University of Bamberg

Professorship for Computer Science

Communication Services, Telecommunication Systems and Computer Networks

Project on

Towards Implementing a MQTT-SN Gateway for Semi-Constrained Devices

Submitted by:

Gabriel Martin Nikol

Supervisor: Prof. Dr. Udo Krieger

Bamberg, October 9, 2017
Summer Term 2017

Contents

LIST OF FIGURES

I. Introduction

"By 2020, IoT technology will be in 95% of electronics for new product designs"[5]. This means Internet of Things (IoT) devices will also build with any kind of consumer hardware technology available. Additionally IoT devices become ubiquitous from small ones like sensors/actuators, smart wearables, smart meters, smart phones, connected vehicles, smart TV, smart furniture[1] and grow to large scale networks over smart homes, smart house infrastructure to smart cities. We need to bring connectivity into these many different device sizes and potential application areas. Not only the size of the hardware but also the cost of single device will becomes interesting too. IoT device vendors are always interested in cheaper devices. This cost reduction becomes important especially when increasing numbers of devices benefit from. But smaller and more constrained IoT devices are not only cheaper. In a lot of cases the smallest, most constrained devices run only with battery or energy harvesting technologies. For example smart wearables, self sustaining sensors/actuators. Often a single chip handles the whole application logic and data transmission. For example a single Bluetooth chip is used in a smart watch to transmit data to a smart phone. The chip also react and shows the user necessary data of the application use case. As a result more energy saving wireless protocols are developed like ZigBee[2], Bluetooth[3] or LoRa[4].

We assume in the future, IoT devices use a lot of different networking technologies which are more energy efficient than TCP/IP based networks. But traditional IoT messaging standards like HTTP and MQTT[6] provides well understood paradigms.

To mix the best of two worlds the well understood publish-subscribe paradigm and energy efficient networking technologies should be used together. We suggest MQTT-SN as a non connection oriented protocol which is easy to implement for most constrained devices. To connect traditional IoT messaging standards and application-layer protocols we provide a prototypical MQTT-SN gateway implementation. During the implementation we want to learn and review which problems we needed and will need to solve for future developments and which problems we already solved.

This project report is structured as follows. In section II we give a definition of different device types and provide a overview over the two most used messaging protocols HTTP, MQTT and MQTT-SN. Then we justify why we think MQTT-SN is well suited for so called constrained devices. Afterwards the MQTT-SN architecture is introduces. In section III we provide the target hardware and software environment for our project, as well as define non-functional requirements and limitations. The MQTT-SN gateway enviroment independent Core Components are introduces in section IV and implement on Linux in section VI. Through a example MQTT-SN client life cycle MQTT-SN procedures we identified gateways procedures. Then we provide a Linux gateway implementation for the Core Components in section VI. This implementation can also used with minor changes on the target hardware and software environment. Next in section VII we implemented a automated unit and regression test environment. Afterwards in section VIII we exchange the UDP socket implementation by a Bluetooth 4.0 Low Energy implementation. Last we conclude our project and provide suggestions for future work.

[1]smart furnitures like smart fridge, toaster and any other you can imagine
[2]http://www.zigbee.org/
[3]https://www.bluetooth.com/
[4]https://www.lora-alliance.org/

II. Related Work

In this section we provide related work and describe why we think a MQTT-SN gateway as IoT middle ware is needed. We start with defining IoT device types in subsection II-A and introduce the application layer IoT messaging standards HTTP, MQTT and MQTT-SN in subsection II-B. Afterwards we compare them and justify why we think MQTT-SN is a good choice as IoT middle ware for constrained IoT devices. Afterwards in subsection II-D we introduce the typical MQTT-SN architecture.

A. IoT Device Types

For the first time the Eclipse IoT Developer Survey asked in 2017 which programming languages and operating systems are not even used in general IoT device but for constrained devices, IoT gateways and IoT clouds, too[6]. One can see that Linux (44.1%) and No OS/Bare-metal (27.%) are the top 2 choices. In the following section we define three categories of device types namely: constrained, semi-constrained and unconstrained devices.

1) Constrained Devices: We define a constrained device similar to so called constrained nodes of RFC7228[7]. Characteristics of a constrained device is limited ROM/Flash, RAM size, limited processing power and a endless power source. For example a Arduino Uno[5] powered by USB-Power bank is a constrained device. Adding a Arduino Ethernet Shield 2[6] makes it to a constrained node after RFC7228. Of course a device can have different energy harvesting modules. A solar cell can refresh the energy source. But even this leads in most cases to a limit device lifetime.

2) Semi-Constrained Devices: A semi-constrained device is the same as a constrained devies defined in subsubsection II-A1. The only difference is that the semi-constrained device is on the grid meaning it has a infinite power source.

3) Unconstrained Devices: A unconstrained device differs from the constrained and semi-constrained device by his greater persistent memory (HDD), RAM size and less or nearly unlimited processing power as well as infinite power source. A unconstrained device can be a Raspberry Pi 3[7], a normal personal computer like our development environment introduced in subsection VI-A or even a non physical computer like a rented Virtual Machine (VM) or Docket container in the cloud[8].

B. IoT Messaging Standards

The Eclipse IoT Developer Survey 2017 shows that the most used messaging standards in the IoT environment are HTTP (60.1%) and MQTT (54.7%)[9][6, p.24]. In the following section we give a short overview over these two standards and MQTT-SN. Afterwards we compare these three standards and justify why MQTT-SN is well suited for constrained devices and as IoT middle ware.

[5]https://store.arduino.cc/arduino-uno-rev3, accessed: 2017-10-08

[6]https://store.arduino.cc/usa/arduino-ethernet-shield-2, accessed: 2017-10-08

[7]https://www.raspberrypi.org/products/raspberry-pi-3-model-b/, accessed: 2017-10-08

[8]https://blog.docker.com/2016/02/containers-as-a-service-caas/, accessed 2017-10-08

[9]The third place is CoAP with 26.7% and all others are blow 20%[6, p.24]

1) HTTP: The Hypertext Transport Protocol (HTTP) is a application-layer protocol[1, p.98] defined in RFC2616[8]. HTTP uses the client-server pattern as shown in Figure 1. The client (PC running Internet Explorer) requests a resource addressed by a Uniform Resource Identifier (URI) via HTTP request message form the server (Server running Apache Web server). The client replies with the resources via a HTTP response message.

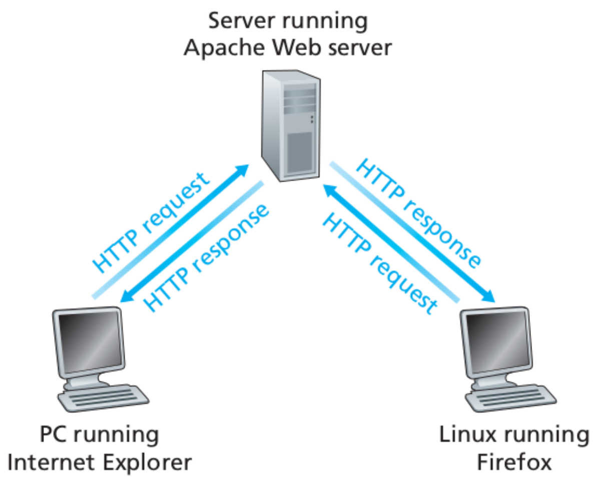


Figure 1: HTTP request-response behaviors[1, p.99]

HTTP has some disadvantages when using it on constrained devices. First HTTP is based on TCP so we have a lot of layer from the application layer to the physical layer adding metadata. Second the whole ISO/OSI layer stack is used and so message can be fragmented for the transmission process. The receiver needs to reassemble these fragmentations.. Both disadvantages consumes lot RAM and energy which is not always available especially for battery powered constrained devices.

2) MQTT: Message Queuing Telemetry Transport (MQTT) is a standardized in the version 3.1.1 by the OASIS Message Queuing Telemetry Transport (MQTT) TC Technical Committee[9]. "It is light weight, open, simple, and designed so as to be easy to implement"[9]. "The protocol runs over TCP/IP, or over other network protocols that provide ordered, lossless, bi-directional connections"[9]. "The publish/subscribe pattern (pub/sub) is an alternative to the traditional client-server model, where a client communicates directly with an endpoint"[3].

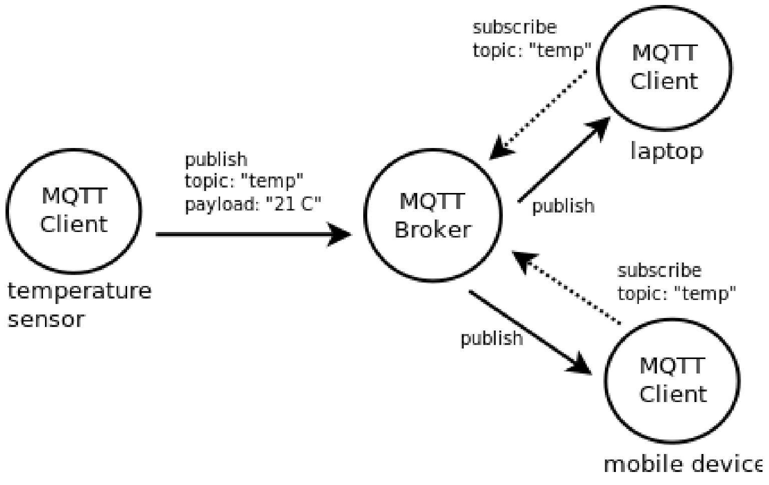


Figure 2: MQTT publish-subscribe example[3]

In Figure 2 one can see simple-publish subscribe example. The three MQTT Client namely temperature sensor, laptop and mobile device are connected to the MQTT broker. First laptop and mobile device subscribe to the topic "temp". Then the temperature sensor publishes a message to the topic "temp" with the payload "21 C". Both subscribed clients laptop and mobile device receive now on the payload "21 C" on the topic "temp". This provide a data centric data model in contrast to the communication centric client-server model of HTTP.

Further MQTT provides three Qualities of Service (QoS) methods shown in Figure 3. The different QoS behave like followed:

- QoS 0 (at most once) - Sends only the publish message. It has the same reliability of the underlying TCP layer.
- QoS 1 (at least once) - Sends the publish message and awaits a puback as reply. It is ensured that the message is delivered at least once.
- QoS 2 (exactly once) - Sends the publish message and expects the messageflow of pubrec, pubrel and pubcomp. It is ensured that the message is delivered exactly once.

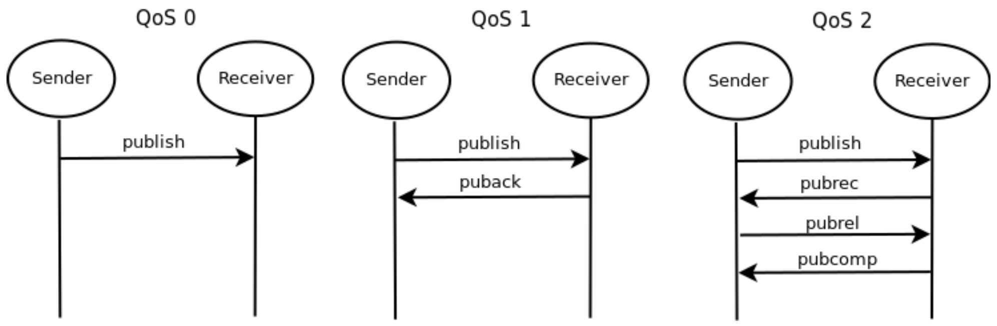
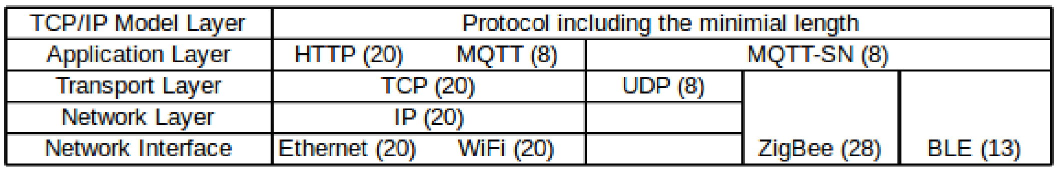


Figure 3: MQTT Quality of Service (QoS) methods[2]

3) MQTT-SN: Message Queuing Telemetry Transport for Sensor Networks (MQTT-SN) is a standard from International Business Machines Corporation (IBM). It is designed for low cost Wireless Sensor Network (WSN)[4]. "MQTT-SN can be considered as a version of MQTT"[4, p.4] which we have described in subsubsection II-B2. But MQTT-SN has some advantages in contrast to MQTT especially for constrained devices. MQTT-SN connect procedure uses three different MQTT-SN packets namely connect, will topic and will message. The will topic and will message are optional. MQTT uses one connect packet. MQTT-SN client's publish to so called short topic names which is a two bytes long topic id. These short topic can be registered client wise at the MQTT-SN gateway via a registration procedure. Additionally to short topic names it is able to pre-defined short topic names for each MQTT-SN gateway. These pre-defined topic id are accessible by every client and so even the registration procedure is optional. These three points reduce the message exchanged and so reduce power consumption. MQTT-SN clients can discover MQTT-SN gateways, no fixed encoded MQTT Broker is needed. Also MQTT-SN clients can publish with QoS -1 without even connecting to a gateway. And last the biggest advantage is that MQTT-SN support so called sleeping devices. During this state a MQTT-SN client can be inaccessible via the transmission protocol. The gateway queues incoming MQTT publishes for the sleeping client. During the sleep procedure the device can go to a power down mode and save a lot of energy. Then the MQTT-SN client can wake up and request the buffered publishes[4].

C. Comparing HTTP, MQTT and MQTT-SN

To illustrate why we chose MQTT-SN over HTTP and MQTT we provide a small example comparing the bytes needed by all protocols in the following section. For each layer of the TCP/IP Model we provide the minimum packet sizes needed and compare them as shown in Figure 4. All TCP/IP Model layers under the application-layer summarized are a so called tranmsission protocols networking technology. A overview over network types and technologies for transmission protocols are given by [10], [11], [12] and [13].



TCP/IP Model Layer	Protocol including the minimial length				
Application Layer	HTTP (20)	MQTT (8)	MQTT-SN (8)		
Transport Layer	TCP (20)		UDP (8)		
Network Layer	IP (20)				
Network Interface	Ethernet (20)	WiFi (20)		ZigBee (28)	BLE (13)

Figure 4: TCP/IP Model Layer including the minimal length of the different layers and transmission protocols in bytes

Figure 4 provides a summary of this comparison example in subsubsection II-C1.

1) Application-Layer Comparison of IoT Messaging Standards: First we compare the minimum application-layer packet sizes. For HTTP and MQTT we assume a TCP connection is already established to the server or MQTT Broker. We further assume a MQTT Client is already connected to a MQTT broker. For MQTT-SN we assume a MQTT-SN Client is already connected to a MQTT-SN gateway. Now we compare the minimal packet size of data transfer with value. For HTTP this is sending a HTTP GET request to the server. For MQTT and MQTT-SN it is sending a publish packet with QoS 0. HTTP need a minimum of 20 bytes[10], MQTT needs 8 bytes[11] and MQTT-SN 8 bytes[12], too. When every byte counts MQTT and MQTT-SN outperform HTTP on application-layer.

2) Transmission Protocol Comparison: Next we do not compare HTTP any more because it is already outperformed as shown in subsubsection II-C1. First we assume MQTT uses TCP and MQTT-SN uses UDP. Both are connected to their endpoints via Ethernet. A empty TCP datagram needs 44 bytes and a empty UDP datagram 32 bytes[13] including the IPv4 header without any options. This means MQTT-SN with UDP outperforms MQTT with TCP by 12 bytes. Now we assume MQTT-SN runs on Bluetooth 4.0 BLE and MQTT over WiFi (802.11). A WiFi generic data frame without payload is 34 bytes long[14]. MQTT-SN using BLE's ATT notifications datagram and needs 13 bytes without payload. The difference between WiFi (802.11) and Bluetooth 4.0 is roughly 21 bytes for BLE[14]. Siekkinen et al. [14] strengthens the result of our small example by showing that BLE has a "very energy efficient in terms of number of bytes transferred per Joule spent"[14, p.1].

One counts together all layer we compared, then we need 82 bytes for the combination of MQTT, TCP and WiFi and 21 bytes for the combiantion of MQTT-SN and BLE[15].

[10]https://stackoverflow.com/questions/9233316/what-is-the-smallest-possible-http-and-https-data-request, accessed: 2017-10-08

[11]message type and flags/QoS/Retain (1 byte) + remaining length (1 byte) + topic name length (2 byte) + topic name e.g. "a" (1 byte) + mesage id (2 bytes) + payload (1 byte) = 8 bytes

[12]Lenght (1 byte) + MsgType (1 byte) + Flags (1 bytes) + TopicId (2 bytes) + MsgId (2 bytes) + Data (1 byte) = 8 bytes

[13]https://stackoverflow.com/questions/1846077/size-of-empty-udp-and-tcp-packet, accessed: 2017-10-08

[14]https://www.safaribooksonline.com/library/view/80211-wireless-networks/0596100523/ch04.html#, accessed: 2017-10-08

[15]Note that it cannot be said that MQTT-SN outperforms MQTT by 61 bytes (74%) because the example is too limited. But it shows a strong tendency.

3) Result of Comparison: As we showed MQTT-SN outperforms HTTP and MQTT when we use other transmission protocols than TCP/UDP. But of course using other tranmission protocols like BLE have a lower datarates. This was completely ignore in our comparison because we assume that constrained devices as MQTT-SN client exchange slightly less data then semi-constrained or unconstrained devices. For semi-constrained and unconstrained devices MQTT-SN can still be an opportunity using UDP. They can use the advantages of MQTT-SN compared to MQTT like described in subsubsection II-B3. Overall we say it would be a benefit to implement a MQTT-SN gateway which can run on cheap hardware of a semi-constrained device.

These are some reason we think MQTT-SN is a good choice for a IoT middleware for constrained IoT devices. But most import reason why we think MQTT-SN is well suited for constrained devices is that it support more energy saving options on application-layer level with sleeping clients, that we can use event non TCP/IP based networking technologies and last we keep the same well understood publish-subscribe paradigm for all device types.

D. MQTT-SN Architecture

An example architecture is shown in Figure 5. The MQTT-SN standard know three kind of MQTT-SN components. Client, gateways and forwarder. As shown in Figure 5 MQTT-SN clients are connected to MQTT-SN gateways via the MQTT-SN protocol. The MQTT-SN gateway is connected to a MQTT Broker via MQTT.

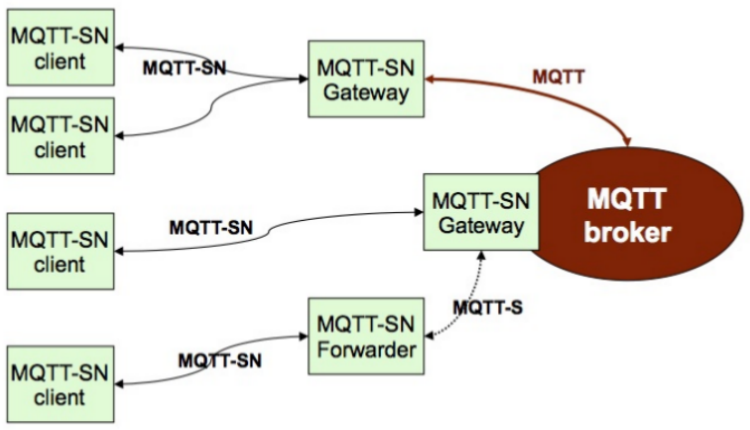


Figure 5: MQTT-SN Architecture[4]

Additionally MQTT-SN client can be connected via a MQTT-SN forwarder encapsulating MQTT-SN packets into MQTT-SN forwarder encapsulation packets. The MQTT-SN encapsulates the received MQTT-SN messages and forward the MQTT-SN forwarder. It encapsulation packets send to the MQTT-SN gateway and expose packets send to the MQTT-SN client. Figure 5 proposes that a MQTT-SN gateway must not necessarily be and independent device. It can also be located at the same device as the MQTT Broker[4].

1) MQTT-SN Gateway Architectures: The MQTT-SN standard propose two kind of gateway architectures: transparent and aggregating gateways. As shown in Figure 6 transparent gateway maintain one MQTT connection for each MQTT-Sn client to the MQTT Broker and so only provide syntax translations. The benefit of the transparent gateway is that it is easier to implement. But large scale MQTT-SN networks may brake the maximum limit of the MQTT Broker's active MQTT connection count. Or the MQTT-SN gatway is a device like a semi-constrained device which cannot maintain a lot MQTT connections. In this cases a aggregating gateway scales better. The aggregating gateway maintains only one MQTT connection to the MQTT Broker and multiple connections to the MQTT clients.

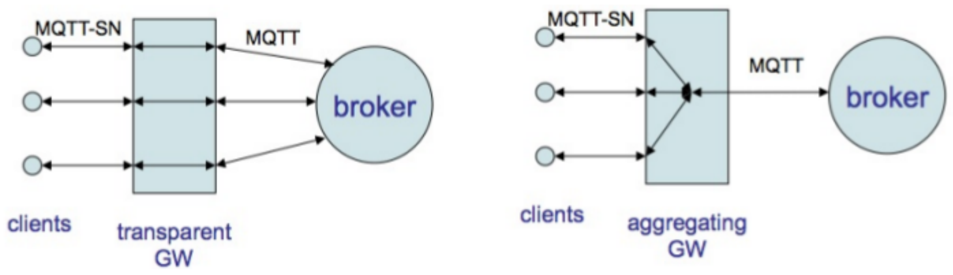


Figure 6: MQTT-SN gateways architecture - Transparent and Aggregating Gateways[4]

2) MQTT-SN Components, Transmission Protocols and Device Types: Figure 7 combines the knowledge of the most used IoT hardware architectures and connectivity protocols from the Eclipse IoT Developer Survey's[6] with the MQTT-SN architecture proposed in the MQTT-SN standard[4]. We added our knowledge of device types. Once can now easily see which device type is used for which purpose. It shows that a typical MQTT-SN client is a constrained device without a operating system or a fixed power source (off the grid). As MQTT-SN transmission protocol we use either WiFi (UDP), ZigBee, Bluetooth 4.0, or LoRa. The MQTT-SN gateway runs on a semi-constrained device connected to a reliable power source (on the grid). As usual MQTT runs over TCP/IP networks and uses WiFi (TCP), Ethernet or maybe 5G in the near future. The MQTT Broker is a unconstrained device like an cloud server or a local Raspberry Pi.

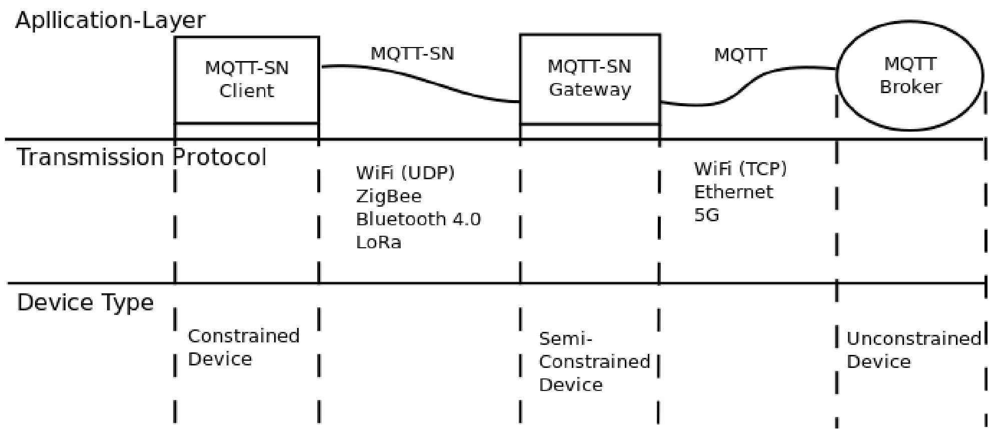


Figure 7: MQTT-SN components the corresponding Device Type and Transmittion Protocols

III. Project Blastoff - Non-Functional Limitations and Requirements

When implementing a MQTT-SN gateway on constrained device we encounter different non-functional requirements and non-functional limitations. The functional requirements are obvious and directly derived from MQTT-SN standard. Of course we need to implement the protocol and message exchanges defined in the standard. These build our functional requirements but are not discusses in detail because they are identified in section V. In this section we define additional non-functional requirements for our implementation by defining a most constrained target platform. This target platform consist of two aspects the hardware environment and the software environment. The hardware environment defines the transmission protocol we can use and the software environment defines which libraries and API we are forced to use. In the following we first describe the target hardware environment then the software environment. Afterwards we discuss the limiting characteristics in comparison to an environment with a Linux operating system. Then we propose the non-functional requirements arisen from these constrains.

A. Target Hardware Environment

For selecting the most constrained device we wanted to pick one of the Boards or Chips supported by the Arduino Environment[16]. This is the reason that we start picking the most constrained device from the official Arduino Boards the Arduino Uno. When we use a constrained device like an Atmel Atmega328P used on an Arduino Uno Boards we have only 2 KB SRAM. This is the greatest restriction[17]. Other micro-controllers like the Atmel Atmega2560 used on the Arduino Mega 2560 Boards have 8 KB of SRAM[18]. But RAM is still the greatest restriction. To compensate the missing RAM, we suggest to use a SD Card for saving data structures.

Selecting a transmission protocol depends on the hardware we can use. For a simplest prototype UDP for the MQTT-SN network and a TCP socket for MQTT can be used. We decide to use UDP because it can run on nearly any Ethernet or WiFi chip supporting a TCP socket, too. So we decided to use the Arduino Ethernet Shield V1[19] as hardware for the transmission protocols. Additionally the Ethernet Shield V1 contains an onboard micro-SD card slot.

We defined an Arduino Mega 2560 Board, Arduino Ethernet Shield V1 and a 4 GB micro-SD card as our minimum target hardware platform. But because we feared that we do not have enough time to optimise the RAM consumption of our implementation we decided to take a chip with more RAM. So we redefined our final target hardware environment to an ESP8266 and a SD Card Slot for a 4 GB SDHC SD card as shown in Figure 8. The only hardware then needed is a simple USB-C to USB-A cable and a power source. The target hardware environment is a semi-constrained device defined in subsubsection II-A2.

[16]https://www.arduino.cc/en/Reference/HomePage, accessed: 2017-10-08
[17]https://store.arduino.cc/arduino-uno-rev3, accessed: 2017-10-08
[18]https://store.arduino.cc/arduino-mega-2560-rev3, accessed: 2017-10-08
[19]https://www.arduino.cc/en/Main/ArduinoEthernetShieldV1, , accessed: 2017-10-08

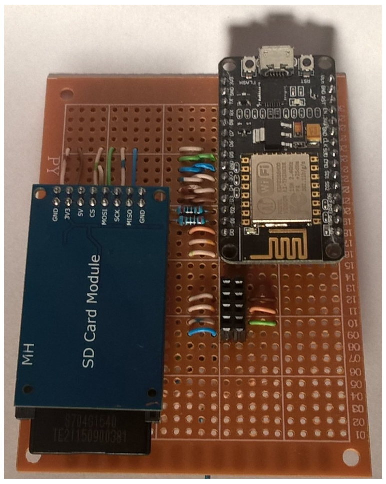


Figure 8: Target hardware environment with a ESP8266 and a SD Card Slot containing a 4 GB SDHC SD card.

B. Target Software Environment

We decided to use the Arduino Environment[20] as target software environment. The Arduino Environment provides a lot of libraries with a unified programing interface and can be used for a lot of different kind of micro controllers and chips.

The ESP8266[21] defined in subsection III-A can be programmed with the Arduino Environment using the ESP8266 core for Arduino firmware[22]. The ESP8266 is already fully working WiFi chip so no external Arduino Shields are necessary. It has 128 KBytes RAM and thus provides a stable platform for a first prototype. The SDHC Card Slot is wired to the ESP8266 by the SPI bus system[23]. It can be use with the SD Library for Arduino [24].

This use of the Arduino Environment gives us the opportunity to run the finished MQTT-SN Gateway on as many different environments as possible. The SD Library for Arduino used by the ESP8266 core for Arduino provides the same API as the SD Library we use with a Arduino Mega 2560 Board and a Ethernet Shield V1. This means we can later, in a future work, optimize the MQTT-SN software for more constrained enviromnents without exchanging the SD Library or rewriting this part. Note that we do not develop the prototype for the target hardware and software environments directly instead we build a prototype on a linux operating system which can be ported to the target environments.

C. Limitations and Requirements

In the following we discuss the limitations of the hardware and software environment. As already mentioned in subsection III-A the RAM size is obviously the greatest limitation. Additionally on micro controllers heap allocation leads to heap fragmentation. As a consequence of the RAM limitation heap allocation should never be used. Second when programming bare metal hardware with the Arduino Enviromnent we need to use C/C++

[20]https://www.arduino.cc/en/Reference/HomePage, accessed: 2017-10-08
[21]https://opencircuit.nl/ProductInfo/1000088/NodeMCU-v2.pdf, accessed: 2017-10-08
[22]https://github.com/esp8266/Arduino, accessed: 2017-10-08
[23]https://www.arduino.cc/en/Reference/SPI, accessed: 2017-10-08
[24]https://github.com/esp8266/Arduino/tree/master/libraries/SD, accessed: 2017-10-08

as programming language. There is also a hidden limitation because the ESP8266 core for Arduino (as well as the Arduino Environment) does not provide all ISO C standard libraries functions like the siscanf() function[25]. Other limitations are that we can only port the finished MQTT-SN gateway to hardware supporting the Arduino Environment. But there is no alternative wich has a similiar hardware support than the Arduino Environment. If implementors want to support their hardware they at least need to map these functionalities to their hardware environment. We decided to implement a aggregating gateway like described in subsubsection II-D1 because our hardware environment is not able to maintain more than e few TCP connections at once. For a first prototype especially for a good IDE and automatic testing we implement an Linuxx.

The resulting requirements are simple, stack only allocation of resources, reduce the stack size as much as possible, map environment functionality to Arduino Libraries or implement it specifically for your environment for example Linuxx.

1) Prototype Limitations: We do not implement the full MQTT-SN standard feature set into our prototype. We decided not to implement receive and send publishes with QoS 2, will message update functionality. The maximum message length is defined to 255 bytes.

IV. Core Gateway Components

In the following section we will propose the Core Components of the MQTT-SN gateway[26]. They are developed after the functional requirements derived from the MQTT-SN standard as so called procedures in section V and the non-functional requirement discussed in section III. The main Task of the Core Components is providing either hardware and software environment independent implementations or interface for functionality which cannot be implemented environment independent. These interface need to be implemented by a vendor or for a operating system as we do in section VI. In this section we first give an overview over the different components in this subproject as well as discuss general design decisions. Then we propose behavior procedures, used during the lifetime of a MQTT-SN client and discuss how we implement them. These behaviour procedures are functional reqierements derived from the MQTT-SN standard. At the end we consider pros and contras of our design.

A. Components Overview

The Core Components consists as shown in Figure 9 of two environment independent classes and five environment depend interfaces. The classes are namely the MqttSnMessageHandler and the Core. The reason that the Core class is divided into CoreInterface and CoreImpl is only that a C++ interface is easier mockable by GoogleMock[27]. In the following Core means both the CoreInterface and CoreImpl as one finished class. The interfaces are the SocketInterface, MqttMessageHandler, System, LoggerInterface and PersistentInterface. In the following we describe the tasks, functionalities, to give implementors of these components an idea.

[25]https://github.com/esp8266/Arduino/issues/404, accessed: 2017-10-08
[26]https://github.com/S3ler/core-mqtt-sn-gateway
[27]https://github.com/google/googletest, commit: 7b6561c56e353100aca8458d7bc49c4e0119bae8

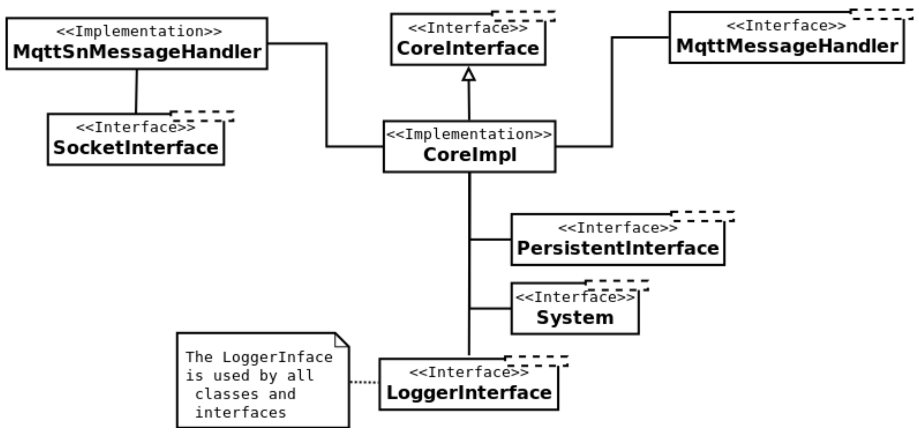


Figure 9: Classes and Interface of the gateway core components

1) SocketInterface: The SocketInterface shall provide an abstraction from the transmission protocol. It only consists of a few method, namely the begin() method to start the component, setter für the Logger and the MQTTSNMessageHandler which shall receive full datagrams. The loop() function shall call MQTTSNMessageHandler's receiveData() function. When data is available it should give with the sender's device_address to the MQTTSNMessageHandler. The device_address is abstract address (implemented as a simple array of bytes) to be translated by the SocketInterface from and the used transmission protocol. This way of addressing keeps the MQTT-SN gateway core component independently from any special networking addressing schema.

2) MqttSnMessageHandler class: The MqttSnMessageHandle has two tasks: parsing incoming datagram and create outgoing datagrams. It provide a set of method that create MQTT-SN packets of all types. These MQTT-SN packets are declared as structs in the mqttsn_messages header file. It is one of the two platform independent implemented classes and uses device_address as identification of MQTT-SN clients. The MqttSnMessageHandler starts parsing a datagram when the receiveData() function is called. When parsing a well-formed MQTT-SN packet the packet's options are identified and the packet is handles by calling the correct function of the Core class. After calling a Core's class function the return value is used to create a proper answer message.

The MQTT-SN standard says that a "server or gateway may also sends a DISCONNECT to a client, e.g. in case a gateway, due to an error, cannot map a received message to a client"[http://mqtt.org/new/wp-content/uploads/2009/06/MQTT-SN_spec_v1.2.pdf]. When a ill-formed MQTT-SN packet is received the MqttSnMessageHandler shall set the sending MQTT-SN client as disconncted via the Core class. This is the equivalent to closing a connection in connection oriented protocol.

3) Core class: The Core class mediates the different calls from the platform depend interfaces and the MqttSnMessageHandler. It performs regular tasks, offering configuration, changing MQTT-SN clients statuses and processing complex behaviour. The regular tasks are MQTT-SN client management like checking frequently the timeout of connected MQTT-Clients and sending out queued MQTT publishes for clients. This is done during the loop() function calls. Other regular task like sending out the MQTT-SN advertisement packet are executed too. It also provides a unified way of getting platform independent configurations. For example the MqttMessageHandler requests the configuration for the MQTT broker and MQTT client not directly from the PersistentInterface and instead getting these value from the Core class. The Core class then requests the PersistentInface to load these data. The MqttSnMessageHandler calls functions on the Core class depend on the received MQTT-SN packets. Some of these packet types change the statuses of a client over the PersistentInterface. Every procedure including more then one Core Component's class is executed and managed by the Core class.

4) MqttMessageHandlerInterface: The MqttMessageHandlerInterface on the other side of the MqttSnMessageHandler manages the MQTT Client's used to connect to the MQTT broker. On different platform we need to use differen MQTT Client implementations thus we need this interface as abstraction. On Linux and the Arduino Environment we can use the Eclipse Paho MQTT C/C++ client for Embedded platforms[28]. The MqttMessage-Handler's tasks is to provide functionality to connect, subscribe, publish to the MQTT broker During a loop() function call receive MQTT publishes shall be forwarded to the Core class.

5) LoggerInterface: The PersistentInterface is only used by the Core class. Here the status of the MQTT-SN gateway, MQTT-SN clients and configurations are accessed. The PersistentInterface saves and retrieves all information living longer than one loop() call of the SocketInterface, the MqttMessageHandlerInterface or the Core class. For example configurations are loaded from configurations files or a new received MQTT publish is saved for a subscribed client. We can say the functionality is similar to the well knwo Create Read Update Delete (CRUD) functionality of databases. Note that the used persistent technology is not specified by us.

6) LoggerInterface: The LoggerInterface shall provide functionality to log message with different log levels. The Logger is designed to start one log message when start_log() function is called. Then multiple messages as characters can be appended until start_log() function is called again. All characters give to the logger, between two start_log() function calls, are written as a single log message. A single log message and shown in one line with a time stamp when start_log() was called for the first time. The LoggerInterface is directly accessed by all classes and interface.

7) System: The System interface provides two functionalities: a frequent heartbeat, relative time measurement and a hard exit() function. The System interface is only accessed by the Core class. It is used to have a timer and a relative time to update the time out values of the MQTT-SN clients. Also MQTT-SN advertisment packets are send out regularly using a instance of the System interface.

V. Gateway Procedures

In the following we describe the so called procedures we identified as functional requirements from the MQTT-SN standard. To understand how we named and investigated the procedure we will first describe the prototypical life cycle of a MQTT-SN client. We divide the procedure into three types: perform frequently or triggered by a timer, triggered on receiving a MQTT-SN packet or triggered on receiving a MQTT packet.

A. MQTT-SN Client Life Cycle

We assume the MQTT-SN client is constrained device. We propose the following typical life cycle.

[28]https://github.com/eclipse/paho.mqtt.embedded-c

- find a MQTT-SN Gateway via advertisement or searching a gateway
- connect to the MQTT-SN gateway with a will message
- register topics
- subscribe to topics
- send and receive publishes
- unsubscribe from topics
- sleep
- wake up and collect queued publishes
- sleep and wake up frequently
- power source is empty - will message is published.

The life cycle was created by investigation the MQTT-SN standard followed by creating the tasks.

B. Advertisment Procedure

The MQTT-Sn gateway needs to send out MQTT-SN Advertisment packets frequently. For this timer triggered procedure we us an instance of the System class. The heartbeat (time value) is set to the MQTT-SN advertisement duration value obtained from the PersistentInterface. In every loop() call of the Core class we check if the heartbeat elapsed (timer timed out). If this is the case we send an MQTT-SN advertisement packet over the MQTTSnMessageHandler.

C. Search Gateway Procedure

When a MQTT-SN client broadcasts for a MQTT-SN Gateway and sending a MQTT-SN search gateway packet. The MQTTSnMessageHandler receive the search gateway packet and forwards this to the Core class. The Core class obtains the gateway id from the PersistentInterface. Then the request is answered with a MQTT-SN gatewy info packet using the MQTTSnMessageHandler.

D. Connect Procedure

The connect procedure starts when a MQTT-SN connect packet is received by the MQTTSn-MessageHandler. There are four different version of this prodecure to handle: connect without will message and without clean session connect without will message and with clean session connect with will message and without clean session connect with will message and with clean session

Connecting without will message and without clean session is the simplest. A valid MQTT-SN connect packet is answered by a MQTT-SN connect achnowledge packet. The MQTT-SN Gateway updates the client's connect duration and sets the client to connected. This is processed by the Core class the PersistentInterface and the MqttSnMessageHandler.

Connecting without will message and with clean session is performed as follows. A valid MQTT-SN connect packet is answered by a MQTT-SN connect acknowledge packet. The MQTT-SN Gateway unsubscribes from all subscriptions saved for the client. Then the client's connect duration is updated and his status is set to connected. This is processed by the Core class the PersistentInterface, the MqttSnMessageHandler and the MqttMessageHandler.

Connecting with will message and without clean session is performed as follows. A valid MQTT-SN connect packet is answered by a MQTT-SN will topic request packet and the client is set to connected. The MQTT-SN Gateway then awaits a MQTT-SN will topic packet. Receiving any other packet type (from the client) is interpreted as an protocol violation resulting in a disconnect of the client by the gateway. A valid MQTT-SN will topic packet is answered by a MQTT-SN will message request packet. The will topic is saved for this client. The MQTT-SN Gateway then awaits a MQTT-SN will message packet. Receiving any other packet type (from the client) is interpreted as an protocol violation resulting in a disconnect of the client by the gateway. After receiving the MQTT-SN will message packet and the gateway saves the will message for this client. Then the gateway answers with a MQTT-SN connect acknowledge packet, awaits a MQTT-SN ping request packet, and the client is successfully connected.

By accepting only the awaited MQTT-SN packet types protocol violations are identified. Protocol violations result in disconnecting the client from the gateway by sending a MQTT-SN disconnect message and set the client to disconnected.

Connecting with will message and with clean session will first perform the version when connecting without will message and with clean session, then the procedure as connecting with will message and without clean session.

E. Register Procedure

A MQTT-SN client can register topics (long topic names) via MQTT-SN register message and get a short topic name represented as a two byte topic id back. When a MQTT-SN register packet is received by the MqttSnMesageHandler the Core is given the long topic name. The Core iterates through the already registered topic list for this client and appends the new topic name if it does not exist. If it already exist the existing topic id is taken. If it does not exist the new topic is the maximum count of topics registered for this client now. Then Core uses the MqttSnMesageHandler and answers with a MQTT-SN register acknowledge packet containing the new topic id.

F. Subscribing and Unsubscribing Procedure

The subscribing and unsubscribing form topic is managed by a so called internal subscription manangement process. We first introduce this process and then the two procedures.

1) Internal Subscription Management Process: In general for subscribing and unsubscribing procedure the gateway manages a counter for each topic at least one client is subscripted to. If the so called subscription counter to a topic is at least one, the gateway is subscribed to the topic at the MQTT Broker. This means if the client is the first one subscribing to a topic, the Core class will subscribe to this topic over the MQTTMessageHandlerInterface and set the subscription counter of this topic to one using the PersistentInterface. Later if other clients subscribe to this topic too, the subscription counter is incremented. When a client unsubscribes from one of his topics, the subscription counter is decremented and checked if it reaches zero. If it reaches zero the topic is deleted and the Core class will unsubscribe over the MQTTMessageHandlerInterface from the topic.

2) Subscribing procedure: The subscription procedure has three different versions: subscribing by topic name, pre-defined topic id or short topic name.

The subscribing procedure is started when a MQTT-SN subscribtion packet is received by the MQTTMessageHandlerInterface.

Subscribing by topic name start internally the same topic name registration like in the register procedure. Then the internal subscription management process is executed. Afterwards it is answered by a MQTT-SN subscription acknowledge packet with accepted as return code and the new topic id.

Subscribing by pre-defind topic id will first lookup the topic in the pre-defined topic list and the found topic name is registered via register procedure. The topic list is accessed via the PersistentInterface. Then the internal subscription management process is executed. Afterwards it is answered by a MQTT-SN subscription acknowledge packet with accepted as return code and the new topic id.

Subscribing by short topic name will lookup the short topic name in the client's registered topic list. the client's registered topic list is accessed via the PersistentInterface. Then the internal subscription management process is executed. Afterwards it is answered by a MQTT-SN subscription acknowledge packet with accepted as return code and the topic id.

3) Unsubscribing Procedure: The unsubscribing procedure is started when a MQTT-SN unsubscribtion packet is received by the MQTTMessageHandlerInterface.

The unsubscription procedure has three different versions: subscribing by topic name, pre-defined topic id or short topic name. For all procedures that processing is nearly similiar. First the topic name is looked up in the client's short topic name list or the pre-defined topic list. Then the internal subscription management process is executed. Last the gateway replies with a MQTT-SN unsubscribe acknowledge packet. This packet has no return code.

G. Message Queueing Procedure

The message queuing procedure handles MQTT publishes from the MQTT broker. Incoming MQTT publishes from the MQTT Client, given the MqttMessageHandlerInterface are forwarded to the Core class. The Core then saves the MQTT publish including the publish options for each MQTT-SN Client subscribed to the MQTT publish's topic persistently (using the PersistentInterface) if client's status is either active, awake or asleep. The Core does not save publishes for lost or disconnected clients. We always use the lowest QoS comparing the QoS of the incoming MQTT publish and the client subscription QoS to the topic.

This implements a real message queuing, each publish is first saved by the PersistentInterface before it will be published to the MQTT-SN client later. The PersistentInterface needs to saved incoming publishes for the each client and keep the order of incoming MQTT publishes.

After the publishes are saved they are retrieved by the Core again during the loop() function. For each loop() call the Core iterates over all MQTT-SN clients. For each awake or active client a MQTT publish is retrieved from the PersistentInterface. If we await from the client a MQTT-SN ping request packet we are able to send the MQTT publish as an MQTT-SN publish packet. Depend on the QoS of the publish we either await a MQTT-SN ping request (QoS 0) or MQTT-SN publish acknowledge packet (QoS 1).

Before Sending a MQTT-SN publish packet we check if the topic id is known by the client. If the topic id of the topic is not know by the client then we call the register procedure. Then we first send a MQTT-SN register packet with a new topic id. Then we expect the client to send a MQTT-SN register acknowledge packet back before sending any other message. After receiving this we send the queued MQTT-SN publish packet during the next loop() call.

We decided to keep only one message in flight between the MQTT-SN Gateway and the MQTT-SN Client This is why we manage the MQTT-SN packets awaited from the client. So additional to the MQTT-SN clients status a second status is handled by the PersistentInterface. This is necessary to detect failures in the protocol and prevents both clients and gateway from running into an illegal state. When no message is in flight between client and gateway we can always expect a MQTT-SN ping request from the client, because this checks the availability of the Gateway and is used in the ping procedure. We can send any MQTT-SN packet type to the client when we expect a MQTT-SN ping request packet.

1) Publish Procedure: The publish procedure handles MQTT-SN publishes from MQTT-SN clients. On receiving a MQTT-SN packet by the MqttSnMessageHandlerInterface the message is parsed and with the QoS, topic id, retain flag and device_address of the sender given to the Core class. Publishing over the MQTT-SN network is only possible via topic id. Then the Core checks in the client's topic list for the topic name. If the topic name is found it is given together with the MQTT-SN publish's payload, options to the MqttMessageHandlerInterface. The MqttMessageHandlerInterface publishes it as a MQTT publish to the MQTT broker. If QoS is one a MQTT-SN publish acknowledge is send back to the MQTT-SN client via the MqttSnMessageHandlerInterface.

If the topic name was not found and the QoS is one. A MQTT-SN publish acknowledge is send back to the MQTT-SN client via the MqttSnMessageHandlerInterface with the return typ rejected invalid topic id. Of course no MQTT publish is send.

2) Ping Procedure: The ping procedure is simple. When a client sends a MQTT-SN ping request packet and is active or awake the Gateway reset the client's timeout value to the duration value during the connect procedure.

3) Disconnect Procedure: When the MqttSnMessageHandlerInterface receives a MQTT-SN disconnect packet the Core class is forwarded this information. The Core class then set the sending client to disconnected using the PersistentInterface. Then a MQTT-SN disconnect packet is send as reply.

When the MqttSnMessageHandlerInterface detect a ill-formed MQTT-SN packet the sending client is set to disconnected and a MQTT-SN disconnect packet is send as a reply.

H. Timeout Procedure

Timeout values in the Core class are handled as follows. For each connected MQTT-SN client a so called timeout value is saved by the PersistentInterface. The timeout value counts down from the value of the duration field during MQTT-SN client's connect procedure. A MQTT-SN client can reset his timeout value by sending a MQTT-SN ping request packet to the MQTT-SN gateway. By Reconnecting and so performing the MQTT-SN connect procedure again a client can update the timeout value.

As a design decision we wanted to reduce updating frequency of the timeout values. Every update perform a potential disk I/O operations which are costly. So MQTT-SN client's timeout value are only updated for example every 10 seconds. To get a time when these 10 seconds pass the Core class uses a instance of the System interface.

The timeout procedure is a timer triggered procedure. We use an instance of a System class as heartbeat timer. In every loop() call of the Core class we check if the heartbeat elapsed (timer timed out). If yes, then we decrease each MQTT-SN client's timeout value by the elapsed time. Depend on the MQTT-SN client's connect duration we either use the 10% timeout tolerance for durations over 60 seconds and 50% for less then 60 seconds [4, p.27]. If a client times out the client's status is set to lost and the will message is send out as a MQTT publish. Afterwards the client's subscriptions are removed (including the Internal subscription management process) and the client is deleted completely.

I. Gateway class

To provide implementors a skeleton on how to initialize the components we offer the Gateway class. As shown in Figure 10 the Gateway class holds a member variable to each Core Component. Classes are directly created by the Gateway's constructor, interface implementation need to be set by the implementor. The begin() function constructs the different Core Component in the correct sequence and then call begin() to each component. The loop() function calls the different loop() function in the order: MqttSn-MessageHandler, MqttMessageHandle, Core.

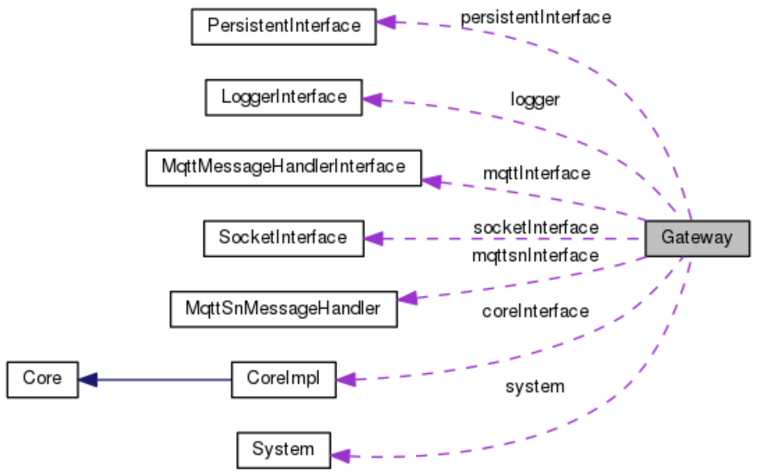


Figure 10: Collaboration Diagram of Gateway class and it's members

J. Discussion

The advantages of section IV and section V are as follows. We can implement the PersistenInterface using only minimal RAM. All platform depend parts are interface and the platform independent classes are without a state. This increase portability and is designed for change. Interface are design so they can run on constrained device and their software environment. No real time clock is needed, only a time giving us the milliseconds since startup of the MQTT-SN gateway.

A disadvantage is the runtime behavior of the Core's loop() function. It is linear to the count of connected MQTT-SN client in most cases. We suggest to provide stress tests to check how this behaves with growing numbers of MQTT-SN client and registrations.

VI. Linux Gateway Implementation

In the following section we will introduce the linux gateway[29] implementation of the MQTT-SN gateway's Core Gateway Components propose in section IV. We provide hardware and software depend implementations of the Core Components interfaces. First we introduce the development environment, then we give an overview over the implemented components and discuss their implementations details. At the end of the section we revisit our implementation and reflect it regarding to portability.

A. Development Environment

As already mentioned in section III we do not directly develop for the target environments. As operating system we use Ubuntu 16.04[30] with a 64 Bit Processor[31], 8 GB RAM[32] and a 240 GB SSD[33]. For a full hardware overview see appendix subsection XI-A. The used integrated development environment (IDE) is Clion2017[34], as build tool we use CMake[35] and gcc[36] with the C++11 standard[15] as compiler.

B. Linux Component Implementations

In this section we describe the interface implemented for the introduced development environment. Figure 11 extends Figure 9 with the implementing classes namely: LinuxUdpSocket, PahoMqttMessagehandler, LinuxLogger, LinuxPersistent, LinuxSystem.

[29]https://github.com/S3ler/linux-mqtt-sn-gateway
[30]Ubuntu 16.04 Kernel 4.10.0-35-generic x86_64 build SMP Wed Sep 13 09:02:42 UTC 2017
[31]64 bit AMD Phenom(tm) II X4 960T Processor
[32]2x 4GiB DIMM DDR3 Synchron 1066 MHz
[33]240GB Crucial_CT240M50
[34]Clion 2017.2.1 Build#CL-172.3544.40, build on August 2, 2017
[35]CMake version 3.6
[36]gcc version 5.4.0 20160609 (Ubuntu 5.4.0-6ubuntu1 16.04.4)

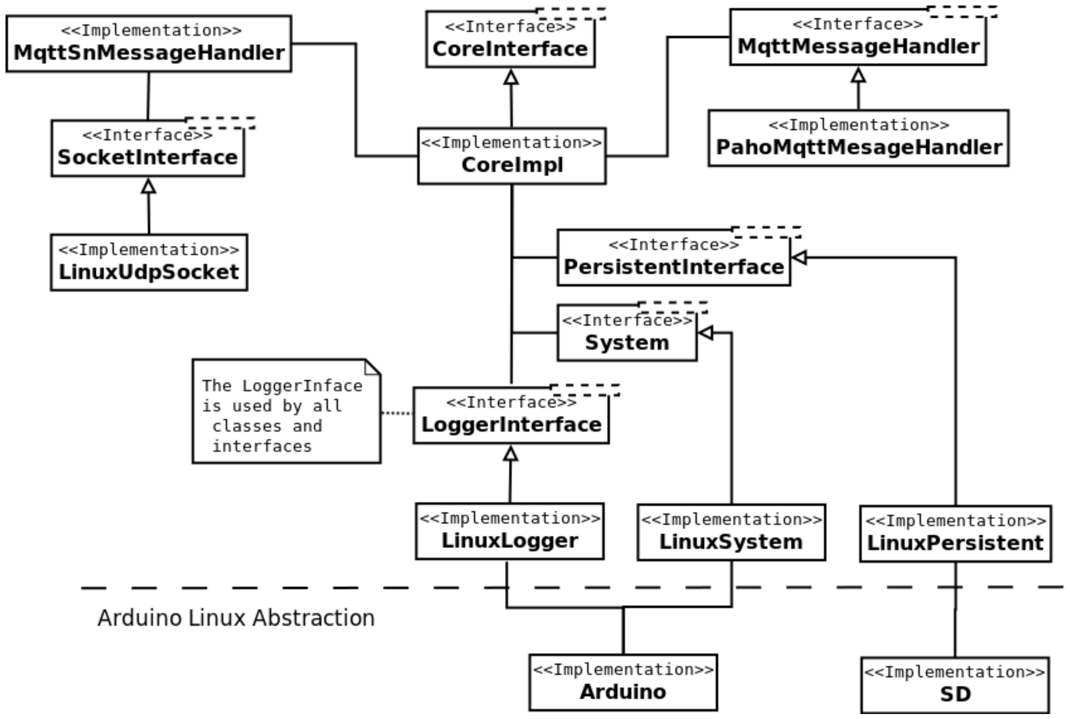


Figure 11: Enviroment depend implementations of interfaces of the linux gateway.

LinuxLogger, LinuxPersistent and LinuxSystem use the Arduino and SD classes from the so called arduino linux abstraction library described in subsection VI-E. In the following we first describe implementation details of implementing classes and then we discuss the result.

C. LinuxUdpSocket

The LinuxUdpSocket implements the SocketInterface and uses UDP as transmission protocol. It implements the function of the SocketInterface as shown in Figure 11. The begin() function of the LinuxUdpSocket initializes two UDP sockets. The first socket is for the message directly addresses to the MQTT-SN gateway (in the following called gateway socket) and the second socket is a IP mutlicast socket (in the followng called broadcast socket) for broadcasting MQTT-SN advertisment, MQTT-SN gateway info packets and receiving MQTT-SN search gateway packets from MQTT-SN clients as well as MQTT-SN advertisment packets from other MQTT-SN Gateways. The gateway socket is fixed coded to port 8888[37] and the broadcast socket uses the fixed coded broadcast group 224.0.0.0 on port 1234. For a prototype this looked easy enough but we suggest to enhance the LinuxUdpSocket class in a later version by loading the ports and broadcast group as configurations during gateway's startup. The maximum message length is fixed coded set to 255 bytes.

Each loop() call checks if the gateway sockets contain a datagram with a short timeout of 300 ms. If a datagram is available the sender IP address including the port number is extracted and converted to a device_address. Then the receiveData() function on the MqttSnMessageHandler is called with the datagram's payload and the device_address. The datagram's payload is interpreted as one MQTT-SN packet. The MQTT-SN standard explicitly not support message fragmentation and reassembly.

[37]we use #define PORT 8888

Sending datagrams is implement by converting the bytes of the given device_address back to an IP including port number and then send a datagram over the gateway or broadcast port back. The getAddress() and getBroadcastAddress() functions of the SocketInterface are implemented by retrieving first the socket's IP address including port, then converting it to a device_address and return it to the callee.

Problems appeared during the automatic the unit testing of the MQTT-SN gateway. The operating system forbids to reuse the broadcast socket's port over multiple processes. We need to add the socket option SO_REUSEADDR fixing this problem. Then multiple processes could access the same port.

D. PahoMqttMessageHandler

The PahoMqttMessageHandler class implements the MqttMessageHandler as shown in Figure 11. The implementation uses an instance of the Paho Embedded MQTT C/C++ Client Libraries[16]. The PahoMqttMessageHandler begin() function initializes the instance of the Paho MQTT C Client on the heap using the linux IPStack[38]. Of course we break with the non-requirement to use stack only memory allocation. but the Paho MQTT C Client is allocated on the heap. This is acceptable as long as only one instance is used during the whole runtime of the gateway and the ownership of the instance is never transfered. The PahoMqttMessageHandler does this. The implementation only maps the MqttMessageHandler functions to Paho Embedded MQTT C/C++ Client functions.

The loop() function is applied by first waiting 300ms for a message on the TCP socket. If bytes are buffered in the socket and it is a MQTT publish message the PahoMqttMessageHandler calls Core's publish() function and the publish will be handled like described in subsection V-G. Because we need to call the Paho MQTT C Client's yield function frequently and we cannot expect the user of the LinuxGateway class to use start_loop() immediately after calling begin(). Thus we decided to connect to the MQTT Broker, when loop() is called on the PahoMqttMessageHandler for the first time. The MQTT Client and MQTT Broker configuration is requested from the Core class which obtains it from the PersistentInterface.

We encountered a problem during programming the PahoMqttMessageHandler with the Paho MQTT C Client. The Paho MQTT C Client is written in C and we had to give a global callback function which will be called when a MQTT publish arrives. We solved it by creating a global pointer to a MqttMessageHandlerInterface interface. This global pointer is initialized during the PahoMqttMessageHandler's begin() function. A second global messageArrived() function calls this global pointer's receive_publish() function. The global messageArrived() function was given to the Paho MQTT C Client as MQTT publish arrived callback.

E. Arduino Linux Abstraction

The purpose of the arduino linux abstraction library is to program the linux gateway to an interface similar available in the Arduino Environment. These are some global functions and the SD Card Library of the Arduino Environment, as already mentioned in subsection III-B and subsection III-C are used. As shown in Figure 11 the Arduino

[38]https://github.com/eclipse/paho.mqtt.embedded-c/blob/master/MQTTClient/src/linux/linux.cpp, accessed: 2017-10-07

class is used by the LinuxLogger and the LinuxSystem. The SD class is used by the LinuxPersistent. In the following we describe the implementation details of the Arduino and SD class. Note that we do not implement all functions of the Arduino Environment only these we needed.

1) Arduino: The Arduino class implements the millis()[39], delay()[40] and various Serial print()[41] and println()[42] functions from the Arduino Environment.

We used std::chrono from the Standard C++ Library[43] to implement the global millis() function. The millis() initializes a timerValue with its first call, similar to starting the Timer in the ArduinoEnvironment. The timerValue is first set to the operating system current time point via now() function[44]. Whenever millis() is called again the difference between the current time pint and the timerValue is returned.

The delay() function used posix nanosleep() function from time.h[45] to suspend the process for the given milliseconds.

Serial output and the different print() and println() functions map these function to std::cout stream of the C++ Standard Library[46] at the standard output stream. The output commands are provide by the internal SerialLinux class.

2) SD: The SD class maps the C++ Standard Library fstream functions[47] and C Standard Library stdio functions[48] to the Arduino SD Library[49]. The arduino SD Library uses two classes namely the SD and the File class. The SD class provide the functions begin(), open(), exists() and remove(). We add the function setRootPath() to set the path to a virtual root file directory on the file system. With the begin() function the SS Pin of the SPI interface is set. This has no functionality in the developlment environment. The open() function creates and returns a File class mapping the C++ Standard Library fstream's file read and write functions to. To remove() a file we use std::remove() function from the C Standard Library. Last the exists() function uses the stat() function[50] to check if a file exists.

Note that es SD Library is built on a sdfatlib. This means not all functionality of modern file system are provided. The most crucial is that it only supports 8.3 names for files[17][18].

F. LinuxLogger

The LinuxLogger implements the LoggerInterface as shown in Figure 11. When using this class on a non ArduinoEnvironment a SerialLinux class from the arduino linux

[39]https://www.arduino.cc/en/Reference/Millis, accessed: 2017-10-07
[40]https://www.arduino.cc/en/Reference/Delay, accessed: 2017-10-07
[41]https://www.arduino.cc/en/Serial/Print, accessed: 2017-10-07
[42]https://www.arduino.cc/en/Serial/Println, accessed: 2017-10-07
[43]http://www.cplusplus.com/reference/chrono/, accessed: 2017-10-07
[44]http://www.cplusplus.com/reference/chrono/system_clock/now/, accessed: 2017-10-07
[45]http://man7.org/linux/man-pages/man2/nanosleep.2.html, accessed: 2017-10-07
[46]http://www.cplusplus.com/reference/iostream/cout/, accessed: 2017-10-07
[47]http://www.cplusplus.com/reference/fstream/fstream/?kw=fstream, accessed: 2017-10-07
[48]http://www.cplusplus.com/reference/cstdio/remove/, accessed: 2017-10-07
[49]https://www.arduino.cc/en/Reference/SD, accessed: 2017-10-07
[50]https://stackoverflow.com/questions/230062/whats-the-best-way-to-check-if-a-file-exists-in-c-cross-platform, accessed: 2017-10-07

abstraction library is used to write log message send them to the standard output. At the beginning of each log message the current milliseconds are written using the global millis() function from the arduino linux abstraction library. We convert the milliseconds from the millis() function to characters using sprintf(). Fortunately sprintf() using integer numbers is supported on the Arduino Environment. As a result the whole LinuxLogger is ready to use on an Arduino Board[51].

G. LinuxPersistent

The LinuxPersistent class implement the PersistentInterface as shown in Figure 11. For using the file system like a local SD card the arduino linux abstraction layer's SD and File classes are used for file manipulations as wells a read/write operations.

The LinuxPersistent class manages some global and client depend files on the file system. Function of the LinuxPersistent interface implementing the PersistentInterface perform CRUD on these fieles.

1) MQTT Configuration: The MQTT Client and MQTT Broker configuration is loaded by the LinuxPersistent class from a simple human readable MQTT.CON which is expected to be located next to the MQTT-SN gateway's binaries. The configuration file is a simple one key value per line file. An example file is provided in the appendix subsection XI-B. The three non optional keys are brokeraddress and brokerport for the MQTT broker to be connected to by the MqttMessageHandler interface, clientid for the client id to be used by the MqttMessageHandler interface. The four optional keys are willtopic, willmessage, willqos and willretain for a MQTT broker connection including a MQTT will message. Note that all four keys are needed to be defined or no will message will be used.

2) MQTT-SN Configuration: The MQTT-SN configration is a simple file called MQTTSN.CON with three key value pairs similiar to subsubsection VI-G1. The keys are gatewayid, timout and advertise where gatewayid is the gateway id used by the MQTT-SN adverstiment and MQTT-NS gateway info packet, time out is how often the gateway updates all MQTT-Sn client's timeout value and advertise is the advertisment duration between sending two MQTT-SN advertisment packets. These value are loaded and used by the Core class.

3) Pre-Defined Topic List: The pre-defined topic list is saved in simple human readable file called TOPICS.PRE. Each line constans a key value pair (space separated). Each line contains topic id as key and topic name as value The TOPICS.PRE file is acces everytime the Core class request a pre-defined topic name.

4) Gateway Subscription List: The gateway subscription list is a binary encoded file named MQTT.SUB. For each topic the gateway is subscribed at the MQTT-Broker it contains a list entry. Such a list entry can be seen in appendix subsubsection XI-C1.

5) Gateway Client List: The gateway client list is a binary encoded file named CLIENT containing a gateway client list entry (see appendix) for each connected or disconnected MQTT-SN client. The gateway client list entry contains values updated when iterating over all connected MQTT-SN client like the timeout value field and the duration field. It also contains a so called file_number. Every device gets a file_number on connecting and four files are created. The *.SUB, *.REG, *.WIL and *.PUB file (where * is replace by the file_number of the client) is a data holder for the client. The *.WIL file contains one

[51]https://playground.arduino.cc/Main/Printf, accessed: 2017-10-07

will message entry (see appendix subsubsection XI-C2). The *.SUB, *.REG contain one list each namely the client's subscription list (see appendix subsubsection XI-C3) and the registration list (see appendix subsubsection XI-C4). The last file, the *.PUB containing the queue MQTT Messages for a client.

H. LinuxSystem

The LinuxSystem implements the System interface as shown in Figure 11. The LinuxSystem measures the heartbeat status by comparing the elapsed time using an internal timer value set when the difference is greater than the heartbeat period. To get a global timer value the class uses the arduino linux abstraction library Arduino millis() function. The sleep() function is implemented by mapping it to the arduino linux abstraction library Arduino delay() function.

I. LinuxGateway

As shown in Figure 12 the LinuxGateway clas inherit from the Gateway class on extends its functionality. In the following section we describe how we adept the Gateway class to our development environment to let the LinuxGateway run.

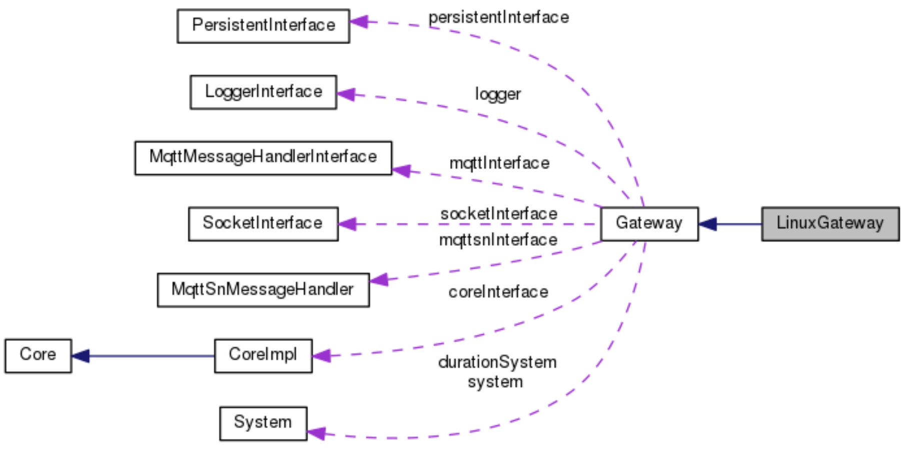


Figure 12: Collaboration diagram of the LinuxGateway class inherit from the Gateway class

First we create instances of the concrete Core Component's interface implementation described in subsection VI-B as member variables of the LinuxGateway class. These implementations are namely: LinuxUdpSocket, PahoMqttMessageHandler, LinuxPersistent, LinuxLogger and LinuxSystem. We added a setRootPath() function to the LinuxGateway mapping the argument to the LinuxPersistent's setRootPath() function. Then we override the begin() function, added the log message "Linux MQTT-SN Gateway version 0.0.1a starting", called from the Gateway class the different setter function with the member variables and last called the Gateway's begin() function. Gateway's begin() function assembles the MQTT-SN gateway class and interface including their setted implementation and calls begin() to all components. On an Arduino we would use Arduino's loop() function[52] to permanently call Gateway's loop() function and call the loop() functions

[52]https://www.arduino.cc/en/Reference/Loop, accessed: 2017-10-07

of all subcomponents. If we create a main function and call the LinuxGateway's loop() function in a while-true-loop we get the same behavior.

We decided to enhance the LinuxGateway and make it embeddable into other application. So we add two new members and three new functions namely start_loop(), dispatch_loop() and stop_loop(). The new members are a std::thread[53] and a std::atomic<bool>[54] variable. The start_loop() function starts the internal LinuxGateway std::thread with the dispatch_loop() function. As long as stop_loop() is not called the thread will continously call LinuxGateway's loop() function.

To show how easily the Linux Gateway can be embedded we create an example main see appendix subsection XI-D.

J. Discussion

As shown in subsection VI-C the implementation of the SocketInterface for UDP as transmittion protocol is straight forward. Of course this is the case because UDP is connectionless like MQTT-SN and we can directly convert one UDP datagram's payload to one MQTT-SN packet. To justify that we also can use connection oriented transmittion protocol we will describe the implementation of BLE as transmission protocol including a connection oriented way of programming in section VIII.

With the arduino linux abstraction library and the use of proper preprocessor commands we can port these Core Component to the target hardware and software environment defined in subsection III-A and subsection III-B.

During the design and development phase of the whole MQTT-SN gateway (including Core Components and linux gateway implementation) we created a test project runnable on the target hardware and software environment defined in subsection III-A. This project is called NodeMCU MQTT-SN gateway[55] and justifies the success of our design.

We benefit from using the Paho Embedded MQTT C/C++ Client Libraries instead of the MQTT C Client for Posix and Windows[19] because now the PahoMqttMessageHandler implementation can be reused with minimal changes on mbed[56], Arduino[57] or FreeRTOS[58][16]. Unfortunately we were not able to port the Paho Embedded MQTT C/C++ Client to the ESP8266 core for Arduino. Instead we implemented the PubSubMqttMessageHandler as MqttMessageHandler using the knolleary PubSubClient[59][20].

In summary for this part project our approach is very suitable for further developments and enhancements.

[53]http://en.cppreference.com/w/cpp/thread/thread, accessed: 2017-10-07
[54]http://en.cppreference.com/w/cpp/atomic/atomic, accessed: 2017-10-07
[55]https://github.com/S3ler/NodeMCU-mqtt-sn-gateway/, commit b0b9c6c9e8f6fee359eaee542a4466302565cee6
[56]https://www.mbed.com/en/, accessed: 2017-10-07
[57]https://www.arduino.cc/, accessed: 2017-10-07
[58]http://www.freertos.org/, accessed: 2017-10-07
[59]https://github.com/knolleary/pubsubclient, commit dddfffbe0c497073d960f3b9f83c8400dc8cad6d

VII. Unit and Regression Testing

Adding more code to support more transmission protocols, platform environments as well as to be sure not breaking existing code we created a test project[60]. By creating compliance test against the MQTT-SN standard we are able to justify the correctness of our implementation, too. In this section we give a short overview over our test environment consisting of the test framework GoogleTest, the used MQTT Broker, the MQTT-SN test client and the MQTT test client. Describing the whole test environment in detail is out of the scope of the project and we advise to take a look at the code directly.

A. Test Environment

As automatic test environment we used the development hardware described in subsection VI-A. All tests are started from inside the Clion IDE. As test and mock framework we use GoogleTest[61] and added it as git submodule directly in out test project. Because we need access to google test main's function argc and argv arguments, we created a global header file named google_test_main_arguments.h allocating two extern variable char **t_argv and int t_argc. Then we changed google test main function to include google_test_main_arguments.h and set the two variables. The variable t_argv[0] contains the execution path of the program, Then we can add configuration files to the correct places needed by the LinuxPersistent class during the test SetUp() function.

1) MQTT Broker: For having access to a MQTT Broker we used a Docker[62] container. The MQTT Broker is Mosquitto[63] using the Docker image jllopis/mosquitto:v1.4.10[64].

2) MQTT-SN Test Client: As shown in Figure 13 we implement a MQTT-SN test client. The MqttSnClient class provides a large amount of function to create well-formed and ill-formed MQTT-SN messages using the structs of the mqttsn_test_message header. With start_loop(), stop_loop() and loo() the MqttSnTestClient is able to run in a own thread and can be used asynchronously inside a single unit test. MQTT-SN messages are send to the tranmission protocol by the FakeSocketInterface, similar to the SocketInterface used in the Core Components. The FakeSocketInterface is very simple. It is only needed to set a MqttSnTestClient where the received datagram are forwarded to and implement a connect(), disconnect(), send() and loop() function. Only the loop() function is not self explaining. During a loop() call the LinuxUdpSocket checks if the gateway or the broadcast socket contain a datagram. If yes, the datagram is forwarded them with the sender's device_address converted from the IP address to the the MqttSnTestClient. The received datagrams are forwarded from the MqttSnTestClient to the MqttSnReceiverInterface. The MqttSnReceiverInterface has a a lot of receive_*() function for each MQTT-SN packet type. It is mocked by the MqttSnReceiverMock. This allows us to use GoogleMock's matcher as a powerful tool for comparing expected and actual MQTT-SN packets.

[60]https://github.com/S3ler/test-mqtt-sn-gateway, commit: cef927611dc5bf3468645342ba0c46e36829106f

[61]https://github.com/google/googletest.git commit 4bab34d2084259cba67f3bfb51217c10d606e175

[62]Docker version 17.05.0-ce, build 89658be

[63]mosquitto version 1.4.10 (build date 2016-10-12 14:51:25+0000)

[64] https://hub.docker.com/r/jllopis/mosquitto/tags/ Tag Name v1.4.10

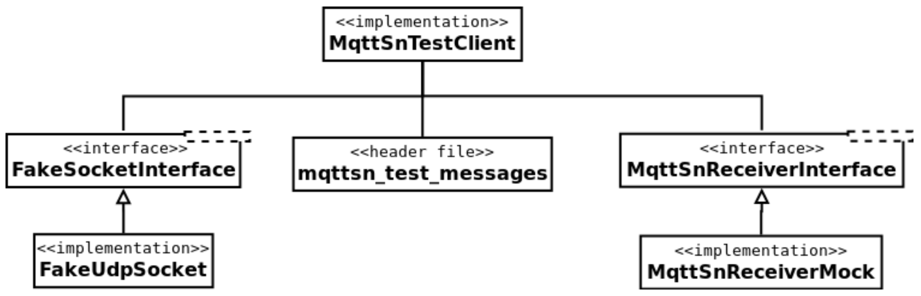


Figure 13: Collaboration diagram of the MQTT-SN test client.

3) MQTT Test Client: As shown on Figure 14 we implement a simple MQTT test client with the same Paho MQTT Embedded C Client already used in subsection VI-D. The main task of the client is to check if certain MQTT publishes and MQTT will message in form of MQTT publishes from the MQTT-SN gateway are received. The PahoMqttTestMessageHandler class is a adapted version of the MqttTestMessageHandler of the Core Components. We added a start_loop(), stop_loop() and loop() function to be able to run the MQTT test client in a thread and so asynchronously from the rest of the unit test. The MqttReceiverInterface and the MqttReceiverMock are necessary to use GoogleMock's functionality to compare the incoming MqttPublishes received by the PahoMqttTestMessageHandler[65].

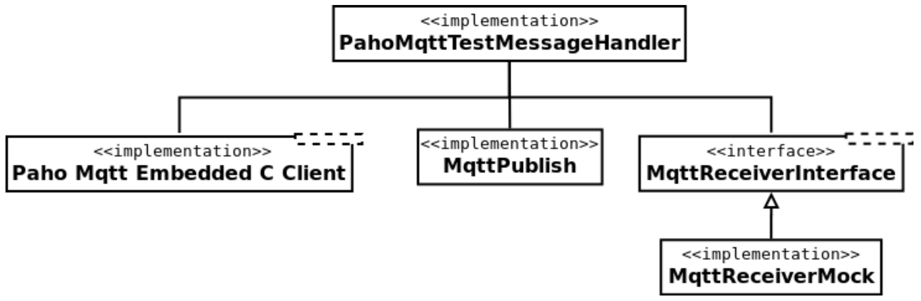


Figure 14: Collaboration diagram of the MQTT test client.

B. Test Overview

We divided out test cases in two group namely compliance and functional tests. Compliance tests stress the interface against well and ill-formed MQTT-SN packets and functional tests check if functionality of the gateway works as standardized or defined by us. Of course something stressing the interface includes testing all functionality - in these cases no (redundant) functional tests are written.

1) Compliance Tests: The compliance tests stresses especially the MqttSnMessageHandler of the Core Components. It shall provide well-formed and ill-formed message of a lot of cases to the gateway's SocketInterface. For every test the expected reply MQTT-SN test packet is created and given to the MqttSnReceiverMock. The GoogleMock framework then will check over some self written macros if each byte of the received and expected packets match. Of course for some tests we need to create a certain status of the MQTT-SN client in the MQTT-SN gateway. For example when we want to test if a client is

[65]This has two reasons:

- with the use of GoogleMock's matcher we have a very powerful tool to comare expected and actual values.
- mocking C++ interface is very easy with GoogleMock - mocking concrete classes very hard.

disconnected by a well-formed disconnect message, we first need to connect the client gratefully to the gateway. These statuses are created in the SetUp() function of GoogleTest. Of course we first created test cases to check if the statuses can be achieved, before we implemented the dependent test cases. For example we first implemented the test cases for MQTT-SN connect packet until we moved on to test MQTT-SN disconnect packets. We oriented ourselves on the procedures of the MQTT-SN client life cycle introduced in subsection V-A. The compliance tests test the following MQTT-SN packet types: search gateway, connect including will topic and will message, register, publish all implemented Qualities of Service, subscribe, unsubscribe, ping request and disconnect.

2) Functional Tests: The functional test shall especially test the functionality of the MQTT-SN gateway implementation against the MQTT-SN standard or the behaviour definitions made by us. We oriented the functional test on the MQTT-SN procedures and MQTT-SN packet exchanges of the MQTT-SN client life cycle introduced in subsection V-A. If test cases are already checked by the compliance tests we do not execute the same tests again. The MQTT-SN client and gateway statuses for each test case are created during the SetUp() function of GoogleTest similiar to subsubsection VII-B1. We implemented test for the following procedures: advertisment, subscription, sleep and will message. The advertisment tests checks if the correct number MQTT-SN advertisment packets are receiving by MQTT-SN test client inside the give time period. The subscription test checks if MQTT-SN publishes are receiving to any kind of topic subscription and QoS methods. The sleep tests check if messages are correctly queued for the MQTT-SN client during the sleep procedure. The will message tests check if will message are receiving for timed out MQTT-SN client after the correct amount of duration. There are no test for connect and publish because they are already tested by the compliance tests.

C. Result of Testing

Our test environment with the MQTT-SN test client's exchangeable transmission protocol using the FakeSocketInterface proposed in subsection VII-A give us the opportunity to reuse the test for multiple transmission protocols. In total we wrote 96 unit tests and at the moment 61 test pass and 35 fail[66]. Investigating the passing and failing test we observed that the most important functionalities are tested and supported. The failing test cases are parse errors of MQTT-SN packet. The MqttSnMessageHandler accepts ill-formed packets without disconnecting the MQTT-SN client. We were not able to implement all test cases until the end and fix all failing tests in time but the most important procedures especially connect, register, subscribe, publish, sleep and will message are tested properly. Unfortunately GoogleTest does not provide a way to measure code coverage of tests out of the box. We recognized too late that it is possible to use gcov[67], , part of the gnu compiler suite, to measure this. Adding more tests, fixing failing tests and measure code coverage is part of a future project.

VIII. Bluetooth Low Energy Socket

We wanted to use a low power transmission protocol and we decided to use Bluetooth 4.0 Low Energy (BLE). One must understand that explaining every detail of the BLESocket

[66]https://github.com/S3ler/mqtt-sn-gateway/blob/master/Test%20Results%20-%20runAll_Tests_including_longWill.html, commit d151dae8d65d8c2a96f1d5c7efe7df8d4f3394fc

[67]http://notsopragmatic.blogspot.de/2012/01/code-coverage-with-gcc.html, accessed 2017-10-08

implementation would be enough material for a whole new project report. This is the reason that we focus on the obstacles and provide a overview so that a basic knowledge of the system and the strategies we used are understood.

To understand Bluetooth one must know that BLE knows two types of devices: central and peripheral devices. Peripheral devices act like server and central devices like clients from a connection oriented perspective[68]. For example central devices connect to peripheral devices and the peripheral device advertises his presence. But from a data perspective saying a server contains the data a client requests, then central devices are the servers and peripherals are the clients. Peripheral device are constrained devices like smart wearables and central devices are unconstrained devices like a Raspberry Pi 3. So our MQTT-SN devices are peripheral and MQTT-SN gateway are a central devices.

First we give an overview over the problems when trying to program BLE central devices using the Arduino Environment in subsection VIII-A. Then we propose our development environment in subsection VIII-B. Afterwads we show how the basic components of the BLESocket in subsection VIII-C.

A. Arduino Environment and BLE

Before we started developing the Linux based Bluetooth 4.0 mqtt-sn gateway we looked for a Arduino Board supporting Bluetooth 4.0 as a Central Device. We found the nRF51822[69] using the Arduino Core for Nordic Semiconductor nRF5 based boards[70] and the ESP32[71] using the Arduino core for ESP32 WiFi chip library[72]. But both projects do not offer a unified or even any non-dirty solution to program the chips as a Bluetooth 4.0 Central Device. There was no Bluetooth 4.0 Central library for the Arduino Environment when the project started in 2017-04-01 until we nearly finished in 2017-09-01, too. We made several attempt and tried out the nRF51822 with some software but creating a own Bluetooth 4.0 Central library for the Arduino Environment is out of the scope of this project. But for the future the ESP32 seems to be the most promising project as it has a lively discussion around this topic[73]. An alternative can be the new released Arduino Primo[74]. But the release date was somehow never really published so we could not expect that the board will become available in time. At the end of the project the board was available. As a result of this whole mess we dropped our attempted to port a Bluetooth 4.0 mqtt-sn gateway to the Arduino Environment and started to program a Linux based version.

B. Development Environment

As development and test environment we use the hardware and software described in subsection VI-A and subsection VII-A. As Bluetooth 4.0 development hardware we use a 7-way hama USB 2.0 Hub with a external power source and four CSL USB Bluetooth Nano-Adapter V4.0 connected to the USB Hub. A closer look of the devices in appendix subsection XI-E. Additionally we use BlueZ version 5.37.

[68]https://devzone.nordicsemi.com/question/232/what-is-a-client-and-server-in-ble/, accessed 2017-10-08
[69]www.waveshare.com/wiki/BLE400, accessed: 2017-10-08
[70]https://github.com/sandeepmistry/arduino-nRF5, accessed: 2017-10-08
[71]espressif.com/en/products/hardware/esp32/overview0, accessed: 2017-10-08
[72]https://github.com/espressif/arduino-esp32, accessed: 2017-10-08
[73]https://github.com/espressif/arduino-esp32/issues/423#issuecomment-331028980, accessed: 2017-10-08
[74]https://store.arduino.cc/arduino-primo, accessed: 2017-10-08

C. BLESocket

For the implementation of the BLESocket class we use the SimpleBluetoothLowEnergySocket which is short discussed in subsubsection VIII-C1. Then we introduce the BLESocket implementation in subsection VIII-D.

1) SimpleBluetoothLowEnergySocket: For implementing the BLESocket we need to write our own SimpleBluetoothLowEnergySocket library[75]. SimpleBluetoothLowEnergySocket support Central Device use only. Explaining this library is out of the scope of this project. But the reasons for creating our own Bluetooth 4.0 Library for Linux are the followings. Little to no tutorials on how to program Bluetooth LE with C/C++ under Linux are available. Programming Bluetooth 4.0 directly against the C hci files of the BlueZ library needs root privileges during runtime, this is not acceptable. Using Bluetooth 4.0 without root privileges is only possible via the DBUS[76] interface. The more obstacles appeared. There is no up-to-date dbus library for C++ available. Then we needed to use D-Bus Glib[77], because the low-level D-Bus library uses a lot more time to unterstand[78]. Then we could not find good tutorials on using Glib, giving us enough information to understand the Glib API Reference and D-Bus at the same time. Last we obtained the BlueZ source code [79] and created a CLion project running the bluetoothctl tool [80]. With this CLion project, the BlueZ documentation, d-feet[81] and countless hours of trial and error we were able finally to create the SimpleBluetoothLowEnergySocket library.

D. BLESocket Implementation

As shown in Figure 15 the BLESocket classes uses four additional classes. When the BLESocket begin() function is called the BLEConnectionAcceptor is started. The BLEConnectionAcceptor scans continously for pheripheral of any kind. It runs in a own thread. When a peripheral is found a new BLEConnection is created by the BLEConnectionAcceptor. Then the BLEConnection's thread is started. The BLEConnectionAcceptor now can create the next BLEConnection without blocking. The BLEConnection started by the BLEConnectionAcceptor tries to connect to the peripheral device. Successfully connected to the peripheral it check for three special GATT[82] services[83].

[75]https://github.com/S3ler/SimpleBluetoothLowEnergySocket, commit: 0ef008057efd6df3b5d10196bc7397f32b18ee2

[76]https://www.freedesktop.org/wiki/Software/dbus/, accessed: 2017-10-08

[77]https://developer.gnome.org/dbus-glib/unstable/, accessed: 2017-10-08

[78]https://dbus.freedesktop.org/doc/api/html/ says: "If you use this low-level API directly, you're signing up for some pain." Accessed: 2017-10-08

[79]http://www.bluez.org/release-of-bluez-5-46/, accessed: 2017-10-08

[80]https://github.com/S3ler/bluetoothctl546_experiments, commit: c25504da0de9742f056094872ab6c31c791cc377

[81]https://apps.ubuntu.com/cat/applications/d-feet/, accessed: 2017-10-08

[82]https://www.bluetooth.com/specifications/gatt/services, accessed: 2017-10-08

[83] three speical BLE GATT service UUIDs

- UUID: "6E400001-B5A3-F393-E0A9-E50E24DCCA9E"
- UUID: "6E400002-B5A3-F393-E0A9-E50E24DCCA9E"
- UUID: "6E400003-B5A3-F393-E0A9-E50E24DCCA9E"

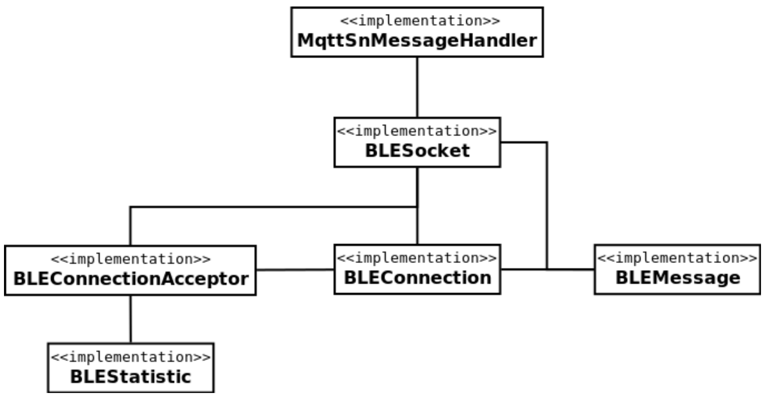


Figure 15: Collaboration diagram of the BLESocket class using the MqttSnMessageHandler

If these services are found the BLEConnection has established a connection to a valid MQTT-SN client via BLE. MQTT-SN message are received via GATT notifications.

Because all BLEConnections run in a own thread and work completely asynchronously from the rest of the gateway, we somehow need to synchronize the interaction between the BLESocket as part of the MQTT-SN gateway loop() thread and the BLEConnections. This is implemented as followed. Each BLEConnection and the BLESocket have access to a shared thread save message queue. This message queue is filled with instance of the BLEMessage classes when a BLEConnection receives a BLE datagram. In each loop() call of the BLESocket one BLEMessage is enqueued and consumed by the MqttSnMessageHandler. This is how we synchronize receiving datagrams from many threaded BLEConnection to one BLESocket.

When the BLESocket shall send data to a MQTT-SN client the device_address given from the MqttSnMessageHandler is converted to a BLE MAC address. Now it iterates over all active BLEConnections and sends the matching one the MQTT-SN packet via GATT service.

One problem occur when we always scan for new BLE peripheral devices. The BLEConnectionAcceptor continously created BLEConnection for not connect able or non MQTT-SN clients (missing GATT services). To prevent this from happening we manage a set of BLEStatistic by the BLEConnectionAcceptor and so BLE peripheral which are too often uncessfully connected are ignored for period of time (for example 10 minutes).

E. Result

We roughly showed how we can add a new transmission protocol, in this case a Bluetooth 4.0 Low Energy to the existing gateway code. The implementation of the connection oriented BLESocket is similar to the coding style of a typical TCP server. Connections are handled in their own thread. We only needed to synchronize the access to the BLEMessage queue and enqueue messages. The BLESocket now behaves like a single threaded socket. The BLEMessage queue bufferes the incoming messages similar to a UDP socket's datagram buffer.

IX. Conclusion

In this project paper we saw how to implement a aggregating MQTT-SN gateway for semi-constrained devices and which design decisions we need to take. We use the Arduino Environment to be able to use a wide range of transmission protocol. With the Core Components independent implementations we provide a basis to distribute the MQTT-SN gateway to nearly any environment. The Linux implementation of the Core Components showed that UDP could be implemented straight forward. The other interfaces are implemented using the arduino linux abstraction library to provide a maximum portability to the Arduino Environment. Then we create a automated unit test and regression test environment to check our implementation. Although not all tests are passing the gateway can be used with real MQTT-SN clients because it offers enough stable functionality. Last we showed how the existing code is to extend when we want to support more transmission protocols. We can say our project was a full success - we learned a lot about using the Arduino Environment and developing portable software for constrained devices using exchangeable implementations.

X. Future Work

There is sill space for further improvements in this project. Some are for example: Implement a MQTT-SN client on a constrained device. Next could be to support more transmittion protocols like ZigBee and LoRa. Support more platforms by implementing examples for windows, MBed or RTOS is also suitablee. Implement the missing MQTT-SN Gateway features like publishing with QoS 2 and updating will topic and will message. Then implement missing functional tests for the missing features. Add more automated testing like stress tests, for understanding were the limits of the design are and measure the used lines of code during the unit tests. Enhance the test environment to even unit test the target hardware directly like a Arduino Board or a ESP8266. Reduce the memory consumption to port the MQTT-SN gateway to Arduino Mega 2560 Board with a Ethernet Shield V1. And last implement the BLESocket on a semi-constrained device using the Arduino Environment.

References

[1] J. F. Kurose and K. W. Ross, *Computer Networking: A Top-Down Approach (6th Edition)*, 6th ed. Pearson, 2012.

[2] H. Ju, H. Kim, S. Lee, and D.-k. Hong, “Correlation analysis of mqtt loss and delay according to qos level,” in *Proceedings of the 2013 International Conference on Information Networking (ICOIN)*, ser. ICOIN ’13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 714–717. [Online]. Available: https://doi.org/10.1109/ICOIN.2013.6496715

[3] dc-square GmbH, “Mqtt essentials part 2: Publish and subscribe,” 2017, accessed: 2017-10-08. [Online]. Available: https://www.hivemq.com/blog/mqtt-essentials-part2-publish-subscribe

[4] H. L. Truong and A. Stanford-Clark, “Mqtt for sensor networks (mqtt-sn) protocol specification version 1.2,” 2013, accessed: 2017-10-08. [Online]. Available: https://de.slideshare.net/IanSkerrett/iot-developer-survey-2017

[5] L. Columbus, “Gartner’s top 10 predictions for it in 2018 and beyond,” 2017, accessed: 2017-10-08. [Online]. Available: https://www.forbes.com/sites/louiscolumbus/2017/10/03/gartners-top-10-predictions-for-it-in-2018-and-beyond/#69ae75f745bb

[6] I. Skerret, “Eclipse iot developer survey 2017 results,” 2017, accessed: 2017-10-08. [Online]. Available: https://de.slideshare.net/IanSkerrett/iot-developer-survey-2017

[7] C. Bormann, M. Ersue, and A. Keranen, “Terminology for constrained-node networks,” Internet Requests for Comments, RFC Editor, RFC 7228, May 2014, http://www.rfc-editor.org/rfc/rfc7228.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc7228.txt

[8] R. T. Fielding, J. Gettys, J. C. Mogul, H. F. Nielsen, L. Masinter, P. J. Leach, and T. Berners-Lee, “Hypertext transfer protocol – http/1.1,” Internet Requests for Comments, RFC Editor, RFC 2616, June 1999, http://www.rfc-editor.org/rfc/rfc2616.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc2616.txt

[9] E. by Andrew Banks and R. Gupta, “Mqtt version 3.1.1,” OASIS Committee Specification Draft 02 / Public Review Draft 02, Tech. Rep., April 2014, http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/csprd02/mqtt-v3.1.1-csprd02.html. [Online]. Available: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/csprd02/mqtt-v3.1.1-csprd02.html

[10] “IEEE Standard for Information Technology- Telecommunications and Information Exchange Between Systems- Local and Metropolitan Area Networks- Specific Requirements Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (WPANs),” Tech. Rep., 2006. [Online]. Available: http://dx.doi.org/10.1109/ieeestd.2006.232110

[11] A. J. Dinusha Rathnayaka, V. M. Podar, and S. J. Kuruppu, *Evaluation of Wireless Home Automation Technologies for Smart Mining Camps in Remote Western Australia.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 109–118. [Online]. Available: https://doi.org/10.1007/978-3-642-27509-8_9

[12] C. Gomez, J. Oller, and J. Paradells, “Overview and evaluation of bluetooth low energy: An emerging low-power wireless technology,” *Sensors*, vol. 12, no. 9, pp. 11 734–11 753, 2012. [Online]. Available: http://www.mdpi.com/1424-8220/12/9/11734

[13] C. Gomez and J. Paradells, “Wireless home automation networks: A survey of architectures and technologies,” vol. 48, pp. 92 – 101, 07 2010.

[14] M. Siekkinen, M. Hiienkari, J. Nurminen, and J. Nieminen, “How low energy is bluetooth low energy? comparative measurements with zigbee/802.15.4,” 04 2012.

[15] ISO, *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. Geneva, Switzerland: International Organization for Standardization, Feb. 2012. [Online]. Available: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_

detail.htm?csnumber=50372
[16] I. Craggs, “Embedded mqtt c/c++ client libraries,” 2017, accessed: 2017-10-07. [Online]. Available: https://www.eclipse.org/paho/clients/c/embedded/
[17] M. Corporation, “How to disable 8.3 file name creation on ntfs partitions,” 2017, accessed: 2017-10-07. [Online]. Available: https://support.microsoft.com/de-de/help/121007/how-to-disable-8-3-file-name-creation-on-ntfs-partitions
[18] A. CC, “Sd library,” 2017, accessed: 2017-10-07. [Online]. Available: https://www.arduino.cc/en/Reference/SD
[19] I. Craggs, “Paho mqtt c client for posix and windows,” 2017, accessed: 2017-10-07. [Online]. Available: https://www.eclipse.org/paho/clients/c/
[20] G. Nikol, “Nodemcu mqtt-sn gateway implementation,” commit b0b9c6c9e8f6fee359eaee542a4466302565cee6, accessed: 2017-10-07. [Online]. Available: https://github.com/S3ler/NodeMCU-mqtt-sn-gateway/tree/master/lib/Implementation

XI. Appendix

A. Development Environment Hardware

```
$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                4
On-line CPU(s) list:   0-3
Thread(s) per core:    1
Core(s) per socket:    4
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 23
Stepping:              10
CPU MHz:               1998.000
BogoMIPS:              5302.48
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              2048K
NUMA node0 CPU(s):     0-3

$ sudo lshw -short
H/W-Pfad                 Gerät      Klasse      Beschreibung
===============================================================
                                     system      To Be Filled By O.E.M. (To Be Filled By O.E.M.)
/0                                   bus         990FX Extreme4
/0/4                                 processor   AMD Phenom(tm) II X4 960T Processor
/0/4/5                               memory      512KiB L1 Cache
/0/4/6                               memory      2MiB L2 Cache
/0/4/7                               memory      6MiB L3 Cache
/0/10                                memory      8GiB Systemspeicher
/0/10/0                              memory      4GiB DIMM DDR3 Synchron 1066 MHz (0,9 ns)
/0/10/1                              memory      DIMMProject-Id-Version: lshwReport-Msgid-Bugs-To: I
/0/10/2                              memory      4GiB DIMM DDR3 Synchron 1066 MHz (0,9 ns)
```

```
/0/10/3                                 memory      DIMMProject-Id-Version: lshwReport-Msgid-Bugs-To: 
/0/0                                    memory      64KiB BIOS
/0/100                                  bridge      RD890 PCI to PCI bridge (external gfx0 port B)
/0/100/2                                bridge      RD890 PCI to PCI bridge (PCI express gpp port B)
/0/100/2/0                              display     GK104 [GeForce GTX 760]
/0/100/2/0.1                            multimedia  GK104 HDMI Audio Controller
/0/100/4                                bridge      RD890 PCI to PCI bridge (PCI express gpp port D)
/0/100/4/0                              storage     88SE912x SATA 6Gb/s Controller [IDE mode]
/0/100/5                                bridge      RD890 PCI to PCI bridge (PCI express gpp port E)
/0/100/5/0                              bus         EJ168 USB 3.0 Host Controller
/0/100/5/0/0            usb8            bus         xHCI Host Controller
/0/100/5/0/0/2                          bus         USB2.0 Hub
/0/100/5/0/1            usb9            bus         xHCI Host Controller
/0/100/5/0/1/2                          bus         USB3.0 Hub
/0/100/5/0/1/2/3        scsi11          storage     USB Storage
/0/100/5/0/1/2/3/0.0.0  /dev/sdc        disk        STORAGE DEVICE
/0/100/5/0/1/2/3/0.0.0/0 /dev/sdc       disk
/0/100/6                                bridge      RD890 PCI to PCI bridge (PCI express gpp port F)
/0/100/6/0                              bus         EJ168 USB 3.0 Host Controller
/0/100/6/0/0            usb10           bus         xHCI Host Controller
/0/100/6/0/1            usb11           bus         xHCI Host Controller
/0/100/9                                bridge      RD890 PCI to PCI bridge (PCI express gpp port H)
/0/100/9/0                              bus         VT6315 Series Firewire Controller
/0/100/9/0.1                            storage     VT6415 PATA IDE Host Controller
/0/100/a                                bridge      RD890 PCI to PCI bridge (external gfx1 port A)
/0/100/a/0              eth0            network     NetLink BCM57781 Gigabit Ethernet PCIe
/0/100/11                               storage     SB7x0/SB8x0/SB9x0 SATA Controller [IDE mode]
/0/100/12                               bus         SB7x0/SB8x0/SB9x0 USB OHCI0 Controller
/0/100/12/1             usb4            bus         OHCI PCI host controller
/0/100/12/1/1                           input       USBGamingMouse
/0/100/12/1/2                           input       USB Keyboard
/0/100/12.2                             bus         SB7x0/SB8x0/SB9x0 USB EHCI Controller
/0/100/12.2/1           usb1            bus         EHCI Host Controller
/0/100/13                               bus         SB7x0/SB8x0/SB9x0 USB OHCI0 Controller
/0/100/13/1             usb5            bus         OHCI PCI host controller
/0/100/13.2                             bus         SB7x0/SB8x0/SB9x0 USB EHCI Controller
/0/100/13.2/1           usb2            bus         EHCI Host Controller
/0/100/13.2/1/2         scsi10          storage     Intenso Speed Line
/0/100/13.2/1/2/0.0.0   /dev/sdb        disk        15GB SCSI Disk
/0/100/13.2/1/2/0.0.0/1 /dev/sdb1       volume      14GiB Windows NTFS Laufwerk
/0/100/14                               bus         SBx00 SMBus Controller
/0/100/14.1                             storage     SB7x0/SB8x0/SB9x0 IDE Controller
/0/100/14.2                             multimedia  SBx00 Azalia (Intel HDA)
/0/100/14.3                             bridge      SB7x0/SB8x0/SB9x0 LPC host controller
/0/100/14.4                             bridge      SBx00 PCI to PCI Bridge
/0/100/14.5                             bus         SB7x0/SB8x0/SB9x0 USB OHCI2 Controller
/0/100/14.5/1           usb6            bus         OHCI PCI host controller
/0/100/16                               bus         SB7x0/SB8x0/SB9x0 USB OHCI0 Controller
/0/100/16/1             usb7            bus         OHCI PCI host controller
/0/100/16.2                             bus         SB7x0/SB8x0/SB9x0 USB EHCI Controller
/0/100/16.2/1           usb3            bus         EHCI Host Controller
/0/101                                  bridge      Family 10h Processor HyperTransport Configuration
/0/102                                  bridge      Family 10h Processor Address Map
/0/103                                  bridge      Family 10h Processor DRAM Controller
/0/104                                  bridge      Family 10h Processor Miscellaneous Control
/0/105                                  bridge      Family 10h Processor Link Control
/0/1                    scsi2           storage
/0/1/0.0.0              /dev/sda        disk        240GB Crucial_CT240M50
/0/1/0.0.0/1                            volume      511MiB Windows FAT Laufwerk
```

```
/0/1/0.0.0/2             /dev/sda2   volume      215GiB EXT4-Laufwerk
/0/1/0.0.0/3             /dev/sda3   volume      8185MiB Linux swap Laufwerk
/0/2                     scsi7       storage
/0/2/0.0.0               /dev/cdrom  disk        DVDRAM GH24NSB0
```

B. Example MQTT.CON file

```
brokeraddress 127.0.0.1
brokerport 1883
clientid mqtt-sn linux gateway
willtopic /mqtt/sn/gateway
willmessage mqtt-sn linux gateway offline
willqos 1
willretain 0
```

C. LinuxPersistent Dataobjects

1) Gateway Subscription List Entry:

```
struct entry_mqtt_subscription{
uint32_t client_subscription_count;
char topic_name[255];
};
```

2) Client Will Messsage Entry:

```
struct entry_will {
char willtopic[255];
char willmsg[255];
uint8_t willmsg_length;
uint8_t qos;
bool retain;
};
```

3) Client Subscription List Entry:

```
struct entry_subscription{
uint16_t topic_id;
uint8_t qos;
char topic_name[255];
};
```

4) Client Registration List Entry:

```
struct entry_registration{
uint16_t topic_id;
char topic_name[255];
bool known;
};
```

5) Client Publish Queue Entry:

```
struct entry_publish{
```

```
uint8_t msg[255];
uint8_t msg_length;
uint16_t topic_id;
uint8_t qos;
bool retain;
bool dup;
uint16_t msg_id;
uint16_t publish_id;
uint32_t retransmition_timeout; // not used atm
};
```

6) Gateway Client List Entry:

```
struct entry_client {
char client_id[24];
char file_number[9];
device_address client_address;
CLIENT_STATUS client_status;
uint32_t duration; // changed
uint32_t timeout;
uint16_t await_message_id;
message_type await_message;
};
```

D. Example Linux Gateway main.cpp

```
#include <libgen.h>
#include <LinuxGateway.h>

LinuxGateway gateway;

void setup() {
    while (!gateway.begin()) {
        std::cout << "Error starting gateway components" << std::endl;
        exit(1);
    }
    std::cout << "Gateway ready" << std::endl;
}

int main(int argc, char *argv[]) {
    gateway.setRootPath(dirname(argv[0]));
    setup();
    gateway.start_loop();

    while(true) {
        // we need to keep the program alive
    }
}
```

E. Bluetooth 4.0 Developement Enviroment

```
$ lsusb
Bus 003 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 007 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 006 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 002 Device 009: ID 0a12:0001 Cambridge Silicon Radio, Ltd Bluetooth Dongle (HCI mode)
Bus 002 Device 008: ID 0a12:0001 Cambridge Silicon Radio, Ltd Bluetooth Dongle (HCI mode)
Bus 002 Device 007: ID 05e3:0608 Genesys Logic, Inc. Hub
Bus 002 Device 006: ID 0a12:0001 Cambridge Silicon Radio, Ltd Bluetooth Dongle (HCI mode)
Bus 002 Device 005: ID 0a12:0001 Cambridge Silicon Radio, Ltd Bluetooth Dongle (HCI mode)
Bus 002 Device 004: ID 0a12:0001 Cambridge Silicon Radio, Ltd Bluetooth Dongle (HCI mode)
Bus 002 Device 003: ID 05e3:0608 Genesys Logic, Inc. Hub
Bus 002 Device 002: ID 1f75:0917 Innostor Technology Corporation
Bus 002 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 005 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 004 Device 002: ID 04d9:1603 Holtek Semiconductor, Inc. Keyboard
Bus 004 Device 003: ID 258a:0012
Bus 004 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 011 Device 001: ID 1d6b:0003 Linux Foundation 3.0 root hub
Bus 010 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 009 Device 003: ID 05e3:0743 Genesys Logic, Inc. SDXC and microSDXC CardReader
Bus 009 Device 002: ID 05e3:0612 Genesys Logic, Inc.
Bus 009 Device 001: ID 1d6b:0003 Linux Foundation 3.0 root hub
Bus 008 Device 002: ID 05e3:0610 Genesys Logic, Inc. 4-port hub
Bus 008 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
```