Development of a home automation system using the communication protocol MQTT through TCP/IP home network

Juan Jose Ros Gimeno

Supervised by:
Juan Carlos Sanchez Aarnoutse

October 2016

Table of Contents

Table of Contents i

Tables iv

Figures v

1 Introduction 1

1.1 Context . . . 1

1.2 Purpose . . . 1

1.3 Scope . . . 2

1.4 Objective . . . 2

2 Scenarios 3

2.1 Comfort scenario “El patrón” . . . 3

2.2 Leisure and security scenario “El vividor” . . . 3

2.3 Energy efficiency scenario “La verde” . . . 4

3 Characteristics of Home Automation 5

3.1 Architecture . . . 5

3.1.1 Centralized architecture . . . 5

3.1.2 Distributed architecture . . . 5

3.1.3 Hybrid architecture . . . 6

3.2 Components of a home automation system . . . 6

3.2.1 Sensing Devices . . . 6

3.2.2 Controlled Devices . . . 10

3.2.3 User Interfaces . . . 11

3.2.4 Controllers . . . 11

3.2.5 Communication protocol . . . 12

4 Documentation and study of MQTT protocol 13

4.1 Introduction to MQTT . . . 13

4.2 Architecture . . . 14

4.3 History of the protocol . . . 14

4.4 Benefits of MQTT . . . 15

4.4.1 Publish/Subscribe pattern 15
4.4.2 Scalability 16
4.4.3 Space decoupling 16
4.4.4 Time decoupling 16
4.4.5 Synchronization decoupling 16
4.4.6 Authentication 17
4.4.7 Quality of Service (QoS) 18
4.4.8 Last Will and Testament 18
4.4.9 Subject-based filtering 19
4.5 Drawbacks of MQTT 20
4.5.1 Central broker 20
4.5.2 Transport layer protocol 20
4.5.3 Security 21
4.5.4 No TTL (Time-To-Live) on messages 21
4.6 MQTT Control Packet format 21
4.7 Description of MQTT Control Packets 23
4.7.1 CONNECT 23
4.7.2 CONNACK 26
4.7.3 PUBLISH 27
4.7.4 PUBACK 32
4.7.5 PUBREC 32
4.7.6 PUBREL 34
4.7.7 PUBCOMP 35
4.7.8 SUBSCRIBE 35
4.7.9 SUBACK 37
4.7.10 UNSUBSCRIBE 38
4.7.11 UNSUBACK 39
4.7.12 PINGREQ 40
4.7.13 PINGRESP 40
4.7.14 DISCONNECT 41

5 Solution design **43**
5.1 Design overview 43
5.2 Functional design 43
5.2.1 *Musquetteer* nodes 43
5.2.2 *ESPutnik* nodes 43
5.2.3 MQTT convention 44
5.3 Technical design 45
5.3.1 Raspberry Pi 3 45
5.3.2 NodeMCU DEVKIT / D1 mini 47

5.3.3 Additional electronics 53

6 Conclusion **57**

6.1 Results 57

6.2 Known limitations 57

6.3 Future improvements 57

Bibliography **58**

APPENDIX 1 **59**

APPENDIX 2 **60**

Tables

4.1 Structure of an MQTT Control Packet 21
4.2 Fixed header format 22
4.3 Control Packet types 22
4.4 *CONNACK* Return code values 27
4.5 *SUBACK* Return code values 38

5.1 Device properties 44

Figures

3.1 Smart Home . . . 7

4.1 *CONNECT* packet structure . . . 24
4.2 *CONNACK* packet structure . . . 27
4.3 *PUBLISH* packet structure . . . 28
4.4 Choreography of a publication with QoS 0 . . . 29
4.5 Choreography of a publication with QoS 1 . . . 30
4.6 Choreography of a publication with QoS 2 . . . 31
4.7 *PUBACK* packet structure . . . 33
4.8 *PUBREC* packet structure . . . 33
4.9 *PUBREL* packet structure . . . 34
4.10 *PUBCOMP* packet structure . . . 35
4.11 *SUBSCRIBE* packet structure . . . 36
4.12 *SUBACK* packet structure . . . 37
4.13 *UNSUBSCRIBE* packet structure . . . 38
4.14 *UNSUBACK* packet structure . . . 39
4.15 *PINGREQ* packet structure . . . 40
4.16 *PINGRESP* packet structure . . . 41
4.17 *DISCONNECT* packet structure . . . 42

5.1 Raspberry Pi single-board computer . . . 46
5.2 Raspbian logo . . . 47
5.3 Mosquitto logo . . . 48
5.4 NodeMCU development board . . . 48
5.5 D1 Mini development board . . . 49
5.6 Flowchat . . . 52
5.7 NA03-T2S05 power supply module . . . 53
5.8 Solar+Battery powered *ESPutnik* node . . . 54
5.9 Relay module . . . 55
5.10 Optocoupler . . . 55
5.11 Rectifier before the optocoupler . . . 56
5.12 Schmitt trigger . . . 56

1 Introduction

1.1 Context

Home Automation is the residential extension of building automation and involves the control and automation of lighting, HVAC[1], and security. Modern systems generally consist of switches and sensors connected to a central hub sometimes called a *gateway* from which the system is controlled with a user interface that is interacted either with a wall-mounted terminal, mobile phone software, tablet computer or a web interface, often but not always via internet cloud services.

A Smart Home, is a home that incorporates advanced automation systems to provide the inhabitants with sophisticated monitoring and control over the building's functions. Smart homes use home automation technologies to provide home owners with intelligent feedback and information by monitoring many aspects of a home. For example, a smart home's refrigerator may be able to catalogue its contents, suggest menus, recommend healthy alternatives, and order replacements as food is used up. A smart home might even take care of feeding the cat and watering the plants.

Many new homes are being built with the additional wiring and controls which are required to run advanced home automation systems. Retro-fitting (adding smart home technologies to an existing property) a house to make it a smart home is obviously significantly more costly than adding the required technologies to a new home due to the complications of routing wires and placing sensors in appropriate places.

The range of different smart home technologies available is expanding rapidly along with developments in computer controls and sensors. This has inevitably led to compatibility issues and there is therefore a drive to standardise home automation technologies and protocols.

1.2 Purpose

Regardless of the technology, smart homes present some very exciting opportunities to change the way we live and work, and to reduce energy consumption at the same time.

[1]Heating, Ventilation and Air Conditioning

While the cost of living is going up, there is a growing focus to involve technology to lower those prices. With this in mind the Smart Home project allows the user to build and maintain a house that is smart enough to keep energy levels down while providing more automated applications. A smart home will take advantage of its environment and allow seamless control whether the user is present or away.

In the other hand, Home Automation suffers from platform fragmentation and lack of technical standards a situation where the variety of Home Automation devices, in terms of both hardware variations and differences in the software running on them, makes the task of developing applications that work consistently between different inconsistent technology ecosystems hard. Customers may be hesitant to bet their IoT future on a proprietary software or hardware devices that uses proprietary protocols that may fade or become difficult to customize and interconnect.

1.3 Scope

The implementation of multiple hardware components is necessary to provide the functionality that will be further discussed in this document. Behind the complex hardware involved in controlling the smart home project there is a fair amount of software architecture that is responsible for driving the hardware components. Each part of the project is built and designed with a different functionality in mind that will be determined by the use cases.

1.4 Objective

The main objective of the project is to propose and design a solution that is simple and effective to overcome typical retro-fitting scenarios. To develop the project, various scenarios has been proposed with different functional focus on which a home automation deployment will be proposed.

2 Scenarios

Scenarios are proposed based on different user archetypes, each one with different interest and expectations.

2.1 Comfort scenario "El patrón"

A smart home contributes to the comfort in the daily activity of the inhabitants, increasing their quality of life.

- **Lighting Control**. The control of lighting is the usual first port of call for any home automation system.
- **Blinds, curtains, louvres and shutter Control**. Controlling heat entering a room is a great way to keep your house cool in summer but warm in winter.
- **Air-Conditioning and environmental control**. The control of heating and cooling within the smart house is one of the key areas where money can be saved on power bills through more extensive automation control. Sub-Segmentation and occupancy control allows only the area needed to be heated or cooled while other areas fall back to a different temperature making considerable energy savings.
- **Remote Control Systems**. For convenience value, The Smart Home of today tends to use your personal devices to double up as remote controls for your home, Whilst dedicated remote devices are also available to enhance your ability to control your Smart Home, increasingly people are becoming more confortable using just one device that they are very familiar with.

2.2 Leisure and security scenario "El vividor"

The fundamental objective is to avoid risks and domestic accidents and to ensure and protect users and their goods.

- **Access control**. Access Control is controlling who enters your home using an entry system that can use Smart Card readers, radio key tags, Near Field Communications NFC sensing, fingerprint recognition or even facial recognition to

ensure a person should have access.

- **Security system control and integration**. A stand-alone security system for your home is a good investment, but link and integrate it with your Smart Home automation system and you will multiply its value. Security sensors can be used in a dual role as occupancy sensors, keeping your home automation system informed of your whereabouts allowing the home to make adjustments to the environment based on these input.
- **Entertainment System control and integration**. There are many areas of home entertainment that can be enhanced through Smart Home integration. Media storage and distribution, multi-zone audio and video systems, personal music device integration, broadband Internet access and game system integration are just a few of the standard techniques that can be implemented.

2.3 Energy efficiency scenario "La verde"

The mission of home automation in the field of energy management is to meet the household needs at minimum cost.

- **Control Energy Efficiency by Monitoring Energy Use**. Every time you turn something on, you're increasing the amount of electricity that's been consumed. Not to mention the appliances that are on and using power 24 hours a day. With an automated Smart Home you have the ability to keep track of the energy use of all components drawing power.
- **Watering system control**. Having a watering system can be a very water efficient concept and saves you lots of time manually watering your garden.
- **External equipment control**. Why stop inside the house? Control the garage door, car access gate, front gate, extending pergola, ponds, water features through the same automation system giving strict scheduled control of these features and full remote control.

3 Characteristics of Home Automation

3.1 Architecture

The architecture of a home automation system determines how the system components (sensors, actuators, controllers, etc.) are connected. From the commercial point of view we have two different types of automation systems: centralized systems and distributed systems, although there are hybrid systems of both.

3.1.1 Centralized architecture

The control is performed by a central element of which depend on other elements, ie, all signals detection and action are addressed in a single point which is the central unit.

The main advantage of this architecture is its low cost, since the elements do not need routing modules or interfaces for different buses. In addition, installation is simple and can be used a large number of commercial items as the requirements that are required are minimal.

The main drawback is the limited flexibility, which leads us to have costly reconfigurations. In addition, if the central processing unit fails, this would make the whole system stops working.

3.1.2 Distributed architecture

No need of a central element, but each element must have a certain intelligence to know who to send the information they collect (sensors) or what information to use (actuators), ie, each element has sufficient capacity to work autonomously.

The main advantage of this topology is that it has great ease for subsequent reconfiguracon of the system, making it very flexible. On the other hand, allows the possibility of using *plug&play* technology, resulting in installation simplicity.

The biggest drawback is that the additional electronics that is needed in the system elements expensive project. In addition, most of the communication protocols used are proprietary and incomplative amongdifferent vendors.

3.1.3 Hybrid architecture

Systems that use a hybrid architecture, also called decentralized systems, are halfway between centralized and distributed systems, so try to take advantage of both.

3.2 Components of a home automation system

A home automation system is composed of a series of elements that detect a change in status of a variable and transmit this information so other elements can act according to rules or standards set by the user.

3.2.1 Sensing Devices

Sensing devices can report values or states. The signals sent by sensors are converted into data that can be displayed to the user or used by a controller program to make informed decisions based on certain conditions. The signals can be converted at the sensor itself, by an intermediate converter or by the system controller.

3.2.1.1 Light sensors

A light sensor may also be known as a photosensor or photodiode. It is used to monitor the ambient light levels and report them back to your home automation controller. This is often used in conjunction with a motion or presence sensor to switch lights on automatically when someone enters a room - but only if they are needed. They can also be used to ensure that security lights only operate after dark, or make outdoor lighting come on automatically at dusk.

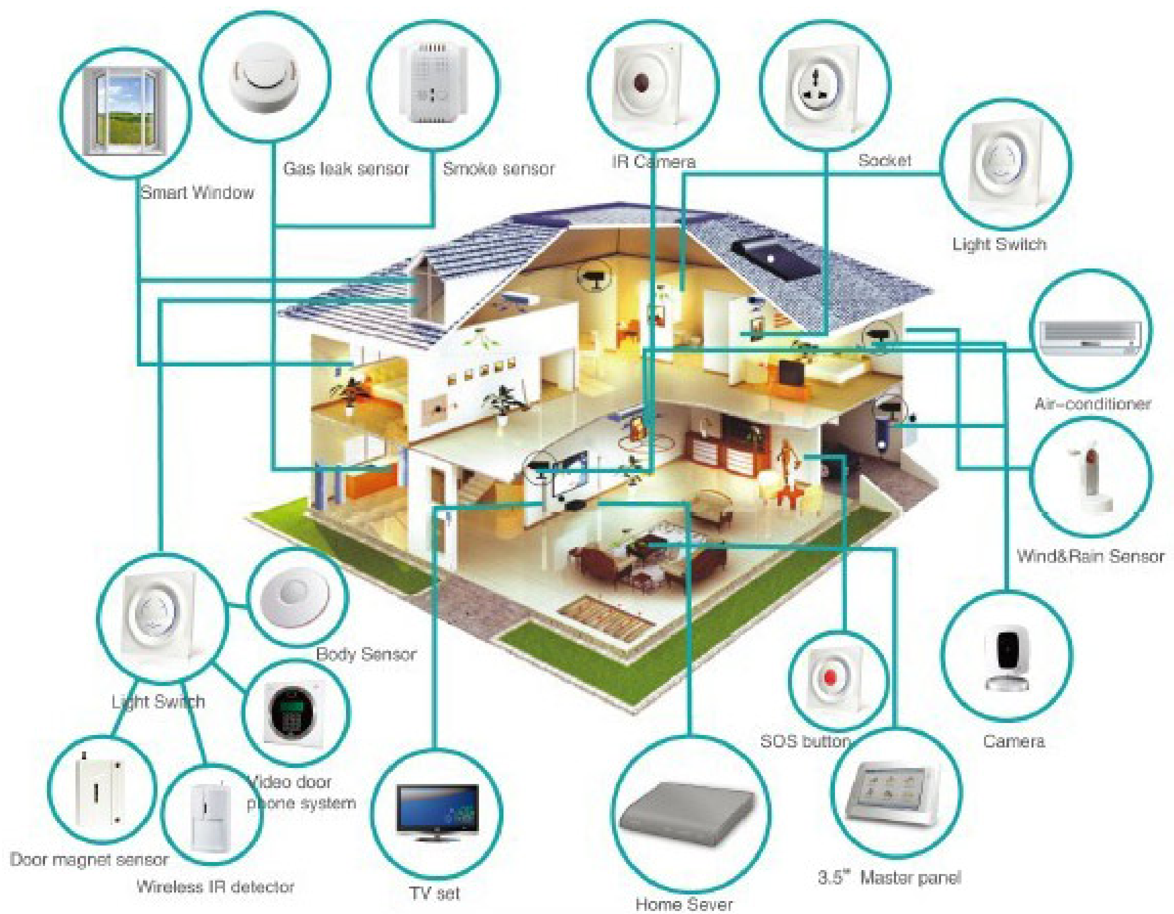


Figure 3.1: Smart Home

3.2.1.2 Temperature and Humidity Sensors

When temperature sensors are used they often come built into a thermostat unit or radiator actuator valve, but there are times when you may like to fit an independent thermometer. Small temperature sensors can easily be embedded into walls to avoid having more unsightly boxes stuck to your walls or ceiling.

Combined with a humidity sensor, they can be used to automatically control air conditioners or de-humidifiers, or even to automatically open windows if a room starts to get too ‘muggy’.

3.2.1.3 Motion and Occupancy Sensors

One of the most popular sensor technologies for domestic automation systems is the motion detector. Often they will be put to multiple uses, such as triggering a burglar alarm if movement is detected when the house is supposed to be empty, or automatically switching lights on and off when you enter or leave a room.

The most common type of motion detector is the PIR[1]. This works by detecting changes in infrared light radiation within its field of vision. When choosing a PIR it is important to make sure that the effective range is sufficient. Although most PIRs will easily be able to cover a room, larger rooms, especially in open plan design buildings may require more than one.

3.2.1.4 Fire Alarm Sensors

There are three main types of fire alarm sensors which are used in domestic properties:

- Optical / Photoelectric: This is the most common type. It uses light beams, and the alarm is triggered if particles of smoke interrupt the beam.
- Ionization: These detect ionized particles in the air, and are more sensitive that the optical type. This can mean, however, than it is more prone to false alarms than an optical sensor. Many modern systems use a combination of optical and ionization.
- Heat: detects anomalous temperatures.

[1]Passive Infrared sensor

3.2.1.5 Gas Sensors

Gas sensors allow us to deal with possible leaks of natural gas, butane, propane or methane. When a gas leak occurs, the internal sensors by catalytic oxidation are capable of detecting if the concentration of these gases is greater than allowed.

3.2.1.6 Flood and Leak Sensors

Flood or leak sensors are commonly fitted underneath baths and kitchen sinks, or in other locations with an elevated risk of leaks. If your plumbing does spring a leak then catching it early can save you a lot of trouble and expense, as water damage can have a major impact on your property.

3.2.1.7 Proximity Sensors

In some systems proximity sensors are used in place of switches. This would allow the user to simply wave their hand over a wall mounted sensor to, for example, switch on the lights, rather than having to actually press a switch.

3.2.1.8 Contact Sensors

Contact sensors are used for a range of applications, the most common of which is in burglary alarms. The sensor itself is basically just a kind of switch, which sends an electrical signal when two surfaces make contact. They can be used to monitor whether a door or window is open or closed.

3.2.1.9 Glass Break Sensors

There are two different kinds of sensor which can detect an intruder breaking glass to enter your building.

The first type is installed on the window itself, limiting its usefulness for protecting an entire property. This is also known as a ‘shock sensor’ and is triggered by sudden high frequency vibrations when the glass it is attached to breaks.

The second type is basically a microphone tuned specifically to pick up the sound of breaking glass. This is more useful, because a single detector can cover even a relatively large room with many windows.

3.2.1.10 Weather Sensors

The number of instruments can vary, but most personal weather stations include instruments to measure temperature, relative humidity, pressure, rain fall, and wind speed and direction.

3.2.2 Controlled Devices

Controlled devices include the tremendous range of equipment that a home automation system is capable of controlling. This components receive orders the controller sends and transform them into physical actions.

3.2.2.1 Motors

They are asynchronous and single phase motors installed inside the control element axis. They are used to raise and lower blinds, running curtains, awnings, etc. You can be activated manually with buttons or switches, or response to a stimulus captured by a sensor

3.2.2.2 Solenoid valves

They are electronic valves that control water connections, gas or electricity housing that open or close as needed or emergency.

3.2.2.3 Relays

Relays are used to open or close a circuit based on an external signal and functions as if it were a switch. A relay can trigger various circuits or several systems at once.

3.2.2.4 Buzzers and speakers

They are sound elements that are activated in response to alarms raised in the installation. They can be accompanied by lighting elements. They are connected to batteries for greater autonomy.

3.2.3 User Interfaces

User interfaces allow the user to interact with the system by sending information to the controller or by presenting information to the user about the system. The form and capabilities vary widely. Typical user interface devices include:

- Push-button panels, with or without visual displays.
- Touch-panel displays, with fixed or programmable screen layouts.
- Computer keyboards and monitors.
- Hand-held remote controls.
- Telephone interfaces to allow long-distance remote control.
- Television controllers with on-screen menus.

3.2.4 Controllers

Controllers are the elements responsible for receiving data from the sensors and user interfaces to analyze and transmit commands to the actuators.

Controllers generally run complex software, allowing them to execute single or multiple actions based on a variety of events. These events can come in many forms but can essentially be broken down to just two categories: timed and triggered

3.2.4.1 Timed events

Most home automation systems integrate an astronomic time clock. By knowing a home's geographic location, the astronomic time clock keeps up with changing sunrise and sunset events and syncs its clock over the Internet to remain accurate. With this feature, the home can perform tasks at specific times every day

3.2.4.2 Trigeered events

Triggered events are actions that the automation system executes based on something happening. Common triggers include:

- A button press, when I press this button, do this action
- A door opening, when this door opens, turn on the light
- Motion being detected, if there is no motion for 5 minutes, turn off this light
- A sensor tripping, if the temperature in this room exceeds 80 degrees, send me an email

3.2.5 Communication protocol

A home automation system is characterized by the communications protocol used, which is nothing other than the language or message format that different control elements of the system should use to understand each other and exchange their information in a coherent way.

While there are many competing vendors, there are very few world-wide accepted industry standards and the smart home space is heavily fragmented. Popular communications protocol for products include X10, Ethernet, RS-485, 6LoWPAN, Buetooth LE, ZigBee and Z-Wave, or other proprietary protocols all of which are incompatible with each other.

4 Documentation and study of MQTT protocol

4.1 Introduction to MQTT

MQTT is a Client/Server publish/subscribe messaging transport protocol. It is lightweight, open, simple, and designed so as to be easy to implement. These characteristics make it ideal for use in many situations, including constrained environments such as for communication in Machine to Machine (M2M) and Internet of Things (IoT) contexts where a small code footprint is required and/or network bandwidth is at a premium. Its features include:

- Use of the publish/subscribe message pattern which provides one-to-many message distribution and decoupling of applications.
- A messaging transport that is agnostic to the content of the payload.
- Three qualities of service for message delivery:
  - "**At most once**", where messages are delivered according to the best efforts of the operating environment and underlying transport protocol. Message loss can occur. This level could be used, for example, with ambient sensor data where it does not matter if an individual reading is lost as the next one will be published soon after.
  - "**At least once**", where messages are assured to arrive but duplicates can occur.
  - "**Exactly once**", where message are assured to arrive exactly one time. This level could be used, for example, with billing systems where duplicate or lost messages could lead to incorrect charges being applied.
- A small transport overhead and protocol exchanges minimized to reduce network traffic.
- A mechanism to notify interested parties when an abnormal disconnection occurs.

4.2 Architecture

MQTT has a Client/Server model, where every node is a client and connects to a server, known as a broker, usually over TCP[1]/IP[2], but can run over any other network protocols that provide ordered, lossless and bi-directional connections.

MQTT is message oriented. Every message is a discrete chunk of data, opaque to the broker.

Every message is published to an address, known as a topic. Clients may subscribe to multiple topics. Every client subscribed to a topic receives every message published to the topic.

4.3 History of the protocol

MQTT was invented by Andy Stanford-Clark (IBM) and Arlen Nipper (Arcom Control Systems) back in 1999, when their use case was to create a protocol for minimal battery loss and minimal bandwidth connecting oil pipelines over satellite connection. They specified the following goals, which the future protocol should have:

- Simple to implement
- Provide a Quality of Service Data Delivery
- Lightweight and Bandwidth Efficient
- Data Agnostic
- Continuous Session Awareness

These goals are still the core of MQTT, while the focus has changed from proprietary embedded systems to open Internet of Things use cases.

The name MQTT comes from “MQ Telemetry Transport”, referencing to IBM MQ Series, a family of message-oriented middleware products which originally supported MQTT, however, MQTT don’t use queues as in traditional message queuing solutions, although in some cases is possible to hold certain messages on memory until they are delivered to the client.

After MQTT had been used by IBM internally, version 3.1 was released royalty free in

[1] Transmission Control Protocol
[2] Internet Protocol

2010. Since then everybody could implement and use it. In addition to the protocol specification, also various client implementation were contributed to the newly founded Paho project underneath the Eclipse Foundation. This was definitely positive for the protocol because there is little chance for wide adoption when there is no ecosystem around it.

Around 3 years after the initial publication, it was announced that MQTT should be standardized under the wings of OASIS[3], a global non-profit consortium that works on the development, convergence, and adoption of standards for security, Internet of Things, energy, content technologies, emergency management, and other areas.

The standardization process took around 1 year and on October 29th 2014 MQTT was officially approved as OASIS Standard.

MQTT 3.1.1 is now the actual version of the protocol. The minor version change from 3.1 to 3.1.1 symbolizes that there were only little changes made to the previous version. The primary goal was to deliver a standard as soon as possible and improve MQTT from there on. There is also a variant of the main protocol called MQTT-SN addressed to devices integrated into sensor networks using UDP[4] transport layer or mesh network topology like ZigBee.

MQTT 3.1.1 was accepted as an ISO[5]/IEC[6] standard at the end of January 2016. Voting for the ISO/IEC 20922 standard it was closed with the approval of 100% and was published in mid-June 2016.

4.4 Benefits of MQTT

4.4.1 Publish/Subscribe pattern

The Pub/Sub (Publish/Subscribe) pattern is an alternative to the traditional Client/Server model, where a client communicates directly with an endpoint. However, Pub/Sub decouples a client who is sending a particular message, called publisher, from one or more clients who are receiving the message, called subscribers. This means that the publisher and subscriber don't know about the existence of one another. There is a

[3]Organization for the Advancement of Structured Information Standards
[4]User Datagram Protocol
[5]International Organization for Standardization
[6]International Electrotechnical Commission

third component, called broker, which is known by both the publisher and subscriber, which filters all incoming messages and distributes them accordingly.

The broker is responsible for subscriptions, persistent sessions, lost messages and security in general, including authentication and authorization.

4.4.2 Scalability

Pub/Sub also provides a greater scalability than the traditional client-server approach. This is because operations on the broker can be highly parallelized and processed event-driven. Also often message caching and intelligent routing of messages is decisive for improving the scalability. But it is definitely a challenge to scale publish/subscribe to millions of connections. This can be achieved using clustered broker nodes in order to distribute the load over more individual servers with load balancers.

4.4.3 Space decoupling

While the node and the broker need to have each other's IP address, nodes can publish information and subscribe to other nodes' published information without any knowledge of each other, since everything goes through the central broker. This reduces overhead that can accompany TCP sessions and ports, and allows the end nodes to operate independently of one another.

4.4.4 Time decoupling

A node can publish its information regardless of other nodes' states. Other nodes can then receive the published information from the broker when they are active. This allows nodes to remain in sleepy states even when other nodes are publishing messages directly relevant to them.

4.4.5 Synchronization decoupling

A node that's in the midst of one operation is not interrupted to receive a published message to which it's subscribed. The message is queued by the broker until the receiving node is finished with its existing operation. This saves operating current and

reduces repeated operations by avoiding interruptions of ongoing operations or sleepy states.

4.4.6 Authentication

When it comes to authentication, MQTT protocol itself provides user and password fields in the *CONNECT* message. Therefore, a customer has the ability to send a username and password when connecting to a broker MQTT

The username is a UTF-8 encoded string and password is binary data with a maximum length of 65535 bytes. The specification also states that a username without password is possible, in the other hand, it's not possible to just send a password without a username.

When using the built-in username/password authentication, the MQTT broker will evaluate the credential based on the implemented authentication mechanism and return one of the following return codes: - 0 (Connection Accepted) - 4 (Connection Refused, bad username or password) - 5 (Connection Refused, not authorized)

When setting username and password on the client, it will be sent to the broker in plain text which would allow eavesdropping by an attacker and an easy way of obtaining the credentials. The only way to guarantee a completely secure transmission of username and password is to use transport layer encryption.

4.4.6.1 Client Identification

Every MQTT client has a unique identifier which is provided by himself in the MQTT *CONNECT* message. This unique identifier can be up to 65535 characters making a common practice to use 36 character long UUIDs[7] or any other unique information available to the client like the MAC address of the network module or the serial number of the device itself. In the authentication process client ids are often used in addition to username and password. A common example to confirm if a client can access the MQTT broker is to validate username/password and the correct client id for that credential combination. While it's not a good security practice, it's also possible to ignore the username/password and just authenticate against the client identifier. For a closed system this kind of authentication may be enough.

[7] Universally Unique Identifier

4.4.6.2 X.509 Certificates

Another possible authentication source from the client is a X.509 client certificate, which will be presented to the broker during the TLS handshake. Some brokers allow to use the information in the certificate for application layer authentication after the TLS handshake already succeeded. This enables the broker to read all informations contained in the certificate and use it for authentication purposes as well.

4.4.7 Quality of Service (QoS)

Quality of Service is a major feature of MQTT as it makes communication in unreliable networks a lot easier because the protocol handles retransmission and guarantees the delivery of the message, regardless how unreliable the underlying transport is. Also it empowers a client to choose the QoS level depending on its network reliability and application logic.

- Use **QoS 0** when you have a complete or almost stable connection between sender and receiver or you don't care if one or more messages are lost once a while. That is sometimes the case if the data is not that important or will be send at short intervals, where it is OK that messages might get lost.
- Use **QoS 1** when you need to get every message and your use case can handle duplicates. The most often used QoS is level 1, because it guarantees the message arrives at least once. Of course your application must be tolerating duplicates and process them accordingly
- Use **QoS 2** when it is critical to your application to receive all messages exactly once. This is often the case if a duplicate delivery would do harm to application users or subscribing clients. You should be aware of the overhead and that it takes a bit longer to complete the QoS 2 flow.

4.4.8 Last Will and Testament

MQTT is often used in scenarios were unreliable networks are very common. Therefore it is assumed that some clients will disconnect ungracefully from time to time, because they lost the connection, the battery is empty or any other imaginable case. Therefore it would be good to know, if a connected client has disconnected with a MQTT

DISCONNECT message or not, in order to take appropriate action.

A connecting client will provide his will in form of an MQTT message and topic in the *CONNECT* message. If this clients disappears without previous notification, the broker sends this message on behalf of the client to all subscribers.

4.4.9 Subject-based filtering

MQTT uses subject-based filtering of messages. So each message contains a topic, which the broker uses to find out, if a subscribing client will receive the message or not. Topics are created by the publisher node and nodes wishing to receive messages should subscribe to them. The communication can be one to one or one to many.

A topic is represented by a string and has a hierarchical structure whose levels are separated using the slash (/) character. In this way customers can create hierarchies that publish and receive data.

A MQTT node can subscribe to any specific topic within the hierarchy or use single-level (+) or multi-level (#) wildcards to subscribe several topics at once. This allows a minimal amount of code and, therefore, reduce memory size and cost.

MQTT differs from traditional message queue protocols in the following points:

- **A message queue stores messages until they are consumed.** When using message queues, each incoming message will be stored on that queue until it is picked up by any client (often called consumer). Otherwise the message will just be stuck in the queue and waits for getting consumed. It is not possible that message are not processed by any client, like it is in MQTT if nobody subscribes to a topic.
- **A message will only be consumed by one client.** Another big difference is the fact that in a traditional queue a message is processed by only one consumer. So that the load can be distributed between all consumers for a particular queue. In MQTT it is quite the opposite, every subscriber gets the message, if they subscribed to the topic.
- **Queues are named and must be created explicitly.** A queue is far more inflexible than a topic. Before using a queue it has to be created explicitly with a separate command. Only after that it is possible to publish or consume messages.

In MQTT topics are extremely flexible and can be created on the fly.

4.5 Drawbacks of MQTT

4.5.1 Central broker

The use of a central broker can be inconvenient for distributed IoT systems. For example, a system can start small with a remote control and a window shade, which does not require a central broker. Then, as the system grows, for example, adding safety sensors, light bulbs or other blinds, the network naturally grows and expands, and may have need for a central broker. However, none of the individual nodes want to assume the cost and responsibility, as it requires resources, software and complexity that are not critical to the function of end node.

In systems that already have a central broker, it can become a single point of failure for the entire network. For example, if the broker is a node powered without a battery backup, then battery-powered nodes can continue to function during a power outage, while the broker is offline, leaving the network inoperable.

4.5.2 Transport layer protocol

Even though MQTT is designed to be lightweight, every MQTT client must support TCP and will typically hold a connection open to the broker at all times. For some environments where packet loss is high or computing resources are scarce, this is a problem.

TCP was originally designed for devices with more memory and processing resources than may be available in typical IoT constrained-node networks. The TCP protocol requires that connections be established in a multi-step handshake process before any messages are exchanged. This drives up wake-up and communication times, and reduces battery life over the long run. Also, in TCP, it's ideal for two communicating nodes to hold their TCP sockets open for each other continuously with a persistent session, which again may be difficult with energy- and resource-constrained devices.

Using TCP without session persistence can require incremental transmit time for connection establishment. For nodes with periodic, repetitive traffic, this can lead to lower

operating life.

These shortcomings are addressed by the MQTT-SN protocol, which defines a UDP mapping of MQTT and adds broker support for indexing topic names so it can be implemented in a practical IEEE 802.15.4 deployments.

4.5.3 Security

MQTT uses unencrypted TCP and is not "out-of-the-box" secure. However, because it uses TCP, it can and should use TLS/SSL Internet security. TLS is a very secure method for encrypting traffic, but is also resource-intensive for lightweight clients due to its required handshake and increased packet overhead. For networks where energy is a very high priority and security much less so, encrypting just the packet payload may suffice.

4.5.4 No TTL (Time-To-Live) on messages

The protocol does not allow to add a TTL attribute per message. So if you use the *cleanSession* parameter, the message will be held indefinitely in the broker. As time goes by, it could create a lot of messages on the broker, so it could affect the overall performances, and use some disk space if you persist the messages.

A possible workaround for this is to periodically check the topics for old messages, but you only get a TTL per topics, not at message level. But it seems that this limitation should be addressed in a next release of the protocol. This way refers to broker's internal TTL mechanisms.

4.6 MQTT Control Packet format

The MQTT protocol works by exchanging a series of MQTT Control Packets in a defined way. This section describes the format of these packets.

Table 4.1: Structure of an MQTT Control Packet

Structure of an MQTT Control Packet
Fixed header, present in all MQTT Control Packets

Structure of an MQTT Control Packet
Variable header, present in some MQTT Control Packets
Payload, present in some MQTT Control Packets

Each MQTT Control Packet contains a fixed header part, a variable header part and payload based depending on its type.

Table 4.2: Fixed header format

Packet	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
PUBLISH	TYPE	TYPE	TYPE	TYPE	DUP	QoS	QoS	RETAIN
PUBREL	TYPE	TYPE	TYPE	TYPE	0	0	1	0
SUBSCRIBE	TYPE	TYPE	TYPE	TYPE	0	0	1	0
UNSUBSCRIBE	TYPE	TYPE	TYPE	TYPE	0	0	1	0
OTHER	TYPE	TYPE	TYPE	TYPE	0	0	0	0

The fixed part of the header consists of 4 bits indicating the packet type and specific packet type flags.

Table 4.3: Control Packet types

Name	Value	Direction of flow	Description
Reserved	0	Forbidden	Reserved
CONNECT	1	Client to Server	Client request to connect to Server
CONNACK	2	Server to Client	Connect acknowledgment
PUBLISH	3	Both	Publish message
PUBACK	4	Both	Publish acknowledgment
PUBREC	5	Both	Publish received (assured delivery part 1)
PUBREL	6	Both	Publish release (assured delivery part 2)
PUBCOMP	7	Both	Publish complete (assured delivery part 3)
SUBSCRIBE	8	Client to Server	Client subscribe request
SUBACK	9	Server to Client	Subscribe acknowledgment
UNSUBSCRIBE	10	Client to Server	Unsubscribe request
UNSUBACK	11	Server to Client	Unsubscribe acknowledgment
PINGREQ	12	Client to Server	PING request
PINGRESP	13	Server to Client	PING response

Name	Value	Direction of flow	Description
DISCONNECT	14	Client to Server	Client is disconnecting
Reserved	15	Forbidden	Reserved

The table shows all messages that make up the MQTT protocol. The most important are *CONNECT*, *PUBLISH*, *SUBSCRIBE*, *UNSUBSCRIBE* and *DISCONNECT*. Others are confirmation messages (*CONNACK*, *PUBACK*, *SUBACK*, *UNSUBACK*) or messages related to quality of service messages *PUBLISH* (*PUBACK*, *PUBREL*, *PUBREL* and *PUBCOMP*).

Some types of MQTT Control Packets contain a variable header component. It resides between the fixed header and the payload. The content of the variable header varies depending on the packet type.

4.7 Description of MQTT Control Packets

4.7.1 CONNECT

After a network connection is established by a client to a server, the first packet sent from the client to the server must be a *CONNECT* packet

A client can only send the *CONNECT* packet once over a network connection. The server must process a second *CONNECT* packet sent from a client as a protocol violation and disconnect the client

The payload contains one or more encoded fields. They specify a unique identifier for the client, a will topic, will message, username and password. All but the client identifier are optional and their presence is determined based on flags in the variable header.

4.7.1.1 Client Id

The client identifier (short ClientId) is an identifier of each MQTT client connecting to a MQTT broker. It should be unique per broker as it is used for identifying the client and the current state of the client. If you don't need a state to be hold by the broker,

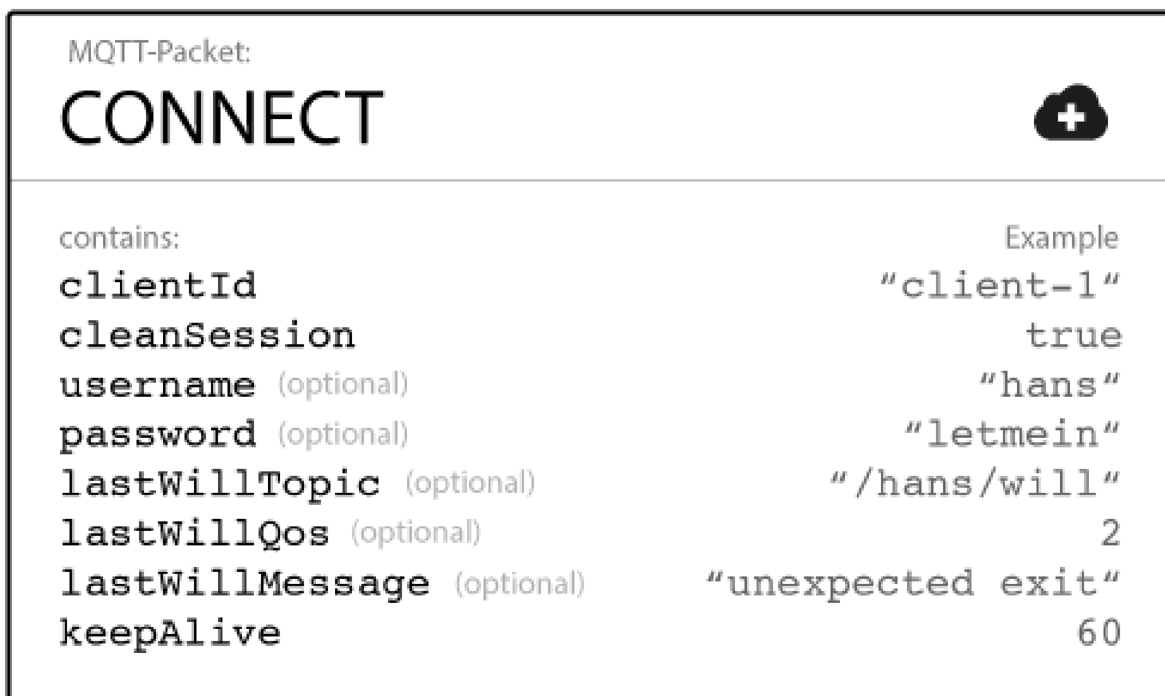


Figure 4.1: *CONNECT* packet structure

it is also possible to send an empty ClientId, which results in a connection without any state.

4.7.1.2 Clean Session

The clean session flag indicates the broker, if the client wants to establish a persistent session or not. A persistent session (CleanSession is set to 0) means that the broker will store all subscriptions for the customer and also all lost messages when they subscribe with QoS 1 or 2. If CleanSession is set to 1, the broker does not store anything for the customer and also purges all the existing information from a previous persistent session. The following information is stored in the broker for a persistent session:

- The existence of a Session, even if the rest of the Session state is empty.
- The client's subscriptions.
- QoS 1 and QoS 2 messages pending transmission to the client.
- QoS 1 and QoS 2 messages which have been sent to the client, but have not been completely acknowledged.
- QoS 2 messages which have been received from the client, but have not been completely acknowledged.

Like the broker, each MQTT client must have the responsibility to keep certain information itself:

- QoS 1 and QoS 2 messages which have been sent to the server, but have not been completely acknowledged.
- QoS 2 messages which have been received from the server, but have not been completely acknowledged.

4.7.1.3 Username/Password

MQTT allows to send a username and password for authentication and authorization of the client. However, the password is sent in plain text so it is highly recommended to use username and password along with safe transportation of it, for example, a hashing algorithm is applied or TLS is used below.

4.7.1.4 Last Will and Testament (LWT)

The LWT feature is used in MQTT to notify other clients about an ungracefully disconnected client. Each client can specify its last will message (a normal MQTT message with topic, retained flag, QoS and payload) when connecting to a broker. The broker will store the message until it detects that the client has disconnected ungracefully. If the client disconnect abruptly, the broker sends the message to all subscribed clients on the topic, which was specified in the last will message. The stored LWT message will be discarded if a client disconnects gracefully by sending a *DISCONNECT* message.

According to the MQTT 3.1.1 specification the broker will distribute the LWT of a client in the following cases:

- An I/O error or network failure is detected by the server.
- The client fails to communicate within the Keep Alive time.
- The client closes the network connection without sending a *DISCONNECT* packet first.
- The server closes the network connection because of a protocol error.

4.7.1.5 Keep Alive

The keep alive functionality assures that the connection is still open and both broker and client are connected to one another. Therefore the client specifies a time interval in seconds and communicates it to the broker during the establishment of the connection. The interval is the longest possible period of time, which broker and client can endure without sending a message.

That means as long as messages are exchanged frequently and the keep alive interval is not exceeded, there is no need to send an extra message to ensure that the connection is still open. But if the client doesn't send any messages during the period of the keep alive it must send a *PINGREQ* packet to the broker to confirm its availability and also make sure the broker is still available. The broker must disconnect a client, which doesn't send *PINGREQ* or any other message in one and a half times of the keep alive interval. Likewise should the client close the connection if the response from the broker isn't received in a reasonable amount of time.

4.7.2 CONNACK

The *CONNACK* packet is the packet sent by the server in response to a *CONNECT* packet received from a client. If the client does not receive a *CONNACK* packet from the server within a reasonable amount of time, the client should close the network connection.

4.7.2.1 Session Present

The session present flag indicate, whether the broker already has a persistent session of the client from previous interactions. If a client connects and has set CleanSession to 1, this flag is always 0, because there is no session available. If the client has set CleanSession to 0, the flag is depending on, if there are session information available for the ClientId. If stored session information exist, then the flag is 1 and otherwise it is 0.

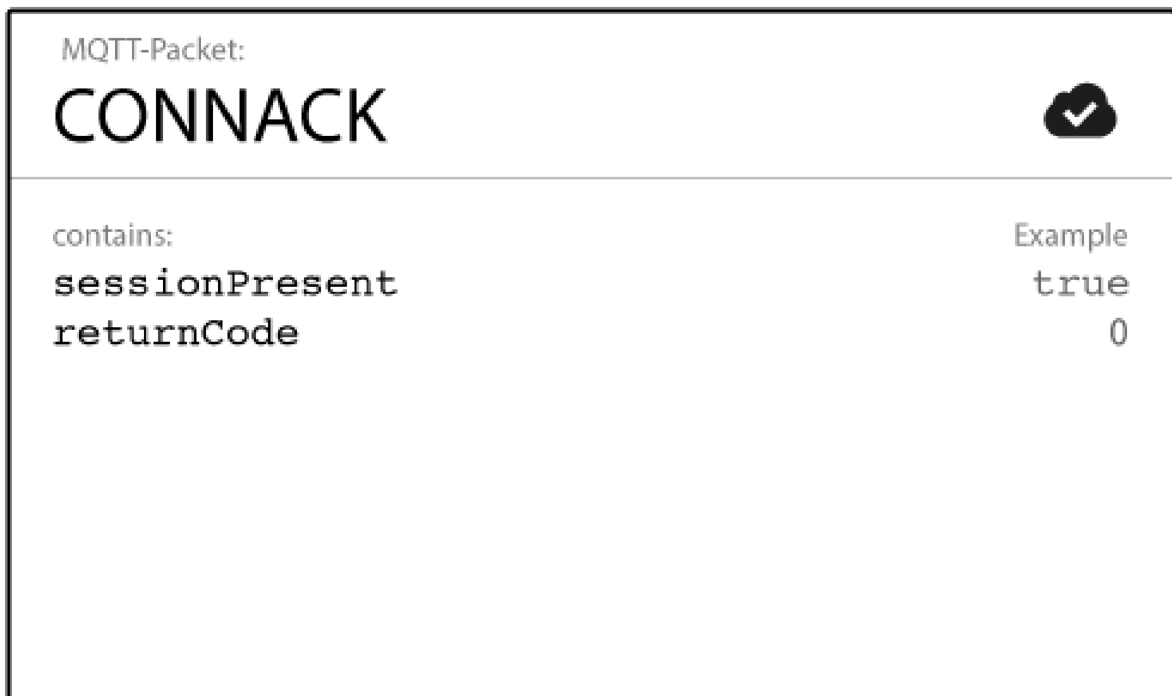


Figure 4.2: *CONNACK* packet structure

4.7.2.2 Return Code

The second flag in the *CONNACK* is the connect acknowledge flag. It signals the client, if the connection attempt was successful and otherwise what the issue is. If a well formed *CONNECT* packet is received by the server, but the server is unable to process it for some reason, then the server should attempt to send a *CONNACK* packet containing the appropriate non-zero Connect return code from this table.

Table 4.4: *CONNACK* Return code values

Return Code	Return Code Response
0	Connection Accepted
1	Connection Refused, unacceptable protocol version
2	Connection Refused, identifier rejected
3	Connection Refused, server unavailable
4	Connection Refused, bad username or password
5	Connection Refused, not authorized

4.7.3 PUBLISH

After a MQTT client is connected to a broker, it can publish messages that must contain a topic, which will be used by the broker to forward the message to interested clients. Each message typically has a payload which contains the actual data to transmit in byte format. MQTT is data-agnostic and it totally depends on the use case how the

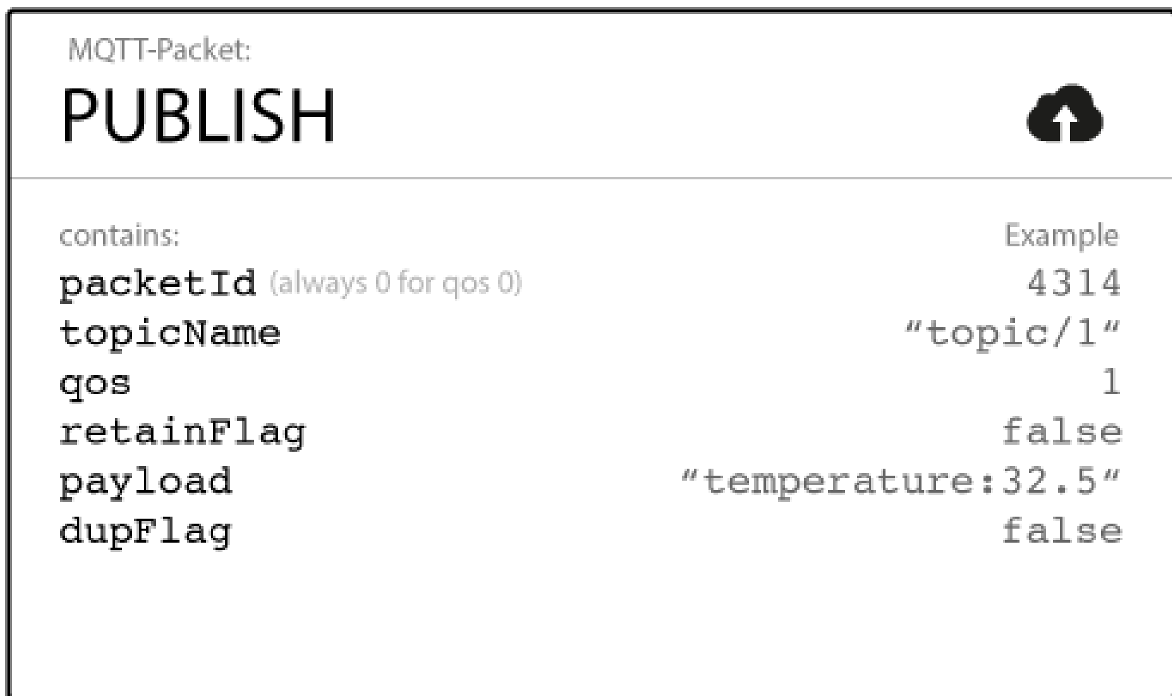


Figure 4.3: *PUBLISH* packet structure

In comparison to a message queue a topic is very lightweight. There is no need for a client to create the desired topic before publishing or subscribing to it, because a broker accepts each valid topic without any prior initialization. Noticeable is that each topic must have at least 1 character to be valid and it can also contain spaces, also a topics are case-sensitive.

When a client subscribes to a topic it can use the exact topic the message was published to or it can subscribe to more topics at once by using wildcards. A wildcard can only be used when subscribing to topics and is not permitted when publishing a message. In the following we will look at the two different kinds one by one: single-level and multi-level wildcards.

- As the name already suggests, a **single-level wildcard** is a substitute for one topic level. The plus character (+) represents a single level wildcard in the topic. Any topic matches to a topic including the single level wildcard if it contains an arbitrary string instead of the wildcard.
- While the single-level wildcard only covers one topic level, the **multi-level wildcard** covers an arbitrary number of topic levels. The hash character (#) represents a multi-level wildwcard. In order to determine the matching topics it is required that the multi-level wildcard is always the last character in the topic

and it is preceded by a forward slash character (/)

In general you are totally free in naming your topics, but there is one exception. Each topic, which starts with a dollar sign ($) will be treated specially and is for example not part of the subscription when subscribing to #. These topics are reserved for internal statistics of the MQTT broker. Therefore it is not possible for clients to publish messages to these topics. At the moment there is no clear official standardization of topics that must be published by the broker. It is common practice to use $SYS/ for all these information and a lot of brokers implement these, but in different formats.

4.7.3.3 Quality of Service (QoS)

MQTT delivers Application Messages according to the QoS levels defined here. The delivery protocol is symmetric, in the description below the client and server can each take the role of either Sender or Receiver. The delivery protocol is concerned solely with the delivery of an application message from a single Sender to a single Receiver. When the server is delivering an Application Message to more than one client, each client is treated independently.

In the figures, the *PUBLISH* message is accompanied by three numbers. The first number corresponds to the level of QoS, the second to the duplicate flag, the third represents the package ID.

- **QoS 0: At most once delivery**. The message is delivered according to the capabilities of the underlying network. No response is sent by the receiver and no retry is performed by the sender. The message arrives at the receiver either once or not at all.

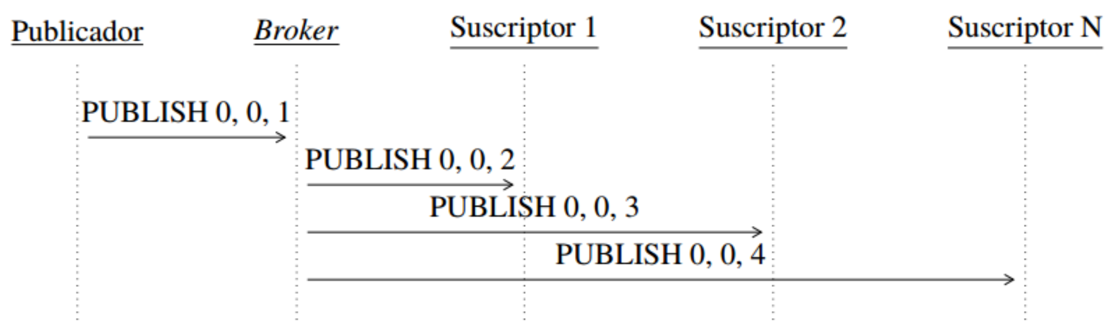


Figure 4.4: Choreography of a publication with QoS 0

- **QoS 1: At least once delivery**. This quality of service ensures that the message arrives at the receiver at least once. A QoS 1 *PUBLISH* packet has a packet identifier in its variable header and is acknowledged by a *PUBACK* packet.
  - The Sender must assign an unused packet identifier each time it has a new Application Message to publish and treat the *PUBLISH* packet as "unacknowledged" until it has received the corresponding *PUBACK* packet from the receiver.
  - The Receiver must respond with a *PUBACK* packet containing the packet identifier from the incoming *PUBLISH* packet, having accepted ownership of the Application Message. After it has sent a *PUBACK* packet the Receiver must treat any incoming *PUBLISH* packet that contains the same packet identifier as being a new publication, irrespective of the setting of its duplicate flag.

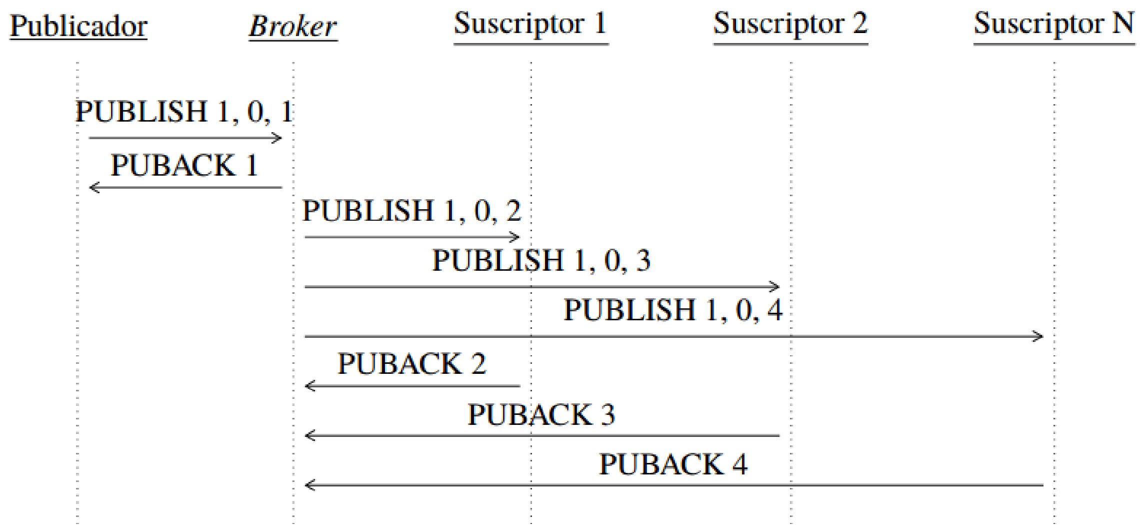


Figure 4.5: Choreography of a publication with QoS 1

- **QoS 2: Exactly once delivery**. This is the highest quality of service, for use when neither loss nor duplication of messages are acceptable. There is an increased overhead associated with this quality of service.
  - The Sender must assign an unused packet identifier when it has a new Application Message to publish and treat the *PUBLISH* packet as "unacknowledged" until it has received the corresponding *PUBREC* packet from the receiver. Also it must send a *PUBREL* packet when it receives a *PUBREC* packet from the receiver (with the same packet identifier as the original

PUBLISH packet) and treat the *PUBREL* packet as “unacknowledged” until it has received the corresponding *PUBCOMP* packet from the receiver. Finally, it must not re-send the *PUBLISH* once it has sent the corresponding *PUBREL* packet.

– The Receiver must respond with a *PUBREC* containing the packet identifier from the incoming *PUBLISH* packet until it has received the corresponding *PUBREL* packet and acknowledge any subsequent *PUBLISH* packet with the same packet identifier by sending a *PUBREC*. Also it must respond to a *PUBREL* packet by sending a *PUBCOMP* packet containing the same packet identifier as the *PUBREL*.Finally, after it has sent a *PUBCOMP*, the receiver must treat any subsequent *PUBLISH* packet that contains that packet identifier as being a new publication.

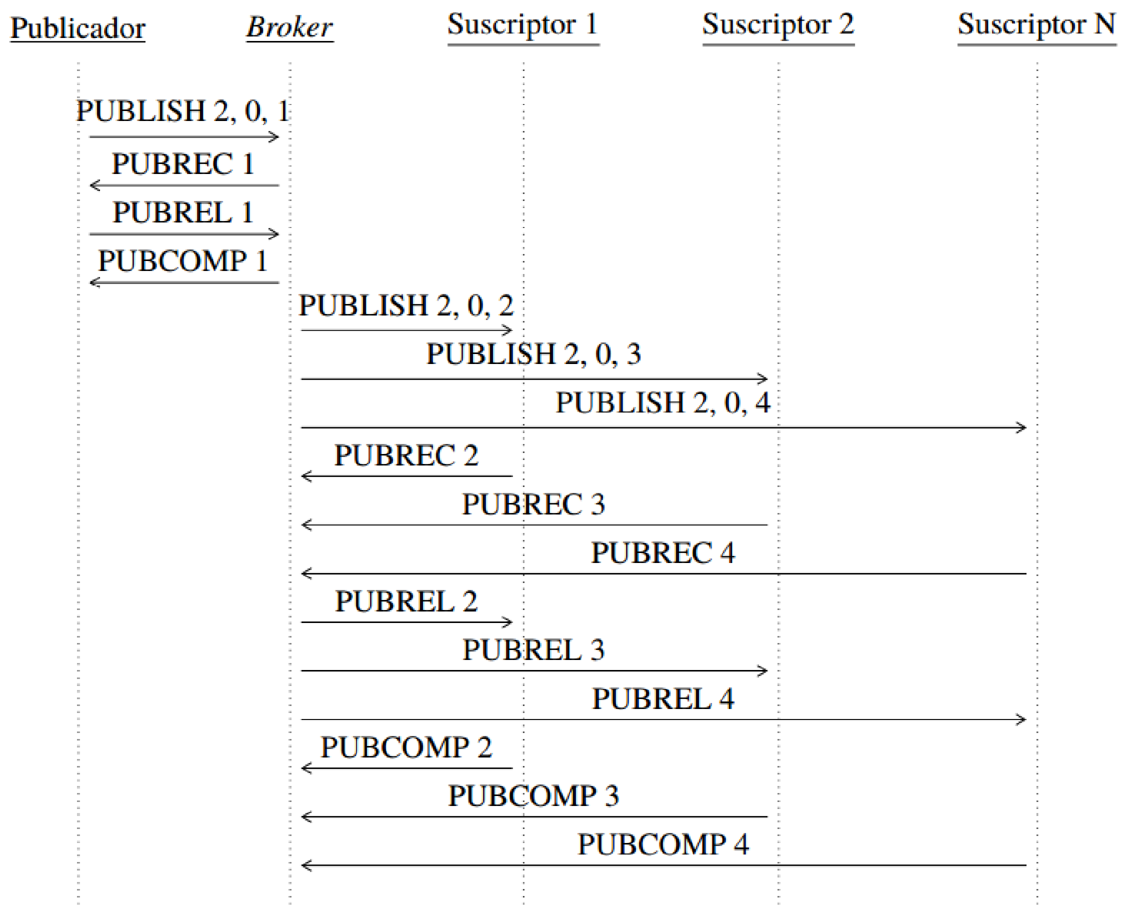


Figure 4.6: Choreography of a publication with QoS 2

4.7.3.4 Retain

This flag determines if the message will be saved by the broker for the specified topic as last known good value. New clients that subscribe to that topic will receive the last retained message on that topic instantly after subscribing.

4.7.3.5 Payload

This is the actual content of the message. MQTT is totally data-agnostic, it's possible to send images, texts in any encoding, encrypted data and virtually every data in binary.

4.7.3.6 Duplicate

The duplicate flag indicates, that this message is a duplicate and is resent because the other end didn't acknowledge the original message. This is only relevant for QoS greater than 0.

4.7.4 PUBACK

A *PUBACK* packet is the response to a *PUBLISH* packet with QoS level 1.

4.7.4.1 Packet Identifier

The packet identifier is a unique identifier between client and broker to identify a message in a message flow. This contains the packet identifier from the *PUBLISH* packet that is being acknowledged.

4.7.5 PUBREC

A *PUBREC* packet is the response to a *PUBLISH* packet with QoS 2. It is the second packet of the QoS 2 protocol exchange.

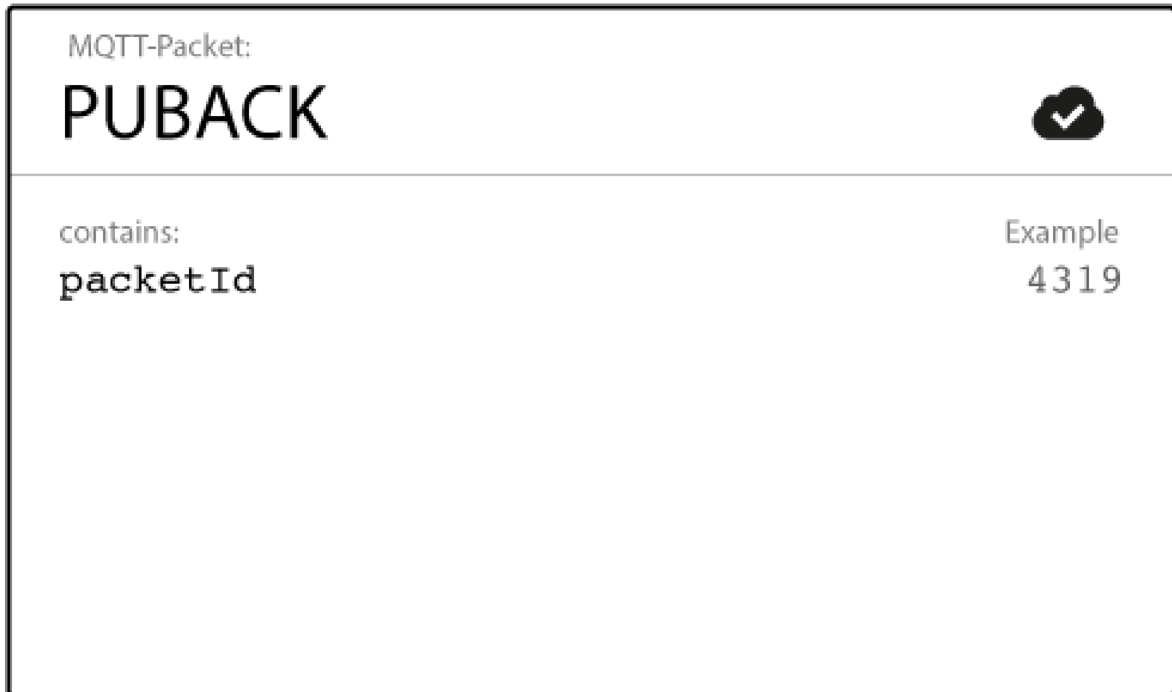


Figure 4.7: *PUBACK* packet structure

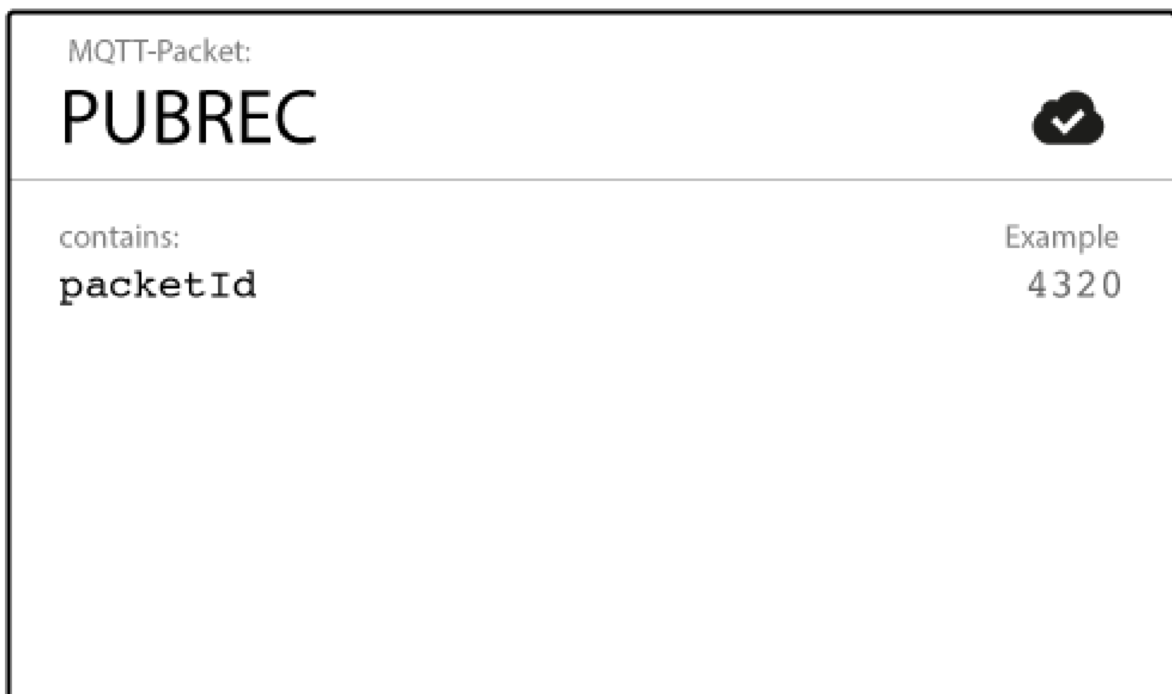


Figure 4.8: *PUBREC* packet structure

4.7.5.1 Packet Identifier

The packet identifier is a unique identifier between client and broker to identify a message in a message flow. This contains the packet identifier from the *PUBLISH* packet that is being acknowledged.

4.7.6 PUBREL

A *PUBREL* packet is the response to a *PUBREC* packet. It is the third packet of the QoS 2 protocol exchange.

Figure 4.9: *PUBREL* packet structure

4.7.6.1 Packet Identifier

The packet identifier is a unique identifier between client and broker to identify a message in a message flow. This contains the packet identifier from the *PUBREC* packet that is being acknowledged.

4.7.7 PUBCOMP

The *PUBCOMP* packet is the response to a *PUBREL* packet. It is the fourth and final packet of the QoS 2 protocol exchange.

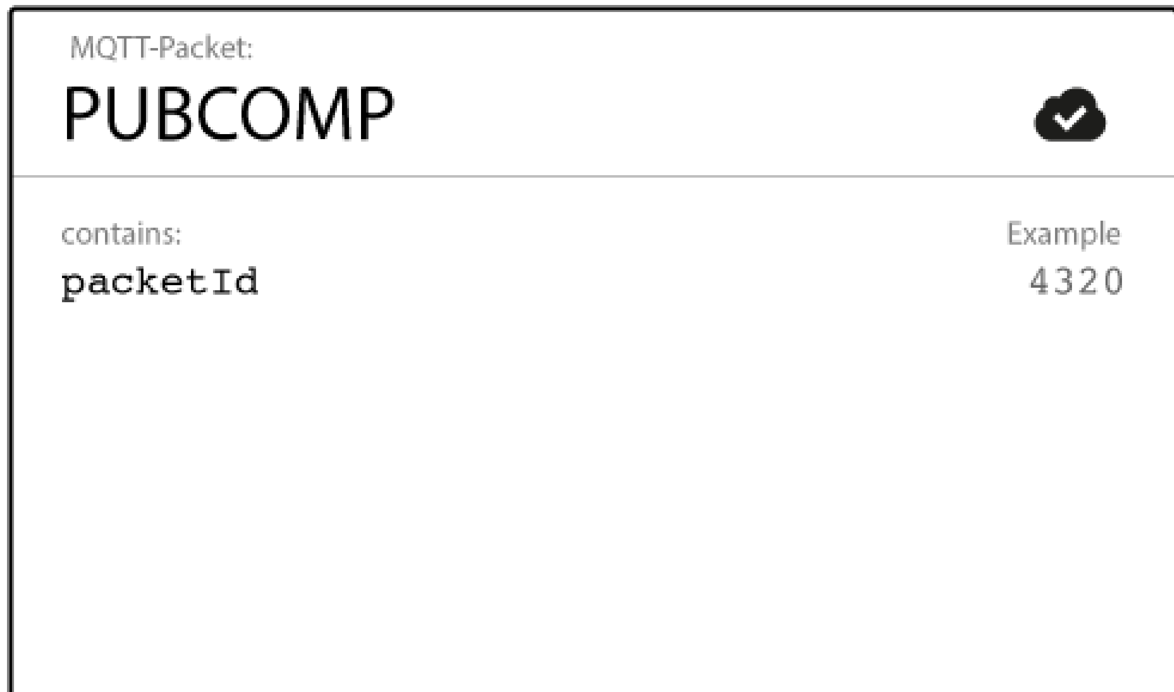


Figure 4.10: *PUBCOMP* packet structure

4.7.7.1 Packet Identifier

The packet identifier is a unique identifier between client and broker to identify a message in a message flow. This contains the packet identifier from the *PUBREL* packet that is being acknowledged.

4.7.8 SUBSCRIBE

Publishing messages doesn't make sense if no one ever receives the message, or, in other words, if there are no clients subscribing to any topic. The *SUBSCRIBE* packet is sent from the client to the server to create one or more Subscriptions. Each Subscription registers a client's interest in one or more Topics. The server sends *PUBLISH* Packets to the client in order to forward Application Messages that were published to Topics

that match these Subscriptions. The *SUBSCRIBE* packet also specifies (for each Subscription) the maximum QoS with which the server can send Application Messages to the client.

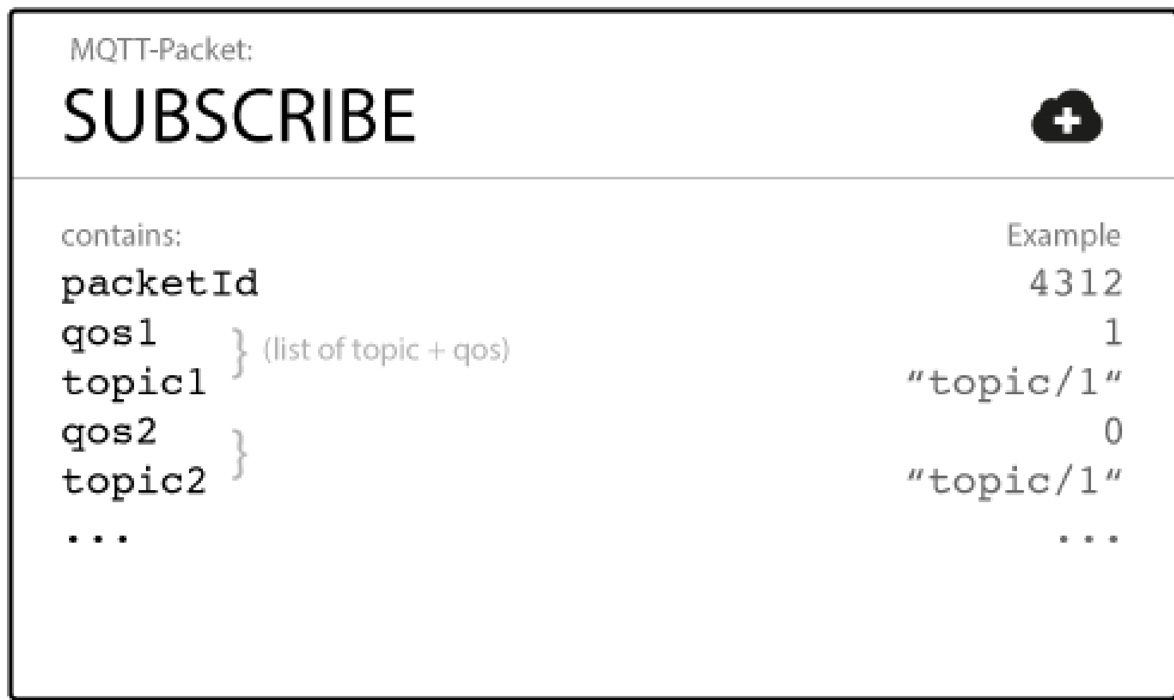


Figure 4.11: *SUBSCRIBE* packet structure

4.7.8.1 Packet Identifier

The packet identifier is a unique identifier between client and broker to identify a message in a message flow. This is only relevant for QoS greater than zero.

4.7.8.2 Topics

A *SUBSCRIBE* message can contain an arbitrary number of subscriptions for a client. Each subscription is a pair of a topic topic and QoS level. The topic in the subscribe message can also contain wildcards, which makes it possible to subscribe to certain topic patterns. If there are overlapping subscriptions for one client, the highest QoS level for that topic wins and will be used by the broker for delivering the message.

4.7.9 SUBACK

Each subscription will be confirmed by the broker through sending an acknowledgement to the client in form of the *SUBACK* message. This message contains the same packet identifier as the original *SUBSCRIBE* message in order to identify the message and a list of return codes.

Figure 4.12: *SUBACK* packet structure

4.7.9.1 Packet Identifier

The packet identifier is a unique identifier between client and broker to identify a message in a message flow. This contains the packet identifier from the *SUBSCRIBE* packet that is being acknowledged.

4.7.9.2 Return Code

The broker sends one return code for each topic/QoS-pair it received in the *SUBSCRIBE* message. So if the *SUBSCRIBE* message had 5 subscriptions, there will be 5 return codes to acknowledge each topic with the QoS level granted by the broker. If the subscription was prohibited by the broker (e.g. if the client was not allowed to

subscribe to this topic due to insufficient permission or if the topic was malformed), the broker will respond with a failure return code for that specific topic.

Table 4.5: *SUBACK* Return code values

Return Code	Return Code Response
0	Success - Maximum QoS 0
1	Success - Maximum QoS 1
2	Success - Maximum QoS 2
128	Failure

4.7.10 UNSUBSCRIBE

The counterpart of the *SUBSCRIBE* message is the *UNSUBSCRIBE* message which deletes existing subscriptions of a client on the broker. The *UNSUBSCRIBE* message is similar to the *SUBSCRIBE* message and also has a packet identifier and a list of topics.

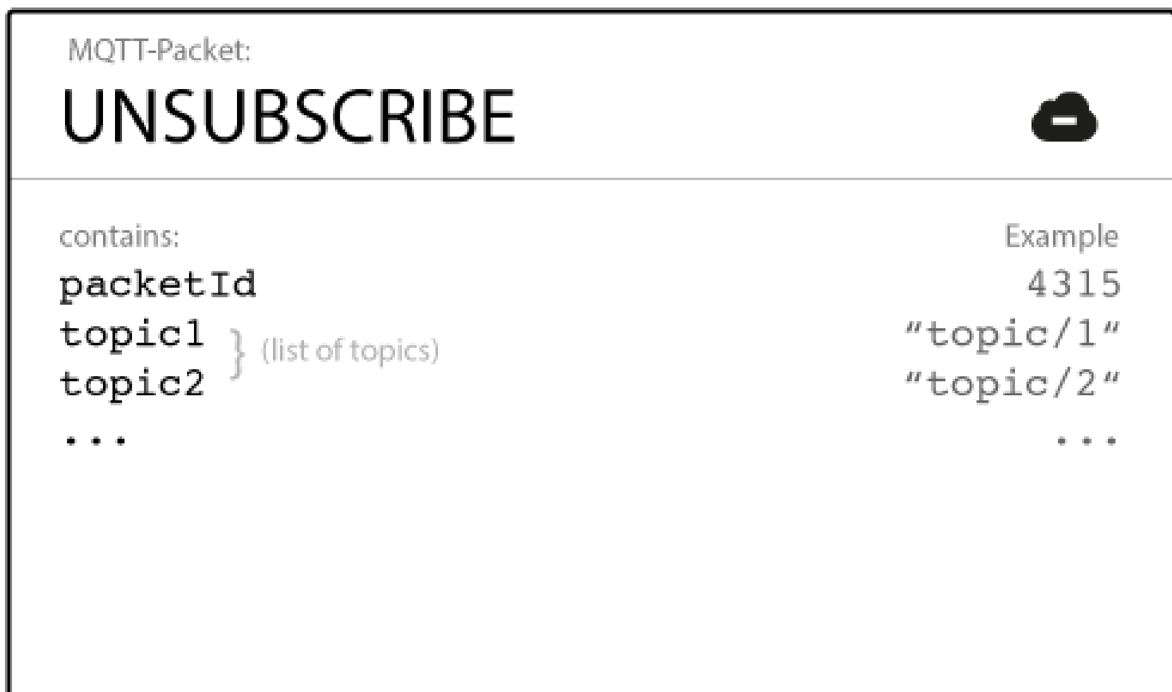


Figure 4.13: *UNSUBSCRIBE* packet structure

4.7.10.1 Packet Identifier

The packet identifier is a unique identifier between client and broker to identify a message in a message flow. The acknowledgement of an *UNSUBSCRIBE* message will contain the same identifier.

4.7.10.2 Topics

The list of topics contains an arbitrary number of topics, the client wishes to unsubscribe from. It is only necessary to send the topic as string (without QoS), the topic will be unsubscribed regardless of the QoS level it was initially subscribed with.

4.7.11 UNSUBACK

The broker will acknowledge the request to unsubscribe with the *UNSUBACK* message. This message only contains a packet identifier.

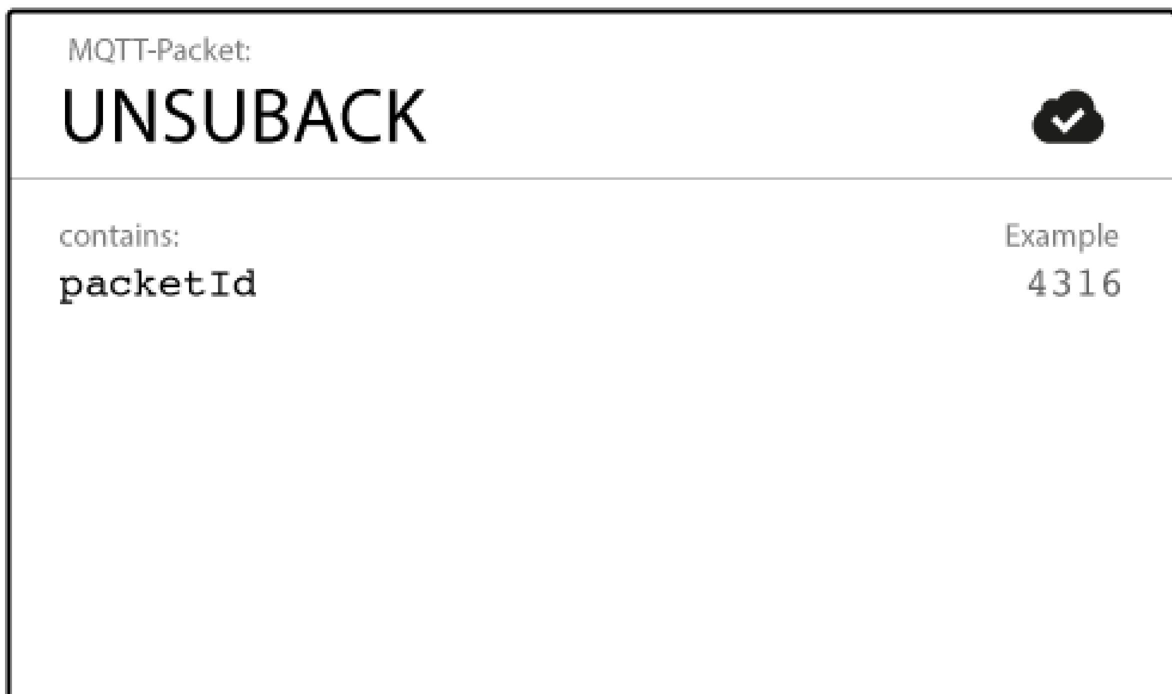


Figure 4.14: *UNSUBACK* packet structure

4.7.11.1 Packet Identifier

The packet identifier is a unique identifier between client and broker to identify a message in a message flow. It is the same as in the *UNSUBSCRIBE* message.

4.7.12 PINGREQ

The *PINGREQ* is sent by the client and indicates to the broker that the client is still alive, even if it hasn't send any other packets (*PUBLISH*, *SUBSCRIBE*, etc..). The client can send a *PINGREQ* at any time to make sure the network connection is still alive. The *PINGREQ* packet doesn't have any payload.

Figure 4.15: *PINGREQ* packet structure

4.7.13 PINGRESP

When receiving a *PINGREQ* the broker must reply with a *PINGRESP* packet to indicate its availability to the client. Similar to the *PINGREQ* the packet doesn't contain any payload.

Figure 4.16: *PINGRESP* packet structure

4.7.14 DISCONNECT

The *DISCONNECT* packet is the final Control Packet sent from the client to the server. It indicates that the client is disconnecting cleanly. The *PINGREQ* packet doesn't have any payload.

- After sending a *DISCONNECT* packet the client must close the network connection and must not send any more Control Packets on that network connection.
- On receipt of *DISCONNECT*, the server must discard any will message associated with the current connection without publishing it.

Figure 4.17: *DISCONNECT* packet structure

5 Solution design

5.1 Design overview

The solution will use a decentralized approach composed of a central node, aka the *Musquetteer*, and several smart nodes, aka the *ESPutniks*. *Musquetteer* and *ESPutniks* communicate to each other using the existing WiFi/Ethernet network.

5.2 Functional design

5.2.1 *Musquetteer* nodes

Musquetteer nodes are small single-board computers and at least one is required in the system.

- **Broker**, it host the MQTT broker and coordinates messages between clients.
- **Access Point**, it can generate the WiFi network that *ESPutnik* nodes connect to.
- **Gateway**, it can act as interface between the sensor network and the internet.
- **Server**, it host a web server and the applications requiered .

5.2.2 *ESPutnik* nodes

ESPutnik nodes are the smallest smart controllers available in the system.

- **Connected**, it connects to the MQTT broker hosted by the *Musquetteer* and sends/receives commands.
- **Controller**, sensors and actuators are connected physically to it.
- **Decentralized**, it have the capability to control its own channels, even if network is down.

5.2.3 MQTT convention

To efficiently parse messages, I will define a few rules related to topic names. This rules will follow a convention to easily offer an abstraction layer between the funtional and technical requirements.

An instance of a physical controller is called a device. A device has device properties, like the current local IP, the Wi-Fi signal, etc. A device can expose multiple channels. For example, a weather device might expose a temperature channels and an humidity channels. A channels can have multiple channels properties which can be settable. The temperature channels might for example expose a temperature property containing the actual temperature, and an unit property.

I will work in a state-based approach. You don't turn on the actuators directly, instead you command your device channel to put it's state to **on**. This especially fits well with MQTT because of retained messages and provides pessimistic feedback, which is important for home automation.

- **ESPutnik** / **device ID** / $ **device property**: a property starting with a $ at the third level of the path is related to the device.
- **ESPutnik** / **device ID** / **channel ID** / **property**: as defined in the $channels device property, **channel ID** is the ID of the channel. **property** is the property of the channel that is getting updated.
- **ESPutnik** / **device ID** / **channel ID** / **property** / **set**: the device can subscribe to this topic if the property is settable from the controller, in case of actuators.

Table 5.1: Device properties

Property	Description
$online	**true** when the device is online, **false*‘ when the device is offline (through LWT)
$name	Friendly name of the device
$localip	IP of the device on the local network
$uptime	Time elapsed in seconds since the boot of the device
$signal	Integer representing the Wi-Fi signal quality in percentage if applicable
$fwname	Name of the firmware running on the device.
$fwversion	Version of the firmware running on the device

Property	Description
$channels	Channels the device has, with format **id:type** separated by a **,** if there are multiple channels
$ota	Latest OTA version available for the device
$reset	**true** when the controller wants the device to reset its configuration. **false** otherwise.

5.3 Technical design

5.3.1 Raspberry Pi 3

Raspberry Pi is a series of credit card-sized single-board computers developed in the United Kingdom by the Raspberry Pi Foundation to promote the teaching of basic computer science in schools and developing countries.

The Raspberri Pi will fit the puspose on the *Musquetteer* node.

5.3.1.1 Hardware

Several generations of Raspberry Pis have been released. All models feature a Broadcom SOC[1], which includes an ARM compatible CPU[2] and an on chip GPU[3]. CPU speed ranges from 700 MHz to 1.2 GHz for the Pi 3 and on board memory range from 256 MB to 1 GB RAM. SD[4] cards are used to store the operating system and program memory in either the SDHC or MicroSDHC sizes. Most boards have between one and four USB slots, HDMI and composite video output, and a 3.5 mm phone jack for audio. Lower level output is provided by a number of GPIO[5] pins which support common electronic protocols. The B-models have an 8P8C Ethernet port and the Pi 3 has on board Wi-Fi 802.11n and Bluetooth.

[1]System on a Chip, an integrated circuit that integrates all components of a computer or other electronic system into a single chip
[2]Central Processing Unit
[3]Graphics Processing Unit
[4]Secure Digital, is a non-volatile memory card format developed by the SD Card Association for use in portable devices
[5]General Purpose Input/Output, is a generic pin on an integrated circuit or computer board whose behavior (including whether it is an input or output pin) is controllable by the user at run time

- **SOC**: Broadcom BCM2837
- **CPU**: 4x ARM Cortex-A53, 1.2GHz
- **GPU**: Broadcom VideoCore IV
- **RAM**: 1GB LPDDR2 (900 MHz)
- **Networking**: 10/100 Ethernet, 2.4GHz 802.11n wireless
- **Bluetooth**: Bluetooth 4.1 Classic, Bluetooth Low Energy
- **Storage**: microSD
- **GPIO**: 40-pin header, populated
- **Ports**: HDMI, 3.5mm analogue audio-video jack, 4x USB 2.0, Ethernet, Camera Serial Interface (CSI), Display Serial Interface (DSI)

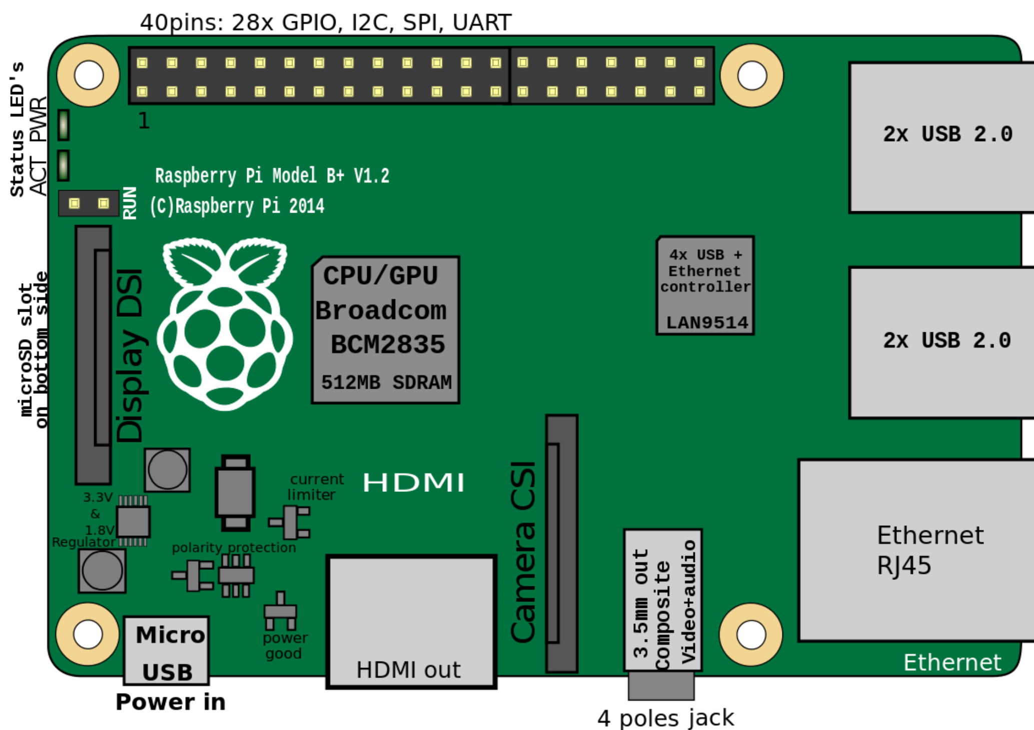


Figure 5.1: Raspberry Pi single-board computer

5.3.1.2 Software

The following software is installed in the node to fullfil it's funtional requirements.

5.3.1.2.1 Raspbian Raspbian is a free operating system based on Debian optimized for the Raspberry Pi hardware.

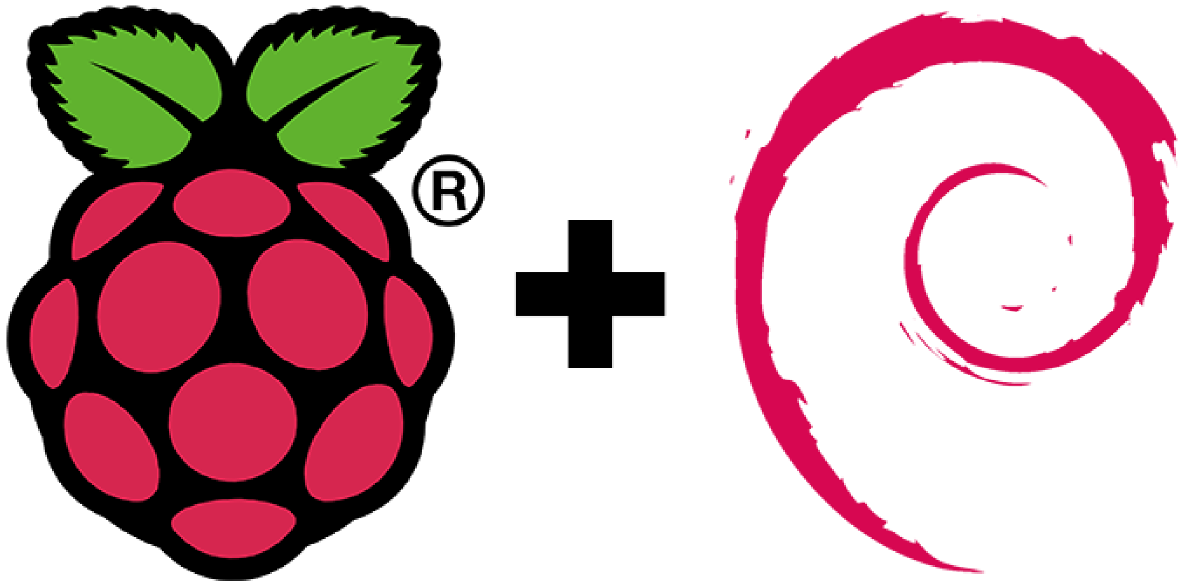


Figure 5.2: Raspbian logo

5.3.1.2.2 Eclipse Mosquitto Eclipse Mosquitto™ is an open source[^Mosquitto-license] message broker that implements the MQTT protocol versions 3.1 and 3.1.1. MQTT provides a lightweight method of carrying out messaging using a publish/subscribe model. This makes it suitable for "Internet of Things" messaging such as with low power sensors or mobile devices such as phones, embedded computers or microcontrollers like the Arduino.

5.3.2 NodeMCU DEVKIT / D1 mini

NodeMCU is an open source IoT platform. It includes firmware which runs on the ESP8266 Wi-Fi SoC and hardware which is based on the ESP-12 module. The term "NodeMCU" by default refers to the firmware which uses the Lua scripting language. The Development Kit integrates GPIO, PWM, I2C, 1-Wire and ADC in one board.

The D1 Mini is a mini wifi board based on ESP-8266EX. It features 11 digital input/output pins, 1 analog input and a Micro USB connection. All of the IO pins (except D0) support interrupt/PWM/I2C/one-wire.

Figure 5.3: Mosquitto logo

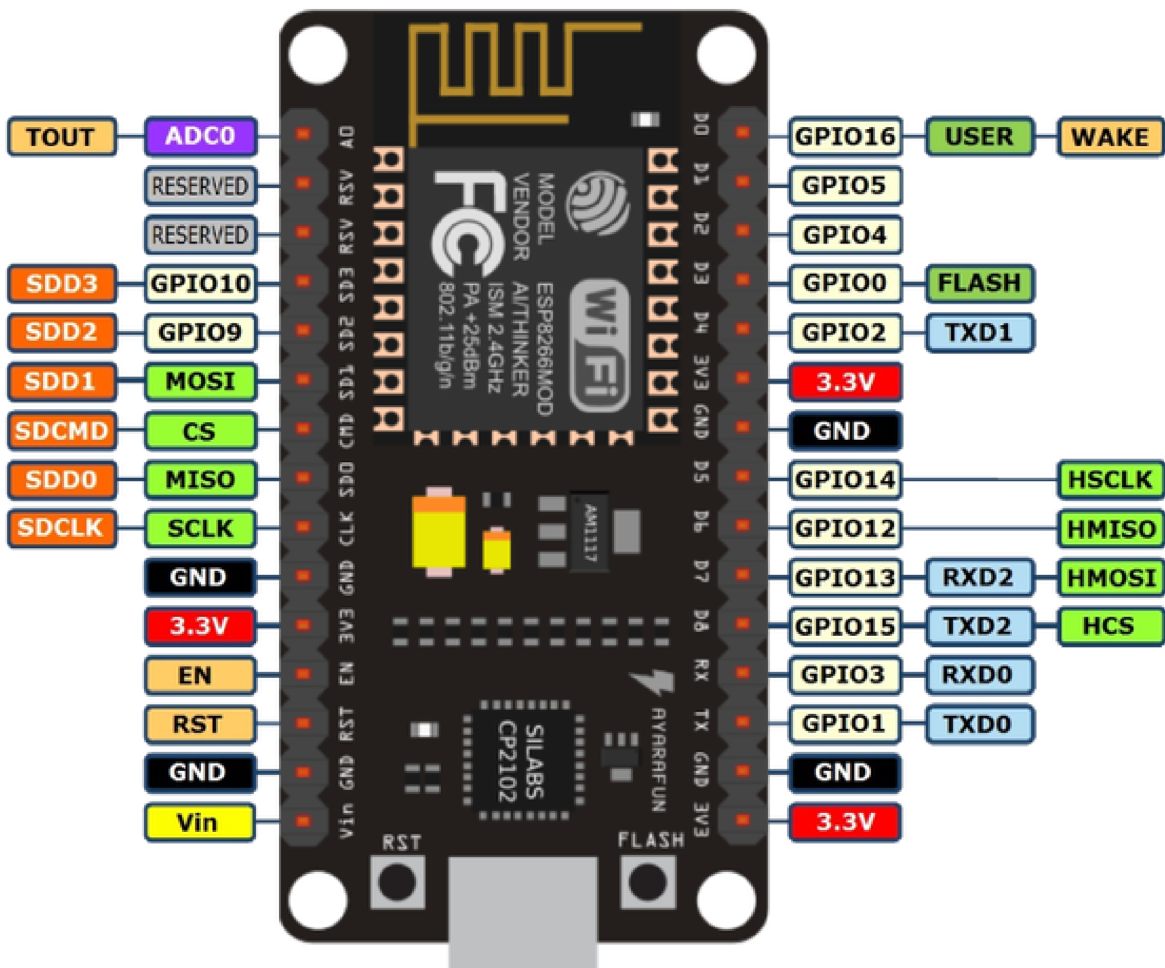


Figure 5.4: NodeMCU development board

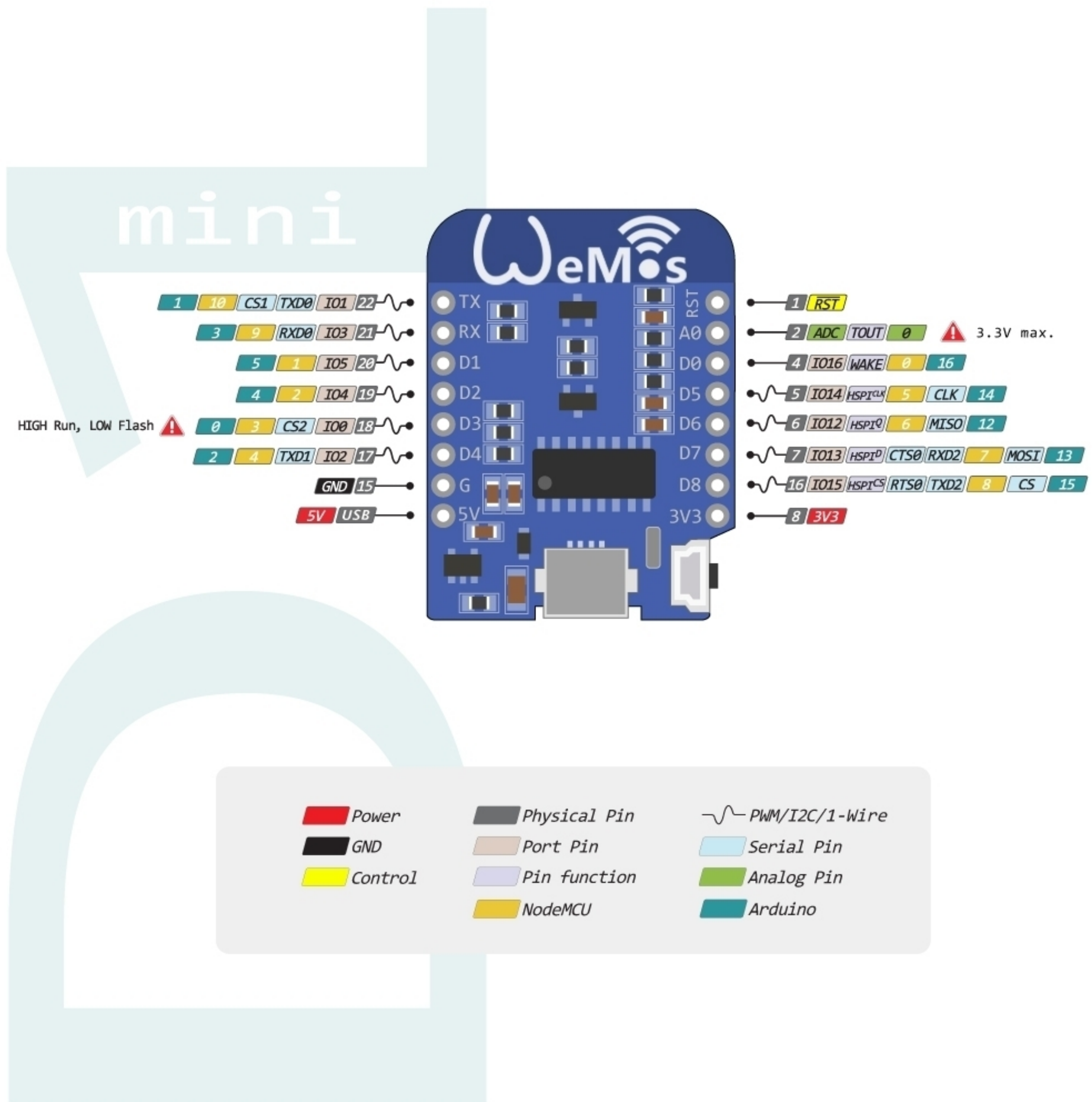


Figure 5.5: D1 Mini development board

5.3.2.1 Hadware

The *ESP8266* WiFi Module is a self contained SOC produced by Espressif Systems with integrated TCP/IP protocol stack and MCU[6] capability that can provide access to your WiFi network. The *ESP8266* is capable of either hosting an application or offloading all Wi-Fi networking functions from another application processor through its AT command set.

As Arduino began developing new MCU boards based on non-AVR processors like the ARM/SAM MCU, they needed to modify the Arduino IDE so that it would be relatively easy to change the IDE to support alternate tool chains to allow Arduino C/C++ to be compiled down to these new processors. They did this with the introduction of the Board Manager and the SAM Core. A “core” is the collection of software components required by the Board Manager and the Arduino IDE to compile an Arduino C/C++ source file down to the target MCU’s machine language. Some creative ESP8266 enthusiasts have developed an Arduino core for the ESP8266 WiFi SoC that is available at the ESP8266 Core[7] webpage.

- **SOC**: Espressif ESP8266
- **CPU**: 32-bit RISC Tensilica Xtensa LX106 running at 80 MHz (CPU can be overclocked to 160 MHz)
- **RAM**: 64 KiB of instruction RAM, 96 KiB of data RAM
- **Storage**: External QSPI flash - 512 KiB to 16 MiB (flash can be overclocked from 40 MHz to 80 MHz)
- **Networking**: IEEE 802.11 b/g/n Wi-Fi with integrated TR switch, balun, LNA, power amplifier and impedance matching circuit
- **GPIO**: 16 GPIO pins
- **Ports**: SPI[8], I2C[9], I2S[10] interfaces with DMA[11], UART[12] on dedicated pins plus

[6] Micro Controller Unit
[7] https://github.com/esp8266/Arduino [[^POE]]: Power over Ethernet
[8] Serial Peripheral Interface, is a synchronous serial bus interface specification used for short distance communication, primarily in embedded systems
[9] Inter-Integrated Circuit, is a synchronous serial bus interface specification used for short distance communication, primarily in embedded systems
[10] Integrated Interchip Sound, is a synchronous serial bus interface standard used for connecting digital audio devices together
[11] Direct Memory Access, is a feature of computer systems that allows certain hardware subsystems to access main system memory independently of the CPU
[12] Universal Asynchronous Receiver/Transmitter, is a computer hardware device for asynchronous serial communication in which the data format and transmission speeds are configurable

a transmit-only UART can be enabled on GPIO2, PWM[13] and one 10-bit ADC[14]

5.3.2.2 Software

Custom firmware based on the Arduino ESP8266 Core. Main features include:

- Parameterizable.
  - Node information
  - Network connection
  - MQTT connection
  - Input/Output configuration
- Persistence
  - The set values are stored in flash memory
  - Configuration changes are applied after a restart of the node
  - If the configuration is invalid or nonexistent during startup, default values are initialized.
- Operating Modes.
  - CONFIGURATION
  - STANDALONE
  - NETWORKED
- OTA[15] updates
- Web server or REST[16] API[17]

5.3.2.2.1 Libraries

- **Async MQTT client**, an asynchronous MQTT client implementation for ESP8266 Arduino.
- **ESPAsyncWebServer**, async HTTP and WebSocket Server for ESP8266 and ESP31B Arduino.
- **ESPAsyncTCP**, a fully asynchronous TCP library, aimed at enabling trouble-free, multi-connection network environment for Espressif's ESP8266 MCUs.

[13] Pulse-Width Modulation, is a modulation technique used to encode information into a pulsing signal or to control the amount of power sent to a load
[14] Analog-to-Digital Converter, is a system that converts an analog signal into a digital signal
[15] Over-the-air programming
[16] Representational State Transfer
[17] Application Programming Interface

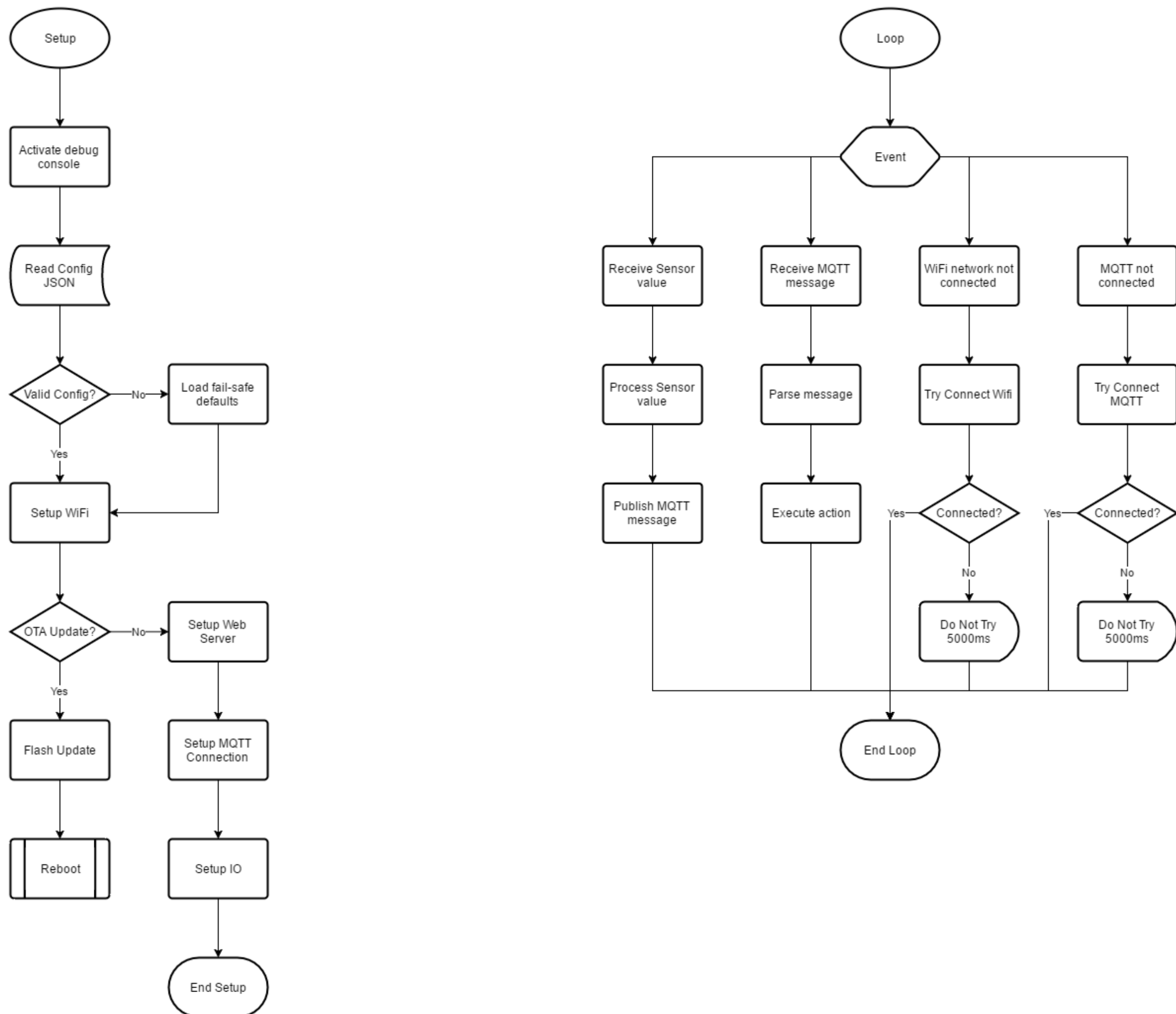


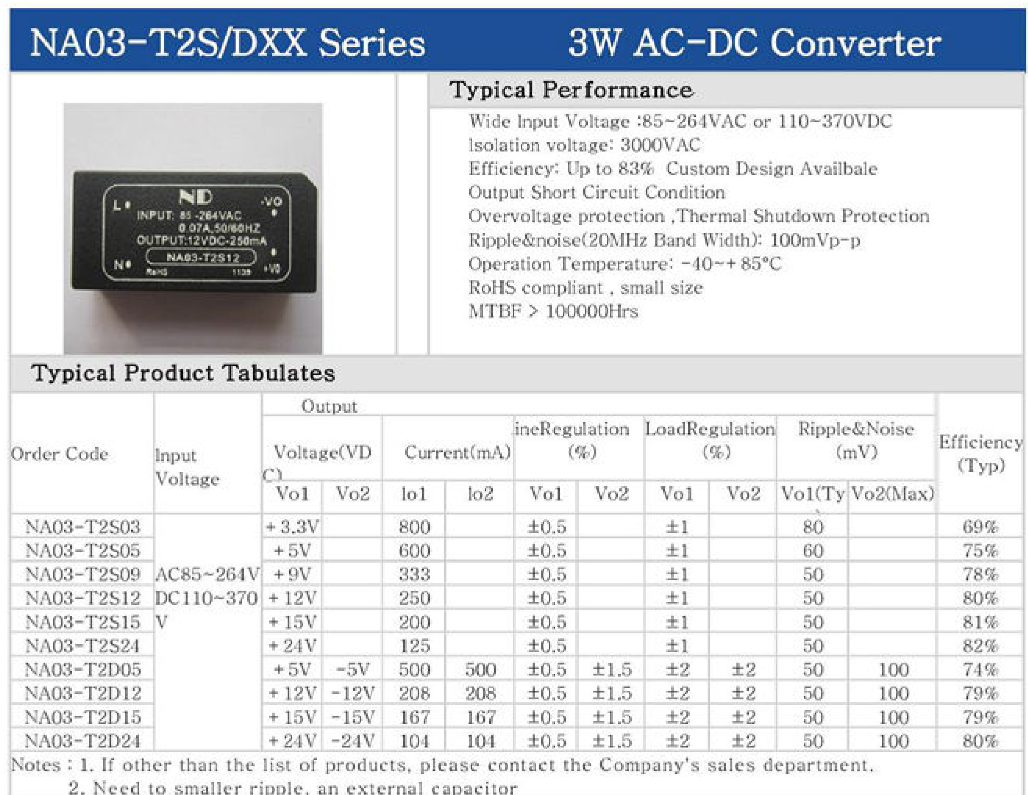
Figure 5.6: Flowchat

5.3.3 Additional electronics

Depending on the final application of each node, you may need different interfaces between the node inputs/outputs and the sensors/actuators. In this section you can find the most common circuitry needed.

5.3.3.1 Power supply

For gerenal purpose indoor *ESPutnik* nodes, power supply is provived a 220V AC to 5V isolated converter.



NA03-T2S/DXX Series 3W AC-DC Converter

Typical Performance

Wide Input Voltage :85~264VAC or 110~370VDC
Isolation voltage: 3000VAC
Efficiency: Up to 83% Custom Design Availbale
Output Short Circuit Condition
Overvoltage protection ,Thermal Shutdown Protection
Ripple&noise(20MHz Band Width): 100mVp-p
Operation Temperature: -40~+85°C
RoHS compliant , small size
MTBF > 100000Hrs

Typical Product Tabulates

Order Code	Input Voltage	Output										Efficiency (Typ)
		Voltage(VDC)		Current(mA)		ineRegulation (%)		LoadRegulation (%)		Ripple&Noise (mV)		
		Vo1	Vo2	Io1	Io2	Vo1	Vo2	Vo1	Vo2	Vo1(Ty	Vo2(Max)	
NA03-T2S03		+3.3V		800		±0.5		±1		80		69%
NA03-T2S05		+5V		600		±0.5		±1		60		75%
NA03-T2S09	AC85~264V	+9V		333		±0.5		±1		50		78%
NA03-T2S12	DC110~370	+12V		250		±0.5		±1		50		80%
NA03-T2S15	V	+15V		200		±0.5		±1		50		81%
NA03-T2S24		+24V		125		±0.5		±1		50		82%
NA03-T2D05		+5V	-5V	500	500	±0.5	±1.5	±2	±2	50	100	74%
NA03-T2D12		+12V	-12V	208	208	±0.5	±1.5	±2	±2	50	100	79%
NA03-T2D15		+15V	-15V	167	167	±0.5	±1.5	±2	±2	50	100	79%
NA03-T2D24		+24V	-24V	104	104	±0.5	±1.5	±2	±2	50	100	80%

Notes : 1. If other than the list of products, please contact the Company's sales department.
2. Need to smaller ripple, an external capacitor

Figure 5.7: NA03-T2S05 power supply module

For nodes that require to have backup power, a lithium 18650 cell with a battery chaging circuit will do. If the node is outdor, i.e. barometric pressure sensing, a photovoltaic cell can be added to the battery charger so it will not need a 220V outlet.

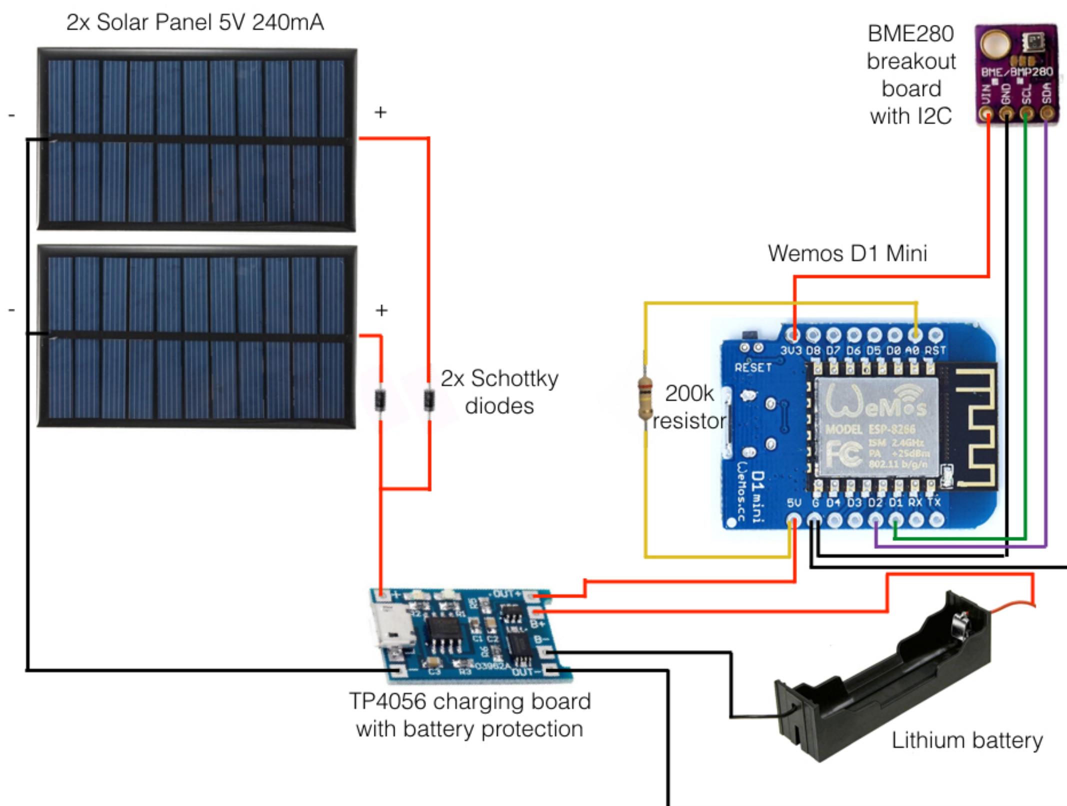


Figure 5.8: Solar+Battery powered *ESPutnik* node

5.3.3.2 Output interface

To drive different voltage or high current loads you need an adaptation circuit, the most simple and reliable is relay based. To further isolate the control signal from the load alimentation, it is advisable to use always optocoupler relay modules. To get rid of the characteristics relay sound when switching is advised to use a solid state relay.

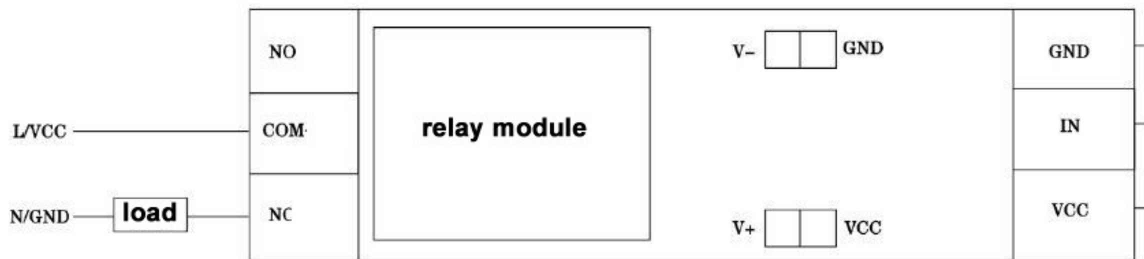


Figure 5.9: Relay module

5.3.3.3 Input interface

Again I will be using optocouplers to isolate input signal at different voltage levels. If the signal is AC, a rectifier circuit is needed.

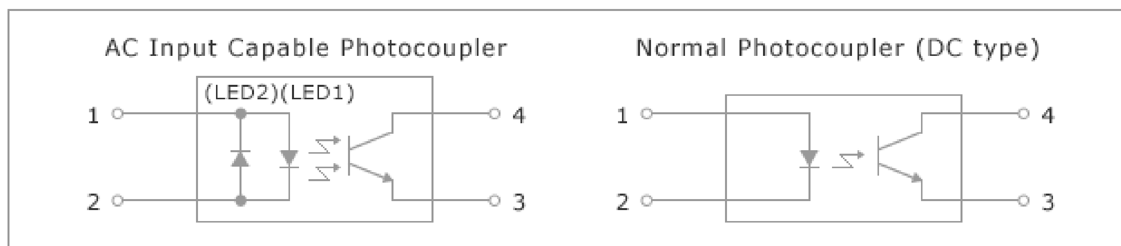


Figure 5.10: Optocoupler

If we need to transform analog input to a digital one based on a voltage threshold, a schmitt trigger allow us to control the set and reset values for the digital signal.

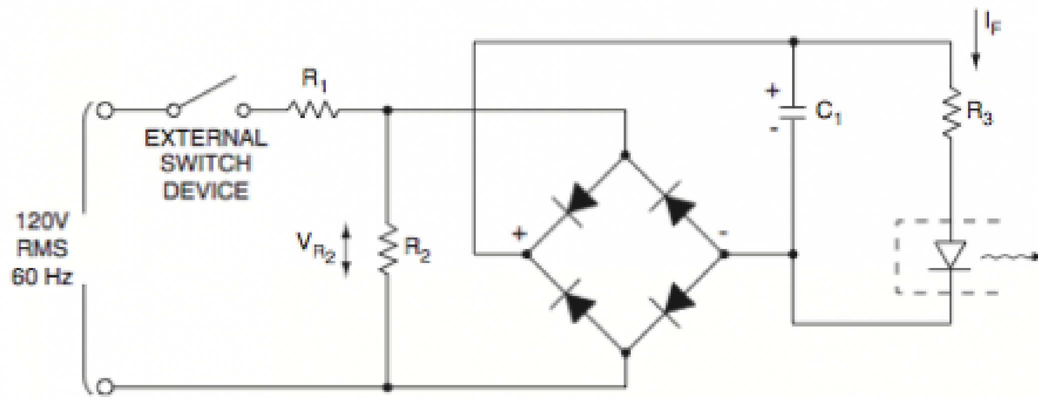


Figure 5.11: Rectifier before the optocoupler

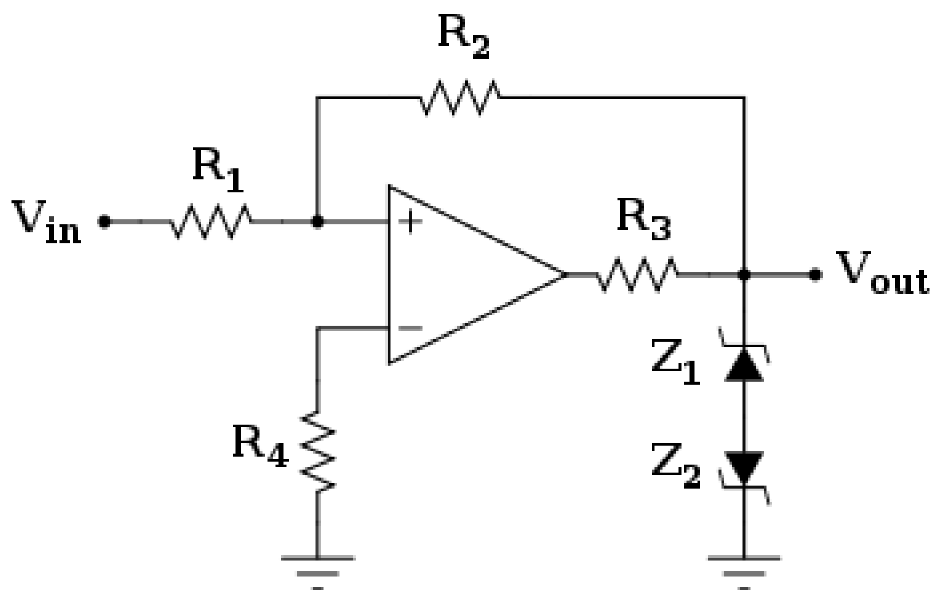


Figure 5.12: Schmitt trigger

6 Conclusion

6.1 Results

We are able to control the actuators attached to different nodes either interacting directly with the regular control interfaces of the home, ie switches, or using a smartphone or tablet conneted to the same network as the *Musquetteer* node which is acting as a gateway between the regular home network and the home automation network.

6.2 Known limitations

- It is only implemented communication over the MQTT protocol.
- Remote control interface is only available to smartphone and tablet devices.
- Area of control is limited to the range of the wifi network.
- The amount of controlled end devices per *ESPutnik* node is rather small.

6.3 Future improvements

- To reduce cost and increase network reliability, *Musquetteer* nodes can be migrated to a router running openWRT.
- To overcome the risk of having a central point of failure, mosquitto broker software should be remplazed by other broker which accepts clustering.
- To increase interoperability, openHAB software can be installed and configured in the *Musquetteer* nodes.
- To extend the wifi area coverage without additional hardware, a mesh network could be used.

Bibliography

[BG14] Andrew Banks and Rahul Gupta. *MQTT Version 3.1.1*. Oct. 29, 2014. URL: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.pdf.

[Hiv15] HiveMQ. *MQTT essentials*. 2015. URL: http://www.hivemq.com/mqtt-essentials/.

[ISO16] 20922 ISO/IEC. *Information technology – Message Queuing Telemetry Transport (MQTT) v3.1.1*. June 15, 2016. URL: http://www.iso.org/iso/catalogue_detail.htm?csnumber=69466.

[Mos16] Mosquitto. *MQTT v5 draft features*. Aug. 15, 2016. URL: https://mosquitto.org/2016/08/mqtt-v5-draft-features/.

[Pip13] Andy Piper. *MQTT community wiki*. Dec. 2, 2013. URL: https://github.com/mqtt/mqtt.github.io/wiki.

[Sta15] James Stansberry. *MQTT and CoAP: Underlying Protocols for the IoT*. Oct. 2015. URL: http://electronicdesign.com/iot/mqtt-and-coap-underlying-protocols-iot.

APPENDIX 1

APPENDIX 2