Async.MQTT5

Notes to Boost Reviewers

About MQTT

MQTT is a lightweight, publish-subscribe, machine to machine network protocol for message queue/message queuing service. It is designed for connections with remote locations that have devices with resource constraints or limited network bandwidth, such as in the Internet of Things (IoT). It must run over a transport protocol that provides ordered, lossless, bi-directional connections — typically, TCP/IP.

The MQTT protocol defines two types of network entities: a message broker and a number of clients. An MQTT broker is a server that receives all messages from the clients and then routes the messages to the appropriate destination clients. An MQTT client is any device (from a micro controller up to a fully-fledged server) that runs an MQTT library and connects to an MQTT broker over a network.

Information is organized in a hierarchy of topics. When a publisher has a new item of data to distribute, it sends a control message with the data to the connected broker. The broker then distributes the information to any clients that have subscribed to that topic.

About Async.MQTT5

Async.MQTT5 is a professional, industrial-grade, header-only C++17 client built on Boost.Asio. This Client is designed for publishing or receiving messages from an MQTT 5.0 compatible Broker. Async.MQTT5 represents a comprehensive implementation of the MQTT 5.0 protocol standard, offering full support for publishing or receiving messages with QoS 0, 1, and 2.

The aim of Async.MQTT5 is to provide a very simple asynchronous C++ interface for application developers. The internal client's implementation manages network and MQTT protocol details. Notably, the client does not expose connect functions (nor asynchronous connect functions); instead, network connectivity, MQTT handshake, and message retransmission are automatically handled within the client.

The following example illustrates a simple scenario of configuring a client and publishing a "Hello World!" Application Message with QoS 0:

```
#include <iostream>

#include <boost/asio/io_context.hpp>
#include <boost/asio/detached.hpp>
#include <boost/asio/ip/tcp.hpp>

#include <async_mqtt5.hpp>

int main() {
      boost::asio::io_context ioc;

      async_mqtt5::mqtt_client<boost::asio::ip::tcp::socket> c(ioc);

      c.credentials("<your-client-id>", "<client-username>", "<client-pwd>")
            .brokers("<your-mqtt-broker>", 1883)
            .async_run(boost::asio::detached);

      c.async_publish<async_mqtt5::qos_e::at_most_once>(
            "<topic>", "Hello world!",
            async_mqtt5::retain_e::no, async_mqtt5::publish_props {},
            [&c](async_mqtt5::error_code ec) {
                  std::cout << ec.message() << std::endl;
                  c.async_disconnect(boost::asio::detached);
            }
      );

      ioc.run();
}
```

Note the absence of `connect()` function and network error handling code. These are deliberately handled automatically by the internal client's code.

The Async.MQTT5 library design

The Async.MQTT5 public interface is designed to be as simple as possible, allowing application developers to avoid dealing with MQTT protocol internals, low-level network events, message retransmission, and other complexities. The goal is to enable developers to publish or receive messages with just a single line of code.

IoT devices using MQTT to communicate with a server (broker) are often deployed in remote or inaccessible locations, and some may even be mobile (such as vehicle GPS trackers). In these cases, network connections are frequently unstable, and due to the nature of TCP/IP over wireless networks, messages are often lost. Client code on IoT devices must be exceptionally robust, as fixing bugs on devices in inaccessible locations can be virtually impossible. Therefore, the client code must be highly reliable, handling network losses, unreliable data transport, network latencies, and other unpredictable events seamlessly.

The MQTT protocol guarantees message delivery at a quality of service (QoS) level chosen by the application developer. For QoS levels greater than 0, the sender must ensure that the receiver will eventually receive the message, even if it requires indefinite retries. To maintain strict compliance with the MQTT specification, the Async.MQTT5 library does not allow developers to interfere with or customize the MQTT-specific transmission and retransmission logic. Instead, Async.MQTT5 ensures that all packets are delivered reliably and in the correct order.

Since the MQTT protocol is fully bidirectional, both the client and broker can send messages to each other at any time. As a result, the network connection, which is always initiated by the client, must remain continuously open during an MQTT session. The client will automatically re-establish any broken TCP/IP connection, regardless of the cause, such as temporary network signal loss, an unavailable broker, expired TLS certificates, network reconfiguration, or DNS changes.

The Async.MQTT5 library supports any compatible user-supplied, ordered, lossless, bidirectional network connection (stream). Users can customize this stream through a template parameter. The library has been tested with a variety of streams, including plain TCP/IP, encrypted TLS, WebSocket, and TLS-encrypted WebSocket streams.

The library provides support for both basic MQTT authentication with a username and password and extended challenge/response-style authentication. This latter form is a customization point, theoretically allowing any authentication mechanism supported by the SASL protocol.

The Async.MQTT5 interface fully adheres to the Boost.Asio asynchronous model. The client's asynchronous functions are compatible with all completion tokens supported by Boost.Asio, enabling versatile usage with callbacks, coroutines, futures, and more. Each asynchronous operation supports targeted, per-operation cancellation as per Asio's specification. The library also supports custom memory allocators, offering additional flexibility and control over memory resources. Async.MQTT5 consistently uses allocators associated with handlers from asynchronous functions to manage objects required by the library implementation.

The library requires Boost 1.82+ and compiles with Clang 12.0+ on Linux (i386, x64, arm64, and armv7 processors), macOS (Intel and ARM processors), GCC 9+ on Linux (x64 processor), and MSVC 14.37+ on Windows (i386 and x64 processors).

The Async.MQTT5 implementation specifics

The composed async operations

Composed asynchronous operations are operations that call two or more other asynchronous operations during their execution. A composed asynchronous operation invokes its final (outer) completion handler only when its entire internal asynchronous process completes, which may involve waiting for several intermediate handlers to finish. These composed operations can be arbitrarily complex and may include the invocation of any number of intermediate asynchronous methods.

In a composed operation, intermediate asynchronous calls are typically chained together in an asynchronous manner. This chaining means that calls to intermediate operations are serialized: the composed operation calls an intermediate asynchronous method, waits for its completion handler, and based on the outcome, determines what to do next. This process may involve calling another asynchronous method, repeating until the entire operational logic is complete.

There is no single “standard ASIO” method for writing composed operations, although most use the ASIO helper function `async_compose` to initiate and govern the execution. In this case, the composed operation itself serves as an ASIO-compliant completion token, with an overloaded `operator()` providing a common completion handler for all internal asynchronous operations. The `async_compose` function automatically manages execution using either the default or a provided executor, propagates custom allocators, and correctly handles cancellation slots, if any, across all intermediate operations.

When using `async_compose`, developers typically implement the overloaded `operator()` as either a stackless coroutine or by maintaining an "execution step" internally to distinguish which of the internal asynchronous operations has invoked this common `operator()`. This method is effective when the composed operation involves only a small number of intermediate asynchronous calls and, importantly, when the completion handler signatures of these internal operations are similar. For example, a composed operation that writes to a stream with `async_write` and then reads a response from it with `async_read` works well with this approach, as both functions share the same completion handler signature. Thus, a single `operator()` overload can serve as a handler for both `async_write` and `async_read`.

However, when a composed operation needs to execute a multitude of asynchronous methods with differing signatures, a single `operator()` overload may not be sufficient, as it cannot adapt (even with defaulted arguments) to match all intermediate operation handler signatures. Take, for instance, the `async_connect` operation within Async.MQTT5: this operation establishes a TCP connection to the broker, optionally performs a TLS handshake, exchanges MQTT CONNECT/CONNACK packets, and handles authentication—all before invoking the final completion handler. The intermediate handler signatures differ so much that a single `operator()` overload cannot cover them all.

Async.MQTT5 addresses this by using a construct that launches composed operations via the lower-level `async_initiate`. Once launched, the composed operation serially runs intermediate asynchronous operations, supplying itself as the completion handler for each. However, for each intermediate operation, the composed operation defines a unique `operator()` overload with a type-distinct first argument, while the remaining argument types match the expected signature for each intermediate operation’s completion handler. The correct `operator()` overload is then selected by binding this distinct first argument to an instance of its respective type.

Here's a brief example of this approach:

```
class composed {
      struct on_read {};
      ...
      void perform() {
            asio::async_read(
                  stream, buff, asio::transfer_all(),
                  asio::prepend(std::move(*this), on_read {})
            );
      }
      void operator()(on_read, error_code ec, size_t bytes_read) { ... }
}
```

By using `async_initiate` instead of `async_compose` to launch composed operations, Async.MQTT5 must manually manage executors, allocators, and cancellation slots. While this approach leads to more verbose code, it provides fine-grained control over how these objects are propagated to intermediate operations. This control is particularly important for enabling the total cancellation type for specific operations.

Binary serialization and deserialization

The MQTT 5.0 protocol message format is quite complex, making the task of parsing and serializing C++ objects into the appropriate MQTT format prone to errors. Serialization also needs to be highly robust and resilient to errors or inconsistencies in transmitted data, as IoT devices are frequently targeted by malicious network attacks.

To minimize—if not completely eliminate—serialization issues, Async.MQTT5 utilizes the Boost.Spirit library for deserialization, complemented by a hand-crafted framework for serialization that follows a similar syntax. For example, here is an actual code excerpt that deserializes a CONNACK message:

```
      auto connack_ = basic::scope_limit_(remain_length)[
            x3::byte_ >> x3::byte_ >> prop::props_<connack_props>
      ];
      return type_parse(it, it + remain_length, connack_);
```

The library uses Boost.Spirit's built-in binary parsers and composes them to convert binary data into the corresponding C++ structures. It also defines a few Boost.Spirit-compatible helpers for data types not covered by the built-in parsers.

Serialization follows a similar process in the opposite direction, using custom-built serializers.

```
      auto packet_type_ =
            basic::flag<4>(0b0010) |
            basic::flag<4>(0);

      auto var_header_ =
            basic::flag<1>(session_present) &
            basic::byte_(reason_code) &
            prop::props_(props);

      auto fixed_header_ =
            packet_type_ &
            basic::varlen_(var_header_.byte_size());

      auto connack_message_ = fixed_header_ & var_header_;

      return encode(connack_message_);
```