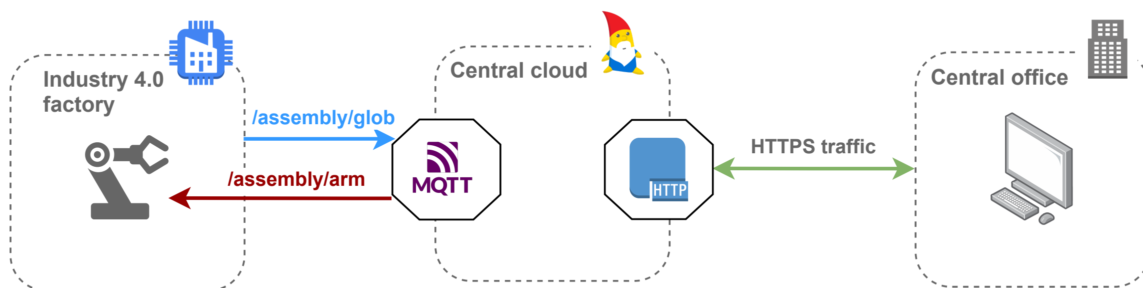
BRNO UNIVERSITY OF TECHNOLOGY FACULTY OF INFORMATION TECHNOLOGY

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

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

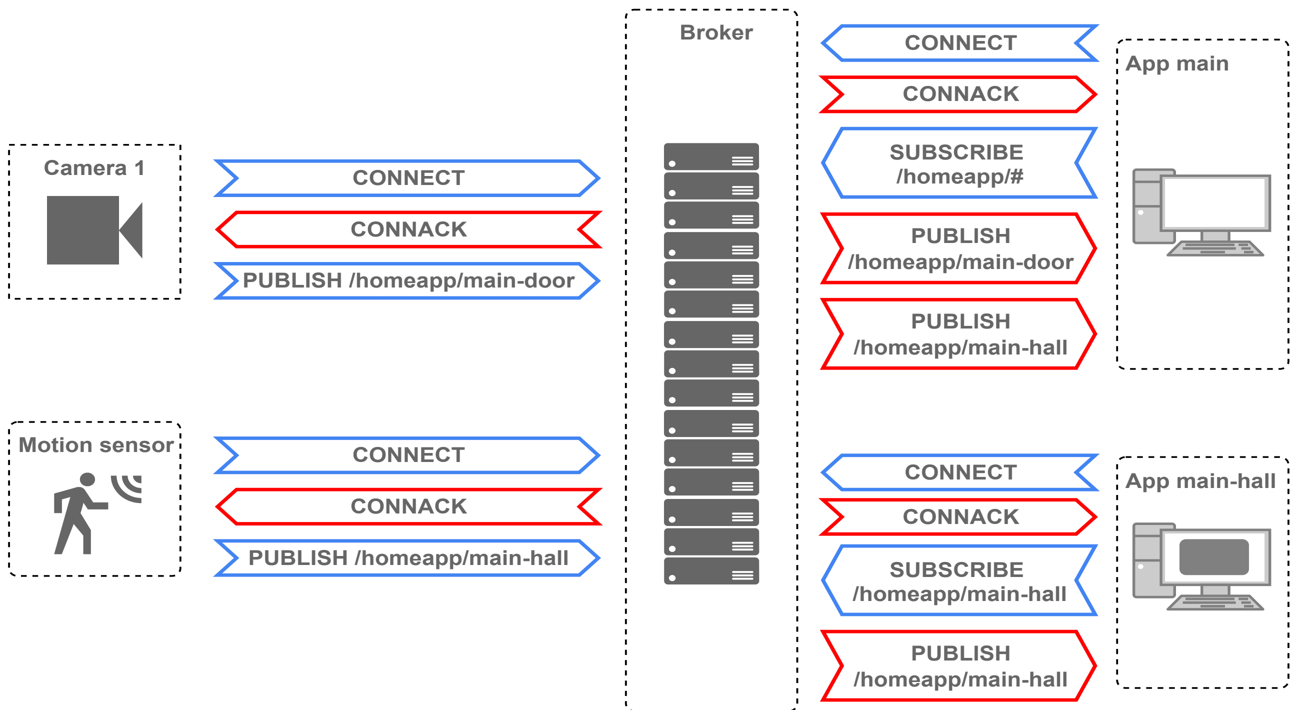


Figure 1. MQTT protocol communication patter in smart home.

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.