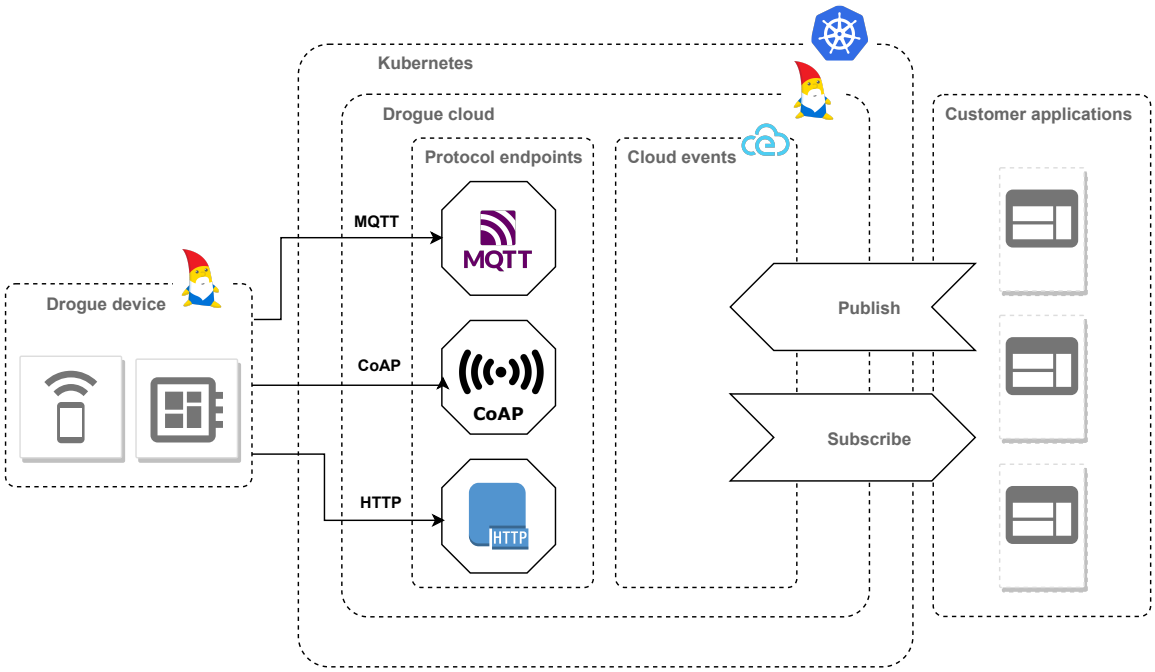


Figure 2.1: Framework architecture expressing Drogue-IoT layout and core components of ecosystem.

2.2 Drogue-cloud

One of the main purposes of the Drogue-cloud is connecting devices and applications displayed in Figure 2.2. Devices need a way how to speak with the application and the same thing reversed, the application needs a way how to command devices. This part of the Drogue-cloud is called *Data plane*.

The second part that is less important but maybe even more complex is called *Control plane*. Both parts are described with more details in Subsections 2.2.1 and 2.2.2. When we say the word *Control* we usually speak about controlling device which includes some kind of security and that is exactly what *Control plane* does.

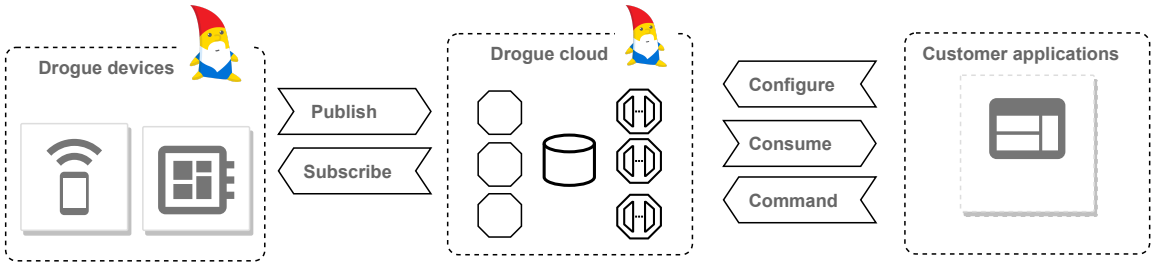


Figure 2.2: Drogue-cloud simplified diagram which express device and applications communication patterns with cloud through protocol endpoints and protocol integrations.

With the gathered information, you are for sure speculating if the *Drogue-cloud* is something different from a simple messaging broker. You could even use a single message broker that implements all protocols, like *ActiveMQ* which implements at least HTTP

and MQTT. However, it will still be only message broker. Drogue extends the broker functionality by very specific IoT functionalities like over-the-air firmware updates or device specific authentication.

2.2.1 Data plane

Data plane serves mainly for a connection. An ideal situation with IoT cloud presented in Figure 2.3 would be the user application supports just one messaging protocol in order to receive or send messages to devices. Drogue provides, let's say, a normalization layer to which the user application connects.

This normalization layer provides multiple APIs on which applications can consume and send messages. Drogue is internally using *Cloud Events*, which means that messages are mapped through external interfaces (protocol endpoints) into *Cloud Events* format used by internal components.

Drogue also allows applications to send messages to devices. Application can send *commands* to the command endpoint (HTTP). These messages are mapped to the Cloud Events format, processed by the Kafka, and forwarded to the subscribed device. This principle, with both sides capable of sending and receiving, is in Drogue named *Push and Pull* model.

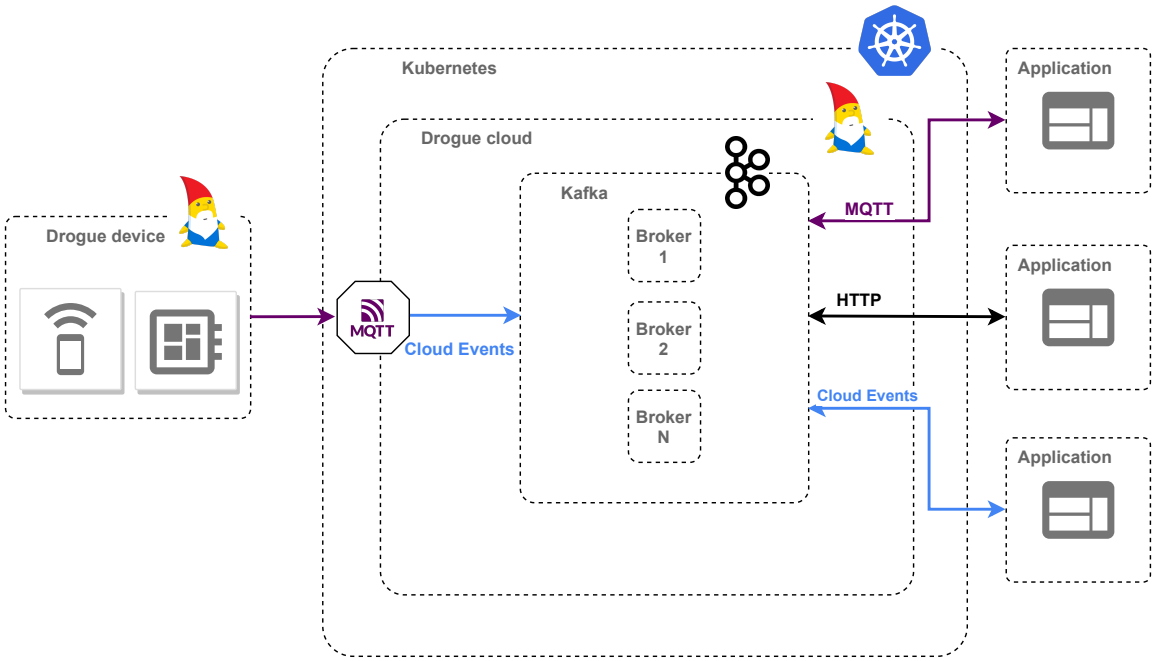


Figure 2.3: Data plane communication schematics with example MQTT device protocol and multiple customer protocols.

Devices are communicating through *protocol endpoints*. In this context, we rarely spoke about a single microcontroller or any kind of sensor. In massive enterprise ecosystems, we also speak about gateways or services.

Gateways act as proxy servers directly connected to the Drogue-cloud (protocol endpoints). Gateways usually enable the use of devices with a non-IP connection like

bluetooth and other low-energy transmissions. These servers are forwarding messages from devices to protocol endpoints.

Services are basically gateways but on a larger scale. Services usually have their own infrastructure and they can be even cloud-based. One of the most popular services is probably *TTN network* [2]. *TTN network* is a service for *LoRaWAN* which is heavily used all around the world.

The protocol endpoints are device facing services. These services offer connectivity to specific messaging protocols. Drogue-cloud currently supports three different protocol endpoints:

- **HTTP** – Includes *TTN* version 3 endpoint
- **CoAP** – Does not include *DTLS*
- **MQTT** – The endpoint provides both MQTT version 5 and MQTT version 3 support. However, it also sets requirements for the client implementation:
  - Clean session flag in **CONNECT** packet must be always *true*
  - Client must support at least *username/password* authentication

Aside just the sending messages from one side to another, Drogue-cloud also provides solution for storing these messages. Message persistence is part of the Data plane layer. When the message hits protocol endpoint and endpoint maps the message to Cloud events format, it is forwarded to *Kafka* server which delivers the message to the business application.

In terms of persistence, we can imagine Kafka as a ''buffer" for messages. Kafka can[3] store messages on a disk until a message consumer is ready to consume. Kafka also offers functionality to share the load of messages between multiple customer instances.

2.2.2 Control plane

The *Control plane* is here to control. In our environment, we speak mostly about security on both device and application sides. With devices, it would be tough to implement any *SSO*[4] service. For the application side is *SSO* an ideal solution. To make everything easier for devices and applications, Drogue provides a solution named *Device registry*. It controls both applications and devices authentication services. These are connected via device registry component, as it is displayed in Figure 2.4.

[2] More information about TTN can be found at https://www.thethingsnetwork.org

[3] Kafka persistence is configurable by user in multiple ways. First is number of bytes which can be stored for each topic and second is time for which are messages in topic stored.

[4] SSO (Single sign-on) is an authentication scheme that allows users/applications to sign on to an authentication service just one time.

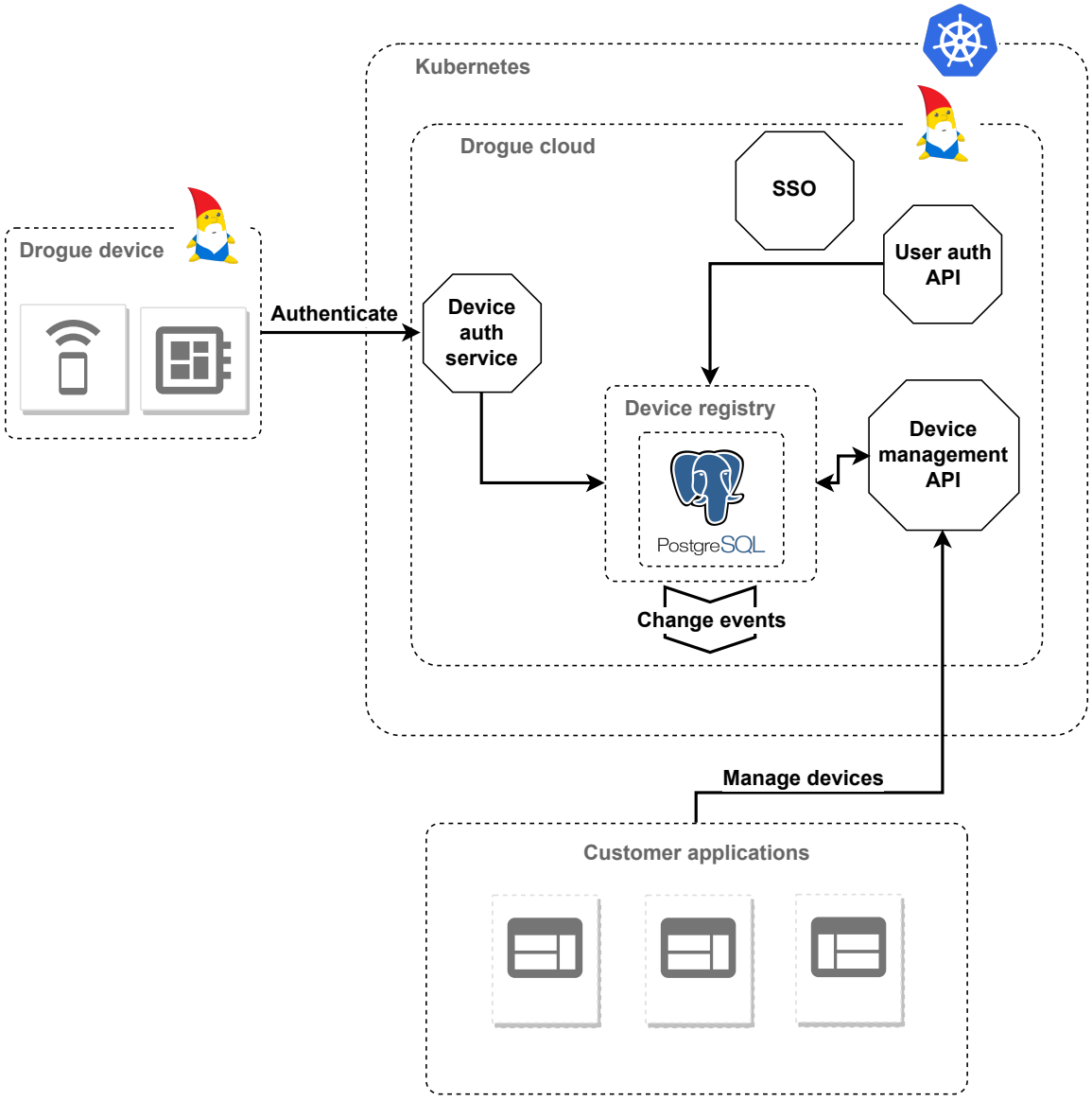


Figure 2.4: Control plane schema describing communication between Device registry and services.

The device registry is basically an access control[5] server for devices and applications. It also stores the configuration for devices and applications. Configuration persistence is provided by *PostgresSQL* compatible database[6]. With devices trying to access the requested resource, you want to make it the easiest as possible to save energy and also a memory of the device. Because of this, Drogue-cloud provides several types of technologies for access control:

- **pre-shared** keys
- **X.509** certificates

[5]When we speak about devices, we usually speak about both authentication and authorization.
[6]Type of database deployment is up to the user. It can be both clustered or non-clustered.

- **simple** username and password

For the configuration, Device registry provides several public and private services that users can use to edit or read both device and application configuration. *Device Auth Service* is the core device access control service. It allows internal components to authenticate and authorize devices. This service is mostly used by protocol endpoints when the device wants to publish or subscribe to the endpoint. Drogue does not forget about users. For user access control system provides *User Auth Service* internal service which allows checking if the user has an access to the target resource.

Both these services have a read-only access to the database. How the name of the last service *(Device) Management API* suggests it is used mainly for the management of information stored in the Device registry. Management API is the only one that has read-write access to the database [4].

Drogue also implements a system named *Change events*. This system is built on top of *Knative*[7] eventing. *Change events* basically allows other components to react to change in the Device registry. As an example, when a user makes a change by Management API, there is an event created which notifies listeners like operators. These operators can eventually react to a change by changing the device certificate or password.

Besides basic services of Device registry, Drogue-cloud also provides *SSO*. Authentication service will propagate information about currently authenticated sessions between other components so these components do not have to authenticate users again. This service is by default implemented by *Keycloak* however any other *OpenID Connect* service can be used instead. Although SSO service is available for authentication only. *User Auth Service* is still responsible for the authorization. This service is also commonly used with external components described in Subsection 2.2.3.

2.2.3 Additional components

Additonal components are not in standard Drogue-cloud deployment but there is built-in support for them so they can be deployed afterward. These components do not carry the core functionality of the cloud stack.

Eclipse Ditto digital twins technology brings a new look into device management problematic. Basically, for each of local devices, there is a twin in the cloud. These twins are synchronized with the actual device in terms of current information or, to be exact, the latest sent and received information to/from the device. We can use these pieces of information for many useful things, such as predicting the future based on input data, connecting machine learning, and others. One of the currently used implementations is *Eclipse Ditto*[8], which also can be used in Drogue-cloud.

Grafana is open-source visualizing software[9] that allows you to visualize a time-series database. Grafana's basic part is dashboard. Dashboards are configurable boards which visualize data from the selected data source. There are several data sources, most used are probably *Prometheus, Graphite, and Influx*. Grafana is not a part of the Drogue-cloud, but the cloud provides all necessary services to adapt Grafana as easy as possible, such as SSO.

[7] Knative is an open source project which provides components for deploying, running, and managing serverless, cloud-native applications to Kubernetes

[8] More information about Ditto can be found: https://www.eclipse.org/ditto/

[9] Deeper grafana specification can be found: https://grafana.com/

2.3 Drogue-device

Drogue-device is a memory and thread-safe async open-source framework for embedded devices that allows creating applications using firmware that contains specific device drivers[10] described in Section 2.3. Framework is based on *Actor model* from Subsection 2.3.

Actor model is a mathematical model of concurrent computation. In this model, the actor is a universal primitive that performs all computations in the system. Each actor has its own state [11], which can be changed only by the owner (actor to which state belongs). Besides, a state actor needs to implement several fundamental rules:

- actors can communicate with each other only through messages
- in response to message actor can do only 3 reactions:
  1. **change** its own state
  2. **send** message to other actor
  3. **create** a finite number of child actors

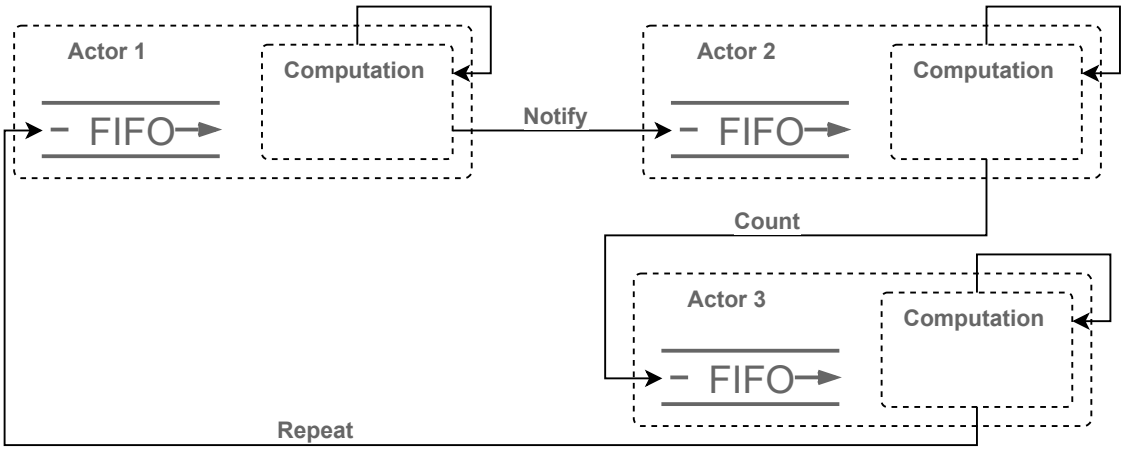


Figure 2.5: Example of actor model communication scheme using fifo queues as actor inbox.

Each actor in Drogue ecosystem is single-threaded thus able to process one message at the time. As embedded processors are single thread, supporting multiple actors require usage of **async** [5] and **.await** [14] functions.

In order to enable communication between actors, each actor has its own address assigned during creation. This address is the entry point for all communication with that actor. The address cannot be used to access actor directly, it can be used only for sending messages. As Figure 2.5 shows, the next to address actors have also attached async message channels (FIFO queue), that enable actors to receive messages during computation.

Besides standard actor model, Drogue provides specific component *Packages*. *Package* connects multiple actors into one semantic component. This component has shared *package* state [4].

[10] Driver provides a software interface that allows embedded applications access hardware without knowing exact details about it.

Drivers that Drogue-device contains[10] are for commonly used sensors and extension boards[11]. Besides that there are also several ways how to implement custom driver:

- **a trait** that defines API for driver to implement
- **a driver** that implements *HAL*[12] or hardware directly
- **an actor** implementation for the driver

In order to create a new driver, the user has to implement one or more of the things above. Each of these has a different outcome and is suitable for different situations. Implementing a new trait makes sense only in the case that board has some very specific behavior that is not yet created. This situation mostly comes only with new boards. For most scenarios, a new driver's implementation will be the best option. The driver will implement existing traits that are already defined.

The use of the last option makes sense only when a peripheral or a board that requires shared access by multiple parts of the system. Actor implementation secures that there is always only one actor accessing the hardware at the time.

Supported devices are based on Embassy supported embedded devices. These devices are:

- **nRF52** – The nRF52 Series[13] devices contain low-power Arm Cortex-M4 processor
- **STM32** – The STM32 family[14] of 32-bit microcontrollers is based on the Arm Cortex-M processor (large range from M0 to M7F).
- **Raspberry Pi Pico** – Based on RP2040[15] microcontroller, contains dual-core Arm Cortex-M0+ processor.

Besides the support for the microcontrollers, Drogue also specifies support for specific ESP8266 firmware. Drogue-device currently supports only the ESP8266 based Wi-Fi chips that are running on *AT* firmware version *1.7.0.X*.

[11] Led matrix display, RAK811 (LORA), HTS221, ESP8266, ESWiFi

[12] A Hardware Abstraction Layer which enables manipulation with hardware accessible from: https://github.com/rust-embedded/embedded-hal.

[13] nRF52 specification can be found: https://infocenter.nordicsemi.com/index.jsp?topic=%2Fstruct_nrf52%2Fstruct%2Fnrf52.html

[14] STM32 specification accesible from: https://www.st.com/en/microcontrollers-microprocessors/stm32-32-bit-arm-cortex-mcus.html

[15] Raspberry Pi Pico specifications can be found: https://www.raspberrypi.com/documentation/microcontrollers/raspberry-pi-pico.html

Chapter 3

Technology evaluation

This chapter aims to go through the main technologies that are necessary to create MQTT client. First, there is a summary of the MQTT features, patterns, and attributes. The MQTT is an OASIS standard messaging protocol. This chapter covers the MQTT version 5, which is the a current latest release of the protocol.

MQTT section provides the most necessary information, to understand the protocol and be able to create a functional messaging client. These pieces of information are topics, packet formats, packet types, and quality of service.

Besides the MQTT, the chapter also describes the technologies that are used to build the client. Rust is a multi-paradigm programming language designed for high performance and safety. Rust provides a way to build an application into an embedded environment with a few limitations in memory and concurrency.

After the examination of the embedded Rust limitations, the chapter outlines a solution for the concurrency on the embedded devices, Embassy. The embassy is an async executor for the embedded devices.

3.1 MQTT protocol

As it is described in Chapter 1 MQTT (MQ Telemetry Transport)[1] is a lightweight messaging protocol that works on a well-known observer pattern. MQTT is built on top of TCP protocol that ensures best effort delivery. MQTT was designed to be suitable for the following typical IoT challenges [8]:

- be lightweight to make possible transmission high volumes of data without a huge overhead
- distribute a small amount of data in high volumes
- make it possible to react to events whenever they happen (event-oriented architecture)
- offer security and privacy for all the data
- be able to provide scalability to distribute data to a huge amount of clients
- offer low power consumption. That basically means to keep size of message header to bare minimum but still keeps all the functionality

[1] There is also MQTT branch that is working on top of the UDP protocol. More information about the MQTT/UDP can be found: https://mqtt-udp.readthedocs.io/en/latest

Currently, there are two versions of MQTT that are mostly being used, 3.1.1 and 5.0. There are some major changes done between these two versions, but the basics of protocol remain the same. MQTT OASIS standard version 3.1.1 [2] does not include negative acknowledgments or extended authentication packets.

Both versions of the protocol need a server, also known as a broker. Broker stands for *Subject* in the observer pattern [6]. Client (*observer*) can subscribe to topic (described in Section 3.1) and also publish messages to some topics. After the message is published to the topic broker, it notifies all subscribed clients with a new message.

Communication pattern

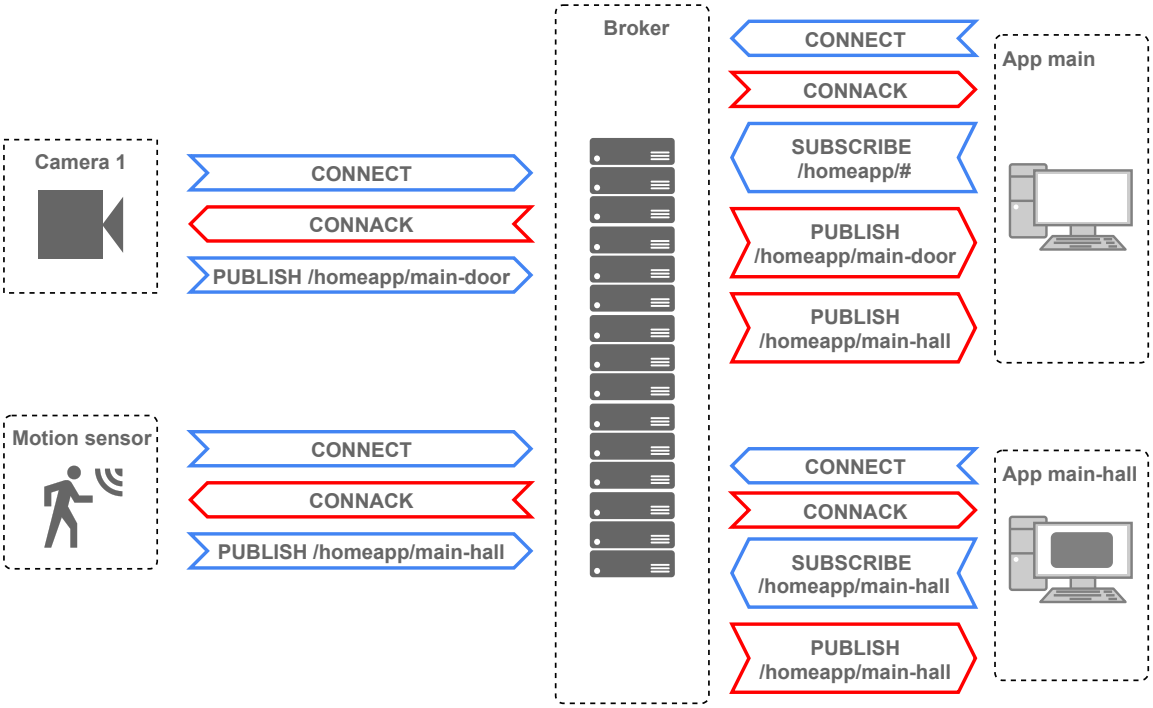


Figure 3.1: MQTT communication scheme expressing communication between security sensor and multiplatform user application.

Standard MQTT communication always starts with **CONNECT** packet, which is sent from the client to the broker. Then follows **CONNACK** packet which confirms or declines connection, with MQTT 5.0 there is also *Negative Acknowledgment* present. *Negative Acknowledgment* brings back information why was connection declined (we can imagine this as return codes from standard command-line programs). If the connection is successfully established, the client can continue sending various control packets [1].

Topics

Topics are message channels that allow MQTT to share messages between clients. Topics are treated as a hierarchy (using slash as a separator) client can create any topic on which will later publish messages. As we can see on communication pattern Section 3.1 topic can

look like **/homeapp/cottage/side-door**. The client can subscribe to one or more topics using wildcards:

- + wildcard is used to subscribe on exact level on topic hierarchy. Client subscribed to **/homeapp/cottage/+** will get messages from all topics under **cottage**
- # wildcard is used to subscribe on remaining levels of hierarchy. Client subscribed to **/homeapp/#** will get messages recursively from all topics lower in hierarchy (for example **/homeapp/cottage/side-door**, **/homeapp/cottage/main-door** and **/homeapp/house/main-door**)

3.1.1 Packet format

All MQTT packets are following the same design. The packet is separated into three parts: *Fixed header, Variable header, Payload*. This dispensation is the same in MQTT version 3.1.1 and 5.0. In addition, version 5.0 includes the *Negative Acknowledgment* in the packet, introduced in Section 3.1.

Byte/Bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
1	Type				Reserved			
2...5	Remaining lenght							

Table 3.1: Table displays format of the MQTT packet fixed header.

Fixed header as packet diagram Table 3.1 displays fixed header have the length from 2 to 5 bytes, where the first byte contains type and second remaining lenght of the control packet. This is the same for both MQTT versions. *Type of control packet* is a 4 bit sequence which determines type of MQTT packet [1, p. 16]. *Remaining lenght* is lenght of $Variable\ header + Payload$ encoded as *Variable Byte Integer* [1, p. 29].

Variable header each MQTT control packet has a different *Variable header* structure. These headers usually carry information about the protocol, QoS, topic, flags, return codes and user properties.

User properties are a list of values formatted as identifier and value combination. Each property has a unique identifier and specific data type that is following (string, byte, four byte integer or other). These properties mostly carry information that is not strictly necessary for the communication (settings, extensions, specific user information).

Payload the payload part of the packet must follow the length specified in Subsection 3.1.1. Content of the payload is specific to packet type and content of the variable header. The payload contains longest part of the packet, like authentication data, subscription descriptors and the application message that is most important part of the payload.

Security

MQTT protocol provides mechanism for several security procedures:

- **Authentication** of users and devices. Authentication is part of **CONNECT** packet and is implemented through a combination of **username** and **password**. Although

this could be sufficient for some of the applications, this can be extended via *LDAP* or *OAUTH* connected to the broker. Where *TLS* is enabled, SSL certificates sent from the client can be also used for client authentication.

- **Authorization** of access to server resources. The broker makes authorization itself and MQTT, as the protocol, does not carry any other information than user identification to support this. The broker usually implements some kind of *RBAC*[2] or *ACL* to achieve authorization.
- **Integrity** of packets can be achieved with *checksum, digital signature or MAC*[3] [9] included in payload of **PUBLISH** packets.
- **Privacy** of MQTT Control Packets, there is no built-in mechanism for data privacy in MQTT. Privacy of packets can be achieved via secure *TLS* connection. Otherwise application can encrypt message but everything except payload will be still visible, such as *Topic* introduced in Section 3.1.

3.1.2 Control packets

Control packets are a fundamental base of MQTT protocol. Each of the control packet serves different purpose. Each packet can cause different actions on the broker/client once it is received. Packets listed in this section are those which are absolutely necessary for the client implementation.

CONNECT is used to connect the client to broker. This packet has to be always first in the communication. This packet is also carries the most information. It contains a unique field in the variable header carries connection flags. These flags specify how will client establish the connection (presence of username and password). Besides that, it also contains a specification of the *Will* behavior.If *Will* client enables sets this flag, broker will send specified *will* message after client disconnects. Last and most important is *Clean Start* flag. It ensures that broker will create a new user session[4] for the connection.

Rest of the packet is composed of *Keep alive* value, optional properties and payload. Keep alive defines time window during which will connection be active. Once this time runs out, broker can send disconnect packet to the client or just close the TCP connection. None of these properties is required. Information which these properties carry are mostly just supportive, not functional (with exceptions). The payload of connect packet depends on the connection flags. It can contain username, password or *will* message, based on specified flag combination.

CONNACK serves as an answer for the connection. This packet can flow only from the broker to client, not in the opposite way. The most important section of this is the reason code. Reason codes are in MQTT slang often called *negative acknowledgments*

PUBLISH is the core of the MQTT communication. Both client and broker can send this packet. A client sends the publish packet in order to send a message towards

[2]Role-based access control is a method of regulating access to whatsoever resource or service based on role of an individual users inside the organization.

[3]MAC – Message Authentication Code

[4]Sessions are used to store information about connected client. These information contain session expiry timer, specific QoS information, subscriptions

MQTT broker. A broker sends publish packets to the subscribed clients once the broker accepts a new message on subscribed topic. This packet contains three unique segments: topic name, packet identifier and specific fixed header flags. Topic name indicates topic to which will broker direct message.

Packet identifier is the specific attribute of the publish packet available only during communication with QoS enabled. It gives both broker and client opportunity to send acknowledge receipt. Fixed header of this packet contains three specific flags. *DUP, QoS, Retain*, where DUP marks packet as re-delivery. QoS is a two bit flag which indicates the expected level of QoS. If the client set Retain flag to packet and send this packet to the broker, the broker has to store this message and send it to each client that successfully subscribed to a specified topic.

PUBACK, PUBREC, PUBREL, and PUBCOMP serve only for acknowledgment purposes. These follow exactly same the format. They contain packet identifier, reason code and optional properties. Packet identifier identifies the packet which is being acknowledged. Reason codes are slightly different for each of the packets. Packets starting acknowledgment (**PUBCOMP, PUBREC**) have a diverse range of reason codes. Including implementation specific error, invalid payload or invalid topic name. **PUBACK and PUBREL** do not have this range of the reason codes, they can carry just one reason code, *packet identifier not found*.

SUBSCRIBE packet completes the send and receive loop between broker and client. Headers of the packet are same as, for example, PUBACK, only difference is in user properties. The most valuable part of this packet is the payload. Payload contains specific subscription descriptors. These contain packet filter and subscription options. Packet filter is string (MQTT represents strings as length on first 2 Bytes followed by UTF-8 encoded string) that contains single topic or wildcard.

Subscription options section is a bit array that represents the settings of the subscription for the specific filter. It allows configuration of Retain, shared subscription and QoS. Retain settings adjust rules (send immediately, send only if the subscription does not exist, do not send) for sending retained messages to the subscribed client. Besides that, there is also *RAP*(Retain as Published). If the client set this flag, it means that messages forwarded by this subscription will carry Retain flag set to one. Last flag is the expected level of Quality of Service. This level is does not have to be final level. Critical is the QoS included in the response for subscribe packet.

SUBACK is the answer for the subscription packet, which has to be sent by the broker. Headers of this packet do not carry any necessary information. Crucial part of this packet is the payload. The payload is formed by a list of the subscribe reason codes. Each entry of this list represents status for subscription descriptor sent by the client in SUBSCRIBE packet. This reason codes differ slightly from other because there are more positive acknowledgments. These positive codes have value from 0x00 to 0x02 and they specify granted QoS (from 0 to 2). Order of these codes have to match order of subscription descriptors.

UNSUBSCRIBE allows the client to remove some or all topics from the subscription . This packet is very similar to the subscribe one. Only difference is that payload of the packet contains only a list of topic filters, not whole subscription descriptor (combination of filter and subscribe options). Response for the unsubscribe is the UNSUBACK.

UNSUBACK follow same scheme as the SUBACK. Headers are minimalistic and a list of ordered reason codes composes the payload.

PINGREQ is something what could be called utility packet. These packets do not have variable header, nor payload and remaining length is always to zero. Only purpose of this packet is to ensure the broker that client who is sending the packet is still alive. Clients are mostly using some timer that is restarted after each message and maximal value of the timer is set to keep-alive value specified in the CONNECT packet. This workflow ensures that broker cannot disconnect client because of timeout.

PINGRESP is an answer for the PINGREQ. It follows same scheme as the PINGREQ packet. Purpose of this packet is to ensure the client that broker still exits. Together with PINGREQ these packets also ensure both client and broker that network connection is alive.

DISCONNECT ends the communication between broker and client. It can be sent in both directions. Packet does not carry any payload. The most important part of the packet is the reason code in the variable header. This reason code shows why client or broker close connection.

3.1.3 Quality of Service

Quality of Service (QoS) is the level on which MQTT guarantees successful delivery of a message to broker and all subscribed clients. MQTT provides three levels of QoS from zero guarantees to double confirmation of a received packet. QoS is configured in each **PUBLISH** packet (fixed header) by number from 0 to 2 in a bit sequence. The number 3 is reserved.

At most once delivery this level of QoS is identified by 0 and means that all messages will be delivered according to network capabilities.

Figure 3.2: Quality of Service level 0 schematics

The receiver (broker, client) is not sending any confirmation MQTT packet to assure the sender that message was delivered successfully. This approach is mainly beneficial for use cases where is acceptable to lose some amount of data (usually there is a huge send rate). All of the possible losses are covered only with TCP's best effort.

At least once delivery is identified by 1. With this QoS receiver will send exactly one **PUBACK** packet to confirm that message was delivered, which includes the packet identifier of the received packet. This identifier will not be used by the sender until **PUBACK** is received.

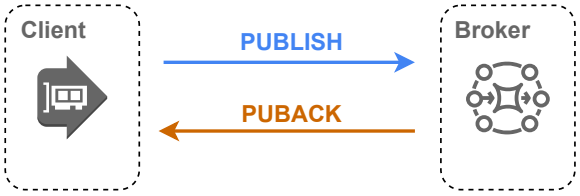


Figure 3.3: Quality of Service level 1 schematics

This approach is ensuring that the packet will be delivered from sender to receiver but does not guarantee that there will not be duplicates.

Exactly once delivery this is the highest QoS that MQTT can assure, and it's identified by 3. This level of QoS is guaranteed with 2 separate confirmation packets.

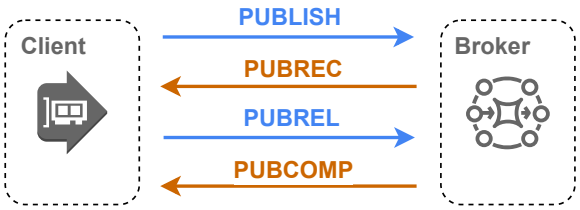


Figure 3.4: Quality of Service level 2 schematics

PUBLISH packet is confirmed via **PUBREC** packet. After confirmation of **PUBLISH** is completed, the sender must send **PUBREL** packet, including packet identifier, and wait for **PUBCOMP** packet. Sender cannot send packet again when received **PUBREC** packet and cannot reuse packet identification until **PUBCOMP** is received. Receiver must send the acknowledgment to any **PUBLISH** packet with the same packet identification until is **PUBREL** received, but it must not cause any duplicates of messages.

As it is clear from the name and description, the result of exactly-once delivery is that message will be received by the receiver once and it is not possible that sending create any duplicates of that message (as it is possible with *at least once delivery*). Because of massive communication overhead, this level of QoS is rarely used by small battery powered IoT devices.

3.2 Rust and Embassy

Rust is a statically typed programming language originally developed by *Graydon Hoare* in 2005 [16]. Currently is Rust open-source that is mainly developed by *Mozilla* and other developers from the open-source community.

The syntax is very similar to languages like *C/C++*. Rust is in most cases very similar to both named languages but solves a couple of problems with which were developers struggling: *memory management* Section 3.2.1 and *concurrent programming* Section 3.2.2.

Programming embedded applications in Rust can be achieved with a specific Rust environment named **no_std**. Using this environment variable, programmer throws away all possibilities to use standard operating system library instead of that **no_std** Rust is using

core library. **Core** library delivers only a minimum of features that are necessary for basic embedded development, features like heap allocation or input-output operations are not included. These features and others can be imported from specific libraries if the platform support them and developer see usage of them as benefit.

3.2.1 Memory management

The memory model in Rust is divided into multiple segments [12]:

- **text** – contains actual code to be executed in compiled binary
- **data** – place for static variables
- **heap** – segment used for store any dynamically allocated data. Dynamically allocated data are those whose size is known only in run-time
- **stack** – used to hold any local variables and addresses of functions (size is known in advance)

Fixed memory is used whenever is binding **let** either as values or as pointers to a heap[3, p. 43], we can summarize this as everything except *smart pointers*[5] and collections. Any time when function or method is called *stack frame*[6] is being created. These values are removed in reverse order, such as **LIFO**. The main benefit of fixed allocation is speed, allocation/de-allocation memory requires just one CPU instruction (increasing/decreasing stack frame pointer). Each *stack frame* is accessible only until the program leaves scope[7] in which was *stack frame* created.

Dynamic memory is used to allocate variables that have to outlive scope and collections on the stack. Reference to allocated data is stored in *smart pointers*[3, p. 55]. So far everything sounds the same as *C++ or C* but the main difference is not in allocation but in de-allocation. Rust uses *semi-automatic* mechanism for de-allocating based on Rust ownership model. Memory is de-allocated if one of the following conditionals is met:

- **Box** (the simplest form of heap allocation) goes out of scope
- **reference** count goes to zero

3.2.2 Concurrency

Rust's concurrency model relies on native operation system threads [12], API delivering necessary tooling to manage threads are delivered in standard library module **std::thread**. When we use threads for concurrency, we need a way how to exchange information between those threads. Rust offers two ways how to achieve this:

- **Message exchange** is mechanism to exchange information between threads. Rust standard library provides implementation to create *message channels*[8] to deliver this approach.

[5] Data that must outlive the scope in which it is declared.
[6] Stack frame is a logical block or memory on a stack that stores context of a function.
[7] Section of a computer program where the binding is valid.
[8] We can image message channels as tunnels where sender stays on the entrance and receiver on the exit. The message is passed to the tunnel where it is safe and no one can achieve this message until it reaches exit where the receiver takes it.

- **Shared state** is the second valid approach. With this mechanism, threads are accessing the same allocated memory. We can recognize this approach from *C* where threads are accessing one allocated memory and using locks to eliminate memory corruption. Rust is no different, shared memory is secured by **Mutex locks**[9]. Difference between rust and *C* is that with Rust we have to respect the ownership model.

  So we cannot move ownerships of mutex variable freely between threads. Luckily, Rust provides yet another concurrency primitive named *Atomic reference*. This primitive solves the problem by implementing synchronization mechanisms.

Threads are the base unit of concurrency in Rust standard library. For situations where standard library cannot be used, Rust also provides building blocks for creating co-routines or non-preemptive multitasking. These blocks are called **async/await**[10].

Embassy

Embassy is an async *executor* which schedules a fixed number of tasks without the need to allocate anything on heap [5]. Embassy is basically delivering a way how to write and manage concurrent applications on embedded devices using **async/await**. Besides that Embassy also provides *HAL* to enable access to peripherals such as *UART, I2C, SPI* and others.

Executor is a function whose only purpose is to poll tasks. Executor stores the tasks in the queue of tasks to poll. After the task is polled and it is completed successfully (there is no other responsibility for a task to be satisfied) task is returning **Poll::Ready** and the executor removes the task from the queue. If there is something more what task needs to do and currently it is blocked (usually a task is **.await**ing an async function) task returns **Poll:Pending** then executor takes a task and put it at the end of the queue.

Interrupts every input from a single button, keyboard, or other peripheral usually raises interrupt. Embassy has built-in support for interrupts which fits into Embassy async architecture. Let us imagine the situation when the executor polled the task which then instructed the peripheral to do something and now it is waiting for a peripheral interrupt. After the interrupt is received, the interrupt handler wakes the task and notifies the executor which polls task again.

Applications working on top of the Embassy as the async executor, work in specific workflow, displayed in Figure 3.5 from MQTT client perspective. This workflow is very important. Based on knowledge of this workflow it is possible to create effective async applications that are need access hardware. In case of this thesis we are talking about the Drogue-device demonstration 6.3.

[9] More information about the mutex locks is accessible from: https://doc.rust-lang.org/std/sync/struct.Mutex.html

[10] **async/await** are special pieces of Rust that enable asynchronous programming in one thread [13]

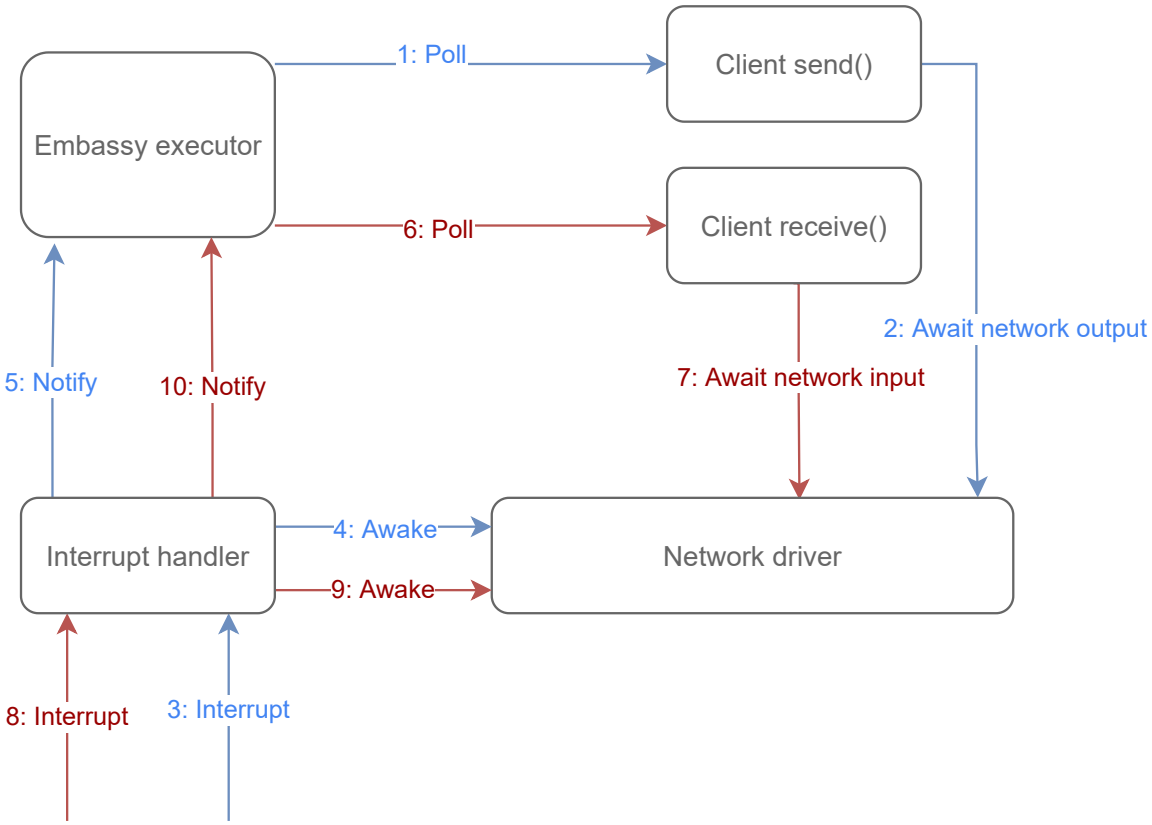


Figure 3.5: Async scheme representing position of the client in the actor model that has to be followed in order to use embassy async executor.

Poll is the first step of the workflow, as it is described in Embassy Subsection 3.2.2. It creates asynchronous running task which is on top of executors FIFO queue.

Await as the name suggests, this step is about waiting. After instructing network peripheral using network trait, the client has to wait until the operation is complete. If the last operation failed, it can either repeat the failed operation or cancel the task. Await goal is fulfilled after the network peripheral raises the interrupt signal.

Interrupt after an operation on the network peripheral is done, the interrupt signal is raised, marking the operation as done. Interrupt handler than routes this signal to driver and notify executor.

Awake the goal of awake is to ensure driver internal state is updated so the operation, after which was the interrupt raised, can continue.

Notify the purpose of notify is to inform the executor that the task which was waiting is now ready to continue (to be polled again).

Chapter 4

Client design

This chapter goes through the requirements for client design and implementation, client design models, and expected use cases for the client. The requirements are derived from the Drogue-IoT limitations for the clients and also the characteristics of the embedded devices.

Client architecture sections contain two diagrams that describe how the implementation of the client should behave, class and sequential diagram. The class diagram displays the client structure composition. The sequential diagram complements the class diagram and shows how the client communicates. This diagram also points to actions that are required from the user to execute client methods.

The last topic described in the section is two expected use cases of the client, Industry 4.0 and Smart home. Both of the areas are a great fit for this client. However, the Industry 4.0 use case is the main area for the Drogue-IoT, so this example is described more deeply in the chapter and also includes an example of such a use case.

4.1 Requirements

The client is going to be used in various situations on large scalce of the devices. The aim of the requirements is to ensure that clients work effectively and follow all constraints derived from expected use cases, platforms, and used technologies.

Functional requirements are derived from the thesis assignment and Drogue-IoT framework. In summary, there are three functional requirements:

- support MQTT version 5 (possibility of extending support for version 3)
- support Quality of Service level 0 and 1
- enable username/password authentication (Drogue-cloud MQTT endpoint authentication 2.2.1)

Non-functional requirements define constraints that affect how the system should perform.

- client memory utilization must be user configurable. While the MQTT packets form the most client's RAM usage, configurable maximal packet size ensures that client will not be limited by hardware on Drogue-device supported devices 2.3
- library can sustain a load of at least 10000 messages with no side effects
- client does not increase hardware utilization over the run time

4.2 Architecture design

Class design of client displayed in Figure 4.1 respects all shortages of Rust:

- rust has no inheritance.
- specific memory model.

Besides the shortages from Rust, the client architecture is created around the fields with a variable length that MQTT protocol allows (properties). Meaning that end-user controls the number of these fields in the whole client library. In the end, design provides an API that is easy to use and is sufficiently maintainable.

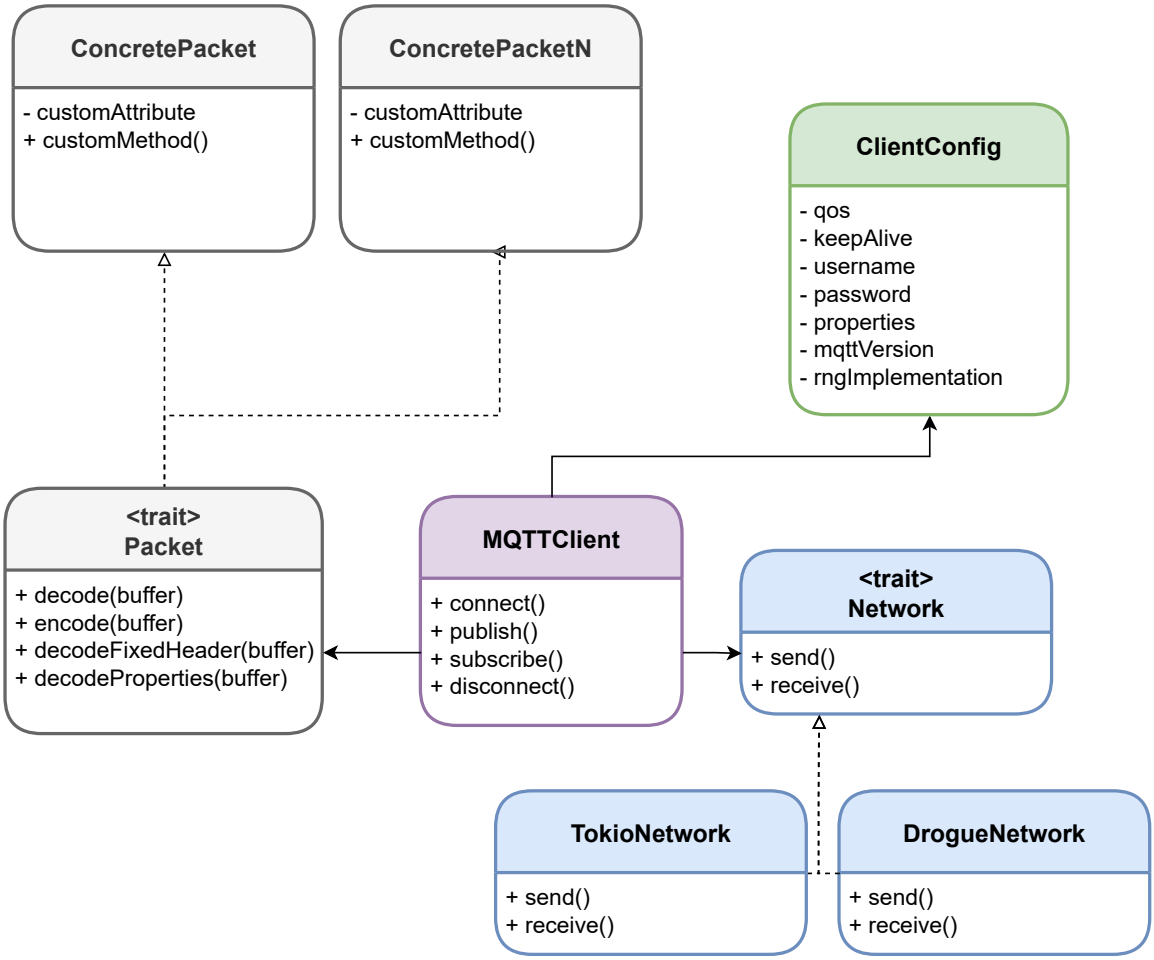


Figure 4.1: Diagram displays the class diagram of the MQTT Client library.

MQTTClient is the core of the client design. This structure should contain client actions for both versions of the protocol. Having separate implementation for each protocol version could be beneficial if the client would support over two versions of the protocol. Otherwise it just complicates an API. Client structure carries all the protocol logic in the client library. Client selects correct actions based on the configuration structure **ClientConfig**.

ClientConfig is the structure that carries client configuration properties. Configuration contains authentication information, version of the protocol, attributes of the connection (Quality of Service, session timeout) and list of properties. This list can contain any property specified in the MQTT standards. Each control packet choose set of the properties from the list and include correct subset to its own property list.

Packet trait specifies the interface that defines the set of the methods and default implementations for specific implementations of the control packet structures. From diagram shown in Figure 4.1, we can notice that **encode** and **decode** methods are requesting buffer without specific size as parameter. Reason for that is the balance between MQTT packet size and device RAM.

Library simply cannot allocate a locked amount of memory for each device, because the size of MQTT packets can be up to 256 MB[1]. That is more memory than most embedded boards have at disposal. This responsibility is handed over to the user. User has to allocate buffer for messages, either statically or on the stack, and hand over a reference to that buffer to client. This way will client always have the information about the maximum size of the message.

Network trait it makes little sense to include many network drivers in the client. Each driver would increase project maintenance needs. For this reason, the client model includes a trait named **Network**. This trait is basically an interface that the user has to implement with some network driver and pass it to the client in order to enable network communication. Besides the trait, client includes two adapter implementations of this trait, DrogueNetwork and TokioNetwork. These adapters are based on the adapter design pattern and they are transforming tokio and drogue network implementation onto compatible network interface [6].

Sequential diagram of the client's example usage is shown in Figure 4.2. The diagram displays position of the receiver and publisher. Although it could look like synchronous communication, the left and the right side of the diagram are time-dependent only from publisher to subscriber. The diagram does not include the network driver position because of the readability reasons. The flow displayed on this diagram proves that usage of the client is simple and there is no necessary overhead.

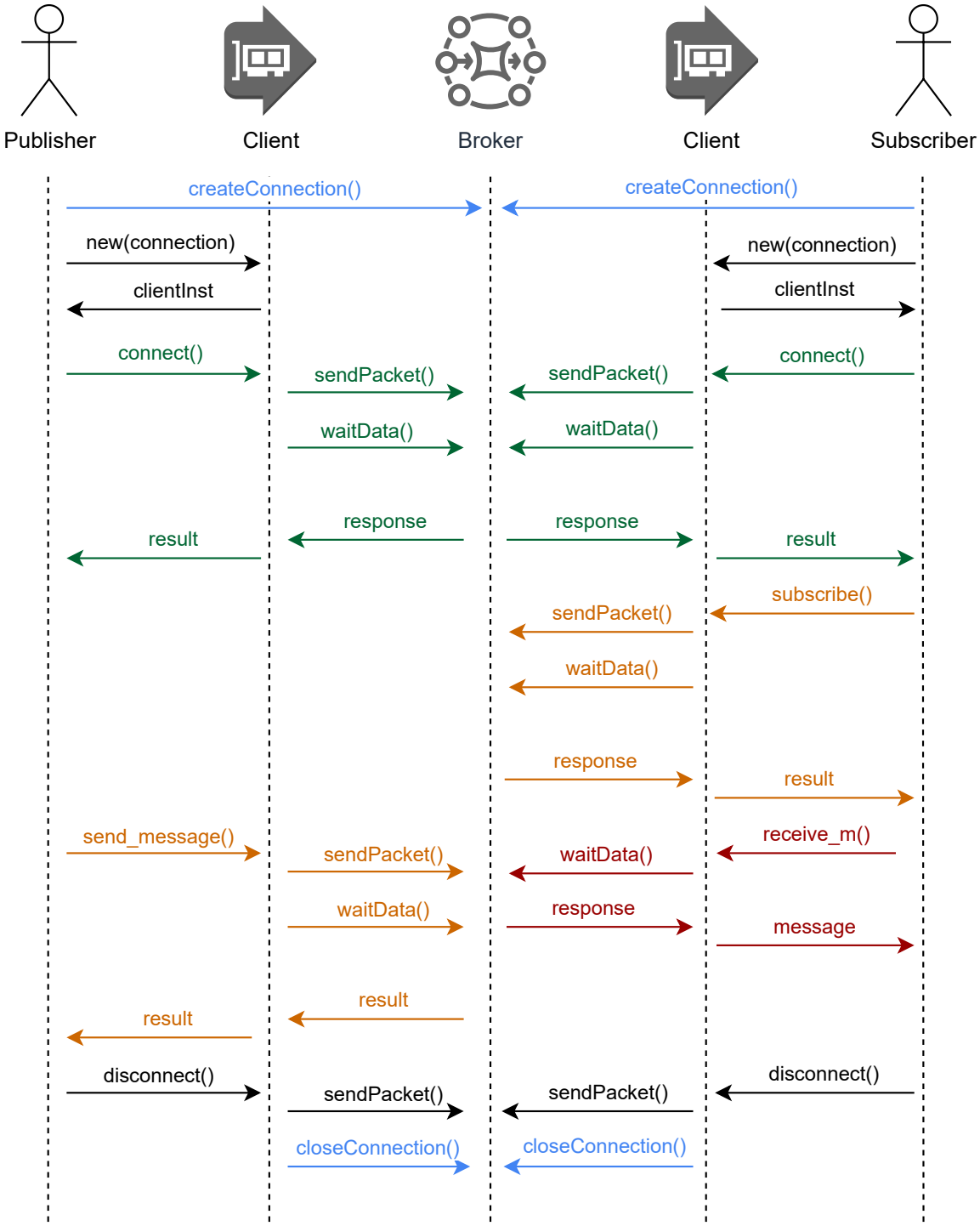


Figure 4.2: MQTT client sequence diagram that displays async communication of the client.

4.3 Use cases

When someone speaks about the word IoT, there are usually two main situations. The first situation is industrial IoT, also known as *Industry 4.0*. The Industrial IoT environment is a perfect fit for this MQTT client. Devices are reporting the state of the monitored machine to some central server. And in some situations, accept commands that machines have to execute, such as shut-down, start-up, and others.

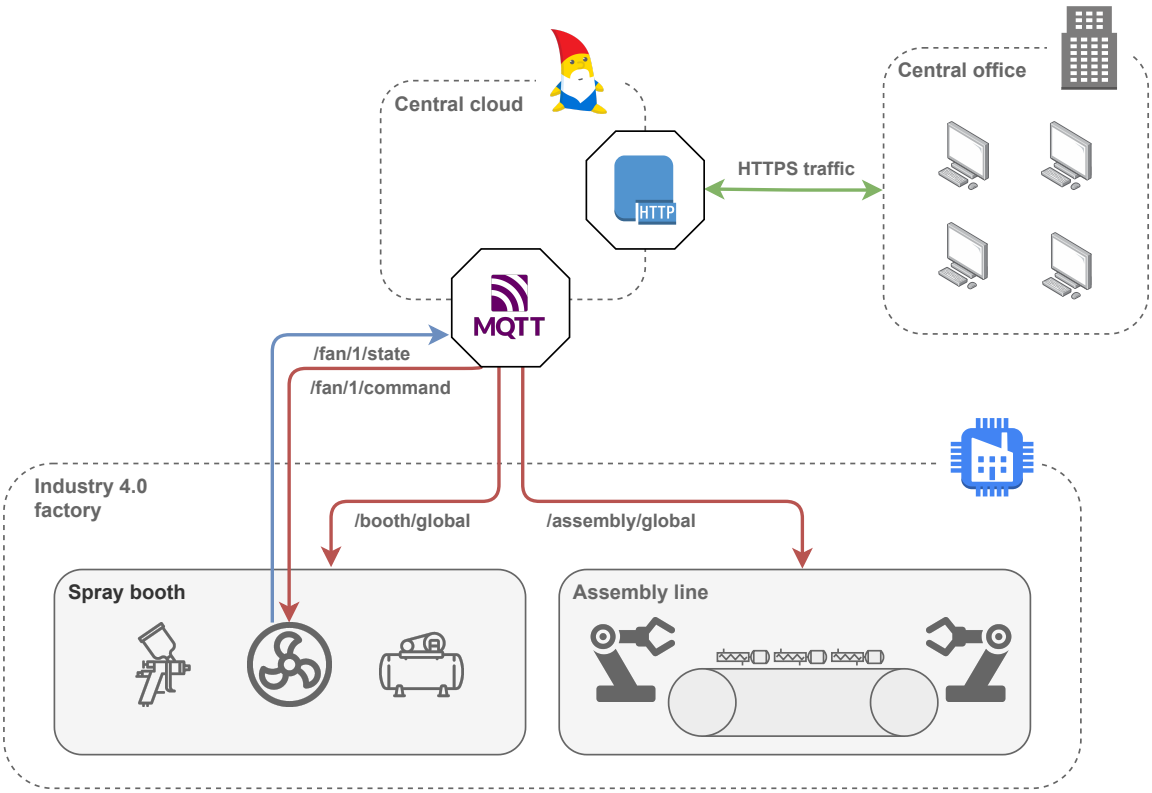


Figure 4.3: Example use case of async MQTT client for embedded devices in Industry 4.0 environment.

In Figure 4.3 we can see the typical use case of this MQTT client in industrial IoT. There are various machines which are controlled via a built-in embedded controller. The controller is monitoring the state of the device and periodically sends it to the unique topic through the MQTT gateway on the central cloud stack. Figure 4.3 also express the model situation which can be reached in industrial IoT.

In this example fan gets stuck because some system failure. Controller detects this failure and send it towards topic **/fan/1/state**. Machine is listening on topics **/fan/1/command** and **/booth/global** for commands. Cloud proceeds the message and sent towards the company central office. Immediately after the central office receives failure, it is evaluated. If machine failure affects other devices, the central office will send command *stop* to topic **/booth/global**.

All controllers listening to that topic will immediately stop their machines to prevent more damage. This principle is repeated after the machine is fixed. The state will be sent towards the central office and evaluated. Based on the state central office will send a command to start all devices through the global topic. Of course, machine failure does not

have to affect other devices. Here, all commands would be sent just to **/fan/1/command** topic so other machines can continue working.

The second situation which IoT covers is a smart home. Smart home architecture is not very different from an industrial one. The only small difference is that smart home aims more for commands (lights on, gate open, and others) than monitoring (speaking only about sensors or actuators that are supposed to send and receive some messages via MQTT). The central stack is usually much smaller than the industrial one, and there is no need to have something scalable.

Example usage of MQTT client in a smart home is mostly remote relay control such as opening a gate to the house. The user takes a phone and clicks on the gate icon, which usually results in some kind of command for the central server. Here, the server is instructed to send the command to the gate controller. The command would be sent to the unique topic where the gate controller is listening. The gate controller receives the message (command) and opens the gate.

Chapter 5

Implementation

The implementation chapter goes through details of the client implementation. It summarizes the project repository[1] architecture together with the purpose of each module. Later in the Client implementation section, there are descriptions and examples of the various implementation obstacles and specific solutions for each of the repository modules.

After the client implementation section explains the core of the client implementation, the next section explains the network trait and adapters implementation. The next section also contains specific information which is necessary to extend the client by new network implementation and also the description of existing adapters, Drogue and Tokio network adapters.

The last two sections show details about used external libraries and Rust package manager. As the client has to work on the **no_std** environment, the number of libraries that can be used is significantly reduced. The libraries section goes through all the external libraries used for the implementation. The Rust package manager section points out specific parts of the build configuration files.

5.1 Client implementation

Client implementation has been logically separated into several packages (modules):

- client
- encoding
- network
- utils
- tokio_net

These packages contain structures and traits that are having a common focus on functionality. Some packages are mostly formed with structures and functions that are used to manipulate buffer, memory and network. These packages and structures will be in this chapter call utility structures and packages. All the packages are joined in *client*, see Subsection 5.1, package.

[1] Repository can be found at:https://github.com/obabec/rust-mqtt

Utils package contains structures and types which are mostly focused on work with memory. The most significant parts of this package are *BuffReader* and *BuffWriter*. These are used for reading and writing into buffer supplied as a parameter.

An interesting aspect of the code below 5.1 is the return type, which is used **Result<(), BufferError>**. That type is specific to Rust, methods return either success with void or error with a variant of specific enumeration.

```
1       pub fn write_binary_ref(&mut self,
2       bin: &BinaryData<'a>)
3       -> Result<(), BufferError> {
4           self.write_u16(bin.len)?;
5
6           return self.insert_ref(bin);
7       }
8
```

Listing 5.1: Example client code displaying error delegation.

That is interesting, but what is even more teasing is operator **?**. This operator is bound to the Result type. If the result is success operator unwraps the value (void in this case) otherwise it takes an error and returns it immediately from the method/function.

Encoding package contains Decoder and Encoder for variable byte integer, which is described in MQTT version 5 OASIS standard [1]. Error delegation in this package works same way as previous package only this time there is a use of special type **VariableByteInteger** that is basically type alias as we know from other languages like *C or C++*.

Packet as Figure 3.1 shows, MQTT provides various control packets. These packets have to be mapped in a protocol to ensure communication functionality. This mapping is stored right in this package. There is a public trait **Packet** which contains a declaration of all methods which have to be implemented for specific packet types and contains a default implementation for common features which are the same for all the packet types.

The rest of the structures for control types simply map packet binary form into Rust structures. At this moment, there is a massive obstacle which has to be overcome. MQTT version 5 enables users to include properties of variable lengths and amounts in the packet.

With embedded platforms in combination with variable lenghts, there is a problem. As it was said, there is no dynamic allocation, so there is no way how this could be variable. We have to know the exact size during the compile-time. Rust provides a solution named *const generics*[2] that allows programmer to parameterize a structure or method with a constant. In the manner of this client, it allows to parameterize structures with the expected length of buffers that store fields of variable length. Let's get through this by the example of publish packet.

[2]Rust generics: https://rust-lang.github.io/rfcs/2000-const-generics.html

```
use heapless::Vec;

pub struct PublishPacket
<'a, const MAX_PROPERTIES: usize> {
    pub properties: Vec<Property<'a>,
                        MAX_PROPERTIES>,
    pub message: Option<&'a [u8]>,
}

let pub = PublishPacket::<'b, 5>::new();

```

Listing 5.2: Code example explaining the Rust const generics usage in the client library.

In the Listing 5.2 is definition of **PublishPacket** structure which contains explicit lifetime annotation **'a** and const generic argument **MAX_PROPERTIES**. This argument sets the size for heapless vec[3] during the creation of packet structure so variable length of properties is maintained and the user can decide how many properties will need with no limitations from the client-side.

Aside from mapping packets also contains implementations of trait methods. The most crucial of these are *decode and encode*. These two methods are the core of the whole client library. Decode methods decode incoming messages from raw format into the usable structures with which can client manipulate. Encode method do exact opposite. Without them client could not work with packets effectively.

Client contains an implementation of MQTT version 5 compatible client structure and configuration structure **ClientConfig** in the client package. Client structure holds config as an attribute and passes corresponding parts of config to each of the control packets. Client contains implementations of *Actions* from the MQTT standard. The most significant obstacle here is the hassle with lifetimes.

```
pub async fn connect_to_broker<'b>
(&'b mut self)
-> Result<(), ReasonCode> {

    let mut connect =
    ConnectPacket::<'b, 2, 0>::new();

    if self.config.username_flag {
        connect.add_username(
                &self.config.username);
    }
}

{ client.connect_to_broker().await };
{ client.send_message(topic, MSG).await };

```

Listing 5.3: Code example that displays problematic of the Rust borrow lifecycle.

We can see the example right in the code above. Method **send_message** also uses one of the client's attributes - *config*. Firstly client is passed as a mutable reference with lifetime **'b** which means the reference will live only in the method's scope. Later

[3]Heapless crate: https://docs.rs/heapless/0.2.1/heapless/struct.Vec.html

is attribute **config.username** as a reference handed to created connect packet (packet lifetime is also set to **'b**).

Once the method is done, all variables and references with lifetime **'b** are destroyed and they can be freely moved to another method. If the lifetime was not specified, the client mutable reference could not be passed to the next method because the reference in the packet would outlive the scope.

5.2 Network trait and adapters

Achieving compatibility with all the network drivers that exist is not possible. Client provides implementation of two network adapters for both embedded and non-embedded network drivers. These adapters are for tokio network and Drogue-device network driver. In order to achieve maximum network driver compatibility, library also provides public network traits:

- **NetworkConnectionFactory** which should be used to establish a connection
- **NetworkConnection** containing all methods necessary for working with TCP stack

Users with specific needs can adapt these traits onto their network driver and pass adapter to the library. Both traits contains specific return types. These types are Rust futures. Future traits represent an asynchronous computation that may eventually produce final value [15] of the **NetworkConnection** or **NetworkConnectionFactory** actions (send, recv, connect, close).

Tokio network adapter first network trait implementation is *Tokio network*. Implementation is stored in a package of the same name. This implementation adapts *Tokio* network that is contained inside *Tokio* async library[4] into providing network traits. Network implementation in Tokio aims to support network driver for standard devices (non-embedded).

Adapting such a network is not really great example for this project because network running on standard devices rarely need to close connections because most of the systems can close these themselves. Having this implementation means the client offers full support for non-embedded devices using *Tokio* runtime. This network adapter is not the primary goal of this thesis but having such implementation is necessary to make whole development easier because debugging and testing are things which are in most time very problematic and time-consuming on embedded devices.

Drogue network adapter second implementation of network traits is *Drogue Network*. This adapter is located directly in Drogue GitHub repository[5].

Having support for Drogue means that the library now supports all devices and Wi-Fi chips that are supported in Drogue framework. This is much more beneficial than having support for just one type of device, which is scope of this work.

Behavior of the Drogue Network differs totally from Tokio network because we have to respect structure of Drogue-device firmware.

[4] Available from: https://tokio.rs/

[5] Drogue Network available from: https://github.com/drogue-iot/drogue-device/blob/0385306/device/src/network/clients/mqtt.rs

```
pub struct DrogueNetwork<A>
where
    A: TcpStack + Clone + 'static,
{
    socket: Socket<A>,
}

```

Listing 5.4: DrogueNetwork implementation code displaying usage of actor model.

As the code above displays, instead of keeping some address to the socket network structure is keeping the address of the socket as the TcpStack. This way client can communicate with an actual TCP connection (open, send, receive, close).

Besides that standard connection, there is also an adapter implemented for the Drogue **TlsConnection**. This implementation allows using TLS. Support for TLS is necessary to allow connection with public Drogue-cloud. This adapter is later used in the evaluation application 6.3 with Drogue-cloud sandbox.

5.3 Libraries

Libraries used during the implementation do not carry any main functionality but they are making implementation, debugging, and even readability of code much more simple. All libraries that are covered in this section are accessible from community crate registry[6].

Regular build described in Section 5.4 does not include all the libraries. These libraries are development libraries. These dependencies are in Rust, usually used for things such as logging or other supportive features for developers.

Heapless library provides friendly data structures that do not require allocation on heap. In, case of this project, we talk specifically about *Vec*. Small disadvantage which is, of course, understandable is that this structure does not support any kind of reallocation (meaning size of *Vec* given during initialization cannot be changed or overridden afterwards). There is an enormous advantage. Having data structure such as *Vec* without memory allocator means developer does not need to worry about usual exceptions such as *OOM (Out Of Memory)*.

Rand core rand core provides traits for random number generation. As some embedded devices have special hardware support for generating random numbers (usually based on hardware timers), we cannot provide strict implementation.

Client provides only very simple implementation of these traits (counting random generator) but users can pass own Rng implementation to the client via configuration visible in Section 4.1. Client uses the rng implementation to generate packet identifiers.

Logging is necessary for every application and this project is no different. Although supporting both embedded and non-embedded runners makes things more complicated. Usually one logger implementation is enough but embedded devices need much more specific approach which is not necessarily great fit for standard devices. Because of that is the client using two different implementations for logging *Defmt* and *Env logger*.

[6] Accessible from: www.crates.io

Defmt library is a high performance logging library for embedded devices. Defmt provides limited formats of log messages and possibilities, which are absolutely sufficient for the embedded world but not for standard environments. We would surely like something more complex. *Env logger* fulfilled this requirement, basic logging library for standard devices which allows configuration expected on non-embedded devices such as logging to std-err or std-out, filtering specific logs and formatting log messages to specific format including timestamps.

Client code provides module that automatically selects which logging implementation it should use based on the Cargo features. This way is all decision making around the logging hidden from end-user and makes API of the client more simple.

5.4 Rust package manager

Cargo is the package manager for Rust. Cargo is basically entry point to programs written in Rust. It downloads dependencies, compiles program and makes distributable packages. We could say that Cargo is very like Maven for Java language.

The main configuration for Cargo is located in *Cargo.toml* file in root of the MQTT package. Four sections form standard configuration:

- **package** – contains information about crate which will be displayed in crate registry once crate is released.
- **dependencies** – section contains dependencies (libraries) that will be linked to application during every compilation. It will include these dependencies every time regardless build features specification.
- **dev-dependencies** – part is very interesting. It contains dependencies which will be used in tests, benchmarks or examples. It will not include these dependencies in final build of the application.
- **features** – section creates *parameters* for the build. Each parameter (feature) contains a list of optional dependencies, that are included in build once is the parameter specified. In scope of this project, features are the key to compile on both embedded and non-embedded environments.

Chapter 6

Testing

Aim of this chapter is to describe the testing process in the client repository. First, it goes through all the test levels that are included in the client test suites. These levels are: unit, integration, load and performance. Unit levels tests each component of the client in an isolated environment. Integration level aims for the interaction between system components. Load test level validates if the client can endure a high number of messages with no side effects. Last, probably the biggest part of the testing is performance.

Performance testing is last part of the test levels section. This level of testing provides metrics gathered during manual performance testing. This metrics, as it is described in the end of the section, helped to find a significant performance issue.

Client's library is open-source. That means anybody from the community can contribute to the project and something has to keep things in best shape possible. System that helps to achieve this is described in the second section of this chapter. CI/CD provides *Continuous Integration and Continuous Delivery* functions. Continuous integration functions serves the automation testing of the pull requests. Continuous delivery is going to be used for automatic release of the library into public crate repository[1]. System that combines both of the functions and is used in this repository is named GitHub actions.

Last part of this chapter is the evaluation of the client implementation. The evaluation was done on the Micro::bit V2 micro controller connected to Adafruit Huzzah ESP8266 microchip that serves as Wi-Fi module. Evaluation displays that client can work asynchronously on a single embedded device. Application contains publisher which sends the messages towards the broker after button is pressed and async receiver that subscribes to the topic and waits for the new messages. Once the message arrives, receiver displays it on the LED matrix display.

6.1 Test levels

Currently there are three automated testing levels wit for this library:

- **unit** – main purpose of the tests on this level is to test functionality of each component in an isolated environment. This test level together with integration level is automated and run in the *CI/CD* 6.2.

[1] Available at: https://crates.io/

- **integration** – level is used to test interactions between the components of the system. That is the main job of this level also in this library, but there are also scenarios included which could be identified as *end to end*.
- **load** – includes tests that aim for behavior of the system under load (high message rate).

Having all these different test levels ensures that all library requirements are met and there should be minimal amount of bugs in the merged software. Tests included in these levels all use non-embedded desktop runner. Possibilities how to run automated tests on some embedded device are very limited and make little sense to this library. Reason for it is that all problems which are specific to embedded are resource based (insufficient memory size) and these are even harder to emulate.

Once we look at this problematic from a higher perspective, keeping the library and Rust design in mind, we can assume that testing on embedded devices does not really bring any higher advantage. If the code is compilable[2] for embedded architecture and all tests for all the functionalities completes successfully (on non-embedded). Only possible failures are those made by user. As example would be a mismatch of configuration for the target device hardware equipment.

Tests could run directly on the embedded devices. But there is second obstacle. Tests for embedded devices would have to live in the Drogue-device repository as the network driver is part of that project and Drogue-device. At the time of writing, does not provide any technology which could run those tests.

Unit tests

Unit tests are a fundamental part of the product testing. Primary aim of these tests is to confirm that each individual component works as expected. Isolated environment accomplishes this, meaning tests can test all components separately and integration between them does not affect functionality of the component.

In the manner of this library, unit tests provide confidence on most important parts of the client. All these parts aim on memory, so we are speaking about buffers, vectors, and streams. Testing functionalities of such components with unit tests is fundamental because unexpected behavior of the component is usually hidden and takes a huge amount of debugging in order to find and fix the failure in production code.

Standard Rust practice is to have the unit tests in the same module as the tested component. For this thesis, unit tests are stored in separate module to achieve more consistency and readability. Module containing unit tests is called *tests/unit* and it is located inside the client library because of that these tests do not use *dev-depenencies* introduced in Section 5.4.

Integration tests

Integration tests are testing interactions between system components. That means tests on this level the test client library as one component. Usually we can find a huge amount of test cases for this test level, but with MQTT version 5 is everything little more difficult.

[2]Compilation of the code to embedded platform secures one the the CI pipelines, more described in the Section 6.2

This protocol version does not enforce exact reactions for most of the actions, so each broker can behave differently.

This behavior is a problem and a big one. One of the test cases cover maximum packet size property, but it simply cannot because, for example, Mosquitto broker simply kill TCP connection with no Disconnect packet with some reason code. This behavior cannot be properly tested because TCP connection could be closed because of totally different reason.

Because of problems described above, integration tests are simplified to most possible configurations of the client, which output can be determined from OASIS standard. These tests are executed against multiple MQTT broker open-source implementations (currently Mosquitto and HiveMQ).

Following Rust best practices about testing integration, tests lay in specific module *tests* that is in the client's core next to *Cargo.toml* cargo configuration file. Tests in this module are using *dev-depenencies*.

Load tests

These tests can be executed against whatever broker. These tests are not executed in regular builds because results are highly depending on the environment (current state of broker, network throughout). This tests level contains tests that tries if the client is working without any side effects under load. Load used for these tests is from hundred to twenty thousand messages.

During the development, load tests were executed a couple of times and results were very satisfying. Client shown no kind of load problem. Tests with enabled quality of service have zero failure percentage.

Performance testing

Performance testing is slightly specific than previous levels. Main point of the performance testing for this library is to collect performance metrics about the behavior of the client.

Application for the performance scenario is in custom fork for drogue repository. Code is prepared for micro:bit V2 board with ESP8266 Wi-Fi chip. Client is waiting for button press then sends the desired amount of messages and also starts the hardware timer which is measuring the time. Once application sends all the messages, it stops the timer and logs the value of timer into the terminal window.

Message count	**Action**	**QoS**	**Time (ms)**	**Stack (KiB)**	**Flash**	**Static RAM**
100	Send	0	2661	14.69	0xe91	0x20dc
		1	7777			
1000		0	37305			
		1	69568			
10000		0	336551			
		1	715689			
100	Receive	1	4385		0xefd4	0x2174
1000			44337			
10000			451740			

Table 6.1: Performance metrics gathered during performance experiments.

Performance experiments were executed on the standard home Wi-Fi network against Hive-MQ broker. Network was not under any bigger load at the time of the execution. All the experiments were executed 15 times. The results in Table 6.1 are arithmetic means of the actual results. Standard deviation of the results differs from experiment to experiments. All the standard deviations are recorded in Table 6.2.

Message count	**100**	**1000**	**10000**
Send QoS0	79.0 ms	183.9 ms	413.2 ms
Send QoS1	80.7 ms	250.3 ms	573.2 ms
Recv	85.9 ms	198.3 ms	440.1 ms

Table 6.2: Standard deviations for each specific action and message count

It is clearly visible that QoS is adding some time and deviations are higher. It is an expected result because client relies much more on the broker and network 3.1.3. There is one more metric, which is not included in the table and that is the number of CPU cycles. This metric is not included because measurement library[3] uses only *u32* register to store this count. With longer experiments, such as these which have been executed count will overflow the register so metric would not be accurate.

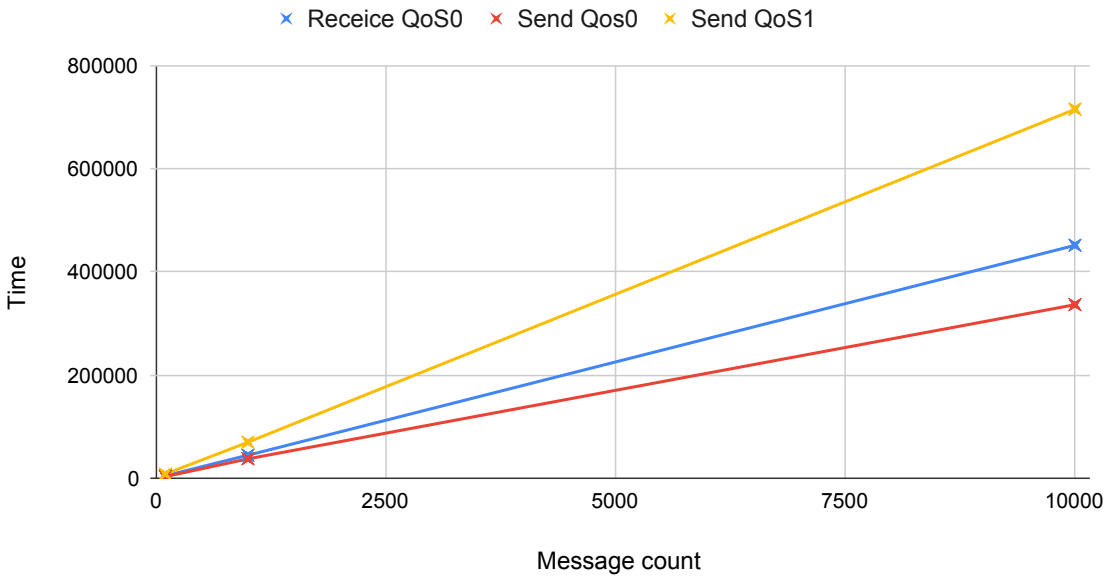


Figure 6.1: Graph displaying trend in the performance results.

Overall, results are highly satisfying. The course of the experiments have proven that library has a higher performance tan hardware which was available for the testing. When we examine Plot 6.1 we can see that implementation reports ideal results. As time needed for execution is growing linearly with the number of messages sent.

During the performance testing, there was also a significant issue found. The issue was in the process of reading packet from network driver. This process discovered that part of

[3]Available from: https://docs.rs/cortex-m/0.5.1/cortex_m/peripheral/struct.DWT.html

the packet could be discarded and that later results in loss of whole next packet. Fixing this issue[4] brought also significant performance improvement (20-25 percent).

6.2 CI/CD

Main point of CI/CD, in this open-source library, is to help with control of the pull requests from community and maintainers and automate the release process to public crate repository, including storage for build archives, documentation and rest of the released artifacts.

GitHub provides a perfect solution that is already integrated into GitHub itself and it is free for open-source projects such as this one. GitHub Actions are configurable pipelines running on container platform. There is also a public marketplace for already created steps.

Actions are composed from several key parts:

- **workflow** – a set of jobs that is triggered after the trigger condition is met (created pull request, published release, and others)
- **job** – pipeline containing steps
- **step** – the individual part which can execute usual system commands

Workflows are configured via yaml files stored in *.github* folder in root of the repository.

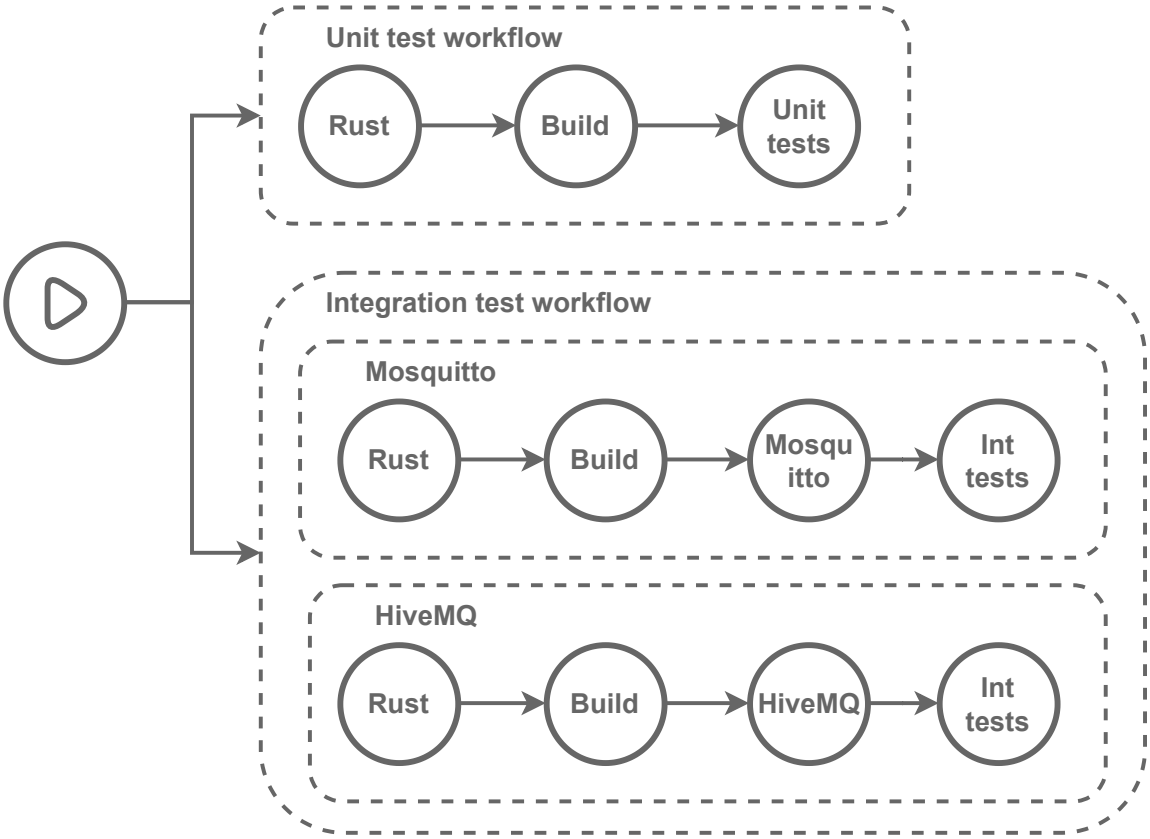


Figure 6.2: Schema displaying GitHub Actions workflows for client library.

[4]The issue was fixed in the commit: https://github.com/obabec/rust-mqtt/commit/0e7bdadf5

Figure 6.2 displays schema of workflows, jobs and steps used in this project. Each circle represents on step in GitHub action job. To keep things simple default GitHub action steps are omitted.

First workflow contains only one job. Job contains three steps installing specific Rust tool chain, building project with default features (meaning with *std*) an execution of unit tests. Second workflow is more complex and is composed of two jobs.

Jobs are almost the same, but there are different broker implementation deployed for each job. Build in the integration test workflow is configured with embedded target without default features, which assures that code is deployable on embedded devices.

Next to the CI workflows, there is one continuous delivery workflow. This workflow ensures automatic releases into public crate repository[5]. This workflow starts once a new release is created in GitHub. There are three steps in this job. The first job runs the unit tests, just to ensure nothing has been broken by accident in the final code of the client. The next step tries compilation to the embedded platform. The last step publishes the library to the crate repository.

6.3 Evaluation

For the evaluation purposes, I have created demonstration application which will connect Drogue-cloud with MQTT client[6] and show the usage of this combination in the real world. This application runs on the Micro::bit V2 and *ESP8266* Wi-Fi chip.

Micro:bit V2

Micro:bit is a single board computer which is build on top of ARM Cortex-M4 nRF52 processor with clock speed of 64MHz and 128KB of RAM[7].

Computing power and memory is definitely smaller than rest of the single board computers but micro:bit offers a big amount of built-in peripheral (led matrix, buttons, gyroscope, and lot other).

Main reason for selecting micro:bit is size and also accessibility. For this board, there is no need to get any other sensor or some other I/O device. It is a complete package.

ESP8266

ESP8266 is one of the most used Wi-Fi chips. These days, we usually cannot say that the boards offering the Wi-Fi module are only Wi-Fi chips. Most of these ESP8266 boards, such as NodeMCU, are completely independent micro controllers. However, Drogue firmware currently does not allow to run on these micro controllers alone. This means we need some firmware that will allow us to communicate with the Wi-Fi module.

Most of the ESP8266 boards include *AT*[8] firmware in order to allow communication to Wi-Fi module alone. This is also the case for the ESP8266 used on demonstration video. Evaluation device contains ESP8266 manufactured by *Adafruit* and is built on top of ESP-12 module.

[5] Available at: https://crates.io/crates/rust-mqtt

[6] There is also second demo, running demonstration app with non-TLS connection and plain MQTT broker. This video is accessible from: https://nextcloud.fit.vutbr.cz/s/zx27acoZHZ23Mt7

[7] Whole specs available at: https://all3dp.com/2/bbc-micro-bit-v2-review-specs/

[8] Firmware for the esp devices from the EspressIF. More information can be found: https://www.espressif.com/en/products/sdks/esp-at/overview

Adafruit supplies the chips with firmware from manufacturer *EspressIF* (AT). However, version of the firmware was not sufficient for Drogue-device. That results in flashing the evaluation Wi-Fi chip with the AT firmware version *1.7.0.4*[9].

Demonstation application

The whole code is aligned around the Actor model 2.5, which allows running and managing asynchronous applications on one thread. Code is separated into two actors.

- **Main** – contains publisher functionality. First, there is a configuration of the board and the following peripherals (Led Matrix and ESP8266 Wi-Fi chip). After that, Drogue-device establishes the TLS and TCP connection. Receiver actor is spawned with passed connection.

  The main application loop follows. This loop contains asynchronous wait for the trigger of button A. User press the button and MQTT client will send *„temp“: 42* json message to the specified application in the Drogue-cloud sandbox instance.

- **Receiver** – contains all configuration and logic of MQTT receiver. Once the client is configured, it connects to the broker and subscribes to the specified command topic. Then the main receiver loop starts. The client is waiting for a new MQTT message. When the MQTT message arrives, it executes display function on the **LED Matrix** driver with the payload of the MQTT message. The

Besides the code of the Rust demonstration application, there is also a script that will create the echo on the cloud side. Meaning it will listen on the MQTT integration. Waiting for the device MQTT message. Once the message arrives, it extracts message data and sends the data to the command topic for the device.

This example represents the expected usage of the client and is displayed on the demonstration video[10]. Several client instances running on the same device with the possibility of different configurations for all these instances.

[9] Available open source at: https://github.com/espressif/ESP8266_NONOS_SDK
[10] Accessible from: https://nextcloud.fit.vutbr.cz/s/n7GL2RtpCzAKxRL

Chapter 7

Conclusion

The assignment for the thesis was to study basic principles of the Drogue-IoT, MQTT protocol, Rust programming language, and Embassy asynchronous library for embedded devices. The second part of the assignment was also to design, implement and test MQTT asynchronous client library (fully supporting MQTT v5) in Rust programming language for embedded devices. Respect all constraints introduced by intended usage on these devices enforced by the Drogue-IoT during the design and implementation.

During the study of technologies, I have been able to identify all requirements introduced in Section 4.1. These requirements mostly come from the embedded platform itself. Embedded devices have various amounts of hardware equipment, so every step in design and implementation has to aim for maximal configurability (to use limited hardware as effectively as possible).

The design described in Chapter 4 respects all the requirements introduced in Section 4.1. Implementation shown in Chapter 5 follows this design. Functionality, usability, quality of design, and implementation were confirmed using not only automatized testing but also demonstration application and performance experiments.

The main future improvement is to add support for the MQTT version 3. Second improvements can be done on the optimization side. Most of the MQTT brokers implement *Nagle's algorithm*[1] to improve the efficiency of the TCP protocol. The client does not implement this feature at the moment. Implementing such feature would surely bring even more satisfying performance results.

The client is fully functional and currently, it is the first open-source implementation of the MQTT version 5 client in Rust language. The client library is accessible on GitHub[2]. The community interest in this project can confirm the usability and quality of the library, as Appendix B shows, one of the companies in the Czech IoT environment will use the client in production systems.

During the work on this thesis, I have also created a proposal for the **Excel@FIT** conference. The proposal, see Appendix C, was accepted by **Excel@FIT** committee and was presented during the poster session. This paper was awarded by the main partner of the conference *EdHouse*[3] as the interesting industrial ready project.

[1] Nagles's algorithm can reduce the number of small datagrams sent over the network. Detailed information can be found here: https://datatracker.ietf.org/doc/html/rfc896

[2] Can be found at: https://github.com/obabec/rust-mqtt

[3] Information about the company can be found at: https://www.edhouse.cz/

Bibliography

[1] Banks, A., Briggs, E., Borgendale, K. and Gupta, R. *MQTT Version 5.0* [online]. March 2019. [visited 2021-01-09]. Available at: https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html.

[2] Banks, A., Cohn, R. J., Coppen, R. J. and Gupta, R. *MQTT Version 3.1.1* [online]. October 2014. [visited 2021-01-09]. Available at: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html.

[3] Blandy, J. and Orendorff, J. *Programming Rust: Fast, Safe Systems Development*. 1st ed. O'Reilly Media, 2017. ISBN 9781491927281.

[4] *Drogue IoT* [online]. 2020. [visited 2021-01-09]. Available at: https://book.drogue.io/drogue-book/index.html.

[5] *Embassy* [online]. 2021. [visited 2021-05-10]. Available at: https://embassy.dev/embassy/dev/index.html.

[6] Gamma, E. *Design patterns : elements of reusable object-oriented software*. 1st ed. Boston: Addison-Wesley, 1995. Addison-Wesley professional computing series. ISBN 0-201-63361-2.

[7] Gubbi, J., Buyya, R., Marusic, S. and Palaniswami, M. Internet of Things (IoT): A vision, architectural elements, and future directions. *Future generation computer systems*. 1st ed. AMSTERDAM: Elsevier B.V. 2013, vol. 29, no. 7, p. 1645–1660. ISSN 0167-739X.

[8] Hillar, G. C. *MQTT Essentials - A Lightweight IoT Protocol*. 1st ed. Packt Publishing Ltd., 2007. ISBN 978-1-78728-781-5.

[9] *MQTT Message Data Integrity - MQTT Security Fundamentals* [online]. 2015. [visited 2021-01-09]. Available at: https://www.hivemq.com/blog/mqtt-security-fundamentals-mqtt-message-data-integrity/.

[10] Naik, N. Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP. In: Ministry of Defence, Defence School of Communications and Information Systems, United Kingdom. *2017 IEEE International Systems Engineering Symposium (ISSE)*. 2017, p. 1–7. DOI: 10.1109/SysEng.2017.8088251.

[11] Nash, M. and Waldron, W. *Applied Akka Patterns: A Hands-On Guide to Designing Distributed Applications*. 1stth ed. O'Reilly Media, Inc., 2016. ISBN 1491934883.

[12] *Rust* [online]. 2021. [visited 2021-01-09]. Available at: https://www.rust-lang.org/.

[13] *Async book* [online]. 2021. [visited 2021-01-09]. Available at: https://rust-lang.github.io/async-book.

[14] *Rust RFC 2394* [online]. 2018. [visited 2021-17-10]. Available at: https://rust-lang.github.io/rfcs/2394-async_await.html.

[15] *Rust RFC 2592* [online]. 2018. [visited 2021-15-10]. Available at: https://rust-lang.github.io/rfcs/2592-futures.html.

[16] Troutwine, B. *Hands-On Concurrency with Rust: Confidently Build Memory-safe, Parallel, and Efficient Software in Rust*. 1st ed. Packt Publishing, 2018. ISBN 9781788399975.

Appendices

List of Appendices

Appendix A

CD Content

- **/rust-mqtt/*** — source code of MQTT client from May 8, 2022
- **/rust-mqtt/README.md** — README with useful informations about MQTT client build and start
- **/docs/*** — documentation of MQTT client from May 8, 2022
- **/drogue-device/*** — source code of Drogue-device used for the evaluation
  - **examples/nrf52/microbit/esp8266/mqtt/*** — demonstration application code
  - **device/src/network/clients/mqtt.rs** — Drogue-device network driver for MQTT client
- **/demo-tls.mp4** — demonstration video with Drogue-cloud and TLS
- **/demo-broker.mp4** — demonstration video with plain MQTT broker
- **/performance.xlsx** — complete performance results
- **/text/*** — source code of this thesis from date May 8, 2022
- **/xbabec00-thesis.pdf** — final version of this thesis from date May 8, 2022

List of Abbreviations

MQTT	Message Queuing Telemetry Transport
HTTP	Hypertext Transfer Protoco
CoAP	Constrained Application Protocol
IoT	Internet of Things
AMQP	Advanced Message Queuing Protocol
TTN	The Things Network
LoRaWAN	Long Range Wide Area Network
SSO	Single sign-on
API	Application Programming Interface
FIFO	First In, First Out
HAL	Hardware Abstraction Layer
TCP	Transmission Control Protocol
OASIS	Organization for the Advancement of Structured Information Standards
RBAC	Role Based Access Control
MAC	Message Authentication Code
ACL	Access-Control List
LDAP	Lightweight Directory Access Protocol
TLS	Transport Layer Security
SSL	Secure Sockets Layer
OAUTH	Open Authorization
QoS	Quality of Service

DUP	Duplicate Delivery of a Publish control packet
RAP	Retain As Published
LIFO	Last In, First Out
CPU	Central Processing Unit
UART	Universal Asynchronous Receiver-Transmitter
I2C	Inter Integrated Circuit
SPI	Serial Peripheral Interface
RAM	Random Access Memory
RNG	Random Number Generator
CI	Continuous Integration
CD	Continuous Delivery
LED	Light Emitting Diode
I/O	Input/Output

Appendix B

Commendation

DATAMOLE

Ondřej Babec
Faculty of Information Technology
Božetěchova 1/2, 612 00 Brno-Královo Pole
Czechia

April 23, 2022, in Prague

Letter of Commendation

Dear Ondřej,
My name is Matouš Hýbl and with this letter I would like to officially thank you for the work you've done on the **rust-mqtt** (Rust native mqtt client for both std and no_std environments) project. The work you've done will save us countless hours of development.

I work for a company called Datamole which is developing cloud based Industrial IoT solutions, where I am in a team developing embedded software for one of our products - a device for long running experiments with liquids in the food processing industry. As a part of the project we aim to use MQTT for communication between parts of the device and this is where we plan on utilising your project.

As per our prior research a library implementing MQTT for embedded Rust has not yet existed or was not in the desired state of completeness, which led us to believe that we'd need to develop it ourselves, but thanks to your project we don't have to. Your project is also a great fit for the reason that it utilises asynchronous programming (implemented using Rust's async/await mechanism), which will make integrating it with our system a piece of cake.

As I am also a contributor to Embassy - a project and an initiative to bring asynchronous programming in Rust to embedded devices, I can confidently say that your project extends the ecosystem with valuable communication capabilities, making it much easier for people to build connected IoT devices with Rust.

I'd also like to thank you for your promise to keep maintaining the project in the future and aiming to bring many other contributions to the whole ecosystem.

Sincerely,

Matouš Hýbl
Datamole, s.r.o.
Tel: +420 720 644 619
Email: matous.hybl@datamole.ai

Headquarters:
Datamole, s.r.o.
Banskobystrická 2080/11
Dejvice, 160 00 Praha 6
Czech Republic

IČ: 037 42 709
Tel: +420 608 535 730
E-mail: info@datamole.ai
www.datamole.ai

Office:
Datamole, s.r.o.
Vítězné náměstí 577/2
Dejvice, 160 00 Praha 6
Czech Republic

Figure B.1: Acknowledgment for the project received from Datamole company.

Appendix C

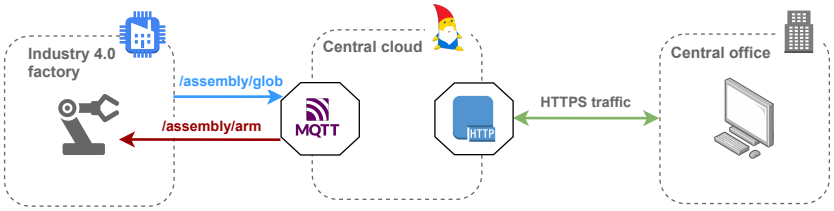
Excel@FIT

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2022
http://excel.fit.vutbr.cz

Asynchronous MQTT Client Library for Embedded Devices Running on Drogue-IoT Firmware

Ondřej Babec*



Abstract

The main target of this work is to create an asynchronous MQTT client, supporting MQTT version 5, in Rust running on embedded devices powered by Drogue device opensource firmware. The number of clients that support MQTT version 5 is highly limited, and currently no client implementation exists in Rust. The main implementation challenge is that MQTT version 5 has properties of variable lengths. Storing these properties of size which is unknown during compile time is a massive obstacle because embedded Rust does not support dynamic allocation as there is no underlying operating system.
The result of the work is a client that has comparable functionalities as other available clients in different languages. The client library is extended with both desktop and embedded async executors. Although the client could be used almost everywhere, leading variants are *Industry 4.0* and *Smart home*.
Keywords: MQTT — Async — Rust — Embedded – Industry4.0 — Smart-home

Supplementary Material: Demonstration Video — GitHub repository — GitHub example

*xbabec00@fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

The motivation for this project was that there is no such client today (at least no client accessible for public use) and community . There arises a question: *Why should I use Rust client when I can use, for example, the eclipse-paho C MQTT client?*. This client benefits from *Drogue Device*[1] firmware. Firmware bring support for several boards, sensors, and wifi chips. This, and all the other features of Drogue, help to build safe and efficient applications in a very short time.

To build such a client, the design and implementation have to overcome several problems and obstacles:

- **Memory allocation** – This is the most crucial problem that the implementation faced. Without a standard crate, the implementation can't use dynamic memory allocation, which means implementation has to either force the user to provide allocated buffers and their length or use buffers with constant size. The absence of dynamic allocation also brings many design obstacles where implementation has to work with the

[1] Available at: https://github.com/drogue-iot/drogue-device

Figure C.1: Excel paper

Rust ownership model described in Section 3.

- **Async executors differences** – There are many async executors which could be used and more executors will probably come in the future as Rust is heavily developed. Client has to use general Rust futures [3] and *async/await* [2] so the end-user can choose final async executor.
- **Support for any network drivers** – For embedded devices, we can find a significant amount of network boards and most of them have different network drivers.
- **Complexity of API** – This point is probably most significant. With MQTT version 5 there come a large number of possibilities as to how control packets can be configured. But how much of this configuration does the user really need? The right balance will make the client usable in the real world.
- **Extensibility** – An objective which is probably needed in all projects these days. With Rust, everything is a little more complicated. With no inheritance and big differences between MQTT versions, there is not much space how to prepare something for future development.

A big advantage of this project is the significant number of potential areas where it can be used. Rust is currently in very active development and the community around it is growing massively. That means Rust is slowly starting to match older languages like *C and C++*. As is common knowledge, development and maintenance project written in those languages is hard so many companies providing *IoT* solutions invest a great deal in Rust development. These solutions are exactly the right fit for this MQTT client.

2. Existing solutions

If we talk about existing solutions there are no solutions that fit the purpose of this client. But there are solutions which are targeting mainly standard devices not embedded ones or supporting only MQTT version 3.

2.1 Microsoft Hub MQTT

Microsoft is providing several clients for MQTT but none of the clients[2] is written in Rust. The only client which could be usable on embedded devices is a client written in *C*. Clients follow the specification of Hub. Most significant ones for MQTTv5:

- Retain not supported
- Maximum QoS is 1
- No subscription identifiers and Client IDs
- Maximum packet size is set to 256 KiB
- Supports only *AUTH* packet authentication

MQTT version 5 features of the Hub are not officially released, so these might not be final specifications.

2.2 Eclipse Paho

Eclipse Paho might be the most used MQTT client of all and, like most clients, Paho uses wrappers of the *C* implementation to deliver the library in different languages. It is no surprise that a wrapper is already created for Rust[3].

This implementation is targeting only memory-managed operating systems, so there is no support for embedded. Although this client will not work on embedded, a big advantage could be support for all configurations which MQTT provides, in all of the current versions. However, in most scenarios, IoT devices use only a limited number of configurations.

2.3 Comparison

All of these client implementations, including this project, have different positives and negatives but none of the clients above is Rust native, or aims mainly for embedded devices. That is a big advantage of this project together with competitive control packets configuration.

Reviewing all of the information above we can say that there is space for this project and there should not be a problem in finding right fit in any of Rust based IoT ecosystems (or just any Rust embedded firmware). However usage of this project on standard devices may not beat any of mentioned clients because embedded support brings several complications which users do not spend time with.

3. Implementation

Client implementation has been logically separated into several packages. These packages contain structures and traits that are having a common focus on functionality. All of these packages are joined together in the *client* package, see section 3.5.

3.1 Utils

Utils contain structures and types which are mostly focused on work with memory. The most significant

[2] Microsoft MQTT client: https://docs.microsoft.com/en-us/azure/iot-hub/iot-hub-mqtt-support

[3] Eclipse Paho: https://github.com/eclipse/paho.mqtt.rust

parts of this package are *BuffReader* and *BuffWriter* which are used for reading and writing into buffer supplied as a parameter. An interesting aspect of this code is the return type, which is used **Result<(), BufferError>**. That type is specific to Rust, methods return either success with void or error with a variant of specific enumeration.

```
pub fn write_binary_ref(&mut self,
bin: &BinaryData<'a>)
-> Result<(), BufferError> {
    self.write_u16(bin.len)?;

    return self.insert_ref(bin);
}
```

That is interesting, but what is even more teasing is operator **?**. This operator is bound to the Result type. If the result is success operator unwraps the value (void in this case) otherwise takes an error and returns it immediately from the method/function.

3.2 Encoding

Encoding package contains Decoder and Encoder for variable byte integer, which is described in MQTT version 5 OASIS standard [1]. Error delegation in this package works same way as previous package only this time there is a use of special type **VariableByteInteger**that is basically type alias as we know from other languages like *C or C++*.

3.3 Packet

As Figure 1 shows, MQTT provides various control packets. These packets have to be mapped in a protocol to ensure communication functionality. This mapping is stored right in this package. There is a public trait **Packet** which contains a declaration of all methods which have to be implemented for specific packet types and contains a default implementation for common features which are the same for all the packet types.

The rest of the structures for control types simply map packet binary form into Rust structures. At this moment, there is a massive obstacle which has to be overcome. MQTT version 5 enables users to include properties of variable lengths and amounts in the packet.

With embedded in combination with variables, there is a problem. As it was said, there is no dynamic allocation, so there is no way how this could be variable. We have to know the exact size during the compile-time. Rust provides a solution named *const generics*[4] that allows programmer parameterize structure or method with constant. In the manner of this client, it allows parameterize structures with the expected length of buffers that store fields of variable length.

Let's get through this by the example of publish packet.

```
use heapless::Vec;

pub struct PublishPacket
<'a, const MAX_PROPERTIES: usize> {
    pub properties: Vec<Property<'a>,
                        MAX_PROPERTIES>,
    pub message: Option<&'a [u8]>,
}

PublishPacket::<'b, 5>::new();
```

Listing above shows definition of **PublishPacket** structure which contains explicit lifetime annotation **'b** and const generic argument **MAX_PROPERTIES**. This argument sets the size for heapless vec[5] during the creation of packet structure so variable length of properties is maintained and the user can decide how many properties will need with no limitations from the client-side.

Aside from mapping packets also contains implementations of trait methods. The most crucial of these are *decode and encode*. These two methods are the core of the whole client library. Decode methods decode incoming messages from raw format into the usable structures with which can client manipulate. Encode method do exact opposite. Without them client could not work with packets effectively.

3.4 Network trait and adapters

Achieve compatibility with all the network drivers that exist is not possible. Client provides implementation of two network adapters for both embedded and non-embedded network drivers. These adapters are for tokio network and Drogue-device network driver. In order to achieve maximum network driver compatibility, library also provides public network traits:

- **NetworkConnectionFactory** which should be used to establish a connection
- **NetworkConnection** containing all methods necessary for working with TCP stack

Users with specific needs can adapt these traits onto their network driver and pass adapter to the library.

Tokio network adapter

First network trait implementation is *Tokio network*. Implementation is stored in a package of the same name. This implementation adapts *Tokio* network that

[4]Rust generics: https://rust-lang.github.io/rfcs/2000-const-generics.html

[5]Heapless crate: https://docs.rs/heapless/0.2.1/heapless/struct.Vec.html

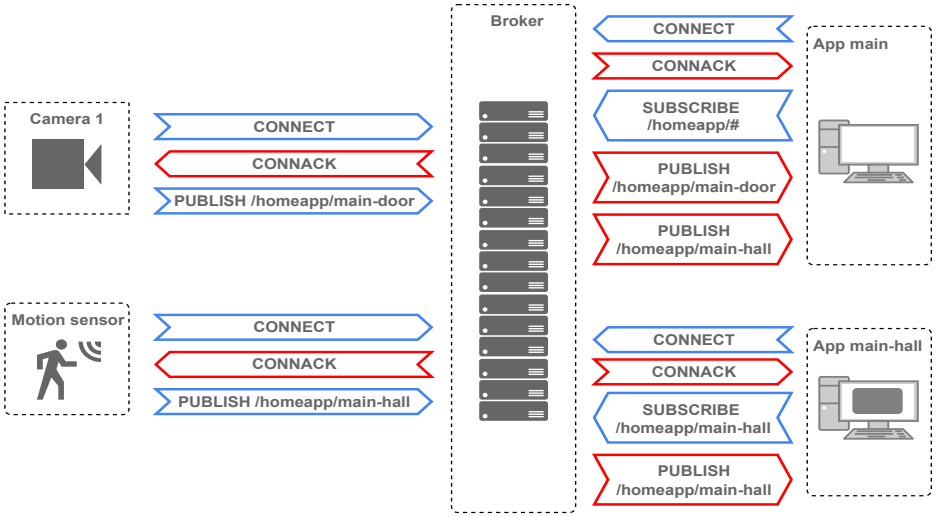


Figure 1. MQTT protocol communication patter in smart home.

is contained inside *Tokio* async library[6] into providing network traits. Network implementation in Tokio aims to support network driver for standard devices (non-embedded).

Adapting such a network is not really great example for this project because network running on standard devices rarely need to close connections because most of the systems can close these themselves. Having this implementation means the client offers full support for non-embedded devices using *Tokio* runtime. This network adapter is not the primary goal of this thesis but having such implementation is necessary to make whole development easier because debugging and testing are things which are in most time very problematic and time-consuming on embedded devices.

Drogue network adapter

Second implementation of network traits is *Drogue Network*. This adapter is located directly in Drogue GitHub repository[7].

Having support for Drogue means that the library now supports all devices and Wi-Fi chips that are supported in Drogue framework. This is much more beneficial than having support for just one type of device, which is scope of this work.

Behavior of the Drogue Network differs totally from Tokio network because we have to respect structure of Drogue-device firmware. Drogue-device async behavior is based on actor model. That is important because manipulating with TCP stack is running in different actor than MQTT client.

```
pub struct DrogueNetwork<A>
where
    A: TcpActor + 'static,
{
    socket: Socket<A>,
}
```

As, code above displays instead of keeping some address to the socket network structure is keeping address of socket as the TcpActor. This way client can communicate with TCP stack (open, send, receive, close).

3.5 Client

There is an implementation of MQTT version 5 compatible client structure **MQTTClient** and configuration structure **ClientConfig** in the client package. Client structure holds config as an attribute and passes corresponding parts of config to each of the control packets. Client contains implementations of *Actions* from MQTT standard. The most significant obstacle here is hassle with lifetimes.

[6] Available from: https://tokio.rs/

[7] Drogue Network available from: https://github.com/obabec/drogue-device/blob/mqtt-client/device/src/clients/mqtt.rs

```
pub async fn connect_to_broker<'b>
(&'b mut self)
-> Result<(), ReasonCode> {

    let mut connect =
    ConnectPacket::<'b, 2, 0>::new();

    if self.config.username_flag {
        connect.add_username(
                &self.config.username);
    }
}

{ client.connect_to_broker().await };
{ client.send_message(topic, MSG).await };
```

We can see the example right in the code above. Method **send_message** also uses one of the client's attributes - *config*. Firstly client is passed as a mutable reference with lifetime **'b** which means the reference will live only in the method's scope. Later is attribute **config.username** as a reference handed to created connect packet (packet lifetime is also set to **'b**).

Once the method is done, all variables and references with lifetime **'b** are destroyed and they can be freely moved to another method. If the lifetime was not specified, the client mutable reference could not be passed to the next method because the reference in the packet would outlive the scope.

4. Testing

Testing of client library is done automatically during pull requests in *GitHub Actions*[8]. The project repository contains 2 workflows:

- Unit tests – these tests aim at fundamental functionalities of structures not the library as a whole. This does not require any other systems to be deployed no real messaging is happening. These tests are most beneficial for working with memory where it is possible to simulate *Index Out Of Bound* error and others.
- Integration tests – testing library as a whole system. This testing is done via *Tokio* test framework and *Tokio net* network implementation. The tests are executed in parallel and all are firing to the same MQTT broker. New approach is currently in development which will allow integration tests to be run mupltiple times agains different broker implementations.

Unit tests are testing a most crucial part of implementation (mapping the protocol). There is a unit test for each packet this tests both *encode and decode* function. After decoding and encoding the packet structure each time the result is compared with the binary or struct created manually, which ensures that the client will create a malformed packet.

5. Evaluation

As a demonstration, there is an example application which is connecting the Drogue device and MQTT client. This demonstrates the usage of the client in the real world even if it seems simple it is basically everything that the end-user will need on the embedded device. The whole code is aligned around the Actor model which allows running and managing asynchronous applications on one thread. Code is separated into three actors.

- **Main** contains publisher functionality. Firstly there is a configuration of the board and the following peripherals (LED Matrix and esp8266 wifi chip). After that TCP connection is established and actors for matrix and receiver are spawned, passing connection to the receiver. The main application loop follows. This loop contains asynchronous wait for the trigger of button A. Once the button is pressed MQTT client will send *Hello World!* message to the specified topic.
- **Receiver** contains all configuration and logic of MQTT receiver. Once the client is configured it connects to the broker and subscribes to the specified topic. Then the main receiver loop starts. The client is waiting for a new message. When the message arrives it sends another message to the Matrix actor to display the received message.
- **LED Matrix** is an actor which is provided as part of the Drogue device for *Micro::bit V2* which is powering this example. The actor is waiting for a new message in the inbox. Once the message arrives at the inbox it displays the message with some refresh rate.

This example is representing the expected usage of the client. Several client instances running on the same device with the possibility of different configurations for all these instances. This example is powered by Micro::bit V2 and *Adafruit HUZZAH ESP8266*[9].

The example was recorded by Drogue contributor Ulf Lilleengen because with recent changes in nightly Rust and the wifi driver the *HUZZAH* which is not

[8] Github Actions: https://docs.github.com/en/actions

[9] https://www.adafruit.com/product/2471

part of common the collection is now only esp8266 currently working without an issue.

6. Conclusion

This work aimed to create an industrial ready asynchronous MQTT client in Rust working on Drogue device firmware. The result of this work is extensible and fully working MQTT client which is, with limitations, fully supporting MQTT version 5, together with automated test suite provides both unit and integration tests (running on Tokio async executor and network driver). The client is ready to support all network and async implementations with minimal effort from the end-user.

At the same time application demonstration was created which confirms that clients API is easy to use and works on small embedded devices supported by Drogue device.

In the next phase of development, I would like to include MQTT version 3 support which could be still useful in the embedded world and release first official version of the library into crate database *crates.io*.

7. Acknowledgment

I would like to thank my supervisors Ing. Jan Pluskal and Ing. Jakub Stejskal for valuable feedback and consultations. I would also like to thank Drogue contributor Siv. Ing. Ulf Lilleengen for feedback and support during the development of the Drogue network adapter and client demo.

References

[1] BANKS, A., BRIGGS, E., BORGENDALE, K. and GUPTA, R. *MQTT Version 5.0* [online]. March 2019. [visited 2021-01-09]. Available at: https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html.

[2] *Async book* [online]. 2021. [visited 2021-01-09]. Available at: https://rust-lang.github.io/async-book.

[3] *Rust RFC 2592* [online]. 2018. Available at: https://rust-lang.github.io/rfcs/2592-futures.html.