KU LEUVEN

FACULTY OF ENGINEERING TECHNOLOGY

CAMPUS GROUP T LEUVEN

Building an easy to use embedded MQTT library

Maarten ARITS

Supervisor: Prof. L. Vandeurzen

Co-supervisor: *F. Van Slycken*

Master Thesis submitted to obtain the degree of
Master of Science in Engineering Technology:
Internet Computing

Academic Year 2015-2016

FACULTY OF ENGINEERING TECHNOLOGY
CAMPUS GROUP T LEUVEN
Andreas Vesaliusstraat 13
3000 LEUVEN, Belgium
tel. + 32 16 30 10 30
fet.groupt @kuleuven.be
www.fet.kuleuven.be

Master of Industrial Sciences 2015-2016
Faculty of Engineering Technology, Campus Group T Leuven
This paper is written by (a) student(s) in the framework of a Master's Thesis

BUILDING AN EASY TO USE EMBEDDED MQTT LIBRARY

Maarten Arits

Master of Science in Engineering Technology: Internet Computing,
Faculty of Engineering Technology, Campus Group T Leuven
Vesaliusstraat 13, 3000 Leuven, Belgium

Supervisor: Prof. L. Vandeurzen
Faculty of Engineering Technology, Campus Group T, Leuven
Vesaliusstraat 13, 3000 Leuven, Belgium
Luc.Vandeurzen@kuleuven.be

Co-supervisor: F. Van Slycken
Intelligent Systems, Altran
Gaston Geenslaan 9, 3001 Leuven
Frederik.Vanslycken@altran.com

ABSTRACT

Daily new devices connect to the internet, making the internet of things a reality. Even though there are big differences in the amount and type of data they communicate, there are some key aspects that return. MQTT is a publish-subscribe messaging protocol relying on TCP that takes care of recurring communication challenges. It is a widely adopted application layer protocol providing abstraction from transport related issues, enabling programmers a shorter time to market. In this thesis a custom implementation of MQTT, pico-mqtt, is discussed. The custom developed library features a very easy to use blocking interface based on BSD sockets. It strongly focusses on reliability and portability resulting in a powerful stack able to fit even the most constraint embedded devices. The comparison between the custom library and existing solutions clearly outlines the many use cases of this library. It will enable every programmer or enthusiast to set up a huge, reliable network that is easy to understand, expand and maintain in a short time.

Keywords

Internet of things, MQTT 3.1.1, Embedded communication, Portable embedded library, M2M

1 INTRODUCTION

Centuries ago the industrial revolution brought machines that started to take over mundane tasks. Since, machines have been getting cheaper, smaller and more complex. Computers where integrated into these machines to augment their capabilities. These integrated computers are called embedded microprocessors and they are capably of complex tasks.

Most electronic devices today contain a microprocessor that controls the device based on sensor readings and user input. Most of devices still work autonomous, limiting its capabilities to the initial program. Connecting these devices to the internet would not only allow for software updates and product information to be collected, it would also allow these simple devices to behave much more intelligent.

If you have for example some solar powered devices in your garden measuring the soil humidity and sending this data to a sprinkler, the sprinkler could activate when the soil humidity is too low. For this standard behavior no internet is needed. By connecting these devices to the internet, the sprinkler could check weather forecasts to see if rain is coming. When it freezes it could automatically disable itself, preventing damage.

Almost no changes need to be made to the sprinkler or humidity sensors for them to have internet access, they only need a radio that is able to connect to the internet. This small change allows the device to save water, prevent frost damage and so much more.

In order to make these kind of devices a reality three requirements need to be met: microprocessors need to be very cheap, they need to be able to communicate to the internet and the development cost has to be very low. At the moment of writing microprocessors are available for less than a euro that are capable of wireless communication. In this thesis a library is discussed. This library enables embedded programmers to develop a connected device in a very short time for a fraction of the development cost. It fulfils the last requirement, making the internet of things economically possible.

1.1 Creating a machine to machine network

To connect embedded devices to a network it is necessary to use a library that implements a protocol. Without using libraries, the development time would be extremely long since the programmer has to solve the same problems as a protocol or library solves. In a sense, the programmer has to create, implement and test a new protocol or library that shares much of the same qualities as existing protocols if no existing protocol is used.

There are several ways to create a network of embedded devices. All of them rely on a transport protocol such as TCP or UDP. There are several protocols available that allow machine to machine communication. Not all of these protocols are suited for embedded applications due to their complexity. As has happened before for IP, TCP and UDP, the industry is moving toward a standard protocol. The benefits of using a standard protocol are that there are many good information sources available and well maintained software libraries are available for almost every processor. Using a standard protocol greatly reduces development cost and makes a solution future proof.

A comparison was made about the different data communication protocols available today and their popularity (IoT Standards and Protocols, 2015) . It is not possible to check how many devices use a certain protocol because most code is not publicly available. Instead Google Trends was used to compare the relative popularity between the protocols (Google, 2016). Google Trends shows the relative number of searches of a specific search term in Google. The maximum number of searches in a data set is used as reference (100). For search terms with alternative hits the term “protocol” was added, the less common protocols where aggregated together to avoid cluttering the graph. **Figure 1.** Google Trends of IoT protocols, clearly shows that MQTT is the most popular protocol at the moment and is rapidly gaining popularity. This is the reason why Altran is interested in supporting MQTT in its TCP stack. Since the internet is a popular way to search for information and google is not biased toward a protocol, the number of searches is valid indication of popularity.

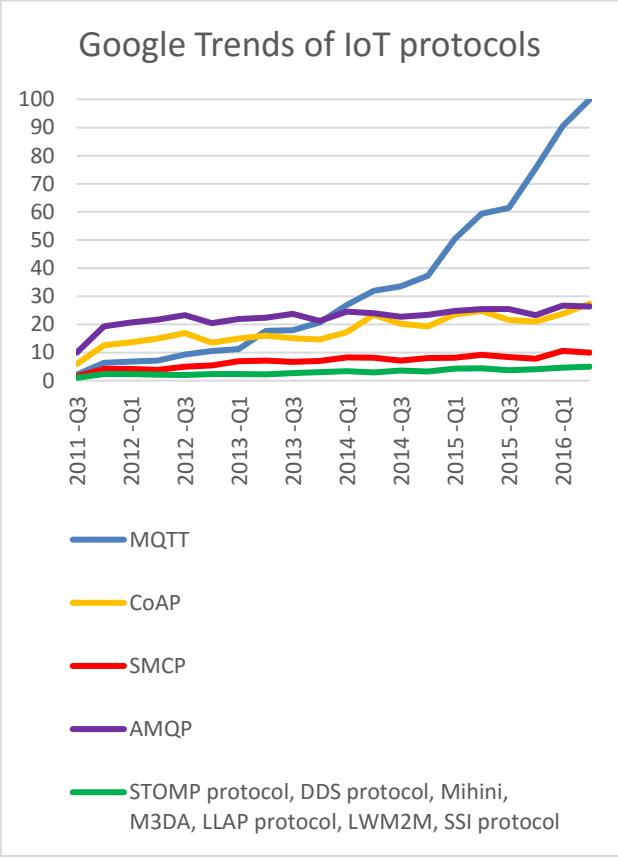

Figure 1. Google Trends of IoT protocols

1.2 Goal of the paper

At the moment of writing there are several MQTT implementations already in use, some of them are suitable for use in an embedded system. In this paper a custom created MQTT library is discussed and compared to existing solutions. The main goal of this paper is to verify if the custom library fulfils the requirements posed by Altran and is complementary to the existing libraries. In order to better understand the problems arising when creating a machine to machine communication network the challenges are described in the next section together with how MQTT solves these challenges.

2 THE MQTT PROTOCOL

When creating a machine to machine network a couple of challenges need to be overcome such as devices that are unreachable, addressing and error handling. In this section is described how MQTT handles these challenges and the internal working of MQTT.

MQTT stands for Message Queuing Telemetry Transport and is a light weight messaging transport protocol (Gupta, 2014). It uses a publish subscribe strategy where devices are called the client and they connect to a single broker; the broker is the MQTT messaging server. The library presented in this thesis is custom written for Altran to be used on top of their existing pico-TCP stack. It is intended to speed up development of custom solution in a professional environment.

2.1 Basic machine to machine networking

In most networks there are different types of embedded devices communicating with each other. New devices will be added over time, some devices will be mobile and only sporadically participate with the network. To enable these devices to communicate, every device needs to be addressable. MQTT solves this challenge by specifying that every active device must connect to the MQTT broker, providing authentication if required.

Routing the messages is the main focus of a network. Not every device will be interested in every message, moreover, the device sending the message may not even know which devices are interested in the message. MQTT provides an elegant way to handle routing. Devices publish a message to a topic. Other devices can subscribe on a topic to receive all the message published on that topic. MQTT can be seen as an online forum for machines. If a device is not connected it will not receive the messages published.

MQTT provides a mechanism to allow devices to disconnect from the network. On every topic a retain message can be published by setting the retain flag, doing so will replace the old retain message. Every time a device subscribes to the topic it will receive the latest retained message.

An automated garden watering system would be a good example of how MQTT could be used in a practical application. The example is presented graphically in **Figure 2.** Garden watering message flow, in the description the letters refer to the deferent steps in the illustration. The example is a system where the garden is watered automatically. Several sensors could be connected to the network by sending a connect message(A.) and publish the ground humidity to the topic "garden/humidity" (B.). A sprinkler would connect to the network (C.) and subscribe to this topic (D.). By subscribing the sensor will be receiving updates on changing humidity from the humidity sensors via the broker (E.). To allow other devices to know the sprinkler started watering the garden, the sprinkler could publish this to the topic "garden/sprinkler" (F.). When done the sprinkler could unsubscribe from the topic (F.) and disconnect from the network (E.). Writing this application with the pico-mqtt library would only require a couple lines of code. Example code is provided in on Github in the file "garden_watering_example.c".

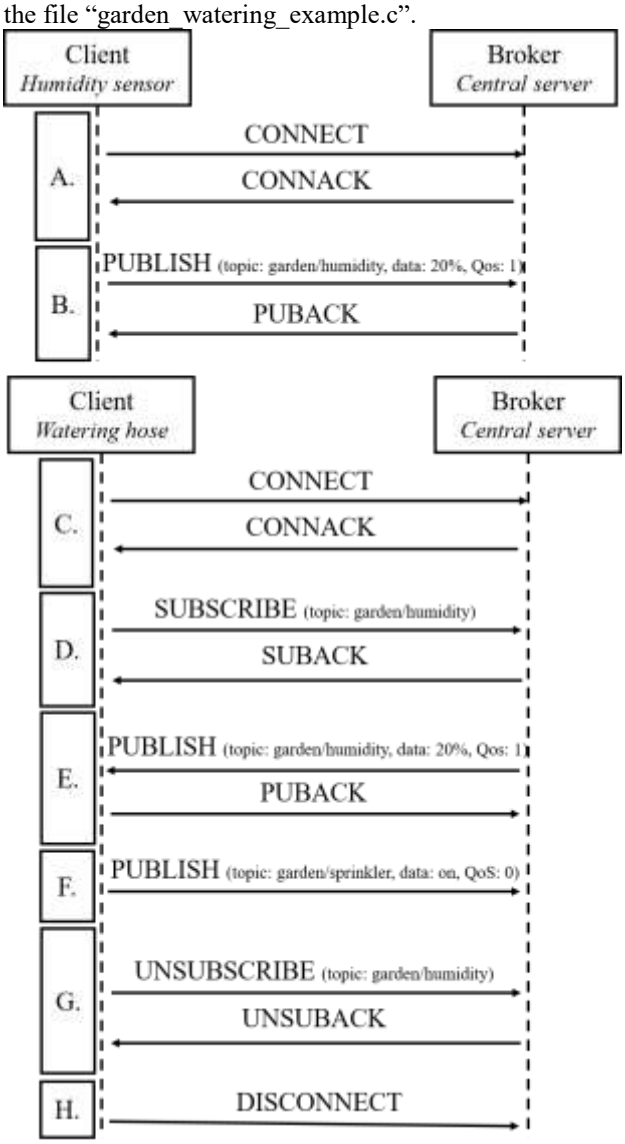

Figure 2. Garden watering message flow

2.2 Handling errors

In a perfect world, no more mechanisms are needed than the ones described above to ensure stable communication. MQTT does however provide mechanisms to cope with packet loss and connection loss. MQTT relies on TCP, preventing byte loss, losing connection during transmission is still possible and can lead to packet loss. For a more reliable way of transmission the QoS (Quality of Service) can be set to 1 or 2.

2.2.1 Quality of Service 0

When messages are published with a QoS of 0 the message will be published only once. If an error occurs during transmission the message will be lost and not delivered. An example of the communication between client and broker for publishing a message with QoS 0 can be found in **Figure 2.** Garden watering message flow.

2.2.2 Quality of Service 1

When an error occurs during the transmission of a message with QoS 1 it will be retransmitted until an acknowledge message is received (PUBACK). An example can be found in **Figure 2.** Garden watering message flow, for communication from the broker to the client this message flow stays the same. Client and broker can be interchanged in the diagram.

2.2.3 Quality of Service 2

QoS 2 will ensure that the message is published exactly once. Publishing the message happens in 2 stages. The first stage is exactly the same as for QoS 1, the publish message will be resend until an acknowledge message is received. Once the acknowledge message is received (PUBACK) by the sender the message transmission goes into a second stage. In this stage a publish release message (PUBREL) is send until a complete message is received (PUBCOMP). This release message signals to the receiver that it can process the message that was published. Published messages with a QoS higher than 0 have a unique ID, allowing the detection of duplicates. This method of transmissions guarantees the message is only processed once by the received. In **Figure 3.** MQTT Quality of Service message flow, the message flow is visualized. Again the client and broker can be interchanged.

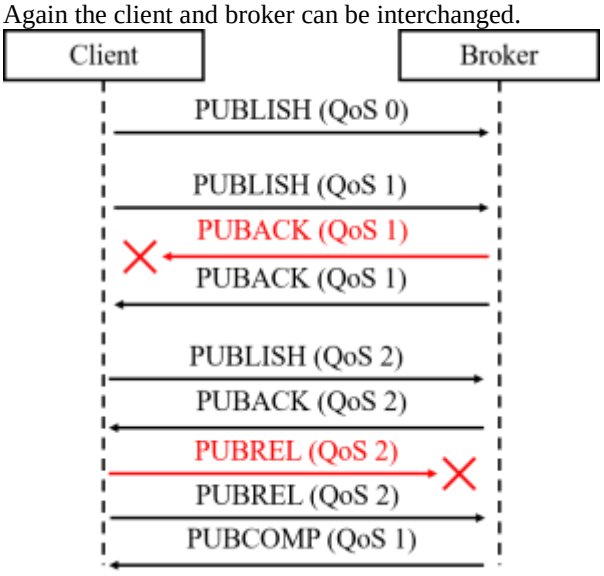

Figure 3. MQTT Quality of Service message flow

2.3 Connection loss

A useful feature MQTT offers is ability to set a will message and topic. This message will be published by the broker on behalf of the client when the client disconnects unexpectedly. The will message and topic are set by the client when connecting to the broker. When the client reconnects it can do so without setting the clean session flag. All the subscriptions from the previous session will be restored, preventing the overhead of re-subscribing after an unexpected disconnect.

2.4 Existing MQTT implementations

There are several MQTT implementation already in use. At the moment the thesis commenced the number of existing projects was lower. Licenses of some projects also prevent free commercial use, a key requirement for Altran. In **Tabel 1.** MQTT library overview, an overview of the different open source libraries is provided. The interface complexity gives an indication how difficult it is to start using the library:

L: large complexity
M: medium complexity
S: small complexity

	Eclipse Paho C	Elcipse paho embedded C	LibMosquitto	Libemqtt	wolfMQTT	Pico-MQTT
Blocking	Yes	Yes	No	Yes	Yes	Yes
Callback functions	No*	Yes	Yes	No	Yes	No
Typedefs	Yes	Yes	Yes	No	No	No
Enums	No	Yes	Yes	No	No	No
Switch statements	Yes	Yes	Yes	No	Yes	No
Double pointers	No	No	No	No	No	No
Interface complexity	L	M	M	M	M	S
Free commercial use	No	No	No	Yes	Yes	No

**tokens are used instead of callbacks.*

Tabel 1. MQTT library overview

2.5 Library comparison

In this section the pico-mqtt library is compared to the other open source embedded C libraries in existence. **Tabel 1.** MQTT library overview, is based upon the latest source code available. The sources are referred to in their respective sections below.

2.5.1 Eclipse Paho

The first version of the paho library provides a big interface that is not as straight forward as the other libraries. The main focus of this implementation is powerful machines with operating system such as Linux and Windows. Instead of blocking function calls or callback function, tokens are used. Functions are provided that use the tokens to block until a specific event occurs. This MQTT library is not competitive with the library developed because it is not well suited for use on embedded processors. (Eclipse, 2016)

2.5.2 Eclipse Paho embedded

The standard Paho project was poorly suited for integration into embedded projects, that was the main reason why an embedded version was developed. This library features an interface depending on callbacks. Callbacks allow the client to efficiently run other tasks concurrent to the MQTT stack. When an event occurs the callback will handle it. This interface allows more flexibility and reliability but is harder to use.

The library also makes use of features that are not clearly defined by C such as enums. The main benefit of using the pico-mqtt library has over the use of the embedded Paho version is the capability to automatically reconnect the client when the connection is lost without requiring user action. The Paho library is not free for commercial use, making it unsuited for use by Altran. (Eclipse, 2016)

2.5.3 LibMosquitto

LibMosquitto is the last eclipse library discussed. It was one of the first ones to be written and also features a broker. This library depends upon callback functions and uses constructs that violate the set constraints. In contrast to the embedded Paho implementation a loop is used to process messages. This loop allows the device to sleep while still processing the MQTT messages. During the loop the callback functions will still be called. This allows to configure the device for MQTT communication and to setup the callbacks before enter an infinite loop. After the loop is entered the device will simply react to incoming messages. The interface itself is much more complicated than the pico-mqtt one due to all the functions provided. The number of arguments of each functions is high as well, 4 arguments is not uncommon in this library. The license does not permit free commercial use. (Eclipse, 2016)

2.5.4 Libemqtt

This library perfectly matches the created pico-mqtt library in its requirements. This library does not support multiple simultaneous subscription, pico-mqtt mirrors this limitation with good reason (refer to 9.3 Multiple subscriptions). The project does feature an interface that is surprisingly similar considering they were developed completely independent. There are still some points of criticism on the interface.

The biggest criticisms are all the exposed private functions. They only use 1 header file and expose the internal workings of the library to the users. This is bad practice. They pass in massages as separate parameters, making for long parameter lists. This creates a big risk of the user forgetting or switching parameters with possibly significant implications. An example of the kind of errors that can result from long parameter lists: switching QoS 1 with a retain flag of 0 will be perfectly valid and create a hard to track bug without any compiler warnings or errors. The library does not include any unit testing and is still in development. The Github page clearly states that the code is in development and should not be used in production.

Unfortunately, the last commit was done in 2014 and the code is maintained by nobody. This library should no longer be used. (Ruiz, 2014)

2.5.5 WolfMQTT

WolfMQTT provides an excellent MQTT library with the added benefit that the company's primary project is WolfSSL, a security library they integrated with their new MQTT implementation. The interface of this library is very clear. It is a bit more complex due to the use of a custom structure for every function instead of using parameters. They also use callbacks for received messages violating the requirements for this project. Together with the Eclipse projects this library is very well documented.

A downside of the close integration with the WolfSSL library is that the SSL library is not free for commercial use. If the project requirements where set aside this library would be the best one for embedded applications. Not only is it implemented very well, it also has good integration with the security library, is very well written and thoroughly documented. (WolfSSL, 2016)

2.5.6 Conclusion of the comparison

There are a lot of projects available, some providing a very good library with beautiful integration. Most interfaces are clean and easy to use. Pico-mqtt offers the simplest interface however. None of the other available projects fit the requirements. If other types of interfaces are required a good fit can be found. The similarities between pico-mqtt and the existing libraries, both in interface and implementation indicate that pico-mqtt is well designed and well structured.

3 REFLECTION ON MQTT

The MQTT protocol is in general easy to understand and use, providing flexibility where needed. It is however far from perfect and is lacking at some points. In this section some important notes are made on the MQTT protocol pointing out areas in which it is lacking, inconsistent or inefficient. By intensively using the MQTT protocol while developing the library these points became clear. This section only contains personal insights and is not based upon work of others. Most of the points discussed require a very thorough understanding of the MQTT protocol and it is advised to refer to the specifications for a broader context (Gupta, 2014).

3.1 Fixed header length field

For every single MQTT message the fixed headers length field is of variable length to reduce overhead. The minimum length is 1 byte and the maximum length is 4 bytes. The most significant bit (0x80) of each byte is used as a flag to indicate if the next byte is a part of the message length or of the payload. The maximum length encoded is 28 bits (268 MB).

While the need for compression is defendable it makes decoding the messages hard due to the resulting relative position of the payload. The length field is however the only place this encoding is used; other places don't encode the length of a message; they are using fixed width length fields. Even harder to understand is why the variable length field is included in fixed length messages. These messages are often 2 or 4 bytes long of which 1 byte will be the length field. The inconsistent way the length is encoded is a big oversight. The only place where the length can be longer than 24 bit (3 byte) is the publish message. Most messages require no length field.

3.2 Omission of length for publish messages

For every instance where a variable length field needs to be transmitted by MQTT, the variable length data is preceded by a length field. The only exception being the payload of the publish message. For this message the length field of the publish message is used to calculate the payload length. Since this is the only instance where a field can be this big, reducing the length field to 16 bit (65536 bytes) as usual would yield a much more consistent packet structure and remove the need for length encoding. Another consequence of setting the maximum length to 256MB is that the receiver needs to allocate 256MB of RAM or buffer the message on a storage device. This will have significant performance consequences. It is also not future proof, since the maximum size is specified it can be reached. An alternative and time proven solution would be to implement a fragmentation algorithm. This solution is already used by TCP, showcasing the possibility of sending endless streams of data.

3.3 Limited length of will message payload

When connecting to the broker a will message can be specified. This message only differs in 2 ways from a regular message. The first difference is that the message will be published by the broker on behalf of the client when the connection is lost unexpected. The second difference is that the payload is proceeded by a 2-byte length field specifying the payloads length.

Other published messages can have a payload size up to 256MB, the will message payload is limited too just over 65kB. This inconsistency could be resolved by making the payload field the last field in the connect message or to change the specification making the maximum payload length always two bytes, if desired combined with fragmentation.

3.4 Subscription QoS returned by broker

When subscribing to a topic the clients sets the maximum QoS that can be used for the subscription. The broker will return the granted QoS for that subscription in an acknowledge message. The broker is free to choose any maximum QoS that is equal or lower than the one requested. If the broker would choose any other QoS than the one requested by the client it would only serve to provide suboptimal service to the client.

If the broker does not implement QoS 1 or 2 for publishing or if another client publishes to the subscribed topic using a QoS lower than the requested one by the client the broker can still publish to the client with a lower QoS than the one requested by the client without violating the specifications. The QoS requested by the client only sets the maximum QoS the broker can use and leaves the broker free to choose the QoS used for each message.

If, however the broker would only grand a maximum QoS of 0 or 1 when the client requests 2 and at a later stage a message would be published to the topic with QoS 2 the broker can no longer send this message to the subscribed client with QoS 2 as requested by this client because the broker limited the QoS on the subscription. If the broker would have granted QoS 2 could still use QoS 1 or 0 to publish the message if so desired without violating the specifications.

The usefulness of the returned QoS by the broker is further reduced by the fact that every message published will contain its QoS. There is no need to specify the maximum QoS ahead of time. In order to keep the MQTT protocol as simple as possible the returned QoS should be removed.

3.5 Subscription QoS with wildcards

At the topic above "Subscription QoS", a clear case is made that the returned QoS for a subscription has no use. When the clients use wildcards this becomes even more problematic. It can lead to hidden changes of the QoS used for other topics. Wildcards allow a client to subscribe to multiple topics at once that match the pattern provided by the client. For each topic subscription the broker will return the maximum QoS granted, for a topic filter with wildcards one return value is send for all matching topics, possibly limiting the maximum QoS of other topics in the process.

Changing the maximum QoS of other topics is a hidden side effect of wildcard usage. This will lead to confusion and hard to track bugs.

3.6 Subscribing with wildcards

The MQTT specification is unclear at some points, leading to confusion and unavoidably to mistakes. The most unclear point is wildcard subscription. At no point in the specification it is stated clearly whether the wildcard subscription also subscribes the clients for future topic that are not yet created. Based on listed requirements the intended working can be deduced. The explanation below should be added to the MQTT specifications.

When a client subscribes for a topic it sends a filter, this filter will be stored by the broker for the client. For every message published, the broker checks if the topic matches the client's filters. The lowest QoS of all the matching filters is used to send the message to the subscribed client. This point should be better described in the MQTT specifications. The sections of the MQTT specification hinting at the correct implementation of the wildcards is at lines 826-830, 1074-1077, 1374 of [mqtt-v3.1.1] (Gupta, 2014)

3.7 No specifications for control

The biggest feature lacking from the MQTT standard is a backend interface. In stark contrast to COAP where protocol control is specified, it remains unspecified for MQTT. The lack of specification prevents cross library configuration of the broker.

A useful feature for backend control would be to gather statistics, access old messages and check upon the current state of the broker. A common way of gathering status information in MQTT is by using system topics which are proceeded by a dollar sign ($). This is common practice but not standardized. There is also no way for a backend to be notified when a new topic is created. In order to keep track of the whole platform an application needs to subscribe to every topic using wildcards, creating massive overhead.

3.8 Quality of Service

Quality of service is an important feature of MQTT, adding a lot of complexity to the platform. The use of QoS on top of TCP is however questionable. TCP guarantees that bytes will be delivered in order. The only way to lose a message is when the TCP connection is broken. TCP does integrate mechanisms to prevent connection loss, even over saturated networks. In networks where packet loss is common MQTT-SN should be used.

The overhead associated with QoS 2 is very low, only 3 messages of 4 bytes. It is important to point out that a TCP datagram is 64 bytes long, for a common Ethernet MTU of 1500 bytes. Most TCP implementations have smart buffers that will aggregate messages together to limit overhead. In situation where MQTT is used, a stable network connection is required, if not MQTT-SN should be used. The stable connection will result in low connection loss, without much need for MQTT intervention to recover lost messages. Overhead of QoS 2 could dramatically be reduced by only initiating the QoS 2 mechanisms when the TCP connection is lost. In most solutions with reliable network connections QoS 0, 1 and 2 will be interchangeable and behave the same.

A possible solution would be to create a FIFO of 2 times the size of the TCP buffer and numbering the messages. Messages pushed out of the FIFO are destroyed, only when the TCP connection is lost the messages inside the FIFO are retransmitted. Using a 16-bit incrementing message ID would insure that duplicates can be detected and discarded. This eliminates the need for QoS and saves memory. The way MQTT currently is implemented the number of messages waiting for confirmation can be infinite requiring lots of memory. In the proposed solution the total length of the messages waiting is the length of the FIFO. A graphical representation of the proposed solution is shown below in **Figure 4.** MQTT QoS alternative.

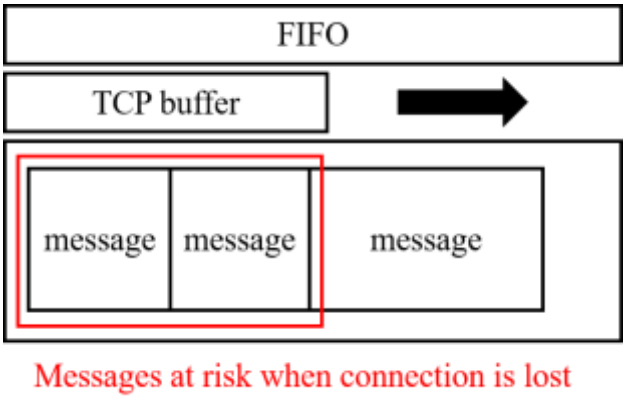

Figure 4. MQTT QoS alternative

3.9 UTF-8 string encoding

MQTT is first and foremost a machine to machine communication protocol, the use of human readable string is questionable and results in unnecessary overhead. In most use cases the client will have a unique ID, it would be reasonable to create a topic and subtopic with this unique device ID to be able to store device specific information or configurations.

By enforcing the use of UTF-8 strings MQTT forces the client to convert the client ID number to ASCII. By relaxing the specification this could be solved. Using binary data as topic ID's MQTT would be more machine friendly. The use of wildcards (#, +) and level separators (/) could still be included. In cases where a human readable format is desired it can be used since UTF-8 or ACHII strings can fit the suggested binary format. For the message itself MQTT already uses binary data, as well as for the password. There are no clear reasons why a distinction was made.

4 REQUIREMENTS

The goal of the MQTT library is to provide an easy to use, portable and reliable way to communicate with an MQTT relaying server; the MQTT broker. Due to the enormous variety of embedded devices and architectures there are significant constraints placed on the implementation of the library to ensure portability. These constrains are described in this section. The terms “ease of use”, “portable” and “reliable” are used from the standpoint of the programmer. An easy to use library for a programmer has logical, well defined functions that behave as expected. Reliable means the library will behave as intended in all possible situations, no matter what happens to the network. Portability means a library can easily be used on other hardware. To avoid copyright infringements, the library does not use any part of existing libraries nor is it based upon existing work. The existing libraries are checked to see if they match the set requirement in 2.4, Existing MQTT implementations.

4.1 Portability

Chip manufacturers provide compilers to allow to program their processors. Multiple compilers and compiler versions are currently in use, most of them only support a subset of C++ (C++ compiler support, 2016). This makes C a far better option for portable libraries.

The C specification do allow compilers to choose how they implement certain features, making these feature vary in size or data layout between compilers and compilations. To avoid problems these features are excluded from use, in particular enums, unions, switch statements and bitfields. To further ensure interoperability between compiler versions flags are added for C99 compliancy. See appendix A for a detailed overview of all requirements and compiler flags. Compiler flags are used to signal the compiler to activate extra checks. Testing the code more rigorously and helping to avoid bugs.

Most embedded devices have limited program memory and RAM, in order to be portable to as many devices as possible memory usage should be kept to a minimum.

As a general rule, the production code may not contain any debug code. Debug code is by definition not needed for normal operation and only increases the memory footprint.

4.2 Ease of use

Ease of use for programmers was one of the main concerns when developing the library. To ensure an easy to use and easy to understand interface, following constraints where set:

- No double pointers,
- No typedefs,
- Blocking API calls,
- No timers or callback functions,
- Maximum 4 function arguments.

The library is an extension of a socket and should only be used in a single thread; as should sockets. The library should be able to handle multiple connections in multiple threads without the use of control mechanisms by the programmer such as mutexes as long as one connection is only used in one thread at the time.

4.3 Reliability

The library will support the latest version of MQTT, 3.1.1. In order to ensure reliability some addition constraints and requirements are enforced:

- No void pointer casts unless well motivated,
- Static type checking,
- Code coverage of unit tests over 75 percent,
- No memory leaks,
- Cyclomatic complexity must be less than 16.

Cyclomatic complexity gives an indication about how hard it is to understand and test a function. At its core it counts the lines of code and gives them a score based upon which type of operation they are. This score is multiplied by the nesting level, code inside and if statement for example has a nesting level of 1. Inserting a loop inside this if statement will add an additional nesting level (Korb, 2011). The lower the complexity score the better. In addition, reasonable compiler warnings and errors are enabled.

5 METHODS AND TOOLS

Portability was not only a key feature of the library, the workflow should be portable and future proof as well. This prohibits the use of many modern tools and IDE’s. Make tool with Makefiles are the preferred build system (GNU, 2016). Also for unit testing the Check framework was selected for its simplicity, portability and reliability (Check, 2016). To check the complexity, the complexity tool by GNU was selected (GNU, 2011). This complexity tool is based on previous research and is open source (National Institute of Standards and Technology (NIST), 1996). These tools are discussed in more detail in this section.

5.1 Build system

The main goal of the build system is to compile the source code. More advanced build systems will integrate unit testing, memory tests, complexity checks… and will automatically generate the scripts needed to perform these tasks.

Make is a tool that uses Makefiles as a script to generate the executable. Make needs dependencies, rules and macros in order to know how to do this. The rules define which command to run to compile a file. Make tracks dependencies that are listed in the Makefile, if one changes Make will automatically rerun the rule to update the objects. This eliminates the need to recompile the whole project if one file changes. The log file created when completely rebuilding the project is added on Github to illustrate how clear and comprehensive the output of the build process is: “log_file_example.txt”. The output format displayed was custom designed and programmed. By making the project a new log file will be generated called “log_file.txt”. This file can be used in a quality assurance system and can be generated in more web friendly formats as well.

5.2 Tools

Several tools have been developed over the years to assist in writing compact, good quality and bug free code. Several of these tools where used in this project and are briefly discussed here.

5.2.1 GNU Gcov

Gcov is a tool that reports on how much percentage of the production code is run by the unit tests (GNU, 2016). It is a part of the GCC compiler. When the unit test executable is compiled, GCC ads additional information to the compilation that allows it to track if a line of code is used by the unit test executable. It copies the source file and adds the number of times each line of code was run in front it. The coverage percentage calculated is the ratio between the lines of code used in the tests (each line will only be counted once) and the total lines of code.

The code coverage percentage outputted by this tool is however easy to misjudge. When writing a single unit test it is easy to have a coverage of 50 % of the code. This because in a normal operation most of the code is used.

In order to have a better code coverage, additional tests need to be written to create (very) uncommon but possible situations for the code to process. Often only 1 or 2 lines of code are needed to detect and return an error or handle an uncommon situation, the unit test coded needed for this situation is often longer than the original unit test code.

This has the unfortunate effect that when comparing 2 blocks of code, one with a test coverage of 50% and the other with a test coverage of 75%, it is easy to mistakenly think that the second block has 25% more test code. In reality the second block of code will have 3 or 4 times more test code associated with it. In order to reach test coverage of 95% or higher huge effort needs to be places upon testing. This forces the programmer to closely examine the codes behavior in any situation leading to excellent code.

5.2.2 Check framework

The check unit test framework is a widely used framework for unit testing in C (Check, 2016). It provides a set of macros and functions to easily set up a test environment. Every unit test runs in a separate threat, this increases the speed at which a test set is completed. Check offers a set of assert functions to check if a functions response is as expected. After running the tests, it will report which test have failed. The code coverage tool forces programmers to add more and more tests, making it much harder to pass all of them. When changes are made it subtle bugs will be caught more easily. In order to finish the library every single unit test needs to succeed.

5.2.3 Memory leak detection

Memory management is typically done after writing the software, making it a tedious process Valgrind is a typical framework used for memory leak detection. Valgrind is a framework providing dynamic analysis tools to benchmark programs, to check for memory leaks and much more (Valgring, 2016). The framework is very powerful and extensive, as a results also complex to use and integrate.

In order to better track memory management and give a detailed view on the memory usage a custom tool was created by me for this project and integrated into the build process. The tool is compiled in during debugging and keeps track of every memory allocation and its release. It also keeps track which file, function and line allocated the memory, simplifying the tracking of memory leaks.

What makes the tool very useful is its integration into the Check unit testing framework. At any time during the unit test the programmer can specify how many memory chunks should be allocated. If the number does not match the test fails. At the end of each unit test there should be no allocations. An allocation report can be generated detailing all the statics about the memory used and also which memory blocks are still in use, facilitating tracking. The tool can even do garbage collection to clean up lost memory blocks. For implementation details refer to pico_mqtt_debug.h on Github. In the unit test files ending with "_test.c" on Github examples can be found on how this tool is used for unit testing. This tool was integrated with the check framework, making it impossible to successfully run a unit test that has a memory leak.

5.3 Automation

The biggest drawback of Make is listing the dependencies. Dependencies are all the files that are linked to a source code file. For every source file the dependencies need to be listed. If a dependency is not listed, the build process Make will not update the executable to the new version, making dependency listing a very tedious and error prone task.

In order to create a better workflow an advanced script was created that automatically list the dependencies of every file. GCC was used for this task. This does not compromise portability because GCC is only used to generate the dependencies and another compiler can be used for (cross) compilation.

The script will first search for all the source files in the folder. Next GCC is used to generate a dependency file for every source file, these dependencies are then loaded into Make. The complete Makefile can be found on Github. The method is based upon proven scripts (Smith, 2016) and is expanded and customized for this project.

In the Makefile, support is integrated for unit testing, complexity checking, test coverage with Gcov and memory leak detection.

The integration of all these tools allows for a smooth workflow with a strong focus on simplicity and reliability. Every time the Makefile is executed it will rerun the unit tests for the changed files, revealing potential bugs. By integrating these tools, compilations are fast and the programmer gets compact and first rate info about the code just written.

During every step in the code development effort was taken to place as much constraints and checks as possible on the code. It was also made very hard to circumvent any of these checks. Doing these checks significantly increased development time, not only for writing the code but also for adding additional checks and expanding the test framework created. The reason why this seemingly

insane choice was made is also the difference between regular code and production code: quality.
This rigorous process forces bugs to reveal themselves, reducing the overall number of bugs. For typical small project this is of less importance but for production code that is intended to be expanded upon this is critical. To illustrate how critical, try to imagine a (small) bug in a TCP stack. This would lead to for example a web server that sometimes lost connection to a client. Since the source code of the TCP stack is not always available to the programmer it would be almost impossible to track down the bug. Adding tests once the code is finished is very hard and violates the principles of test driven development. In the next section even more checks added will be discussed.

5.4 Debug code

Program complexity is a very important metric to keep track of. Reducing complexity will lead to less errors and faster development. Memory in embedded device is also at a premium forcing code size and memory usage to a minimum. This means that debug code which by definition is not necessary for production should never be in the source code files. The debug code never the less is very useful in the source code as it will check for programming mistakes, reduce bug count, ease development for future library users and generally improving code quality. The perfect solution to this conundrum would be debug code that is extremely simple (no added complexity) and that takes up no memory in production code. Removing the code after development is not an option because development on a library never stops and heaving two versions is in direct violation of best practices.
In order to be able to add debug code in the source code without violating conventions a clever set of macros was devised. The most important macros created for debugging are:

- CHECK (*boolean check, error message)*
- CHECK_NULL (*pointer*)
- CHECK_NOT_NULL (*pointer*)

These macros are compiled out when PRODUCTION is defined and do not take up any memory in production code. They are always used at the beginning of a function and are clearly distinguishable, in fact they improve the programmers understanding of the function by clearly stating which assumptions are made.
If a check fails the program will exit with an appropriate error message. Most checks are done on input pointers to avoid null pointer dereferencing. Also array boundary checks are performed using these macros. These checks almost completely eliminate two common hard to track errors: null pointer dereferencing and writing out of bound. An example of their usage:

```
uint8_t pico_mqtt_list_peek(
    struct pico_mqtt_list* list )
{
  CHECK_NOT_NULL(list);
  return list->first;
}
```

To print a message to the debug console extra macros where provided that can be enabled separately, they all use the familiar printf arguments:

- PERROR(const char*, …)
- PWARNING(const char*,…)
- PINFO(const char*,…)
- PTODO(const char*,…)

Other macros where provided to enable or disable error reporting in the unit tests. These macros will be compiled out completely in production code.

6 DESIGN

Up to this point all the requirements and restrictions posed on the implementation are described. Creating good quality code that fits these requirements is a tedious task that often requires rewriting huge portions of the code base. In the next sections is described how these strangling requirements are met successfully using elegant code.
The library interface is declared in the pico_mqtt.h header file. It provides all the functions needed by the programmer to start using the library.
The design of the library is centered around this interface and the socket. The socket is a byte oriented connection where the MQTT protocol and the interface are message oriented. The stream class was created in order to buffer the messages and handle concurrent input and output. The main protocol logic is implemented in the MQTT object; it also implements the user interface.
To store incoming, and outgoing messages, lists are used. The wait list contains messages that are waiting for a response from the broker before being destroyed. To serialize and de-serialize messages the serializer is used. In **Figure 5.** Library class diagram, the class diagram of the library is visualized. When the user creates a new message with QoS 1 the data flow through the different blocks is described in **Figure 6.** Message flow QoS 1. Please note that C is not an object oriented language but object like structures where added for a better and more efficient design. Please refer to **Figure 5.** Library class diagram and **Figure 6.** Message flow QoS 1, for more information.

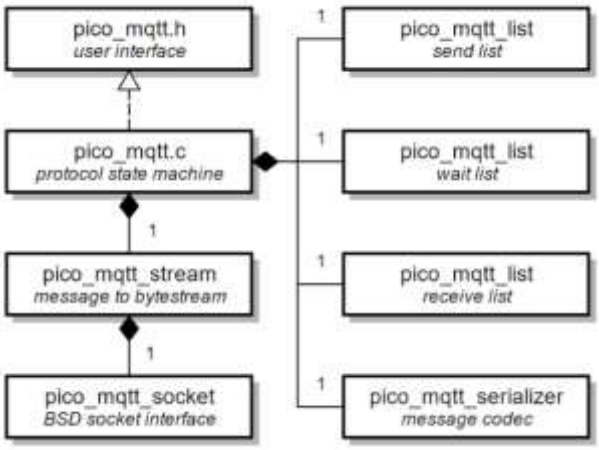

Figure 5. Library class diagram

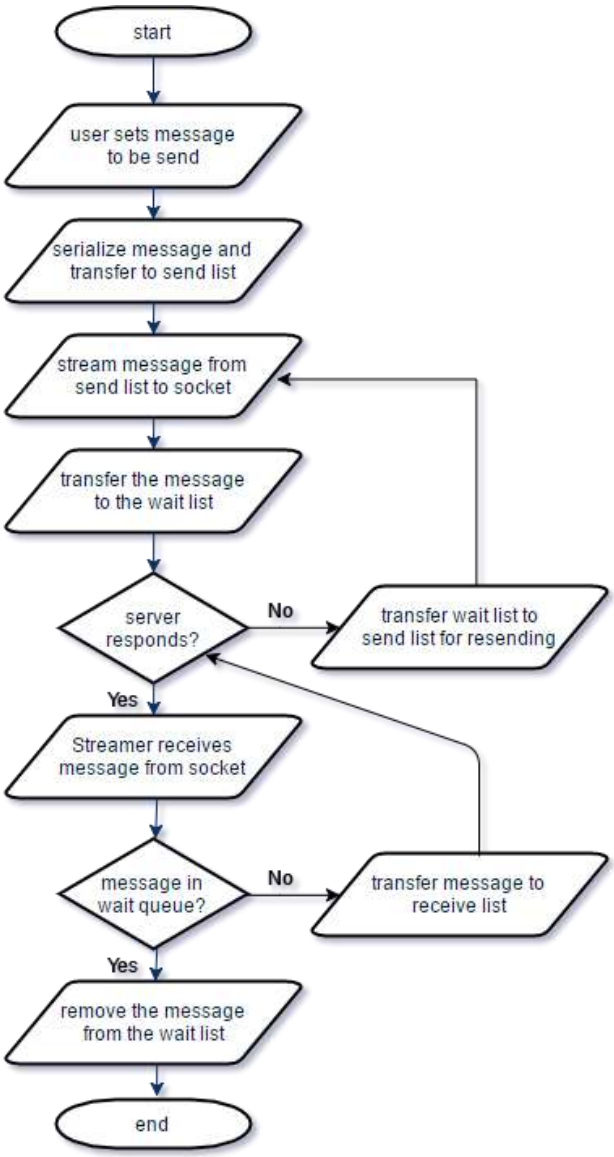

Figure 6. Message flow QoS 1

6.1 Sending a QoS 1 message

This section describes what happens internally in the library when running an application that publishes a QoS 1 message. This can be a humidity sensor in the garden for example.

6.1.1 Preparing to connect

Before the application can send a message to the broker it needs to connect to the network. Depending on the configuration of the broker identification may be required. The setting of the broker can be duplicated in the pico_mqtt_configuration.h file, this allows the library to validate the connect message before sending it. If for example a username and password are required by the broker, the USERNAME_REQUIRED flag and PASSWORD_REQUIRED flag can be set to 1 in the configuration file. When the application attempts to connect to the network without setting a valid username and password an error will be returned. The MQTT protocol allows some choice in the implementation of the broker, the configuration allows to library to match any of these choices. Depending on these configurations extra functions and unit tests will be added and run.

The first step the application has to take is to create a new pico_mqtt object. This object will contain all the information about the connection. Once a pico_mqtt object is created the application can set a client ID, username, password, will message and will topic using the functions provided by the library interface. Every time the application sets these fields the fields are checked for errors. If an error is detected the setting is refused and an error is returned to the application.

6.1.2 Connecting to the network

When all the parameters are set to connect to the network the application calls the pico_mqtt_connect function with the address of the server and its port. The serializer will be used to serialize the connect message. Any error that occurs will be returned to the user. If the connect message is serialized correctly it will be send to the broker via the streamer and socket. If the broker returns a publish acknowledge message (PUBACK) without errors the application is connected. A detailed description on how messages are send and received will be provided in the next section.

6.1.3 Publishing the QoS 1 message

A visual representation of sending a QoS 1 message is shown in **Figure 6.** Message flow QoS 1 and is discussed in detail here. The application can send a message by calling the pico_mqtt_publish function and providing a message to be send. The message will be checked for invalid characters, wildcards, system topic and more. If the check succeeds the message is serialized by the serializer and pushed to the send list. A state machine will check if time is left and if the message is send completely. If not send yet, the message will be send to the streamer. The streamer will send the message to the broker byte by byte while time is left. The message is also transferred to the wait list. If the connection is lost unexpected the message will be transferred back to the send list for retransmission. If time is left the state machine will instruct the streamer to receive the response from the server. If no time is left the task will be completed the next time the application sends or receives a message. Once the publish acknowledge (PUBACK) is received the message is removed from the wait list.

6.1.4 The state machine

The biggest benefit of the way the library is implemented is the state machine with the lists. If a connection is slow or lost the application will not notice it. Once the message is serialized successfully no errors will be returned to the application unless the connection cannot be restored. If there was not enough time to complete a transmission when the user created the message, it will automatically continue the next time there is sufficient time. When the application sends more data than the network can handle an error will be returned, preventing the device from running out of memory. The state machine is used to detect if previous messages needs to be completed. These messages are stored in the lists. If a message is received while the user was sending one, it will also be stored. This

has the benefit that when the user requests a message at a later time it will be returned immediately.
Complicated scenarios can arise when messages are in the process of being transferred and errors occur. In other implementations it is always up to the user to resolve these situations. This library takes care of all of these problems. When every solution has been tried the application will be notified. At this point it is quite possible multiple brokers have been contacted in the libraries attempt to deliver the messages. The only option the application has after an error returns is to write the data to permanent storage and reconnect later. By taking care of these scenarios the application becomes much easier to write.

7 IMPLEMENTATION

The C programming language was selected for portability; the downside is the limit constructions provided by C. Even the most basic programming structures like objects are not defined. Challenges faced during implementation where solved by implementing well known concepts from other programming languages in C.
In contrast to other thesis's is the code developed is used in a production environment. More than 70% of code written is test code. Writing this test code requires critical reflection on the production code written, this resulted to more or less 3 rewrites of the whole production code base in order to fit the requirements in an elegant way.
In contrast to code that is written to solve a problem it is much harder to implement a library. When solving a problem, the only restrictions posed are the interface and practical restriction. When implementing a library, it is necessary to adhere to every single restriction defined or derived. If a code choice is made it is also critical to reflect if this choice will force protocol violations in a later stage. When failing to foresee a future challenge it will become necessary to rewrite huge portions of the code written and tested. To adhere to the MQTT library specification a set of 150 rules needs to be adhered to, this list can be found at the end of the protocol specification (Gupta, 2014).
The full source code is available at https://github.com/tass-belgium/picotcp-modules/tree/master/pico_mqtt.

7.1 Data structures

In order to make the library well-structured object-like data structures where created. These structures can be seen as primitive objects as they contain all the data needed for the proper working of the associated functions. They can also contain other objects (data structures). The main goal of these data structures is to hide the objects implementation without overhead.
The basic objects created in C are structures, unlike objects know in other programming languages they only contain data and no functions or a reference to self. It is possible to add functions to an object with function pointers but this adds unneeded overhead and makes the code more complex. In order to create functions that are private to the object, separate files for each object are created containing static functions. Public functions have the objects name in them to avoid name collision.
There is chosen not to use function pointers for the object related functions to avoid the associated overhead. In order to allow multithreaded use of the library no global variables are used and the object is passed to the functions. Every object can only be used in a single thread, as are sockets, but multiple object may be used.
Interfaces are defined in the header files, providing the separation needed for easy testing. Other constructs used in Object Oriented Programming (OOP) such as inheritance are not supported because there was no need. For testing every object and its implementation was mocked. This means a second version was created that always behaved the same (same return value and so on), based upon settings in order to test functions that used these objects.

7.2 Memory management

A key aspect of any program that is required to run over longer periods of time is its memory management. Any memory leak will result in problems if the program runs over longer periods of time. Java solves this problem by employing automatic garbage collection, in C++ this problem is solved by the use of smart pointers. In the MQTT library the concept of data ownership and object lifetime from C++ smart pointers is used.
When an object such as the serializer allocates memory for a new object it takes ownership of this object. In the case of the serializer by setting a flag. If the serializer would be destroyed it is responsible for destroying all objects it owns as well. When an object owned by the serializer is requested, the ownership of the object transfers as well.
When the user requests an object, for example a message, he takes ownership of this message and should destroy it when done. While the concept is simple, it's implementation assures the memory is handled correctly between different modules. Unfortunately, it is not possible to automate these checks in C and during every step of the way test where added that nevertheless ensured proper behavior.

7.3 Streams

Serializing and de-serializing transforms an array of bytes to a message. When dealing with arrays keeping track of the boundary's is of the utmost importance. In other languages the programmer doesn't have to worry about this and constructs to do boundary checking are common place.
To prevent boundary issues, streams where added to the library. To interact with the streams read and write functions where implemented. These allowed to read or write a number of bytes from the memory without worrying about the boundaries. Different versions where provided to read different datatypes directly. Also objects can be read or written directly from and to the stream.

7.4 Error handling

Exception handling is an important feature of most programming languages but is lacking in C. A common way to add this, is for functions to return an error code. When an error is raised by a function it will ripple through to a function that handles the error, interrupting the normal flow of the program.
In the implementation only 1 error variable is used and is shared by the whole library. If an error is raised either the main state machine will close and reopen the TCP connection following the MQTT specification or the error will be reported to the user. Most errors requiring restarting the connection result from protocol violations by the broker or client library implementation, these should be handled by the library itself. The errors returned to the user mostly originate from erroneous inputs or system limitation and requires the user's intervention.

8 RESULTS

In this section will be checked if the MQTT library coheres to the requirements set in section 4, requirements. For detailed instruction on how to start using the library refer to "getting_started.txt" on Github. All code is available on Github.

8.1 Portability

In order to be portable no compiler specific functionality can be used. Throughout the library only standard C is used. For the unit testing the source file is included into the test source file in order to access the static functions. Functions from other files where mocked using special mock files. This alleviated the need of compiler specific options such as the gcc attribute "__attribute__((weak))", allowing function to be overwritten, commonly used for testing.
The MQTT library written can be compiled using the prerequisite flags without errors or warnings, meeting the requirements.
None of the restricted functionality was used in order to create the library, more specifically bitfields, switch statements, enums and unions are never used.

8.2 Code quality

The most complex function of the library tops out on a complexity of 6. This is well under the maximum complexity of 16, other functions have a score less than 4 with some exceptions of 4 and 5.
The average unit test case coverage is over 95% of most files, some are even 100%, well surpassing the required minimum of 75%. The memory leak detection tool was used to detect memory leaks. This tool is able to guarantee no memory leaks are present. When compiling the application this needs to be rechecked because it is still possible that unique untested situation arises that lead to memory leakage. All efforts have been taken to minimize this possibility.
The ratio between the number of lines of production versus test code gives an indication about the work required to achieve this level of testing. The total number of lines is around 15 000 lines of code, of which 3000 to 4000 is production code.

8.3 Interoperability

Every effort was made to select tools that are ported to as many platforms as possible. The library was used with the Mosquitto broker; this is one of the most used MQTT brokers (Eclipse, 2016). No connections where refused indicating that pico-mqtt follows the standards. In the future the library should be checked with other brokers as well.
The code size of the library is only 2,5kB with a ram footprint of 500 bytes. If some features are disabled in the configuration file this can become even less. The basic Pico TCP stack of Altran on top of which this library can be run is 16kB, this is one of the smallest TCP stacks available on the market.

8.4 Functional test

The most important test of the library is the functional test. This test checks if the library correctly performs the tasks required from the library. The setup of this test is easy. A local Mosquitto broker runs on the local machine (Eclipse, 2016). This Mosquitto broker is used a lot as a standard MQTT broker, it follows the MQTT standards and will close the TCP connection if the library does not follow the specifications. The library is checked by connecting two clients that sends messages to each other, the Mosquitto broker will disconnect the client if it does not follow the MQTT specifications. By sending every type of message in every possible configuration, the library can be fully tested for normal functionality. If the connection is never closed it can be concluded with reasonable certainty that the library does not violate the MQTT specifications.
Once the local broker is running the next step is to run two applications using the MQTT library created. Both applications will connect to the broker using the different options to connect. Next one will subscribe to a topic and the other application will publish to the topic. This is repeated for the different quality of service levels. After this is done the application pings the server and unsubscribes from a topic. Also the multi topic subscription is tested. By performing all these tasks, the full MQTT protocol is used. At no point the TCP connection was closed or unexpected results where encountered. This proves that the MQTT library created is fully compliant to the MQTT protocol and can perform al tasks required from an MQTT library. The functional test code can be found in pico_mqtt_functional_test.c on Github. Further functional testing is required in order to check for all possible errors that can occur.

9 DISCUSSION

In this section some choices made while developing the library are discussed.

9.1 Void pointer casting

In the requirements was clearly stated that the library should be statically type checked and no void pointer casts are allowed. One exception was made for streaming the payload of the publish message. In order to allow the user to send any type of data, the data object contains a void pointer. This void pointer will be set by the user to the allocated memory containing the message. Any other pointer types would result in the user having to cast their pointers to this type. Since the data type the users prefers is unknown, void pointers are the only option.
If the user would use only one data type a macro could be used but this would severely limit the libraries flexibility. The library never uses this pointer and simply passes it along until the point the data needs to be send. C does not permit operations using void pointers, these operations are required in order to read the referenced data byte by byte. To overcome this limitation, the data was cast to uint8_t*. As of the cast the data is considered to be an array of bytes of a known length. One by one these bytes are send by the streamer. The inevitability of the use of void pointer casting justifies it's use in this case.

9.2 Returning subscription QoS

In section 3: Reflection on MQTT, a clear case is made to show that the returned QoS of a subscription is not only useless but can lead to unintended site effects and confusion. In order to prevent library users to run into these issues the subscription QoS is not made available. This does not in any way limit the user.

9.3 Multiple subscriptions

The MQTT standard allows clients to subscribe to multiple topics at once. The use of this functionality is limited and poorly fits with the requirements of the library. The benefit of the multi topic subscription is that only one message has to be send for multiple subscriptions, saving the overhead of the fixed header. The MQTT broker is also able to respond with only one acknowledgement. There are several reasons why this feature is unimportant and not implemented.
The first one is that the fixed header length is only two bytes for short messages and up to five bytes for extremely long messages, making the headers overhead when performing separate subscriptions very small.
The second reason is that there will be much more messages from other types send than there will be subscription messages send. This for the simple reason that only one subscription must be send to receive multiple messages of other types and publicizing a message does not require any subscriptions to be send at all.
The availability of alternatives is the last reason why multiple subscriptions in one message is an unnecessary feature. A client can opt to use wildcards for multiple subscriptions or a client can connect to the broker and requesting to preserve all subscriptions from the previous session.
The small overhead associated with single versus multi topic subscription combined with the availability of alternative techniques to achieve the same results does not justify the added complexity and program size a multi topic subscription implementation requires. In order to allow for multi topic subscription the interface should ether allow to pass in an array of messages with its length or provide functions to load topics into an internal array. This would unnecessarily complicate the interface.

9.4 UTF-8 support

The MQTT specification requires the use of UTF-8 encoded string. The reason why is not clear and is discussed in detail in 3.9, UTF-8 string encoding. In order to add support for UTF-8 on embedded devices extra libraries need to be added. Additionally, most embedded libraries don't support UTF-8 strings, forcing the use of several extensive conversion libraries. URI's for example allow UTF-8 by replacing an UTF-8 characters by %... codes. In some cases, conversions may even be impossible.
To avoid this pitfall, the library only supports ACHII encoded strings. ACHII is a subset of UTF-8 and therefor any valid ACHII string is a valid UTF-8 string. Since there is no need for human readable data in machine to machine communication no limitation is set by only supporting ACHII. Specifically, there is no reason why special UTF-8 characters would be used.

10 CONCLUSION

The MQTT library created is clearly up to specifications. The interface of the library is easy to use and the quality of the work is good and well tested. There are alternatives available that provide the same functionality, also the way they are implemented similar, proving that the structure of the developed library is done well. Other libraries don't meet the requirements of the project however. In order to be a fully competing library supports for security should be added. The build system used together with the tools integration form a complementarity solution and is highly recommended for new designs. Especially the introduction of leak detection and CHECK adds powerfully new capabilities for programmers and will aid in the development of applications using this library.

11 REFERENCES

C++ compiler support. (2016, July 26). Retrieved from cppreference: en.cppreference.com

Check. (2016, 7 12). *Check unit testing framework for C*. Retrieved from libcheck.Github.io: https://libcheck.Github.io/check/

Eclipse. (2016, Jun 15). *Eclipse Mosquitto*. Retrieved from Github: https://Github.com/eclipse/mosquitto

Eclipse. (2016, jun 20). *paho.mqtt.c*. Retrieved from Github: https://Github.com/eclipse/paho.mqtt.c

Eclipse. (2016, Jun 23). *paho.mqtt.embedded-c*. Retrieved from Github: https://Github.com/eclipse/paho.mqtt.embedded-c

GNU. (2011, 5 15). *Complexity - Measure complexity of C source*. Retrieved from GNU: https://www.gnu.org/software/complexity/manual/

GNU. (2016, 4 27). *gcov - a Test Coverage Program*. Retrieved from gcc.gnu: https://gcc.gnu.org/onlinedocs/gcc-6.1.0/gcc/Gcov.html#Gcov

GNU. (2016, 5 22). *GNU Make*. Retrieved from GNU.org: https://www.gnu.org/software/make/

Google. (2016, 8 21). *Google Trends*. Retrieved from Google: https://www.google.nl/trends/explore?q=mqtt,coap,smcp,AMQP,STOMP%20protocol%20%2B%20DDS%20protocol%20%2B%20Mihini%20%2B%20M3DA%20%2B%20LLAP%20protocol%20%2B%20LWM2M%20%2B%20SSI%20protocol

Gupta, A. B. (2014, 10 29). *MQTT Version 3.1.1*. Retrieved from OASIS Standard: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html

IoT Standards and Protocols. (2015, 3 1). Retrieved from postscapes: http://postscapes.com/internet-of-things-protocols/

Korb, B. (2011, May). *Complexity - Measure complexity of C source*. Retrieved from GNU: https://www.gnu.org/software/complexity/manual/complexity.pdf

National Institute of Standards and Technology (NIST). (1996). Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric . *National Institute of Standards and Technology Special Publication 500-235*, 123.

Ruiz, V. (2014, Dec 12). *menudoproblema/libemqtt*. Retrieved from Github: https://Github.com/menudoproblema/libemqtt

Smith, P. D. (2016, January 4). *Auto-Dependency Generation*. Retrieved from mad-scientist.net: http://make.mad-scientist.net/papers/advanced-auto-dependency-generation/

Valgring. (2016, 9 22). *Valgrind*. Retrieved from Valgrind: www.valgrind.org

WolfSSL. (2016, Aug 1). *wolfSSL/wolfMQTT*. Retrieved from Github: https://Github.com/wolfSSL/wolfMQTT

APPENDIX A

Building an easy to use embedded MQTT library

MQTT is a publish-subscribe protocol often used for IoT applications. The goals of the thesis is to implement the MQTT protocol for an embedded C environment on top off a standard BSD socket. The main focus is on ease of use, portability, stability and memory use.

Constraints and requirements:

- The library cannot use bit fields, typedefs or enums
- The library should be statically type checked (no void pointer casts), exceptions should be well motivated.
- The library should compile with the flags -Wall -Wdeclaration-after-statement -W -Wextra -Wshadow -Wcast-qual -Wwrite-strings -Wunused-variable -Wundef -Wunused-function -Wconversion -Wcast-align -Wmissing-prototypes -Wno-missing-field-initializers
- The library should be easy to understand and use
- The library should be well documented.
- Memory should be considered at a premium. Less memory use is better
- Code coverage by unit tests should be >75%
- Functional test should confirm interoperability with other implementations, especially but not exclusively mosquitto. Preferably 2 other brokers.
- The cyclomatic complexity of must functions should be less than 16, it the complexity is higher it needs to be well motivated
- API calls need to be blocking
- No timers can be used. The library can request the system time
- There should be no memory leaks, memory management is a bonus
- Error handling and error reporting should be easy for the user
- The API cannot use callback functions, functions with more than 4 arguments, double pointers or other advanced techniques
- The library should conform to the MQTT 3.1 specifications

Optional extra's

- Conditional compiling for features.
- Enable the use of https.
- Benchmarking to test stability and confirm claims

Research questions

- How does MQTT compare with other relevant protocol throughout different use cases with a focus on performance and stability?
- How does MQTT behave when using it for new use cases (ftp, …)?
- What are the weaknesses of MQTT?
- Is it possible to port MQTT to different test platforms and is the behavior of the test cases as expected?