J Inf Process Syst, Vol.13, No.1, pp.26~39, February 2017
https://doi.org/10.3745/JIPS.04.0028

ISSN 1976-913X (Print)
ISSN 2092-805X (Electronic)

JOURNAL OF INFORMATION PROCESSING SYSTEMS JIPS

Efficient Implementation of the MQTT Protocol for Embedded Systems

Olivier Deschambault*, Abdelouahed Gherbi*, and Christian Légaré**

Abstract

The need for embedded devices to be able to exchange information with each other and with data centers is essential for the advent of the Internet of Things (IoT). Several existing communication protocols are designed for small devices including the message-queue telemetry transport (MQTT) protocol or the constrained application protocol (CoAP). However, most of the existing implementations are convenient for computers or smart phones but do not consider the strict constraints and limitations with regard resource usage, portability and configuration. In this paper, we report on an industrial research and development project which focuses on the design, implementation, testing and deployment of a MQTT module. The goal of this project is to develop this module for platforms having minimal RAM, flash code memory and processing power. This software module should be fully compliant with the MQTT protocol specification, portable, and inter-operable with other software stacks. In this paper, we present our approach based on abstraction layers to the design of the MQTT module and we discuss the compliance of the implementation with the requirements set including the MISRA static analysis requirements.

Keywords

Embedded Systems, Internet of Things, Message Queue Telemetry Transport, Quality of Service

1. Introduction

The need for connected devices has grown rapidly in the last few years. Starting with personal computers using the internet, continuing with smartphones and new wireless technologies and reaching a point where even small devices, not necessarily controlled by a human, should be able to communicate. They will be able to exchange any kind of data to enable new opportunities that we barely start to foresee. The growth of this new domain of connected devices talking to one another has been called the Internet of Things (IoT).

The IoT describes a paradigm shift where a multitude of 'things', such as small sensors or actuators, will communicate with other machines enabling and access to new options of technologies and usage. The IoT has already started, for example with the nest thermostat or with lightbulbs that can be remotely-controlled from any smartphone from anywhere in the world. These examples are simple ones and do not reflect the potential behind the IoT wave that is building. There could be as much as 50

※ This is an Open Access article distributed under the terms of the Creative Commons Attribution Non-Commercial License (http://creativecommons.org/licenses/by-nc/3.0/) which permits unrestricted non-commercial use, distribution, and reproduction in any medium, provided the original work is properly cited.
Manuscript received June 17, 2016; first revision November 11, 2016; accepted December 3, 2016.
Corresponding Author: Abdelouahed Gherbi (abdelouahed.gherbi@etsmtl.ca)
* Ecole de Technologie Superieure, Montreal, Canada (olivier.deschambault.1@ens.etsmtl.ca, abdelouahed.gherbi@etsmtl.ca)
** Micrium, Verdun, Canada (christian.legare@micrium.com)

Copyright© 2017 KIPS

billion devices by 2020, which would be about 7 devices for every human being on the planet [1]. This would lead to an abundance of data and the need for machines to communicate with each other in order to exchange the data gathered so that it can be analysed and to take action based on this analysis. This means that our everyday environment could gather information and answer adequately to any stimuli it perceives. This could mean less energy consumption, more fluid circulation for vehicles and gain in time, comfort and money.

In order to achieve the full potential of IoT, several software components have to be (re-)engineered with adequate quality of service levels taking into consideration the IoT context constraints. For example, several communication protocols already exist and have been designed exclusively for small devices than previous protocols. One of these protocol is the message-queue telemetry transport (MQTT), which is designed to work on any quality of network and where the largest part of the processing is done by the server, not by the devices, the things, which can and should remain fairly small and simple. Several software implementation of the MQTT protocol exist, but they are intended to work on computers or smartphones and not in the context of resources-constrained of embedded systems. Indeed, embedded systems often use lower-level language as C instead of Java, C++ or C#. In addition, the amount of memory and code space available is nowhere as high as when executing on a computer. Moreover, several kernels and networking stacks are available. The MQTT-client (MQTTc) stack should therefore be able to interface and to easily operate with any of them consistently.

In this paper, we report on an industrial experience, which consists in a research and development project. The goal of this project is to design, implement and test a software stack that should to be used in embedded systems and that fully implements the MQTT protocol standard specification. In addition to the specific requirements sketched earlier, software engineering best practices for developing software for an embedded context had to be respected and allowing both an ease of configuration and an ease of use.

The remaining part of the paper is structured as follows. In Section 2, we present a review of the MQTT protocol and an analysis of his features and the context in which this protocol should be preferred. Section 3 will detail the requirements for the MQTT client stack. This section will also provide more information about the challenges and particularities of an embedded system. In Section 4, we will explain the design choices made to achieve the requirements. We present and discuss in Section 5 our analysis of the design choices made, their impact on the stack, suggestions regarding future work and possible improvements. Finally, we conclude the paper in Section 6.

2. Background

Fig. 1 depicts a typical IoT infrastructure where several sensors communicate with a local gateway and possibly with each other over a local network. The local gateway may pre-process the data and transmit it via the internet to other services. In this section, we discuss the protocols which can be used to build this kind of IoT infrastructure.

2.1 Message Queue Telemetry Protocol

MQTT is an emergent lightweight bi-directional communication protocol that is intended to be used in an IoT context to enable the communication between a device and a server, called broker. The

MQTT protocol uses TCP to ensure the reliability the packets transmission and requires a persistent connection between the server and the client. It is a publish/subscribe protocol, as described in the specification [2] abstract. Since it is the responsibility of the broker to keep track of the clients and of what topics they are subscribed to, it reduces the complexity on the client's side. It is stated that publish/subscribe protocols such as MQTT are becoming important particularly for sensor devices in the IoT since messages need to be sent as efficiently as possible without consuming too much energy and needing few processing capabilities [3]. The MQTT protocol also supports several levels of quality of services (QoS), to ensure that even on a faulty network messages will be received ”at most once” (QoS 0), ”at least once” (QoS 1) or ”exactly once” (QoS 2).

Of course, the higher the QoS level, the higher the complexity and overhead of the exchange between the client and the broker. This is reflected by the higher end-to-end delay when using a higher QoS. It has been found that when using a wired network, the end-to-end delay of a payload transmitted via MQTT will typically be below 0.1 seconds for QoS 2, about 0.05 seconds for QoS 1 and below 0.05 seconds for QoS 0. This means that determining the right QoS for various applications leads to better performance.

A formal analysis of the MQTT found no flaws in the specification when the exchange between the client and the broker was unhindered. A flaw was found in the specification if the two suggested methods for handling QoS 2 were used simultaneously in the broker and that the network could be considered hostile [4].

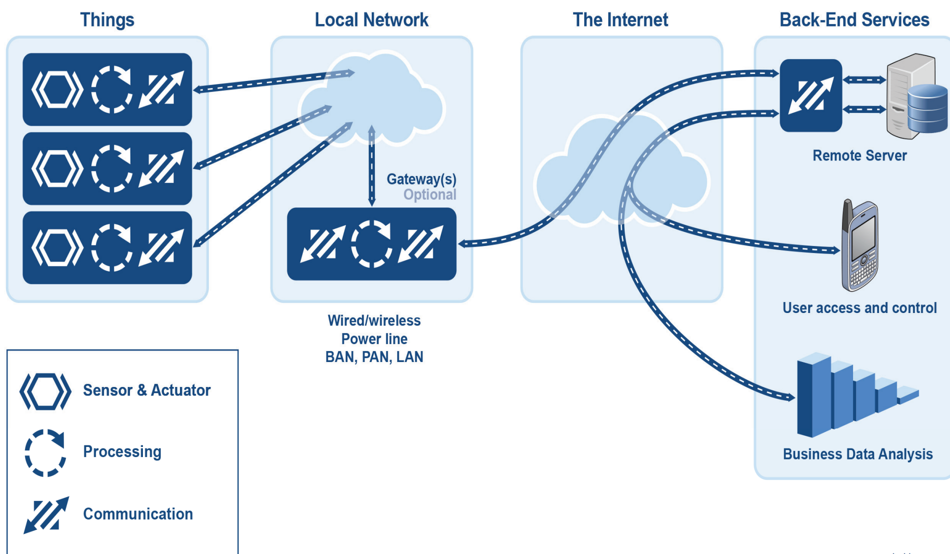

Fig. 1. Internet of Things ecosystem.

2.2 Constrained Application Protocol

There is also a lighter protocol, the constrained application protocol (CoAP), which has been designed to be easily translated to a hypertext transfer protocol (HTTP) equivalent. It uses UDP instead of TCP, which reduces the overhead for every packet, but also reducing reliability. That is why an optional confirmation scheme has been integrated in the protocol itself. A CoAP protocol implementation

must also potentially reorder the packets received, since it cannot rely on TCP to do so. A comparative analysis about the delays and bandwidth usage efficiency in messages exchanged with CoAP and MQTT in various conditions regarding the quality of the network revealed that when more re-transmissions are needed, the TCP overhead makes it so that MQTT has a longer delay. But with few re-transmissions, less or equal to 20%, MQTT has a lower delay than CoAP [5]. It has also been found that when the loss rate is low, CoAP needs less overhead to transmit data to a server than MQTT does. But, with a high packet loss, if the message is big (around 300–350 bytes), the probability of losing a message using UDP, as CoAP does, is higher than if using TCP, as MQTT does. CoAP then needs to re-transmit the whole message more often than MQTT does, which leads to better performance under these conditions for MQTT.

Another comparison between MQTT and CoAP found that the round-trip time was less for CoAP than for MQTT: 127 ms vs. 160 ms, respectively [6]. It was also determined that CoAP almost always uses less bandwidth than MQTT for the same amount of data sent, which is confirmed by other results [5,7]. MQTT has a better reliability, partly due to using TCP and to their QoS scheme and it was found that MQTT has better features regarding security and multi-cast, while CoAP has a better compatibility with HTTP and REST.

Also, whenever CoAP would need to communicate with another device, it would need to know the IP address of every device, possibly needing some discovery mechanism, which would complicate the application. This is not the case for MQTT and other protocols that support some kind of multi-cast. Resources and the processing required to process the requests of CoAP would make it hard for power constrained devices to support, especially when concurrency is high [8].

2.3 Other Protocols

Other protocols include the HTTP, widely known for its use on the internet. The HTTP protocol, while vastly used for several applications can be quite demanding to implement and support, which means that it cannot easily be used by very small devices requiring communication, but could still be a viable option for higher-ends systems which have the resources required to correctly support the HTTP protocol requirements.

Another protocol developed to answer the needs of that niche is the advanced message queuing protocol (AMQP). It uses TCP, like the MQTT protocol, as a transport layer. It does not use a publish/subscribe scheme and is more oriented to exchange data between two nodes, nothing in the protocol enables to easily relay some of the data to some other nodes, as is the case with MQTT. Also, according to [9], MQTT is more efficient energy-wise than AMQP. Also, MQTT would be the better suited of the two protocols for simple sensors or actuators using a constrained environment.

3. Requirements

3.1 MQTT Protocol Specification Details

It is required that the MQTTc stack complies with every item in the MQTT specification [2], including the three QoS levels that can be used to publish, as summarized in Fig. 2.

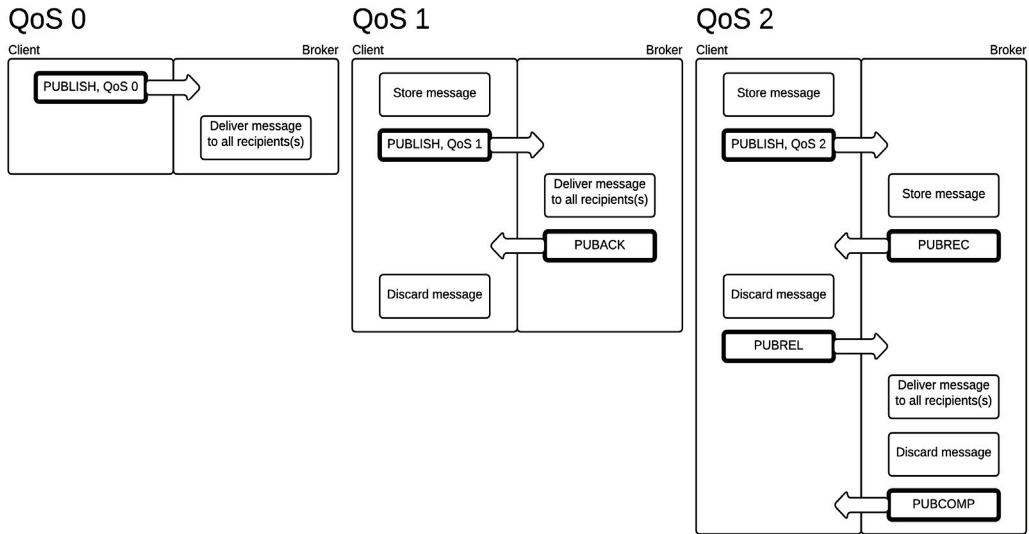

Fig. 2. Sequence for publishing at all QoS levels.

3.2 Embedded System Requirement Details

Requirements when designing for embedded systems differ a lot from requirements and constraints when designing for higher-level systems such as computers or smartphones. It is therefore important to consider the following several aspects.

3.2.1 Resources

An embedded system is typically a device that uses a processor that has limited processing capacity; a small amount of RAM memory, rarely more than a few hundred MB, sometimes less and a flash memory where the code resides that can also be quite limited. A microcontroller operating at a frequency of 32 MHz that has about 1 MB of code space and 128 kB of internal RAM could be considered to be medium-range. This means that when designing software for embedded systems, the amount of memory and the memory footprint that the code will occupy should be kept as low as possible. Also, anything that can be delegated to either the hardware or to a platform with a higher processing capability should be. Failure of doing so can mean that the software will not be able to be executed on some lower-end platforms, that the performance will be limited or, in order to avoid this, a better chip will need to be used, which leads to a more costly end-product. The software stack implemented hence requires using as few resources as possible, regarding code space, memory usage and CPU processing time.

3.2.2 Hardware portability

Another aspect of designing for embedded systems is the higher number of different architectures, peripherals and toolchains (compilers/linkers) that is available compared to their non-embedded counterparts.

In order for some software to work correctly on as many platforms, with their particular peripherals using as much toolchains as possible, some precautions must be taken. To be able to run on both 16-bits or 32-bits architectures and since the C standard does not set a standard size for the 'int' type, using width-constant types that will not vary depending on the platform on toolchain is essential. Also, in order to adjust to the various peripherals on which the software will run, software drivers are required. That is, instead of accessing directly a particular peripheral's registers, a standardized driver should abstract the exact way the hardware works and instead offer a public and constant interface to the software using that hardware. This way, the only thing that needs to be re-written each time a new peripheral needs to be supported is the driver, not the stack itself. In this case, the MQTTc module will use existing software from Micrium that already provides both the width-constant types and the network drivers that work with Micrium's network stack also used by the MQTTc module.

3.2.3 Software portability

Since software vendor for embedded systems often sell their software as blocks that each can be used distinctively, without using all the blocks from a single vendor, it may be required to write abstraction layers for software components, too. To be able to easily interface with any other component, a software block should specify clear boundaries and interfaces where other components should be inserted and, if need be, adjusted in some way.

Since particularities are hidden from the other side of the interface of the layer, it may cause some features that could have been useful to be hidden or some code to be less than optimal in certain cases. Because of this and also to reduce code complexity and increase the ease of maintenance, abstraction layers should be kept at a minimum.

3.2.4 Configuration

Another aspect of portability is the fact that in order to run efficiently on so many different architectures and to be able to handle the needs and requirements of as many applications as possible. To allow that, the software written should be easily configurable, either at compile-time using pre-processor directives or at run-time, if allocating data structures.

Also, in order to allow the user to have access to some debugging features when developing while avoiding to use too much resources in "release-ready" code, it is possible to use some pre-compiler directives, based on a compile-time configuration value to either include or not the debugging capabilities. Features that could be disabled at compile-time may include statistics, additional checks or asserts and logs or any other kind of output. The MQTT module therefore requires to be able to support a number of pertinent configurations to enable or disable features in order to decrease the resources required by the stack. It must also support some configuration options that could be used to tailor the behavior of the stack to the needs of his application or to the platform used.

4. Design and Implementation

This section will provide more details regarding the actual implementation of the MQTT-client (MQTTc) stack. A general overview of the implementation will be detailed after which a description for each of the requirements discussed previously will also be given.

4.1 Implementation Overview

Fig. 3 depicts the general behavior of the MQTTc internal task, in the context of which all the processing for every MQTT connection instance and every MQTT control packet is done. Having a single task doing everything MQTTc-related allows minimizing memory usage since only a single task stack is required. The internal MQTTc task has three main duties: receive any incoming data from a previously opened connection, send any data requested by the application and poll the message queue used by the application to request a new message to be sent. Indeed, all the functions requiring to send data use a queue to communicate with this task, in order to avoid long lock times as much as possible. The application tasks push a MQTT message to the queue, which is polled periodically by the internal task. If the task has a message to process, it will enqueue the correct data structures at the correct locations in its internal data structure and start the communication whenever the socket will be ready to send data. Most messages also require receiving some data from the broker and the task is constantly verifying if it has received something. It could be either a PUBLISH from the broker or the response to a previously sent message. Either way, the task will process it correctly, update the internal state of the message, receive additional data and/or call the callback specified by the application if the message has completed, as required. The internal task is written as an infinite loop, so it will constantly execute these operations, being idle for just the time specified by the application at initialization.

Having a single task and using a queue to communicate with external processes also allows this MQTTc stack to be thread-safe. This is not always the case for simple MQTTc libraries, such as Paho. They sometimes only provide synchronous APIs and are not thread-safe, meaning that no two calls to that MQTT library can overlap; one must complete before the subsequent one can start. Also, offering an asynchronous API and a callback system instead of a synchronous one allows application tasks to continue running while the message is being transmitted and processed, reducing the need for more tasks, instead make them more efficient and saving memory by reducing the number of tasks stacks needed.

4.2 Portability

Portability can be separated in two sub-sections, hardware and software, but since the MQTTc stack uses common modules provided by Micrium to abstract the CPU architecture and compiler that is being used, the MQTTc stack did not really need to make any effort to be portable on multiple hardware platform. The MQTTc is making sure to use the features that are provided by the common modules of Micrium regarding the width of the various data types and it was enough.

Regarding software portability, two abstraction layers have been used, as illustrated in Fig. 4. A first one to abstract the MQTTc from the kernel stack it is using, provided by Micrium. This has the benefit of allowing this stack to work with any kernel, since it is thread-safe by design, or no kernel at all, without much effort. The other abstraction layer is used to abstract which networking stack is being used, this one written exclusively for the MQTTc module. Functions composing this abstraction layer are related to socket-level operations in the networking stack. These functions are to open and close a socket, receive or transmit data or configure a socket to wait on some events and actually wait on those events. It has been decided to have this abstraction layer for the MQTTc module to be able to interface with any networking stack, since there are many options available, for example the ones provided by the chip manufacturer.

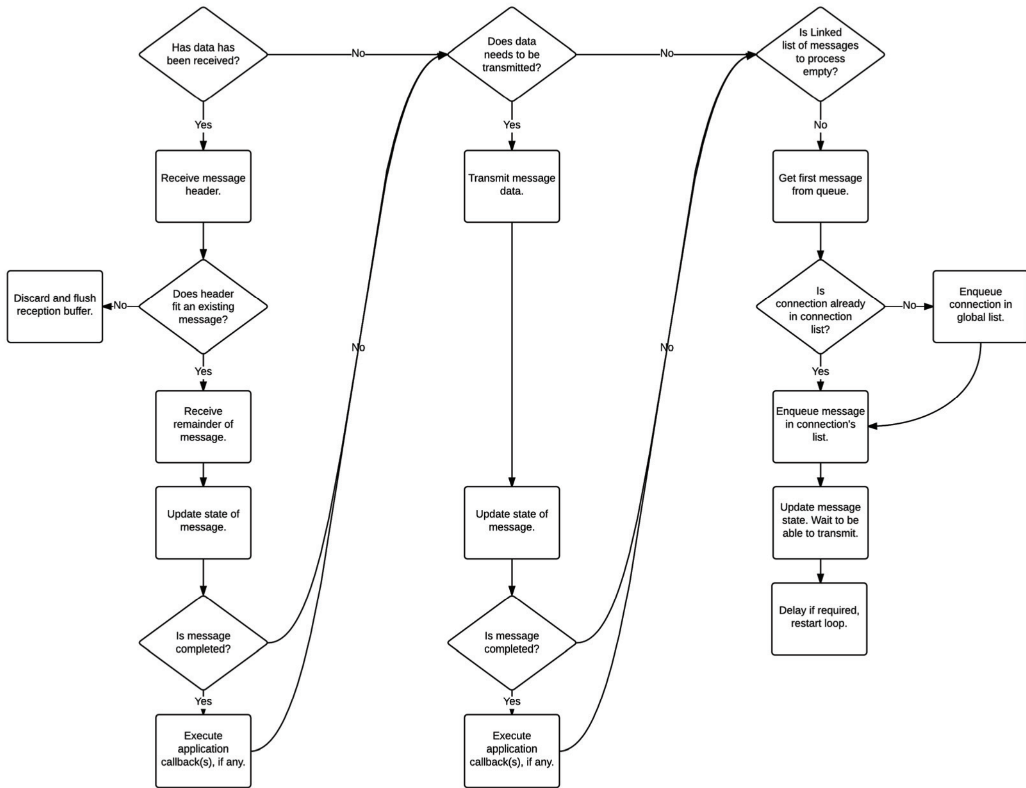

Fig. 3. Internal MQTT-client task processing.

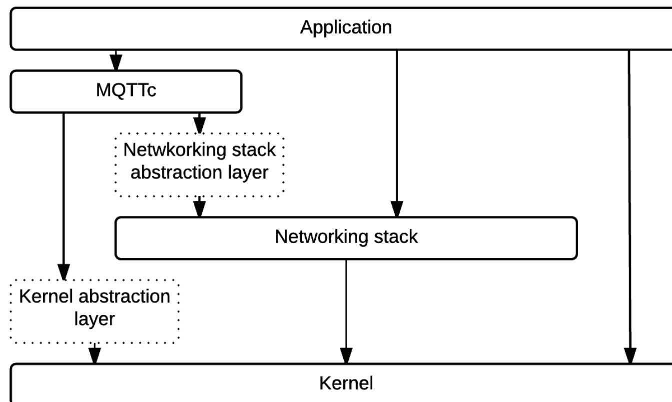

Fig. 4. Block diagram of the MQTT-client stack, its dependencies and its abstraction layers.

4.3 Code Footprint Optimization

To reduce as much as possible the memory footprint usage, and a bit related to the 'Configuration' requirement, a compile-time #define allows the user to enable or disable the optional argument checks. This will mainly be used in debugging mode or when developing but will be turned off, to reduce the code size and increase the execution speed when development will be over.

Also, it was decided, both to optimize the memory footprint and the resource usage, to force the user to allocate the data structures needed by some of the MQTTc functions in his application rather than putting this in the stack.

This does not require any code, where allocating these objects from a pool, the heap or anywhere else and managing them would have increased the code footprint without any significant gain.

The usage of the abstraction layer possibly increases the memory footprint usage, but it was decided that the increase in code footprint was largely compensated by the option to run the stack using any other networking module.

4.4 Memory Usage Optimization

In order to use as few resources as possible, a couple of solutions were implemented. The simplest one is that every data type used is always the smallest possible to do what it needs to do. Also, related to the 'Configuration' requirement, the user will provide the maximum number of message he wishes to be able to have on the network or queued within the MQTT-stack at the same time. This allows allocating the bare minimum of message IDs instead of keeping track of too many for nothing. It was also decided that as few internal structures as possible were going to be used, instead of placing the burden on the user to allocate any data structure needed by the stack, since the application knows exactly how much it needs and when it can re-use some of them.

Tables 1 and 2 illustrate where every data structures will be allocated and their typical sizes. It demonstrates that everything allocated by the stack is quite small, where a connection instance or a message instance or buffer will be bigger. That is why it has been deemed preferable to leave the application to allocate these data structures, in order to avoid allocating too much of them for the application's needs.

For the resource usage optimization, this has the advantage that since the application knows its needs, it can allocate and potentially re-use these data structures more easily than if it was the stack that was doing so. Also, there is no need for a complicated and resource-consuming overhead for managing the allocated objects since the application will know when it is done with any data structures, via the various callback mechanisms integrated in the stack.

Table 1. Application data structures size

Data structure types	Typical size (bytes)
MQTT-client connection instances	111
MQTT-client message instances	26
MQTT-client messages' buffers	From less than 10 bytes to hundreds of bytes

Table 2. MQTT-client stack data structures size

Data structure types	Typical size (bytes)
MQTT-client connection instance head pointer	4
MQTT-client message list head pointer	4
MQTT-client message list tail pointer	4
MQTT-client message ID reference table	8–16
MQTT-client message index	2
MQTT-client stack configuration pointer	4
Total	26–34

4.5 Configuration

The user will be able to specify at the stack the maximum number of messages that can be queued at any time, to reduce resource usage as much as possible. The stack will also allow the user to enable or disable both a temporary local buffer that can be used to analyze the data sent and received and checks on arguments that are coming from the user, in order to indicate, in development, if there is a problem with any given parameter.

Every possible configuration related to the MQTT protocol such as the broker IP address or name, the user name and password, the keep alive time-out can also be configured for every different MQTTc connection. It is also possible to configure the size of the buffer for each message used either to receive or to transmit by the stack. This has the advantage that if the user can know that for a given topic the payload to send will never be more than *N* bytes, he can configure that particular buffer to be *N* bytes instead of needing to account for the worst case.

The user can also configure which callback for every MQTT-connection he wants to be called, since the callback registering mechanism allows the user to fit the notifications to his needs.

5. Results and Discussion

This section will analyze the design choices that were made and see what impacts they have on the final result.

5.1 Portability

As mentioned before, software portability is always a trade-off between integration and therefore performance and optimization and the possibility to use the module with various stacks handling its needs. In the case of the MQTTc module, if a user would want to use it with another OS or with another networking stack, it would require him to write the abstraction layer, which was intentionally kept as simple as possible. The abstraction layers used only rely on simple constructs and functions that should be available in any decent OS and networking module.

Having this kind of portability is one of the biggest advantages of this MQTTc stack over comparable implementations. For example, some stacks, like Mosquitto, need a particular platform to work correctly, be it Windows, Linux or Arduino. Other stacks require particular software components or compilers, such as wolfMQTT which requires the wolfSSL library to support SSL/TLS and offers no easy method of using an alternative. This is not the case with this stack since it provides abstraction layers for both the hardware on which it runs and the other software components it is relying on.

5.2 Code Footprint Optimization

To calculate the code used by the MQTTc module, a test has been done, using the IAR compiler with Optimization set at High, Balanced. The total of the code footprint was 3456 bytes, when the extra checking was done, which is typically only during development. If the extra checks were disabled at

compile-time, 260 bytes were saved, giving a total for 'release' code of 3196 bytes. This is quite reasonable considering that the smallest microcontrollers have about 16 kB of flash for the code to use and that it increases rapidly into the hundreds of kB for other lower-end products. Of course this calculation does not take into account the size of the OS and of the network stack needed for the MQTTc module to work. Considering that a networking stack can be nearly 100 kB, the MQTTc stack is small enough to avoid being a factor when calculating the code flash requirement.

5.3 Memory Usage Optimization

In order to provide the memory usage for various use cases, a formula has been developed, based on the size of the various constructs required by the MQTTc module.

In order to provide examples of the memory usage of the MQTT stack, four scenarios were elaborated, from very simple to a pretty demanding one. Table 3 displays the results obtained. The configuration structure must also be accounted for, it needs 8 bytes, but can be located either in the flash memory or use RAM, it is not included in the calculations. The task stack also needs to be allocated, which can range from a few hundred bytes on small architectures to a few kB on more complex ones.

These results show that the MQTTc stack memory usage will scale with the complexity of the application, based on the number of MQTT connections that need to be opened simultaneously and the number of MQTT control packets, or messages, that need to be available at the same time. For a very small application on either a 16-bits or a 32-bits architecture, the amount of RAM used is between 100 and 200 bytes, depending on the number of control packets. Since even low-end microcontrollers have a few kB of RAM, this usage is quite reasonable, even considering the tasks stack that is needed by the OS, which could be a few hundred bytes on simple architectures. For more complex applications, requiring more connections and control packets available, the RAM usage grows steadily, requiring a higher-end platform to run on. Considering that these messages will need to be processed in a reasonable delay, it is to be expected that applications needing that many messages require more powerful platforms. With the most complex scenario, even when including the tasks stack, the amount of RAM used can easily stay below four kB, which can easily be available on most higher-end platforms.

Also, since the data structures are declared by the caller application and are allocated by the compiler, this allows them to remain valid only for the duration they are needing, without having to use any kind of dynamic allocation to allocate and free them. Several other MQTT stacks require a malloc and free implementation which increases code use and, on embedded systems, can lead to memory fragmentation if not used with caution.

Table 3. Memory usage scenarios

Scenario	Pointer size (bytes)	Number of connections	Number of messages	Total (bytes)
16-bits arch, small app	2	1	1	107
32-bits arch, small app	4	1	1	189
32-bits arch, medium app	4	2	16	664
32-bits arch, large app	4	3	64	2027

5.4 Configuration

The method of configuration used has several advantages but also some disadvantages. It is very flexible and easy to use by the user by allowing him to only allocate exactly what he needs and this also means that the overhead for maintaining these resources is not present in the MQTT module. The disadvantages related to this method is that many pointers must be kept and that some may never be used by the user. The primary example is when registering a callback, if a user never wants to register a callback for a given operation there is no way for the compiler or the MQTT module to strip the unused function pointer from the MQTTc connection structure, even if it will never be used.

Also, some code paths could go unused without the compiler and linker being able to know, which could lead to using more code flash space than specifically required. For example, even if a user knows that his application will never receive any message from the broker, the MQTT module has no way of knowing it is the case. Therefore, it will still have all the functions and data to decode anything received from the broker but will never actually use it.

5.5 Test and Validation

Several tests have been made on the stack to confirm it is compliant with the MQTTc stack. First, using a simulation platform running C/OS in a Windows environment, many use cases have been setup and tried with several brokers. This allowed testing the more trivial aspects of the specification like the correct handling of the control packets. Once that was confirmed the same tests and a more complex scenario were tested using a Freescale TWR-K70F120M evaluation board that uses a K70FN1M0 Microcontroller which is a 32-bit ARM Cortex-M4 that runs at up to 120 MHz and has 1 Mbyte of program flash and 128 kbytes of static RAM. This revealed some timing issues as well as limitations when an error was detected and when a connection was closed inadvertently. The more complex scenario also showed how the offered API could be modified so that it was more intuitive for the user. Still, after all these tests, some aspects of the specification had not been tested, particularly error cases or more limit cases. A partial MQTT broker has been written to test these remaining situations. Some problems were found when handling particularly large packets or when a very slow connection was emulated and that every byte of the MQTT packet was received separately one from the other. This combination of tests and testing methods has allowed to confirm that every aspect of the specification had been respected and that the MQTTc module created was fully functional, both in controlled environments as well as when used in a completely genuine context.

5.6 Improvements

In order to improve the MQTT module, it could be possible to add some very specific and simple configuration defines, to change the available features in the MQTT stack. For example, configurations could be added to indicate if an application only wishes to transmit to the broker and never receive. This would enable removing some code paths and data from structures. Another option could be to disable specifically unused callbacks, which would also reduce both RAM usage and the quantity of code space used a bit. Another possible improvement would be to add a polling mechanism if the developer does not wish to use the internal task but instead prefers periodically calling the task's handler from another task. This would reduce performance and increase latency but would allow saving the task stack in memory.

6. Conclusions

The objective of this work was to design and implement a MQTTc software stack fully compliant with the specification and that could be used efficiently and easily in embedded systems. This meant that the module needed to be as light-weight as possible, using as little memory resource as possible and not requiring too much processing power. Also, in order to run on as many platforms as possible, abstraction layer and a good level of configuration was required, to be efficient and usable in trivial applications as well as in very complex ones.

In order to attain this objective, it was decided to keep the API simple and to let the user manage the allocation of the buffers and the data structures, to avoid allocating anything that was not needed and to help re-use the buffers and structures as the application allowed, not as the stack would require. Also, strategically placed abstraction layers have been designed so that the MQTTc module would be able to run with virtually any operating system and network stack.

Several tests were done to confirm that the MQTTc stack is compliant with the requirements of the specification and that it worked correctly. Based on the results obtained during the testing and the analysis made regarding resources usage, portability and configuration, the objective has been met. The MQTTc stack will be able to be used in an embedded systems context, on a variety of processors, including ones with fewer resources. The MQTTc module will also scale well if the platform has more resources and the application is more complex.

Acknowledgement

This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

[1] Aricent Corp., "Nevermind the IoT... Here comes the third wave," [Online]. Available: http://www.embeddeddeveloper.com/documents/aricent_nevermindtheiot_herecomesthethird_wave.pdf.

[2] OASIS MQTT Standard version 3.1.1 (2014) [Online]. Available: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.pdf.

[3] S. Lee, H. Kim, D. K. Hong, and H. Ju, "Correlation analysis of MQTT loss and delay according to QoS level," in *Proceedings of International Conference on Information Networking (ICOIN)*, Bangkok, Thailand, 2013, pp. 714-717.

[4] B. Aziz, "A formal model and analysis of the MQ telemetry transport protocol," in *Proceedings of 2014 9th International Conference on Availability, Reliability and Security (ARES)*, Fribourg, Switzerland, 2014, pp. 59-68.

[5] D. Thangavel, X. Ma, A. Valera, H. X. Tan, and C. Y. Tan, "Performance evaluation of MQTT and CoAP via a common middleware," in *Proceedings of IEEE 9th International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP)*, Singapore, 2014, pp. 1-6.

[6] N. De Caro, W. Colitti, K. Steenhaut, G. Mangino, G. Reali, "Comparison of two lightweight protocols for smartphone-based sensing," in *Proceedings of IEEE 20th Symposium on Communications and Vehicular Technology in the Benelux (SCVT)*, Namur, Belgium, 2013, pp. 1-6.

[7] S. Bandyopadhyay and A. Bhattacharyya, "Lightweight internet protocols for web enablement of sensors using constrained gateway devices," in *Proceedings of International Conference on Computing, Networking and Communications (ICNC)*, San Diego, CA, 2013, pp. 334-340.

[8] C. Zhou and X. Zhang, "Toward the Internet of Things application and management: a practical approach," in *Proceedings of 15th International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, Sydney, Australia, 2014, pp. 1-6.

[9] J. Luzuriaga, M. Perez, P. Boronat, J. Cano, C. Calafate, and P. Manzoni, "A comparative evaluation of AMQP and MQTT protocols over unstable and mobile networks," in *Proceedings of 12th Annual IEEE Consumer Communications and Networking Conference (CCNC)*, Las Vegas, NV, 2015, pp. 931-936.

Olivier Deschambault

Olivier is working as an embedded developer for nearly 5 years. He has varied interests such as software design, parallel systems or machine learning. He holds B.S. and M.S. degrees in electrical engineering from the École de Technologie Supérieure, Montreal, Canada.

Abdelouahed Gherbi

Abdelouahed Gherbi is an associate professor at the software and IT engineering department in the École de Technologie Supérieur (ETS) in Montreal, Canada. He received his Ph.D. degree in computer engineering from Concordia University, Canada. He has spent one year as visiting researcher at the Defense Research and Development Canada DRDC in Valcartier, Quebec. His research interests include mainly model-driven software engineering, modeling and analysis techniques for real-time and critical software systems, software performance, high availability and security.

Christian Légaré

Christian Légaré is a thought leader in developing RTOS solutions for the Internet of the Things. Before joining Micrium in 2002, Légaré spent 22 years in the telecom industry as an executive in such large-scale organizations as Teleglobe Canada and in engineering and R&D startups. He was in charge of an IP (Internet Protocol) certification program at the International Institute of Telecom (IIT) in Montreal, Canada. He is a regular speaker at the Embedded Systems Conferences in Boston and Silicon Valley and has published several articles on embedded systems. Légaré holds B.S.E.E. and M.S.E.E. from the University of Sherbrooke, Quebec, Canada.