

Master thesis : MQTT broker with in-line, real-time data visualiser for the Internet of Things (IoT)

Auteur : Detienne, Martin

Promoteur(s) : Leduc, Guy; 12788

Faculté : Faculté des Sciences appliquées

Diplôme : Master en sciences informatiques, à finalité spécialisée en "computer systems security"

Année académique : 2021-2022

URI/URL : <http://hdl.handle.net/2268.2/14496>

Avertissement à l'attention des usagers :

Tous les documents placés en accès ouvert sur le site le site MatheO sont protégés par le droit d'auteur. Conformément aux principes énoncés par la "Budapest Open Access Initiative"(BOAI, 2002), l'utilisateur du site peut lire, télécharger, copier, transmettre, imprimer, chercher ou faire un lien vers le texte intégral de ces documents, les disséquer pour les indexer, s'en servir de données pour un logiciel, ou s'en servir à toute autre fin légale (ou prévue par la réglementation relative au droit d'auteur). Toute utilisation du document à des fins commerciales est strictement interdite.

Par ailleurs, l'utilisateur s'engage à respecter les droits moraux de l'auteur, principalement le droit à l'intégrité de l'oeuvre et le droit de paternité et ce dans toute utilisation que l'utilisateur entreprend. Ainsi, à titre d'exemple, lorsqu'il reproduira un document par extrait ou dans son intégralité, l'utilisateur citera de manière complète les sources telles que mentionnées ci-dessus. Toute utilisation non explicitement autorisée ci-avant (telle que par exemple, la modification du document ou son résumé) nécessite l'autorisation préalable et expresse des auteurs ou de leurs ayants droit.



University of Liège - School of Engineering and Computer Science

MQTT broker with in-line, real-time data visualiser for the Internet of Things (IoT)

Author
Martin Detienne

Advisors
Guy Leduc (ULiège)
Emmanuel Tychon (Cisco)

*A master's thesis presented in partial fulfillment of the requirements for the
"Computer Sciences" master degree by Martin Detienne*

Academic year 2021-2022

Abstract

Internet of Things (IoT) devices are more and more used in our today's life. However, as the amount of connected devices increases, the communications between them are more and more difficult to scale. The MQTT communication protocol has come to solve this issue. The MQTT protocol is a communication protocol between IoT devices that aims to centralise the communications inside one broker. However, a typical MQTT communication involves hundreds of messages per minute. This is then hard to analyse an MQTT communication with packet sniffer tools like Wireshark. The goal of this master thesis is to provide an appropriate and publicly available tool to visualise and analyse MQTT communications for debugging purposes. This report reviews the goals of this project: create several MQTT broker environments on the same system, display the set of published messages, the state of a broker, filter the published messages, and display telemetry on the number of published messages. This thesis report then goes through the MQTT protocol and its various types of packets. The high-level architecture is discussed with the first decisions for the role of each component that will be developed: a broker to record the MQTT data and a viewer to visualise it. We then decide on the type of database to store the recorded data persistently, the kind of data to record which is interesting to display to the user, and how this data should be stored. The architecture of the broker is reviewed as well as the technique used to manage the several independent MQTT broker environments. With this, we discuss how to save the events that happen in an MQTT broker and the techniques used to scale the project with a big amount of MQTT messages. To conclude the discussion about the architecture, we discuss the plugin system that allows extending the viewer, how the MQTT data should be stored and represented in the viewer, and how the user should have access to the required features. We then review how complex filters can be built into the viewer. We then discuss the development methodology of this project. The implementation details are then reviewed with the different technologies used in this project. We will talk about the difficulties encountered during the deployment and explanations on how to deploy this project easily. The results of this project are finally reviewed with the latencies encountered when the broker faces a lot of traffic and some screenshots of the viewer. A discussion is made about the further possible improvements on this project. Additional documents and documentation are provided at the end of this report.

Contents

1	Introduction	5
2	Goal	7
2.1	Purpose	7
2.2	Features	7
2.2.1	Message visualisation	7
2.2.2	MQTT broker state visualisation	8
2.2.3	Dynamic MQTT broker creation	9
2.2.4	Message filters	9
2.2.5	Telemetry	9
3	What is MQTT?	11
3.1	Overview	11
3.2	MQTT packet types	11
3.2.1	CONNECT	11
3.2.2	PUBLISH	13
3.2.3	SUBSCRIBE	16
3.3	Usage	18
4	Architecture	19
4.1	High level architecture	19
4.2	The database	20
4.2.1	Data representation	20
4.2.2	To a document-oriented database	22
4.2.3	Database indexes	24
4.2.4	Atomicity	25
4.3	The broker	25
4.3.1	Independent brokers	25
4.3.2	Broker architecture	27
4.3.3	Events	28
4.3.4	Traffic limitation	31
4.3.5	Volatile data	32
4.4	The Viewer	32
4.4.1	Data representation	33
4.4.2	How to update the broker state	34
4.4.3	Plugins	36
4.4.4	How to display the data	38
4.4.5	Filter building blocks	41
4.4.6	Complex composition of filters	43
5	Development methodology	49

6	Implementation details	53
6.1	The Database	53
6.1.1	Technology used	53
6.1.2	Indexes	53
6.1.3	Atomicity	53
6.2	The Broker	53
6.2.1	Technologies used	53
6.2.2	How to generate a broker ID	54
6.2.3	Broker Architecture	54
6.2.4	MQTT Events	54
6.3	The Viewer	55
6.3.1	Technology used	55
6.3.2	React components tree	55
6.3.3	Plugins	55
6.3.4	Filters	56
6.3.5	Scalability	56
7	Deployment	57
7.1	MQTT Broker	57
7.1.1	Manual installation	57
7.1.2	Docker	58
7.1.3	Configuration	59
7.2	MQTT Viewer	59
7.2.1	Manual installation	59
7.2.2	Docker	60
7.2.3	Configuration	60
8	Results	61
8.1	Performances of the broker	61
8.1.1	MQTT Publish latency	61
8.1.2	API latency	62
8.2	Viewer screenshots	62
9	Possible improvements and limitations	67
9.1	The MQTT username	67
9.2	Message modification	67
9.3	Plugins	67
9.4	Replica sets	68
9.5	Embedded MQTT clients in the viewer	68
10	Additional documents	69
10.1	Documentation	69
10.2	Deployment example	69

10.3 Docker images	69
10.4 Gitlab repositories	69
11 Conclusion	71

1 Introduction

Internet is more and more used to connect devices of our daily life. Thermometers, solar panels, traffic lights, cameras, ampules, etc may need to transmit and receive data with other devices. For instance, the heaters of a house need the data produced by the thermometers to heat the house correctly. However, those devices may be separated by large distances and make communications difficult. The most used solution in those recent years to solve this issue is to rely on the already-in-place network: the Internet. The devices will simply have to be connected at the edges of the internet and communicate their information between each others using the infrastructures and the routing of the internet. Those devices are the so-called "Internet of Things" devices.

While the devices do not need to manage the infrastructures and the routing of a network, they face another problem: the communication protocol. Indeed, each pair of devices needs to establish a protocol of communication between them in order to have comprehensive, reliable, and consistent communication. And this is where this scheme faces an exponential complexity: each **pair** of devices has to establish a protocol. A device which may want to transfer data to other devices will have to establish N protocols and then increase the complexity of its development. This scheme is also very static: what is happening when a completely new heater joins the system and wants to take the information of a thermometer? Furthermore, all those communications will have to be supported by the internet infrastructure which could be overwhelmed by this exponential amount of traffic with the number of IoT devices. But most of all, the devices have often a limited amount of battery while a lot of the traffic may be redundant (for instance, when a thermometer sends the temperature to multiple heaters). It would then be a waste of electrical power to replicate data that should be sent to several peers. In this context, the MQTT protocol has been created.

The aim of the MQTT protocol is to provide a unified protocol to handle the communication between a lot of IoT devices. They do not need to communicate with each other, they just need to communicate with a central server called the MQTT broker. The protocol works like the mailing list with the emails. Each device decides to subscribe or not to a mailing list called a "topic" and then receives the messages published on this topic (figure 1). In this way, the IoT devices will be able to reduce the complexity of the development of their communication protocol, their power consumption by sending only one message while there are several recipients, and not overload the internet network.

The MQTT protocol is very well designed to handle a lot of traffic from a huge amount of clients. But a drawback comes with this scalability: the interpretation of the traffic. Indeed, developers and engineers may want to debug and analyse the communications between IoT devices. The amount of traffic and the physical distances make the MQTT traffic hard to analyse with network sniffing tools such as Wireshark or tcpdump. Those tools provide useless information (as we are only interested in MQTT) and do not provide

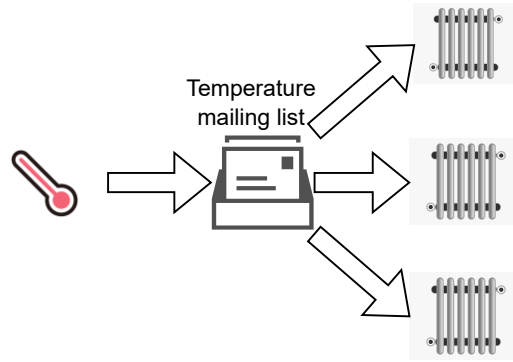


Figure 1: High level view of the idea behind MQTT.

adapted filters to analyse the traffic correctly. Furthermore, it is hard to understand the current state of an MQTT broker. For instance, this is hard to know if a device is subscribed to a topic except if we catch the packet corresponding to a subscription of this client.

In this context, the need for a dedicated and visual tool to analyse and understand the messages exchanged by IoT devices through the MQTT protocol is high. Like POSThere (a public tool to analyse POSTed HTTP requests), we need a tool to visualise the content of each message and to understand the communications between the MQTT clients.

2 Goal

2.1 Purpose

The purpose of this project is to provide a visual of what happened in an MQTT broker. The goal is to develop a modified MQTT broker which is able to record recent events and an interface used to visualise and analyse the recent MQTT messages that go through it. The typical usage of such a tool is for debugging purposes to verify if the IoT devices using MQTT communicate properly.

A typical example of an application that we should be inspired by is POSThere. POSThere provides a way to publish HTTP POST requests on an URL and we are able to get the posted data with an HTTP GET request. This tool allows analysing and debugging APIs that use the HTTP protocol. POSThere provides useful information about what has been posted such as: the body of the POST messages, the query parameters, and the header. The idea is to do the same with the MQTT protocol.

We want an MQTT visualiser where it is possible to see the list of published messages and interesting data linked to it. As the MQTT protocol may have a complex state with all the client subscriptions to the topics, we also want to have an overview of this state. Further explanations of what we want and requirements are listed below.

2.2 Features

2.2.1 Message visualisation

The first main feature is to be able to visualise the content of MQTT messages going through the MQTT broker. We want a list of messages that have been published ordered by the time when they have been published. The important information about each message should be displayed.

The view of the content of a message is the priority. In MQTT, the content of a message is not restricted to special formatting. The IoT devices can then choose the character encoding and the scheme of their messages. This is the role of the MQTT clients to agree on a format understood by all clients. The widely used format are the JSON and XML formats. Our tool would need to have a special display to visualise those widely used formats.

The display of each message should also identify the sender of this message as well as the list of receivers. Indeed, as a client posts a message on a "mailing list", the other subscribed clients will receive this message and we want to identify them. We also want to see in the message the mailing list on which it has been posted called the topic.

In addition, we also want to see the time at which the messages as been published on the MQTT broker and the QoS value associated (more information about the MQTT QoS in 3.2.2).

2.2.2 MQTT broker state visualisation

With only a list of messages, we miss a lot of information about the events that occur in an MQTT broker and its current state. For instance, we do not know easily if the heaters are subscribed to the "temperature" topic and receive all the updates of a thermometer. This information can be found in the set of messages by verifying if each temperature message has been received by each heater but this process is complicated and not very visual.

We then want a feature where we are able to visualise the state of the MQTT broker at a given moment: who is sending messages on which topics, who is receiving messages from which topics, who is subscribed to which topics, and which client is currently connected to the broker. Those information needs to be visual and easily understandable, an example of a possible visualisation is in figure 2.

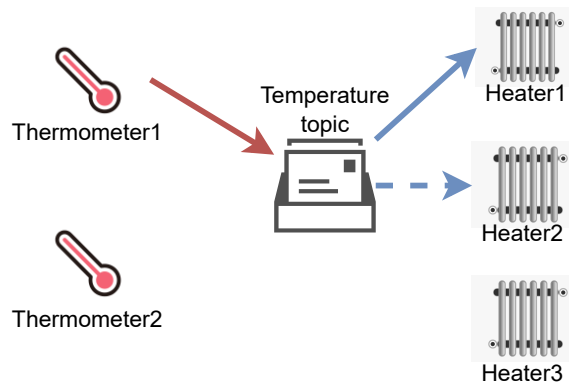


Figure 2: Example of the state of a broker. The red arrows represent the flows of messages that has been published on a topic in the past, the blue arrows represent the flows of forwarded messages to the subscribed clients, and the dashed blue arrows represent the flows of messages forwarded to the clients which are currently subscribed to this topic.

In this example, without having to search on the list of published messages, we see that the Thermometer2 does not send anything, the Heater3 does not receive anything, the Heater1 has received temperature data but is not subscribed so it will not receive data anymore, and only Thermometer1 and Heater2 work correctly.

2.2.3 Dynamic MQTT broker creation

We want our tool to be publicly available. All the users will use our broker to test and analyse their MQTT traffic. The problem with this scheme is that the usage of a user will affect the usage of others. For instance, if a user uses a topic named "Temperature", the other users will then see this usage and this is not what we want. We want independent environments for each user.

The users should then be able to create on the fly a new MQTT broker to get a new environment when he wants. Those created brokers need to be independent as if the user was using the tool alone. Then, a user publishing on the "Temperature" topic should not affect the usage of another user who is also using a "Temperature" topic in another MQTT environment.

2.2.4 Message filters

One big problem with the MQTT protocol is the number of published messages. IoT devices typically send hundreds of messages in a minute and this makes the content of messages difficult to analyse. The user should then be able to select the kind of messages he is interested in.

The creation of filters is then one important feature. We want to be able to filter the messages by setting the senders, topics, or even the content of the messages so that the user can easily find an MQTT message if the flow of messages is too big and the messages difficult to find.

2.2.5 Telemetry

When analysing a flow, we may need to understand how evolves the amount of traffic over time. This is exactly the goal of telemetry. This feature aims to provide a visualisation of the number of messages that are sent at given a time and the evolution of this number.

This feature could be useful to analyse the correctness of a protocol having several states depending on the time. For instance, solar panels may want to send less information during the night to save power and because they do not have useful information to send as there is no sun. This feature would help to figure out if it is indeed the case.

3 What is MQTT?

3.1 Overview

The MQTT protocol [7] is a communication protocol between IoT devices. The architecture of an MQTT communication is composed of one central MQTT broker and any number of MQTT clients which are the IoT devices. The broker is composed of several "mailboxes" called topics where each client can subscribe and publish messages. Each of those clients connects to the broker, subscribes to any number of topics, and publishes messages to any number of topics. The clients, once they subscribe to a given topic, will receive every message sent on this topic during the period it is subscribed to it. In figure 3, we can see a typical exchange where a client subscribes to a topic and receives the published messages. The clients can unsubscribe from the topics to stop receiving messages that are published on this topic.

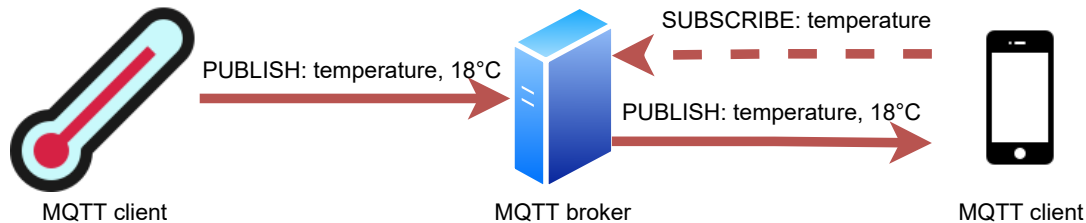


Figure 3: An example of MQTT exchange.

The MQTT protocol relies on the TCP/IP layer. The underlying TCP allows having a reliable connection with no duplicates where the packets are received in order during a TCP session [9]. The default port used by an MQTT broker to listen and communicate is the port 1883.

3.2 MQTT packet types

Several types of MQTT packets are necessary to manage an MQTT communication. Those are listed below.

3.2.1 CONNECT

The first thing an MQTT client needs to do to be involved in an MQTT communication is to connect to the MQTT broker. To do so, a CONNECT MQTT packet is sent by the MQTT client to the broker to connect to it. This packet contains the ID of the client, a clean session flag, and a keep-alive period. Additionally, the packet can contain a username, a password, and a last will. The broker acknowledges this connection with a CONNACK

packet. The client can stop the connection with the broker with a DISCONNECT packet.

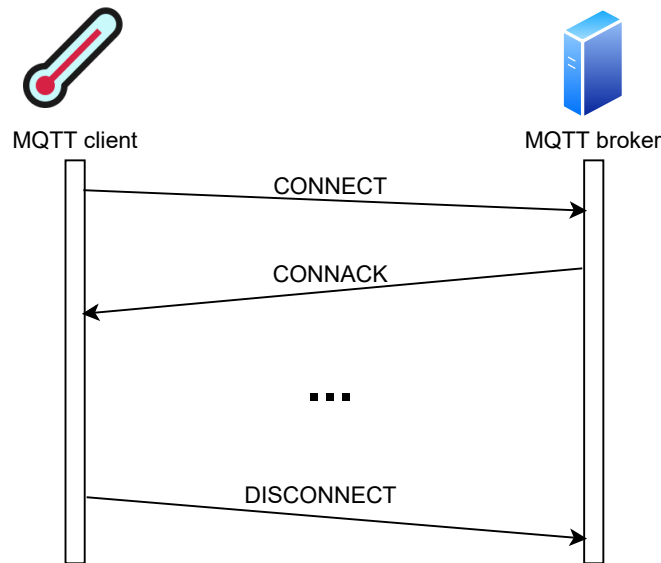


Figure 4: Example of a CONNECT exchange.

The goal of the CONNACK packets is to confirm to the client that the connection is accepted or not. If the connection is refused, a code in the CONNACK packet will indicate to the client why the connection has been refused. For instance, if the client tries to connect with an ID already used by another client, this issue will be communicated through a code (the code 0x02 in this case).

When a client wants to disconnect from the broker, it should send a DISCONNECT packet. The broker and the client should then close the TCP connection. If the TCP connection is closed or stopped before the reception of a DISCONNECT packet, the connection is considered as unexpectedly closed.

ID The ID is a string used to identify each client and must be unique, several clients cannot have the same ID at the same time.

Clean Session The clean session flag is used to establish a non-persistent session. When the flag is true, the broker does not store anything from the previous sessions and does not resend missed messages with a quality of service value of 1 or 2 (additional information about QoS in 3.2.2).

Username and password The username and password fields are optional strings used for authentication purposes. Those can be used to simply allow an MQTT client to connect with the broker. Those fields are not ciphered.

Last Will It may happen that the underlying TCP connection between the broker and client stopped unexpectedly. Indeed, a DISCONNECT message has not been sent meaning that the disconnection was not intended. The problem with the central broker architecture is that the other clients do not have direct communication with other MQTT clients and then are not aware of their connection state. A heater could wait a long time for more data from a thermometer while this thermometer has crashed. We then need a mechanism to notify other peers that a client has crashed.

This is in this context that the last will message has come. This last will message is carried in the last will fields of CONNECT message. The last will fields (topic, QoS, message, and retain) are used to send a message on a given topic with a given message with the given QoS (3.2.2) if the MQTT client stops the connection unexpectedly. With this solution, our thermometer could tell to the broker since the connection to publish a given message on a "Status" topic to notify the heaters when the thermometer connection stops unexpectedly (the heaters need to be subscribed to this "Status" topic to receive this information).

Those fields are not mandatory. If they are not present in the CONNECT packet, no last will message will be published.

Keep alive period The keep-alive period is a field that identifies the period between each keep alive-packet. Those packets are used to maintain the connection. The clients will send PINGREQ packets and the MQTT broker will have to respond with PINGRESP packets.

3.2.2 PUBLISH

The PUBLISH messages are used by clients to publish messages on a topic on the broker. The MQTT broker will then forward the message to the subscribed MQTT clients with the same PUBLISH message. The PUBLISH packets contain a packet ID, the name of the topic where this message is published, a QoS, a retain flag, a duplication flag, and a payload. The receiver of a PUBLISH message will answer depending on the quality of service of the message.

A message published on a broker is typically volatile. Once the broker has been assured that the message has been delivered to all the subscribed MQTT clients (it depends on the QoS value of the message), the message will be forgotten by the broker. An MQTT broker should not forward a published message to an MQTT client who subscribed to the topic after the PUBLISH packet has been received.

Payload The payload of a message can be anything, the formatting of the content is application dependent. An MQTT client can send binary, JSON, and XML payloads and it is not verified by the broker. If they choose so, the MQTT clients can publish messages

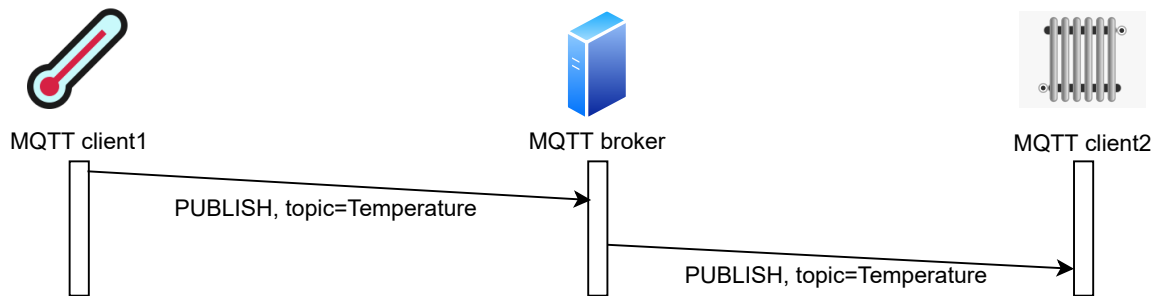


Figure 5: A simple example of PUBLISH packets.

in different formats on the same topic. This is the responsibility of the clients to agree on the formatting of the payloads.

Topic The topic field is a string used to identify the topic where this message will be published. Topics are "mailing lists" to which each client can subscribe and will receive the future messages published on it. The topics are identified by a string. The topic names are case-sensitive: a client subscribed to a "Data" topic will not receive messages published on the "data" topic. The name of the topics must have at least one character.

The topics are not "predefined" and cannot be explicitly created. A topic is "created" when a client starts to publish messages on it or if a client subscribes to this topic.

QoS As we said earlier, the MQTT protocol runs over the TCP transport layer. This TCP layer allows us to have a reliable connection between the broker and its clients. The packets will be transmitted in order, without duplicates, and without losses. However, the TCP layer does not guarantee anything between the TCP sessions.

A TCP connection can stop unexpectedly (if there is a problem in the network infrastructure or if the client crashes for instance). If the broker was transmitting a message to a subscribed client and the client crashes at this moment, the client will reboot, reconnect, and resubscribe to the topic but will miss the published message as the TCP protocol does not re-transmit between TCP sessions. For sensitive usages, we then need guarantees on the transmissions between TCP sessions. This is in this context that the Quality of Service field was introduced in the PUBLISH packets.

The QoS value specifies the quality of service with which this message should be delivered. The quality of service has 3 levels that are listed below.

QoS 0 When the QoS value is 0, the message is transmitted in best effort. It means that the message will be transmitted only once and will not be re-transmitted between TCP sessions. A typical exchange is shown in figure 5.

QoS 1 When the QoS is 1, the message will be delivered at least once to the MQTT broker/clients, the broker/clients will then acknowledge the packet with a PUBACK packet to notify the sender of the PUBLISH packet he received this packet. If the PUBACK packet is lost between 2 TCP sessions, the sender will not be notified that the packet has been transmitted and will then re-transmit it. That is why there is the possibility that the receivers receive this message twice. A typical example is shown in figure 6.

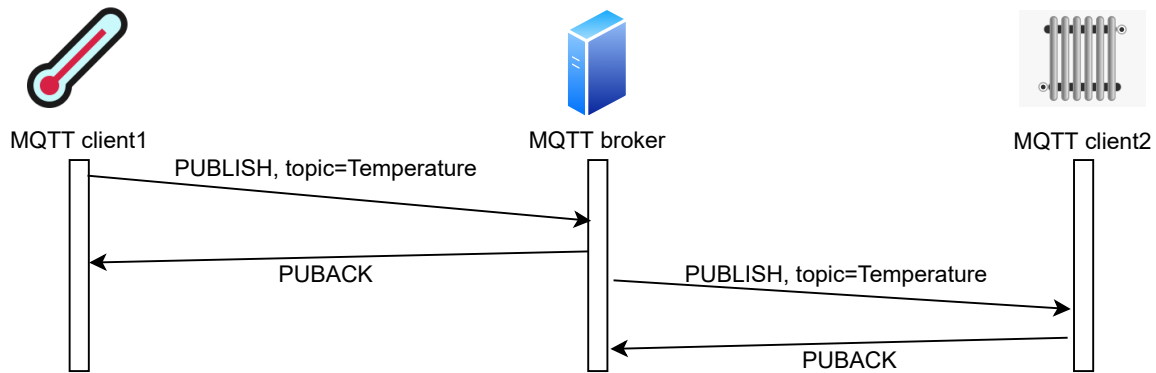


Figure 6: A simple example of PUBLISH packets in QoS 1.

QoS 2 With the QoS of 2, the message will be delivered exactly once. The receiver of the message will then need to be assured that the sender is aware of its reception before accepting it. Indeed, with the QoS 1, the message can be sent twice if the PUBACK packet is lost between 2 TCP sessions. We then need to be sure that the sender is aware that the message has been delivered and that this is not needed to resend it. The problem is similar to the closure of a TCP connection. This problem is mitigated thanks to a four-part handshake to acknowledge the PUBLISH packet (thanks to PUBREC, PUBREL, and PUBCOMP packets) that makes the possibility to have duplicates improbable. The receiver of a message with the QoS 2 will then wait until the end of the four-part handshake before accepting it. An exchange is shown in figure 7.

ID and duplicate flag and clean sessions The ID is used to identify the PUBLISH messages that have been acknowledged with PUBACK packets or PUBREC packets (the ID of the PUBLISH packet is used in the PUBACK and PUBREC packets to respond to it). When the sender replies to a PUBREC with a PUBREL packet, it also identifies this response with the same packet ID. The idea is the same with the PUBCOMP packets.

The duplicate flag indicates if the sender of the PUBLISH packet is trying to re-transmit it or if it is the first attempt to transmit it with the QoS 1 or 2.

The clean session flag used in the CONNECT packets (3.2.1) indicates if the previous PUBLISH messages with QoS 1 or 2 that failed to be delivered have to be re-transmitted. If the clean session is true, the quality of service value of saved messages

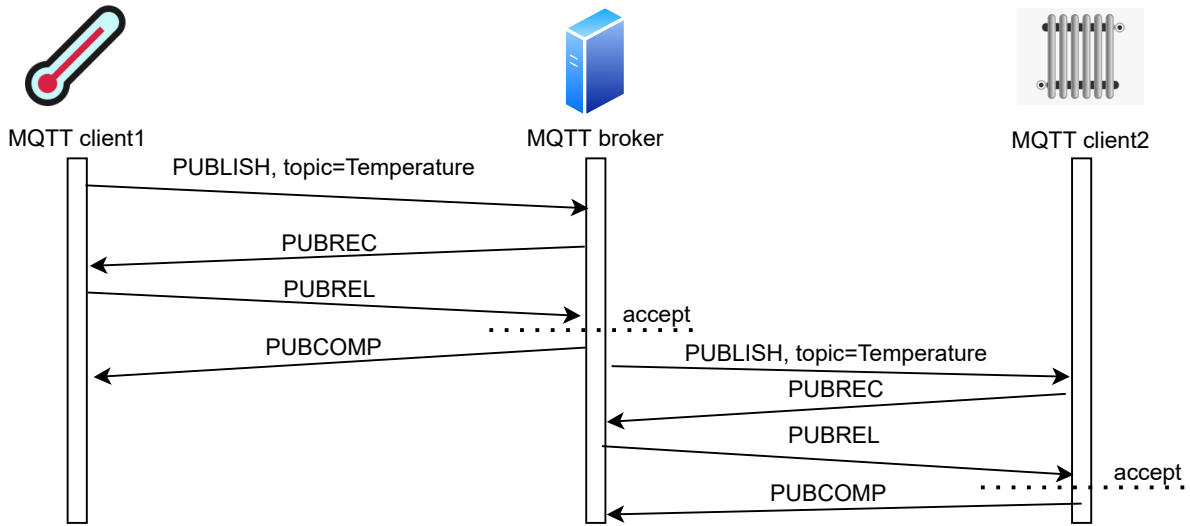


Figure 7: A simple example of PUBLISH packets in QoS 2.

to be re-transmitted (because the TCP session ended unexpectedly) will be ignored and the receiver will not receive those messages.

3.2.3 SUBSCRIBE

The SUBSCRIBE packet is sent by clients to notify the broker to forward messages from a given topic to itself. This packet contains a list of topics to subscribe to with an associated maximum QoS value for each subscription. From the moment when a client subscribes to a topic, the subsequently published messages on this topic will be forwarded to the subscribed client with PUBLISH packets. The broker acknowledges this subscription with a SUBACK packet. Clients can unsubscribe with UNSUBSCRIBE packets which are acknowledged with UNSUBACK packets. An SUBSCRIBE exchange is shown in figure 8.

Maximum QoS The maximum QoS value is used by a client to indicate to the broker that he wants to receive messages from a topic but with a maximum quality of service. For instance, with a message published with QoS 2 on a topic but with a client subscribed to this topic with a maximum QoS 1, the broker will forward the message with QoS 1. The QoS used to forward published messages to subscribed clients will be

$$\min\{publishQoS, subscriptionQoS\}$$

Wildcard topics Wildcard topics are a technique used to subscribe to several topics at once. This is a special kind of topic which is the union of other topics.

Let's take the example described in figure 9. There are several rooms with one thermometer in each of them. Each of those rooms has one or several heaters that fetch the

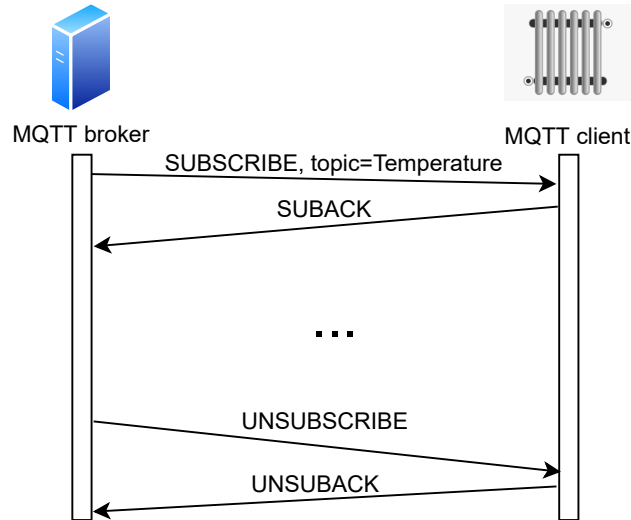


Figure 8: A simple example of SUBSCRIBE exchange.

temperature of the room they are in. In addition to the heaters, a terminal used to visualise the temperature in the whole building is also interested to fetch the temperature in each of the rooms. The creation of the subscription for each of the room topics may be very complex if there are a lot of rooms. Furthermore, this configuration is very static. We may want to subscribe to every topic that starts with "temperature" at once and we do not have to worry if there are hundreds of rooms or if the number of rooms changes over time. This is the idea behind the wildcard topics.

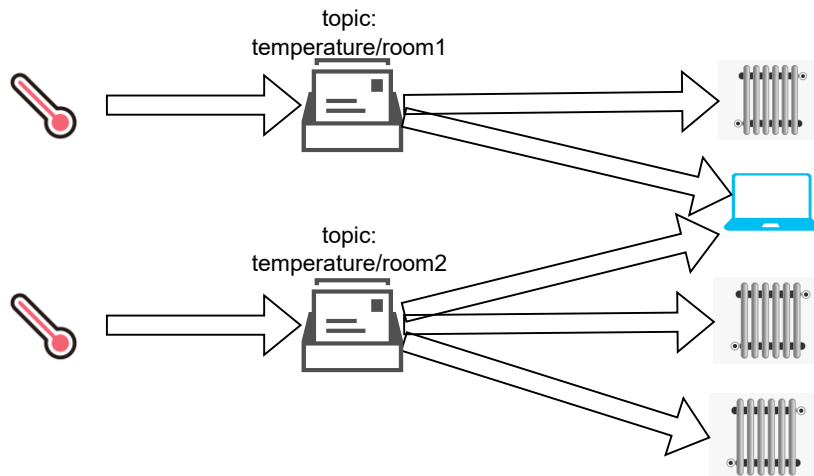


Figure 9: Example of MQTT a client which subscribes to a large range of similar topics.

A wildcard topic is a string in the "topics" field of the SUBSCRIBE packets (like every other subscription to normal topics) that contains a wildcard character. There are two

possible wildcard characters: # and +.

The character # comes at the after of the last separator (the character /) to match every topic that starts with the topic name before the separator. For instance, the wildcard topic `foo/#` will match the topics `foo`, `foo/bar`, or even `foo/bar/foo`. But it will not match topics starting by something else than `foo/` (`foo` will match but not the topic `foobar`). The wildcard topic # will match every topics. `foo#` or `foo/#/bar` are not valid wildcard topics.

The character + comes between two separators, after the last separator, or before the first one. It matches every topic which is the same as the wildcard string where we replace the + character with a string without separators. For instance, `foo+/bar` will match `foo/foo/bar` and `foo/example/bar`. But it will not match `foo/bar`, `foo/example/bar/foo`, or `foo/example/example/bar`. + will match every topic without separator like `foo` but not `foo/bar`. `foo+` or `+foo` are not valid.

The 2 wildcard characters can be combined and + can be used several times in a topic. For instance `+/+/#` will match every topic with at least one separator.

3.3 Usage

The typical usage of the MQTT protocol is to handle the communications between IoT devices. It allows devices to create only one connection with the broker instead of creating multiple connections with each device. The topics are used to filter communications that other devices are not interested in.

This allows scalable communication with a big amount of IoT devices without overwhelming the network. This also allows the devices to save their limited amount of energy for communication. Indeed, this is not the job of a device to duplicate every data for every device that is interested in the data. This solution is also less complex to configure as the clients do not need to worry about the number of other peers and their identity. The topics can easily be used to filter the information which is needed by some devices but useless for others. This communication protocol also allows big flexibility as the content of the messages is not restricted to one special formatting or encoding.

4 Architecture

4.1 High level architecture

To build the high-level architecture, we need to think about the requirements of the project: we want an application which is able to display the events that occurred in an MQTT broker in an understandable way (2). To do so, we need to:

1. Catch the events that happened in a running MQTT communication;
2. Record those events persistently;
3. And display those events in a way which is useful to the user.

We can then define a high-level architecture in 3 parts: a broker that carries the MQTT communications and catch the important information that happened in this communication, a database to save those events, and a viewer to visualise the important information.

The broker is responsible to handle the MQTT connections and recording the events which happen in the broker. In other words, the broker will run an MQTT broker instance to run the MQTT protocol such that MQTT clients can communicate through it. When interesting events happen (when there is a SUBSCRIBE, PUBLISH, or CONNECT packet arriving for instance), the data is recorded and saved on the database. The broker is also responsible for transmitting recorded data to the viewer through an API.

The viewer will be responsible to present the data in an understandable way. This is the viewer which will be in charge to handle all the features required for the goal of the project (2): displaying the list of messages that have been published in the broker, displaying the broker state, the telemetry, etc.

For efficiency purposes, the load of the viewer should rely on the client-side to not affect the performance of the broker. The big point of this architecture is that the code that computes what should be displayed on the viewer should be executed client-side (in a web browser for instance).

With this decision, we then need a component between the database and the viewer to fetch and communicate the recorded events. Indeed, the viewer should not be able to fetch data directly from the database since the code is executed client-side. This may cause severe security issues (for instance a malicious user could modify the code client-side to build a query that drops the database). In order to facilitate the deployment, it has been decided that the broker will integrate this component to minimise the number of programs to deploy. The broker is now also responsible to fetch the data from the database when the viewer asks for it. The viewer will query the data through an API on the broker and the broker will give back the asked data.

In this figure 10, we can see the high-level architecture and the communications, the MQTT client is not part of this project and is present for the example.

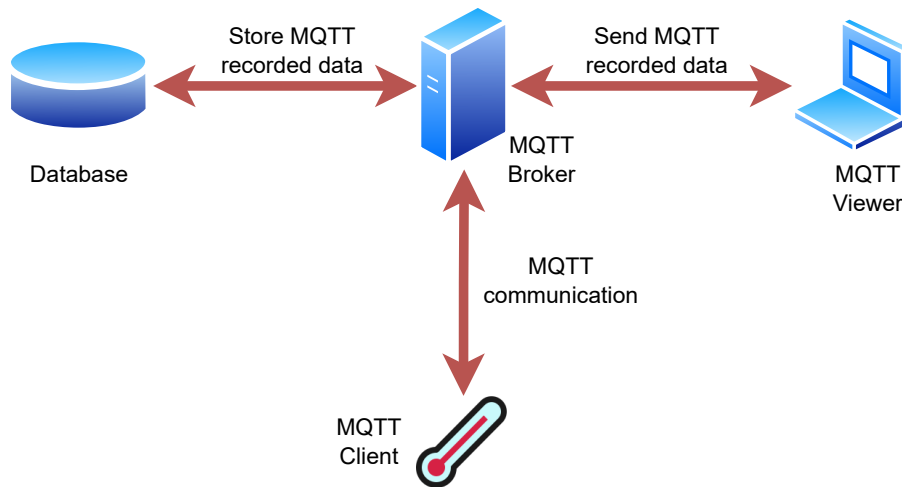


Figure 10: High level architecture of the project.

4.2 The database

This project consists of recording interesting events in an MQTT broker and displaying it to the user in a convenient way. The first thing that we have to do is to have a good understanding of the data we want to store and display to the user.

We will then see in this section what kind of data is important and how this data is stored. We will talk about the architectural decisions made on the database.

4.2.1 Data representation

The abstract entity-relation representation of the MQTT data contained in the database is represented in figure 11. This is important to notice that our "broker" and database need in fact to manage several MQTT brokers independently. This means that the same program will handle the events of several independent MQTT brokers. The way the encapsulation between MQTT brokers is done is explained in 4.3.1. The representation of the data below represents the messages, topics, and clients recorded on several independent MQTT brokers.

MQTT Broker An MQTT broker has an ID which uniquely identifies it among all other MQTT brokers created by the users. This broker records the count of messages published on this broker and the current number of tokens in the token bucket. More information on the token bucket can be found in 4.3.4. A broker contains messages, topics, and clients

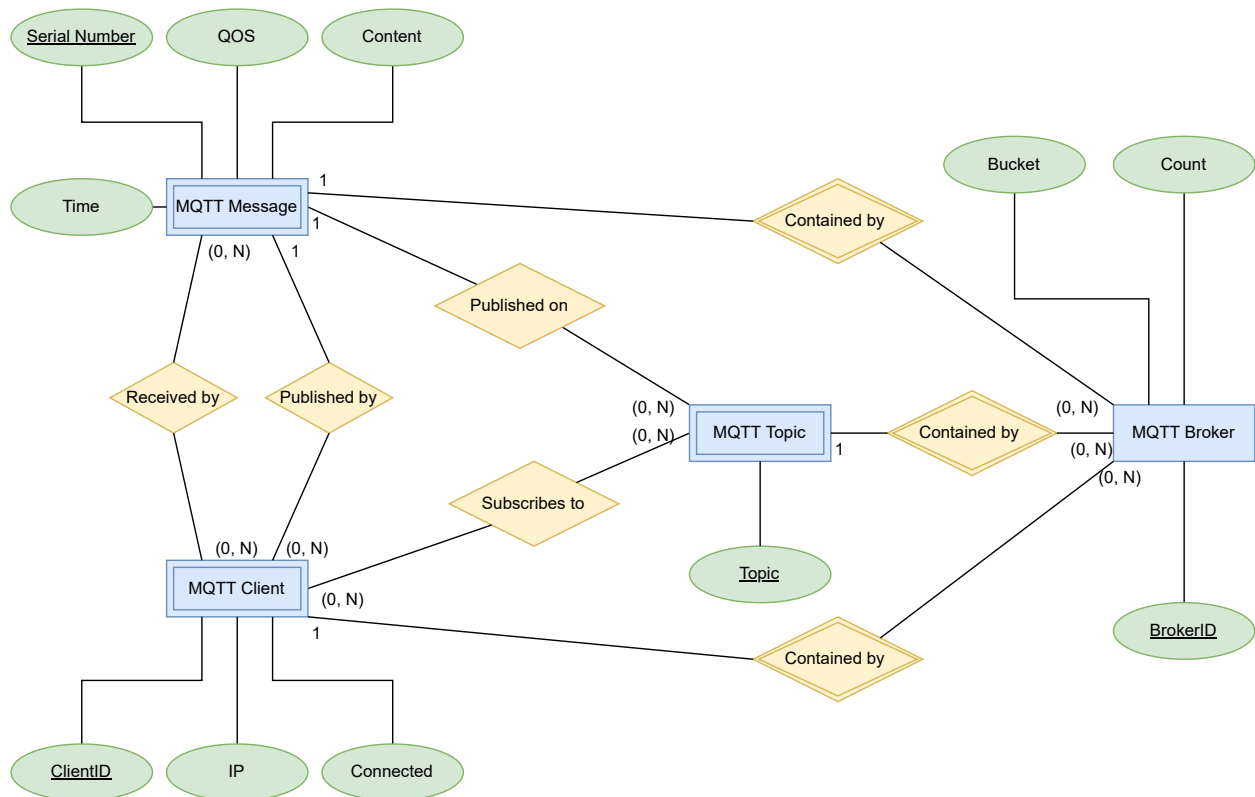


Figure 11: The entity-relation representation of the data.

and identifies them as well (as a message, topic, or client belongs to one broker and will be completely independent of the other brokers). Indeed, as the MQTT brokers have to be independent between them, the ID of clients and the name of topics should be reusable in other brokers independently (2.2.3).

MQTT Message An MQTT message is contained in an MQTT broker and has a serial number which uniquely identifies the message inside this broker but also identifies the order of publication of the messages. This number is not part of the MQTT protocol, it has been added to identify them in the database and the viewer. A message has also a QoS value, a payload, and a time at which the message has been received by the broker. A message is published on a given topic by an MQTT client. This message will eventually be received by any number of MQTT clients. All those information needs to be stored in the database in order to be displayed in the viewer.

MQTT Topic An MQTT topic is identified by its name and the broker in which it has been created. Any number of MQTT messages are published on it and any number of MQTT clients subscribe to this topic currently. As we want to represent the state of the current broker in the viewer (2.2.2), we need to store which client is currently subscribing

which topic.

MQTT Client Finally, we have the MQTT Client which has an ID (the MQTT ID of a client, see 3.2.1), an IP, and a field to identify if the client is currently connected. That information is useful to display the state of a broker in the viewer as it is asked in the requirement (2.2.2).

4.2.2 To a document-oriented database

After the choice of the data that we will have to store, we need to decide how this data will be stored. This will be the second decision on the architecture of the database.

To make a decision on how our data should be stored, we need to ask ourselves what the viewer will need to fetch. Let's take the example where the viewer wants to fetch a big number of recorded messages. In addition, the viewer wants to have more data about each message such as the set of clients who received this message. This is typically the kind of request the viewer will do: one of the features of the viewer is to display the list of messages published on an MQTT broker. A message will be represented by its payload but also by the set of clients who received this message (with other information).

In a relational database, we would need to store this relation of "Received by" in an additional table and then we would need to duplicate data (see fig. 12). This duplication of data would be useless as there are no attributes linked to the "Received by" relation. Furthermore, to recover the set of receivers of each message, we would need to make an additional transaction to get all the receivers of this message. To fetch N messages, we would then need to do N+1 transactions. We need one transaction to fetch all the messages. But once we received the messages, we need to fetch the "Received by" table to fetch the clients' IDs associated with the serial number of a given message for each message.

This is then a better idea to use a document-oriented database. With a document-oriented database, we only need an array of receivers in a field in the message documents. To add a receiver, we only need to find the document of a message and add it to this array. We only need one transaction to fetch an MQTT message and its receivers as this information is included with the field of the receivers.

We can then translate the entity-relation representation in a document representation of the data (in fig 13).

MQTT Broker This entity is transformed into the collection of documents counters where `_id` is the identifier of an MQTT broker. The remaining attributes stay unchanged from the entity-relation attributes.

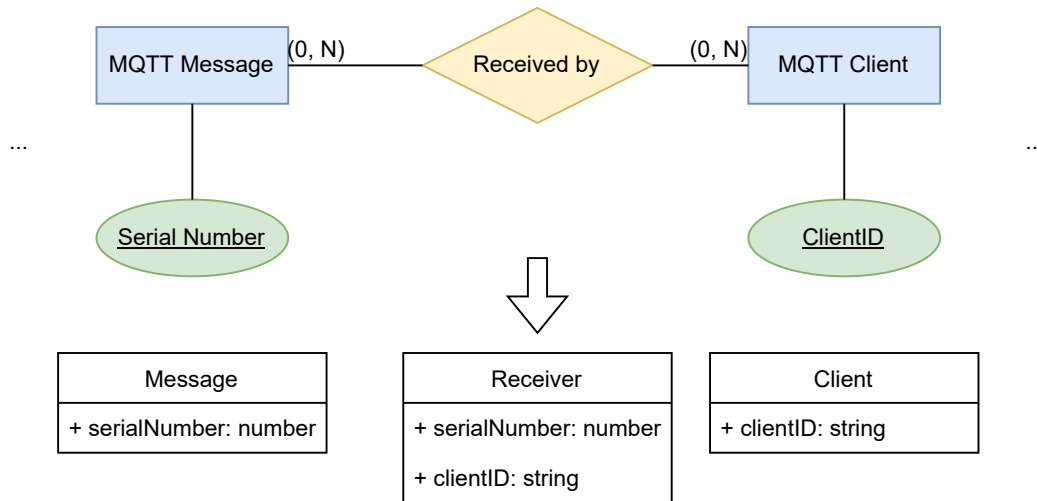


Figure 12: Example of the translation from the entity-relation representation to the relational representation.

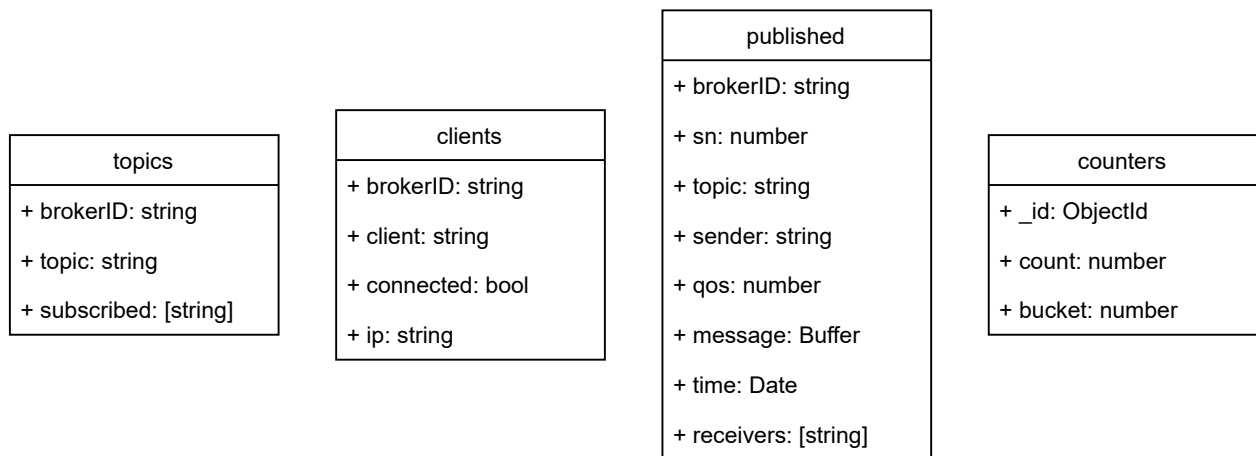


Figure 13: Data representation with a document-oriented database.

MQTT Message This entity is transformed in the collection `published`. The field `Serial Number` is changed into `sn` and the field `Content` into `message`. `brokerID` is the `_id` used in the collection `counters` to identify the broker where the message has been published. The client ID of the sender is used to identify the sender and represents the relation `Published by`. `Received by` is transformed into a set of `receivers` which is an array of strings representing the ID of the clients who received this message. The remaining attributes stay unchanged from the entity-relation attributes.

The decision has been made to use an array of `receivers` in the `published` collection rather than to use an array of received messages in the `clients` collection. Both are technically possible but this decision is guided by the usage of the data. The viewer will be

more interested to display the list of receivers of a message rather than finding the list of messages received by a client (because the main goal is to display the list of messages published on an MQTT broker (2.2.1), and one of the information about each message is who have received it).

MQTT Topic This entity is transformed into the collection `topics`. Those documents contain the `brokerID` which is the `_id` of the associated broker and the topic name. It also has a set of subscribed clients in an array of strings which represents the IDs of the clients that are subscribed to the topic.

Again, the decision has been done to store an array of subscribed client IDs in `topics` rather than to use an array of subscribed `topics` in each document of the `clients` collection. In this case, both solutions were completely equivalent.

MQTT Client This entity is transformed into the `clients` collection. This collection contains the `brokerID` which is the `_id` of the broker, the field `client` is the identifier of this client inside this broker, the field `connected`, and `ip`.

4.2.3 Database indexes

To make the database faster to fetch documents when the viewer asks for them, indexes are needed. The choice of indexes is application dependent, this is then interesting to know how the data is fetched thanks to the documentation in 10.1.

Fetching data is something that could take time. If we look for particular data in a database, we could scan the entire database until we find the record of interest. But scanning an entire database is not something that we want because disks are slow. Databases then use a tool to speed up queries to avoid scanning their entire data: this tool is called an index.

An index is a tool to guide and speed up queries by giving clues on where particular data is stored on the disk. An index is like a table of content in a book. If we search for particular information in a book, we look at its table of content rather than reading the whole book until we find what we want. Indexes rely on an attribute of a table to point where records with a specific attribute value are. For instance, an index on the "age" attribute of a table "Employees" will point to each position on the disk where the employee data with a given age is. There are several types of indexes but we will talk about 2 of them: hashed and sorted indexes.

Hashed indexes have a similar structure to a hashed table where the key is the attribute used by the index and the value is the position of the data on the disk with this attribute. We hash the value of the attribute used by the index to get all the positions on the disk where there is a record with the given attribute value in the collection. This kind of index

is particularly efficient with equality queries (we want every employee with an exact age).

Sorted indexes are sorted arrays containing again the values of an attribute of a table and the list of positions where a document with this given attribute value is stored. To search the documents with a given attribute value, we do a dichotomic search on the index to find the good attribute value and we follow the links. This kind of index is efficient with range queries (we want all employees with a salary between 2200€ and 2400€).

We want the viewer to build a representation of a given broker, the viewer then needs to fetch clients and topics. And it does it for all the clients and topics with a given broker ID (a viewer is only interested in one MQTT broker at a time). It is then interesting to create an index of the `brokerID` field to speed up queries (see fig. 13). As only equality queries are done with the `brokerID`, a hashed index on this field is created for this two collections (`clients` and `topics`).

The viewer also fetches the MQTT messages to build a list of messages from one MQTT broker. He will then be interested only in published messages given a broker ID. But the viewer may fetch a subset of the messages of the broker thanks to a range query on the serial number. As the `brokerID` is still used with an equality assertion but the serial number with a range assertion, the best index is a 2-attribute index where the first field of the index is the hashed index on `brokerID` and the second is a sorted index on `sn`.

4.2.4 Atomicity

During the testing of the broker on a very intense flow of MQTT messages, it has been noticed that several messages had the same serial number in the same broker (while this serial number has to be unique, see fig. 11). This was due to the fact that the `count` field of the `counters` documents was used to compute the serial number of the next published message to arrive in an MQTT broker. Transactions with ACID properties have then been added to solve this problem and have atomicity. We do not have write conflicts anymore with this solution.

4.3 The broker

The broker has to be able to run in a cloud. It is responsible for the communication with the MQTT clients of all the brokers created by the users, for saving the data on the database, and sending this data to the viewer through an API.

4.3.1 Independent brokers

The user should be able to create a new MQTT broker dynamically with the web viewer (2.2.3). Like in POSThere, the user needs to get a new environment to test its

MQTT requests independently from the other environments. This new MQTT broker needs to be independent of other MQTT brokers already created, meaning that a user should not notice that other MQTT brokers are running in the same system.

The choice made to have several independent brokers on the same tool is the first decision we need to do. This decision will influence the whole architecture of the broker. Indeed, the architecture will not be the same at all if we choose to create several instances of an MQTT broker or if we use a cleverer solution with only one MQTT broker.

It has been considered to create a new MQTT instance each time the creation of a new broker is requested. This new broker would use a new available port of the system and we would give to the user the port to use to communicate with the broker. However, this solution has several problems:

1. We do not use the default port of MQTT.
2. We need to open a large range of ports to the system where the MQTT brokers are hosted. This may be incompatible with the security policy where our tool is hosted.
3. The number of ports in a system is limited, and then the number of instances we can create too.
4. The MQTT broker instances that are not used anymore use ports and computation power to listen on this port while those resources may be reallocated elsewhere.

Therefore, the ideal solution should not rely on the ports used. A better solution to differentiate MQTT connections from one broker to another was the usage of the username field in the MQTT protocol (see 3.2.1).

The usernames (and passwords) fields in the CONNECT packets of the MQTT protocol are only used for authentication purposes **to the MQTT broker**. The username is not used in the communication between IoT devices. There is no way a device can know the sender (except if the sender says it in the payload) of a message. Therefore, it is impossible to know the username of the sender of a message. Their usage to differentiate connections inside the same MQTT broker is then not a problem and does not remove any features to the MQTT protocol.

The broker program would be in fact using only one MQTT broker. At the connection, the broker differentiates the broker connections thanks to the username, the used username must be the broker ID generated at the creation of a broker in 4.3.3. At the moment to receive the connection with a CONNECT message, the broker will modify the ID of the client into <brokerID>/<clientID>. With this technique, we can identify the broker environment of a client easily and differentiate brokers when an event occurs with this client. The same technique is applied to differentiate the topics between brokers, the topics are changed into <brokerID>/<topic>. When an event from a client occurs, the client

IDs and topics are translated by adding the broker ID, and when a message needs to be forwarded, the broker ID is removed so that the client does not see that topics and IDs have been changed inside the broker.

We then need to modify the events that happen in a regular MQTT broker: we need to modify the topic field of the PUBLISH packets entering the system, the id field of the CONNECT packets entering the system, the topic field of the PUBLISH packets exiting the system, etc. An example can be found in figure 14. Broker ID and username will be used to point to the same thing in the rest of this document as we use the username to identify the MQTT broker environment of interest.

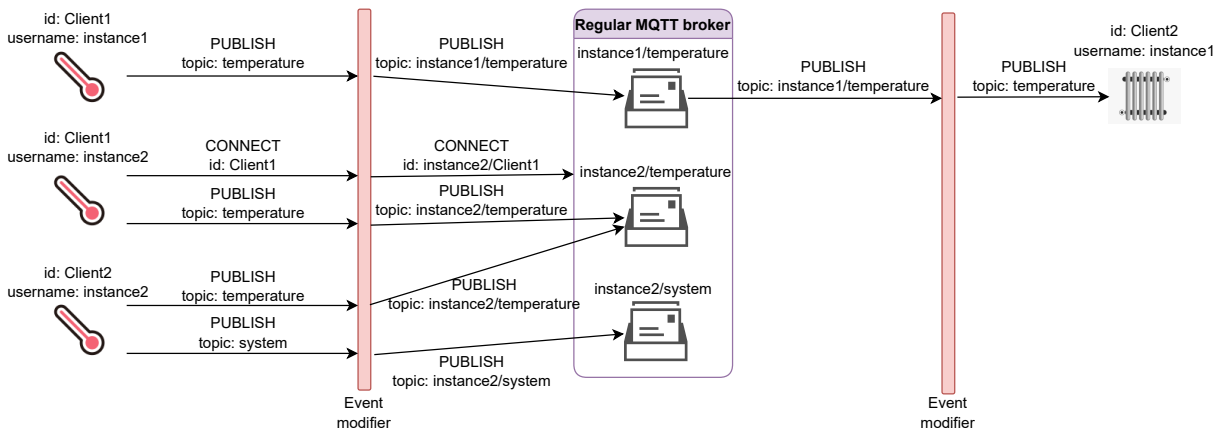


Figure 14: Example of communication with the modification of events before arriving in the regular MQTT broker. The acknowledgement of the packets is not represented.

The problem with this idea is that the username field is sent to the broker only in the CONNECT packets. The broker then needs to remember this username and link it with the new TCP connection associated with this MQTT client. With this technique, when a PUBLISH packet from a client arrives, we can modify it with the username linked to the TCP connection from which the packet arrives.

4.3.2 Broker architecture

Henceforth, we know that we only need one MQTT broker instance thanks to our technique to differentiate several MQTT environments (4.3.1). We can then build our broker architecture on top of this technique. We need to be able to:

1. Run an MQTT broker;
2. Modify the events arriving in this broker and exiting from the broker;
3. Catch and record those events;

4. Listen for viewer queries that ask the events on an API (remember that the broker is also responsible for communicating the events to the viewer);
5. Fetch the recorded events and send them back to the viewer.

The MQTT broker will be contained in one module together with the modifier of those events, this module will also catch the events that occurred in the broker. Another module will have the responsibility to listen for viewer queries and send them the events, this is the API with the viewers. Another module will be responsible to record the MQTT events persistently and fetch those recorded events. This module is a proxy between the MQTT broker, the API, and the database. The relation between those modules is represented in figure 15. A database will be used to store the data persistently.

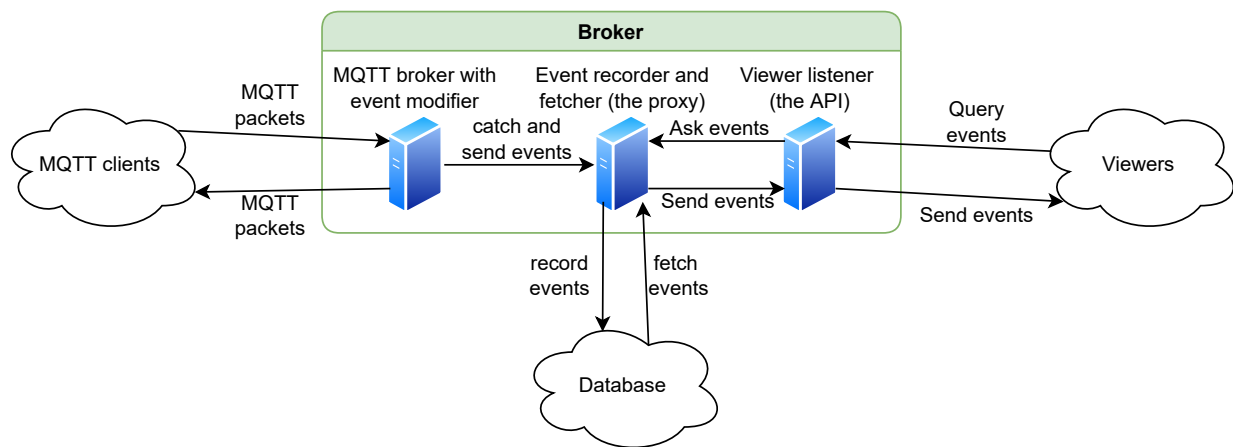


Figure 15: Representation of the architecture of the broker.

4.3.3 Events

We know that the broker comes in three parts: an MQTT client with which the events are modified, an API to communicate with the viewers, and a proxy which is the interface with the database to store and fetch the events. You can find a representation of the events and their consequences in figure 16. Let's take a look at what is happening in each of those events.

createBroker We know that a user should have the possibility to create a new independent MQTT environment to use the MQTT broker as if he were alone. This is the goal of createBroker.

createBroker will create an MQTT broker environment and send back the ID of this broker. This ID will have to be used as username in the CONNECT messages of MQTT clients. We will also create a counters document for this MQTT broker and set the count of messages to 0 and the number of tokens in the bucket to the maximum (see 4.3.4).

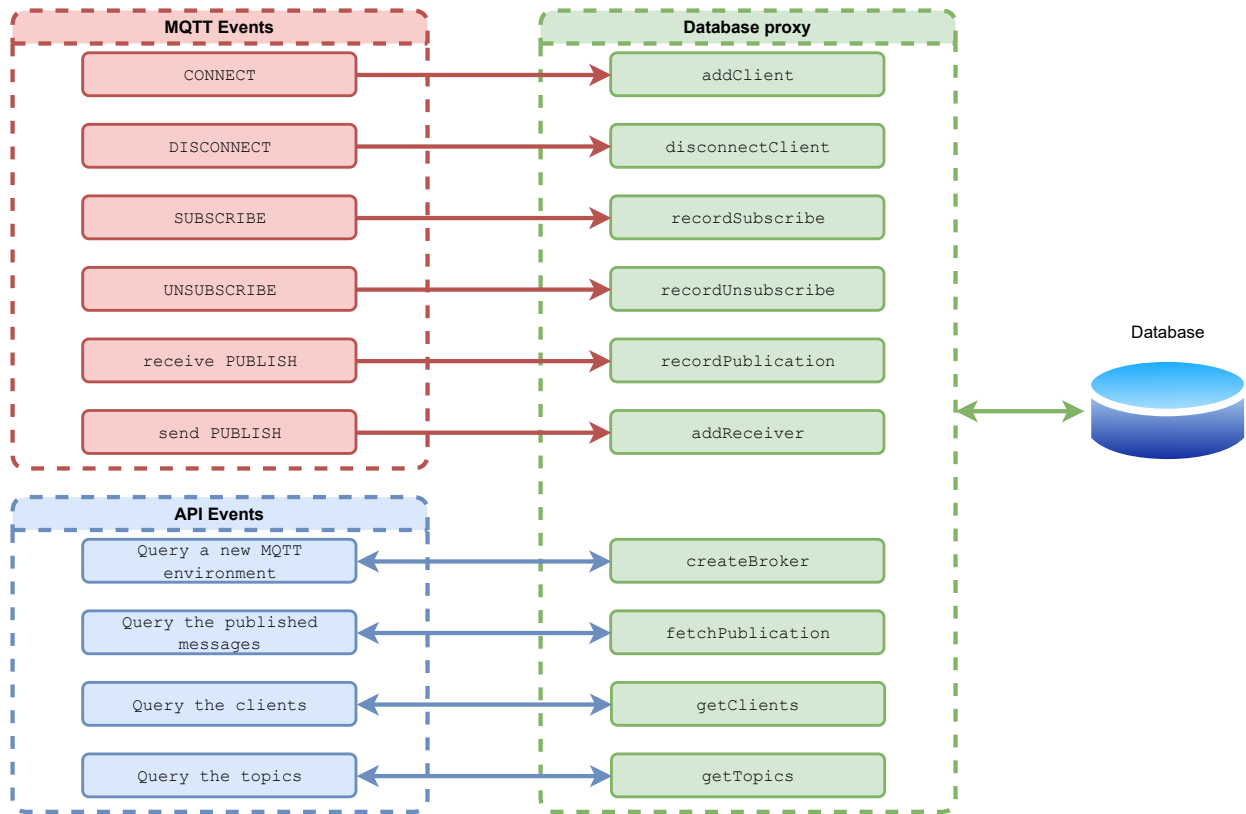


Figure 16: The architecture of the events happening in the broker.

fetchPublication One of the features of the viewer is to display the list of messages that have been published on an MQTT broker. This will be the responsibility of `fetchPublication` to fetch this data and to return it to the viewer. The viewer only needs the messages of one MQTT broker environment at a time, the viewer will then communicate the ID of the broker of interest in this request.

getClients & getTopics Another feature of the viewer is to display the state of the broker. The state includes the list of messages that have been published that we already have with the previous point but we also need the set of MQTT clients and topics which has been used in this MQTT environment. `getClients` and `getTopics` will be responsible for fetching this data. Again, the broker will communicate the ID of the broker of interest. We are only interested in the state of one MQTT broker environment at a time.

addClient When a `CONNECT` packet arrives in the broker, we need to modify the ID used by the client to have independence between brokers (4.3.1). It will be replaced by `<username>/<clientId>` where `username` is the broker ID in the username field and `clientId` is the ID chosen by the client. The connection is refused if there is already a client with the same ID or if the username is a broker ID that does not exist.

Then `addClient` will insert a new client for this broker in the database. If the client is already inserted (can be found thanks to its ID), we update its connection status (we set the `connected` field to `true`) and its IP.

Please note that the recorded client ID in the database is the ID of the client without the `<username>/` as the broker ID is already contained in the field `brokerID`.

disconnectClient The DISCONNECT packets do not need to be modified since there are no interesting fields in those packets.

Then `disconnectClient` will update the connection status of the client (unset the `connected` field) and will remove this client from every subscribed topic (we do this by searching the id of the client in the `subscribed` field of the `topics` collection and by removing it).

recordSubscribe When a SUBSCRIBE packet arrives in the broker, we modify the topic fields of this packet to concatenate the broker ID at the beginning of the topic names. The new topic fields will be `<broker ID>/<topic>`. Remember that we have access to this broker ID because we linked the TCP session of each client with the username they used in the CONNECT message (and the username is the ID of the broker at which they connected).

Then `recordSubscribe` will create the topics if the subscribed topics do not already exist or add the client ID in the array of subscribed clients of those topics otherwise.

Please note that the recorded topic name on the database is the name of the topic without the `<username>/` as the broker ID is already available in the field `brokerID` of the `topics` documents.

recordUnsubscribe Again, the UNSUBSCRIBE packets will be modified to replace the topic names with `<brokerID>/<topic>`.

Then `recordUnsubscribe` will remove the client ID from the array of subscribed clients in `topics`.

recordPublication When a PUBLISH packet arrives, this packet will be modified to replace the topic field of the packet by `<brokerID>/<topic>`.

Then `recordPublication` will first update the value of the number of tokens in the token bucket of the MQTT environment with the given broker ID. Then we check if the token bucket is not empty. It will refuse the publication if it is the case (see 4.3.4 for more information). It will then remove the old publications from the database (see 4.3.5). The publication will then be inserted and the count of messages for this broker (in the collection `counters`) will be incremented. Finally, if the topic in which the message has been published does not exist yet, the topic will be added to the `topics` collection. The PUBLISH packet is finally sent to the regular MQTT broker after all those verifications.

addReceiver When a PUBLISH packet is forwarded by the regular MQTT broker to the subscribed clients, the topic field is modified to remove the broker ID in front of the topic name. `<brokerID>/<topic>` will become `<topic>`. The modification of the topic name is then completely hidden from the MQTT clients with this technique.

`addReceiver` will then add the subscribed client ID in the array of receivers in the documents of the published message. We are able to know the list of clients who received the message only thanks to the forwarded PUBLISH message (we have the client identity thanks to the TCP session in which this packet is sent). Indeed, this information is not present in the PUBLISH packets from the MQTT clients as they do not know who will receive this message.

4.3.4 Traffic limitation

Too intensive traffic may be a problem and make the broker slow. As several MQTT broker environments run on the same regular MQTT broker (4.3.1), the intensive traffic of one environment may impact the performances of another while they are supposed to be independent. Furthermore, the goal of this project is to provide a tool to debug and analyse communications in an MQTT exchange, its goal is not to support a deployed MQTT system.

The solution to this problem is then to drop the published messages of a client when the traffic on its MQTT broker is too high. The technique used to figure out if the traffic is too high or not is the token bucket.

The token bucket is a technique to limit the number of messages or packets going through a system. We have an imaginary bucket which can contain tokens. This bucket can contain a maximum number of tokens. Each time a packet goes through the system, a token is removed. Once we run out of tokens (the bucket is empty), we do not allow the packet to continue. However, the bucket is filled with one token at a time at a given rate (but we stop when we reach the maximum number of tokens). This technique allows to limit the average throughput of a flow of packets and limit the size of the bursts.

The shape of the maximum number of messages that a broker can handle is described in figure 17.

We do not want to query the database at a given rate to update the number of tokens in the bucket of each broker environment. Indeed this would be extremely costly. To replace this, we compute the number of tokens that should be added each time a message is published.

Each time a message wants to be published, the number of tokens in the bucket is first incremented by the number of tokens that should be added since the last published message (this is computed thanks to the published time (the field `time`) of the last published

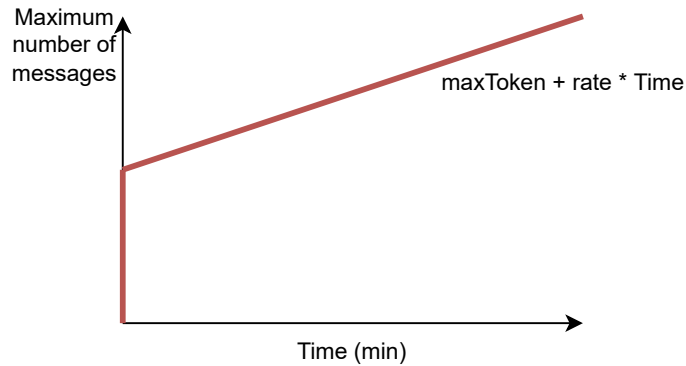


Figure 17: Maximum number of messages allowed with the token bucket.

message). If the number of tokens exceeds the maximum, we set the number of tokens to the maximum number of tokens. Then this number is decremented by one to count the passage of the current message. Finally, we verify if the bucket still has tokens (if the number of tokens is superior to zero) and update the bucket and publish the message if it is the case. If the message is refused, the bucket is unchanged and the message is dropped.

4.3.5 Volatile data

The goal of this project is to debug and analyse the traffic of an MQTT communication in the short term and not to fetch data from a message published a long time ago. There is then no reason to keep old messages that are no longer fetched by the viewer because the viewer only fetches the last X messages. Keeping those messages will only take disk capacity indefinitely while they are not fetched anymore. Those messages are then deleted when there is a given amount of messages published after them on the broker environment of this message. The deletion is done after a message has been published. This number is configurable in 7.1.3.

4.4 The Viewer

The viewer will be responsible to fetch the recorded data on the broker and display it in a user-friendly way. The user should be able to visualise the messages that go through the broker, see the current broker state, and have an idea of what is the rate of messages that go through the broker over time. The viewer is also responsible for requesting the creation of new MQTT brokers if the user wants to create a new MQTT environment. Ideally, the viewer should update its content when the state of the broker has changed (when a message is published or when a client is disconnected for instance) so that the user can see the evolution of what is currently happening in the broker.

4.4.1 Data representation

Before deciding on how the data should be displayed, we have to decide on what should be displayed and how this data should be stored on the viewer side. A part of this decision has been taken while building the database.

The first element that we need is a representation of a broker state. A broker state contains the ID of the MQTT environment we are interested in (4.3.1). Indeed, the viewer is interested in only one MQTT broker environment at a time. This is not useful to display the state of several brokers at the same time. A state of the broker contains a set of messages that have already been published, a set of MQTT clients who already have interacted with this broker, and the set of topics used in this broker.

We need all the data we want to display the representation of a message. We want to display: the topic where the message has been published and the ID of the client who sent the message (we use the MQTT ID because this is the only way we can differentiate clients between each other in MQTT as we already use the username as broker ID (4.3.1)). We also want the content of the message to be displayed as well as the set of receivers, the receivers will be represented by their client ID. Additional interesting information is the quality of service requested for this message to be transmitted and the time at which it has been published on the broker. The serial number will also be useful because we want to display the messages in the order of submission.

The other big important feature required of the viewer is to display in a user-friendly way the state of the broker. That means that we have to see the interactions between the clients and the topics, which client is subscribed to which topic, which client is connected, etc (2). We then need more than this set of messages. We also need the set of topics and clients. The interesting information about a client to display is the IP used by this client and its connection status (if he is currently connected or not to the broker). The set of topics also needs to carry the information about which client is currently connected to this topic.

The MQTT data recorded on the broker and sent to the viewer is represented in figure 18.

There are two reasons why receivers, sender, topic, and subscribed clients are represented by string IDs and not by the actual object representing a `Client` or a `Topic`. The first reason is that the whole information about a client or a topic is not needed to represent the data to the user, only their ID is needed in this case. The second reason is that it would be too costly to update each object rather than replace them all with their new version. Replacing the set of clients is linear with the number of clients while updating the clients is linear with the number of newly fetched clients times the complexity of searching the client to update inside the data structure storing the clients in the viewer.

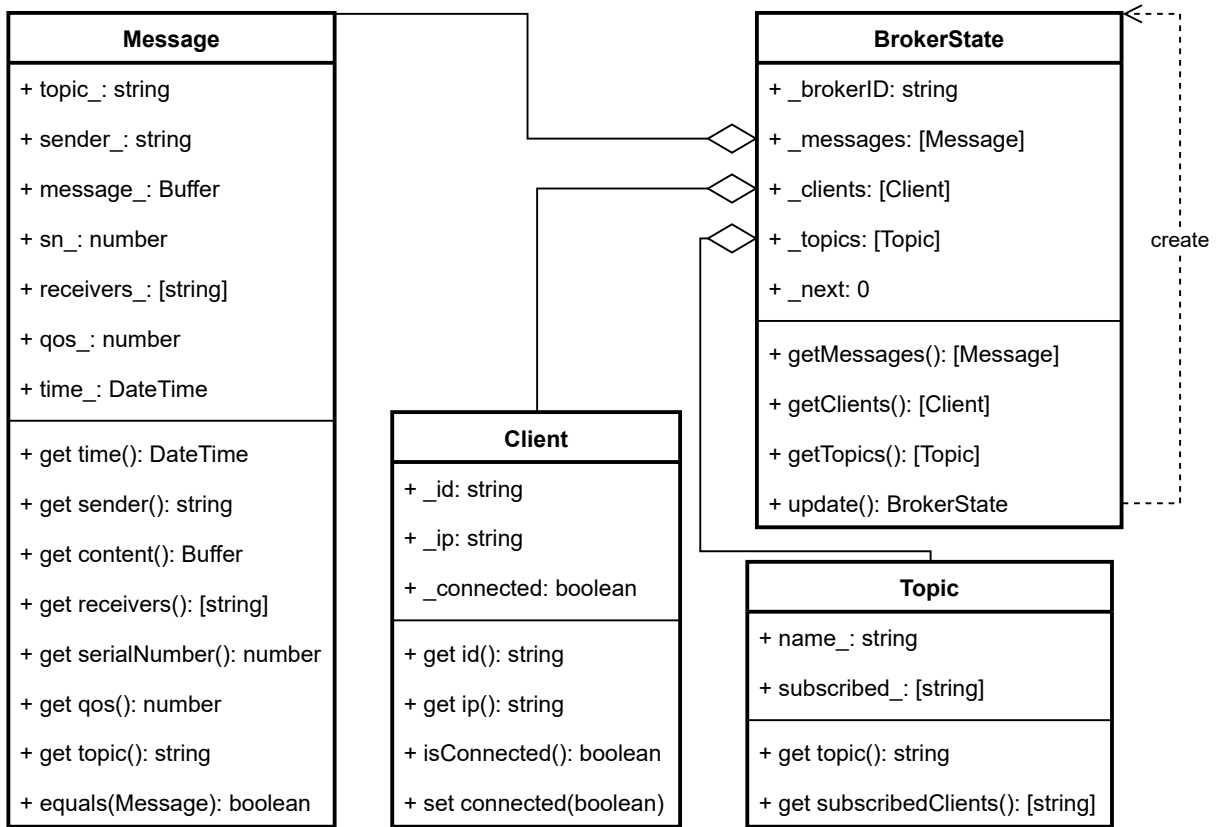


Figure 18: The model representation of the MQTT data.

4.4.2 How to update the broker state

In figure 18, we have seen that the viewer maintains the state of one broker at a time. However, this state needs to be updated regularly so that the user sees the changes in the broker dynamically. When an update is requested to the broker state component, it will create a request to the broker to get the new data.

This request for new data is made through the API of the broker. Once the broker has handled this request by fetching the recorded events on the database, this data will be returned to the broker state. The broker state will build a new broker state containing all the newly fetched information. The data contained in the old broker state component is not updated, the broker state component is building a successor to replace itself. Once the state is built, the old broker state component returns this new broker state and the viewer replaces it with the new state (figure 19).

As the connection status or the IP of a client may change over time, the viewer will request the whole set of clients (of only one MQTT broker environment) to the broker at

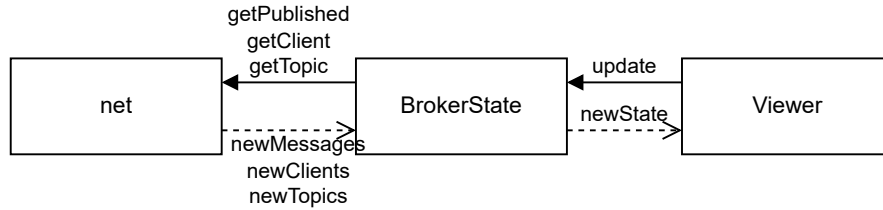


Figure 19: The state management of the viewer.

each update. The idea is the same with the topics where the set of subscribed clients may change over time.

However, the idea is different with the set of published messages on the broker: the already published messages should not change over time (actually they can but we will see that later). An MQTT communication can be composed of hundreds of MQTT clients which typically send hundreds of messages per minute. The flow of messages can then be very big. Fetching this big amount of data at every update is very costly and not useful as a message is not supposed to change over time.

The first step to scale the viewer with the number of messages is by fetching a subset of messages at every update. Instead of requesting all messages at every update, the viewer will save the serial number of the latest recently fetched message and will request all messages from that serial number at the next update. With this technique, the viewer will only fetch the messages that have been published on the broker after the last update.

However, there is one field on the fetched messages which can be modified over time: the set of receivers of the message. Indeed, when a message is forwarded to the subscribed clients of the message, there is a small delay between the moment the PUBLISH message arrives and the same PUBLISH message is forwarded. The receiver ID will be added to the set of receivers with this second event (4.3.3). There is then a small moment where an already published message can be modified. To handle this, we will make sure that each message is fetched exactly twice so that, at the second update, we are sure that the message had the time to be forwarded to all receivers. To do this, when the broker state is asked to build a new broker state, it will give to this new broker state the serial number of his last message plus one (before the update) and this value will be stored in the variable `_next`. When this new state will be asked to update itself, it will fetch the messages from the serial number given by its successor state and replace the messages that have been only fetched once. There is then an overlap between the old set of messages and the newly fetched messages so that each message is fetched twice.

4.4.3 Plugins

Before reviewing how the data should be displayed, let's talk about plugins. The plugin system is not a required feature but could help to extend the viewer application without having to change the core of it. As the world of the IoT evolves very quickly, this is really important that such tools are extensible and can evolve with the IoT world.

The way how plugins are made is explained in the implementation part of this document. But let's take a look at what we want for our plugin system.

We want to provide "mount points" to plugins such that they can mount on the existing application what they have produced. The core application will spread around the viewer and identify those mount points with a name. The plugins will build components for the application and identify the place where they need to be mounted thanks to this mount point name.

There are 4 mount points available in the viewer:

messageContentViewer The format of a message payload can be anything: binary, JSON, XML, etc. This is the decision of the MQTT clients to agree on the kind of formatting and character encoding to use. An MQTT broker is completely independent of the type of content. This implies that the range of types of payload can be very large. This means that we will have difficulties on how to display the payload of the messages: we prefer to represent a JSON with all the keys aligned rather than in a representation without formatting (figure 20).

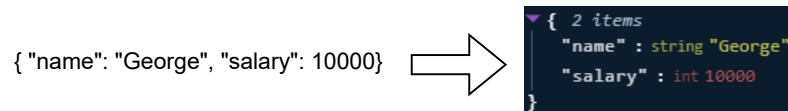


Figure 20: Example of a better representation of a JSON.

But with the number of used formats in the IoT world, this is impossible to implement all of them to have an appropriate representation. This is then important that the representation of payload may be extensible to handle the representation of new kinds of formatting. This is the goal of the `messageContentViewer` mount point.

messageContentVerifier In order to represent a message payload correctly, we have to choose the right plugin to display it. For instance, we should not use an XML plugin to represent a JSON payload. However, there is no way in the MQTT protocol to specify what is the formatting of the payload. We will then have to guess it based on the payload itself. We know that a JSON is a composition of keys and values where the values are either integers, strings, booleans, arrays, or other JSON [11]. All of this respects a

given syntax that other format types would not have and we can then try to detect those payloads.

However, again, we cannot do this as the number of payload formats is huge and increases over time. We will then let plugins specify if the payload of a given MQTT message has a good format to be handled by this plugin. The plugin will provide a function with the additional mount point `messageContentVerifier` that will decide if a payload should be displayed by the `messageContentViewer` of this plugin or not (fig 21).

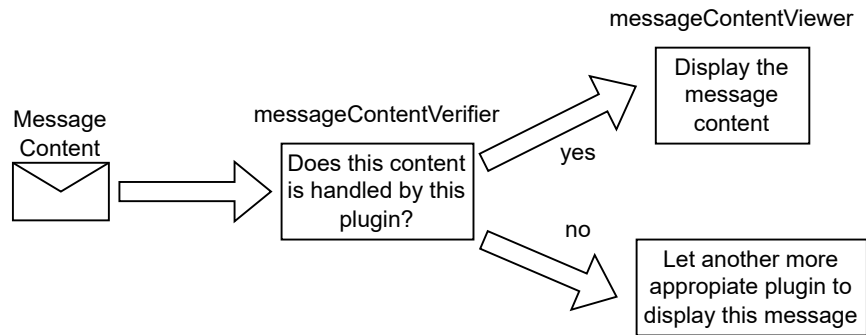


Figure 21: Decision scheme to display a message payload in the plugin.

messageFilters One of the features that the viewer should have is the ability for the user to build filters to only see a subset of the messages. The user may want to only select the messages with a given message payload format or even create filters specific to that format. For instance, a user may want to select every message where the payloads are JSON where the value of the key "foo" is equal to "bar".

But again, this is impossible to do those kinds of filters with all message formats. The mount point `messageFilters` then allows the plugins to implement filters for a given message payload format. The `messageContentVerifier` will again be used to figure out if a message content type is managed or not by this plugin.

additionalFeature This mount point allows the plugins to add a completely new feature to the viewer. The plugins can implement whatever they want with this mount point, there are no restrictions. The components mounted on this mount point will have access to the whole broker state so they can implement any feature that needs the full knowledge about an MQTT broker.

For instance, this mount point can be used to: count the number of JSON messages that go through the broker, make analytics about the content of the data, implement an MQTT client that communicates with the broker, etc.

4.4.4 How to display the data

After choosing what kind of data we want to display and how to store and update this data, we need to decide how this data should be displayed to the user. The simplest way to build user interfaces is by building small simple building blocks and fusion them to build more complex ones. This is then possible to build a complex user interface made of simple components.

The figure 22 shows the dependence graph of the components in our project for the viewer. A component at the top of the graph is made of the components below it. The purely visual components (which contain no information) are not represented in this graph.

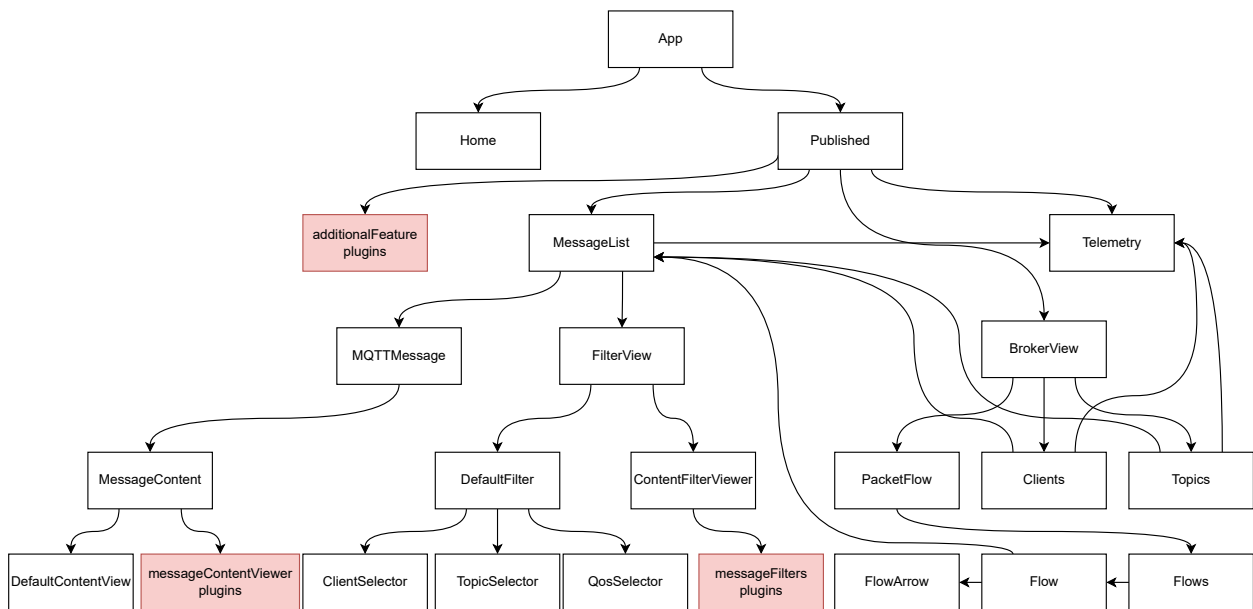


Figure 22: User interface component dependence graph.

We will list below the utility of each component and its interaction with the others.

App The App component contains the whole user interface of the viewer. This is the root of our project.

Home This component is the first content that the user will see. This is a welcome page which invites the user to create an MQTT broker environment by clicking on a button. When the button is pressed, the broker ID of the newly created environment should appear so that the user can use it as username for his MQTT clients to analyse their MQTT communications.

Published This component gathers all the tools to analyse the events that occur in an MQTT broker. The Published component will be responsible to display all the information about one MQTT broker (remember that we are only interested in the data of one MQTT environment at a time) thanks to its child components. This component will also be responsible to maintain the broker state by storing it and requesting an update every second. Once the broker state is replaced in the Published component, it will be spread in the other child components.

MessageList The main feature of this project is to be able to display the messages that go through an MQTT broker. This is exactly the goal of this component. It will be responsible to display the MQTT messages with their payload and additional data from the oldest ones to the most recent ones. The big amount of messages that can be published on an MQTT broker makes that this is complicated to render all of those messages at once. In order to scale the viewer with the number of messages, we then need to render only the messages that the user is seeing.

MQTTMessage As the name indicates, this component displays one MQTT message. It will be in charge to display the sender of the message, the topic in which it has been published, the set of receivers, the time at which the message has been published and the QoS value used. This component relies on the MessageContent component to display the payload of the message.

MessageContent This component is used to display the payload of a message. The payload is either displayed by DefaultContentView which will simply decode the payload with a variety of character decoders and display it as a text. Or it will be displayed by a special visualisation provided by a plugin.

For instance, JSON is widely used with the MQTT protocol and it would be better to display it in a way where each key is aligned, etc. It would be then great if we can have plugins that extend the viewer to handle such payloads by providing special components to display this kind of payloads (we can have a better visualisation of JSON with that technique). To do so, the component needs to know which payload has an appropriate format for a component provided by a plugin (we cannot provide an XML payload to a component that renders JSON). The plugin will then provide a function which will decide if a given payload can be managed by the provided component or not. If the payload is accepted by the function provided by the plugin, the payload will be displayed by the messageContentViewer provided by this plugin.

FilterView One of the big features of the viewer is to be able to filter the messages so that the user can select the messages he is interested in. This is the role of the FilterView component. The user will be able to create a filter (4.4.6) so that the MessageList component only displays the messages of interest. This component will be responsible to return the filters created by the user to the MessageList component.

DefaultFilter This component will provide filters for the additional data of the message (everything except the payload). `ClientSelector` will be responsible for the selection of the senders and the receivers of a given message, `TopicSelector` will be responsible for the selection of the topics where the displayed messages should be published, and `QosSelector` will be responsible for the selection of the QoS values that the displayed messages should have.

ContentFilterViewer This plugin will be responsible to provide an interface to filter messages according to their payload. As the payload may have many formats, this responsibility is again delegated to the plugins with their provided components `messageFilters`.

BrokerView One of the required features for the viewer is to be able to provide a good visualisation of the state of a broker as it is explained in the figure 2. This is the responsibility of this component.

Clients This component will represent the list of clients who already interacted with the broker and display interesting information about them as their connection status and IP. This component can create a `MessageList` or display the telemetry of the messages that have been sent or has been received by a given client.

Topics This component is responsible to display the set of topics used in this broker environment in the broker view. This component can create a `MessageList` or display the telemetry of the messages that go through a given topic.

PacketFlow These components will display the flows in the broker view as displayed in figure 2. This will display the flows of messages from the clients to the topics and from the topics to the clients thanks to arrows. It will also represent the subscriptions of clients to topics thanks to dashed arrows. By clicking on those flows, it will make a `MessageList` appears to represent the messages that go through this flow.

Telemetry One of the required features of the viewer is to be able to see the number of messages that are published over time. This responsibility is delegated to the `Telemetry` component which will display the number of published messages per minute.

additionalFeature Those are plugin-added components that aim to provide additional features to the viewer. This is a way to extend the viewer without any constraint. Those components are given the broker state so they can display any information about the broker state.

4.4.5 Filter building blocks

As said earlier, we want the user to be able to build filters to only see a subset of messages of interest. We want to be able to build complex filters with complex expressions that rely on several parts of the data of a message. For instance, I want every JSON message which has the key "foo" equals to "bar" where the MQTT sender ID is "George" and "Jeanne" is one of the receivers.

To do so, as we have done with the visual components of the viewer, we will build simple filter components that we will fusion to build complex expressions. There are 2 types of filters: basics filters that rely on the attributes of an MQTT message and composition filters that take into account one or several underlying filters. The model can be found in figure 23. The function `filter` will return the set of messages that match the filter and the `check` function will verify only one message. Every filter extends `Filter` where the function `filter` is already implemented so the child filter components inherit it.

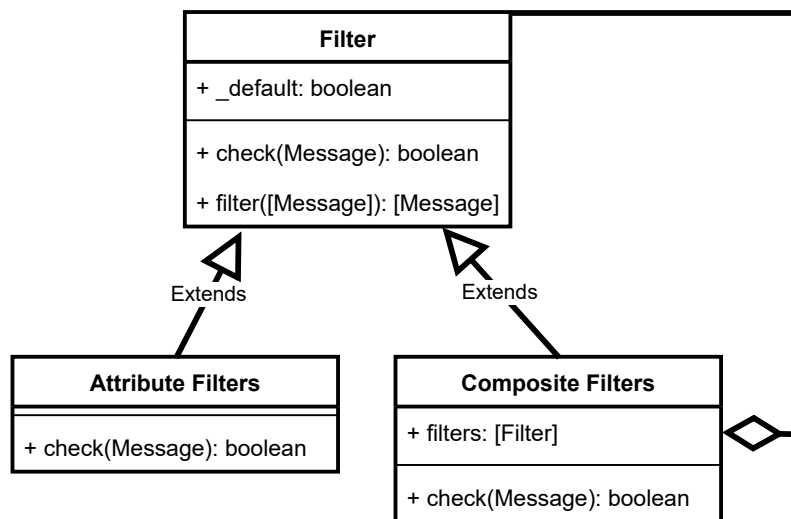


Figure 23: The model representation of the filters.

Below will be listed all the core filter building blocks. Please note that others can be added by plugins thanks to the `messageFilters` mount point.

Filter either accepts or refuses every message based on its default value. All other filters need to inherit from it.

SenderFilter will accept every message where the id of the sender corresponds to the one given to this filter (to its constructor).

ReceiverFilter will accept every message where one of the receivers of the message corresponds to the client ID given to this filter.

QosFilter will accept every message with the QoS value corresponding to the one given to this filter.

TopicFilter matches every message which has a topic name that matches the topic given to this filter. Remember the wildcard topics (3.2.3). Those topics will be visible to the user in the viewer. If a user selects all the messages published on `foo/#`, he should be able to see the messages published on `foo`, `foo/bar`, or `foo/bar/foo` but not those published on `foobar`.

To match a wildcard topic, we will replace the name of those topics in the filter with a regular expression that should match every topic name accepted by the given wildcard topic. The wildcard characters will be replaced by the following regular expression:

1. The characters `"/#"` will be replaced by `(/.*)?` (matches an empty string or every string that starts with `"/`). In this way, the wildcard topic `foo/#` will be replaced by the regular expression `foo(/.*)?` which matches `foo`, `foo/bar`, or `foo/bar/foo` but not `foobar`. The wildcard topic that matches everything (`#`) will be replaced by the regular expression `.+` (matches every string which is not empty).
2. The characters `"/"` will be replaced by the regular expression `[^/]+` (matches every non empty string that does not contain the character `/`). The wildcard topic `foo+/bar` will be replaced by the regular expression `foo/[^/]+/bar` which matches `foo/example/bar` but not `foo/bar`.

The viability of the wildcard topics is verified by the regular MQTT broker so we do not have to worry about malformed wildcard topics (like `foo/#/bar` or `foo+/bar`).

OrFilter is a composite filter which will accept messages which are accepted by one of the 2 filters given to this filter (in the constructor of the filter). If the first given filter accepts the message, the second filter is not checked.

AndFilter is a composite filter which will accept messages which are accepted by both filters that are given to this filter. If the first filter refuses the message, the second is not checked.

NotFilter is a composite filter which will refuse messages which are accepted by the filter given to it.

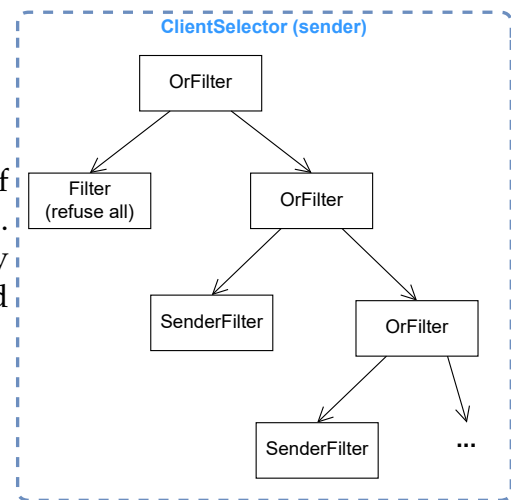
ContentFilter will accept every message from which the payload is accepted by the function given to the filter. This filter will be used to check the payload format of a message to figure out if this payload is managed or not by this plugin. This function will be provided by the plugins with `messageContentVerifier` mount point. For instance, the plugin that manages JSON will create a filter that only accepts JSON messages thanks to its `messageContentVerifier` function.

4.4.6 Complex composition of filters

The building block filters will be fusion to build complex filters. This section will describe how this is done.

The user will interact with the visual components to build a complex filter (4.4.4). Each time a visual component senses a new change made by the user on the filters, this component will create a new filter from which it is responsible and forward it to its parent component. The parent component will in its turn create a new filter and forward it to its parent etc. This is called a callback cascade. The simple filters will be modified in the leaf components of the component tree (fig. 22) and will recursively go up in the component tree to be fusion with the simple filters produced by the other leaf components. The final filter tree obtained is represented in figure 24. We will review each part of this tree. The resulting filter will be given to the `MessageList` component so that it can display only the selected messages.

In this component, the user will select the set of clients who have sent the messages he wants to see. This chain of `OrFilters` with `SenderFilters` will only keep the messages that have been sent by the selected clients.



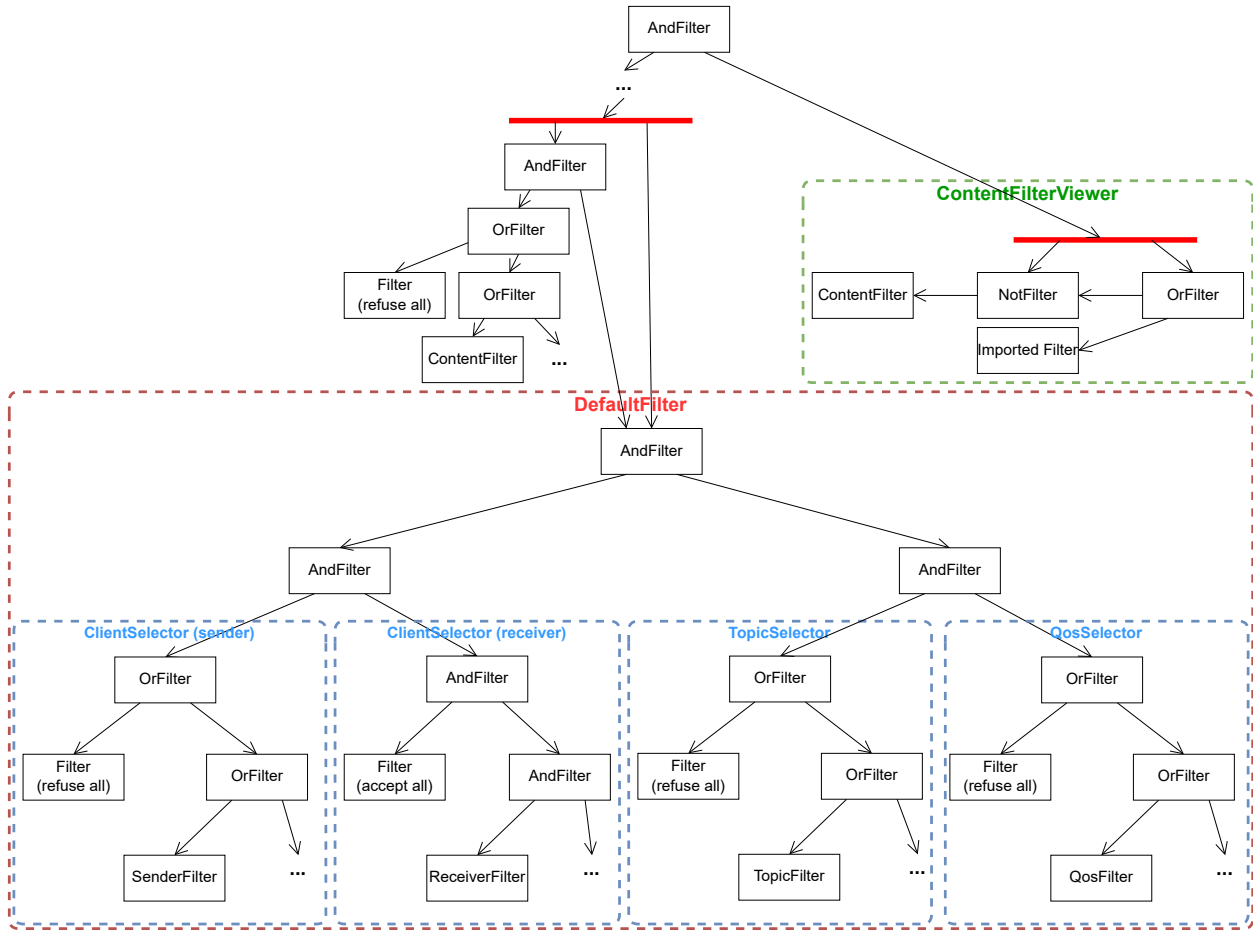
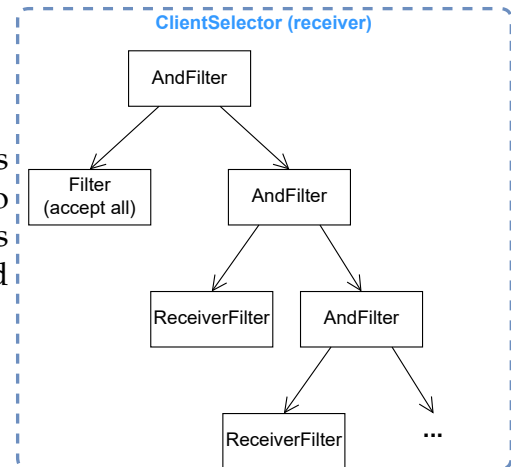
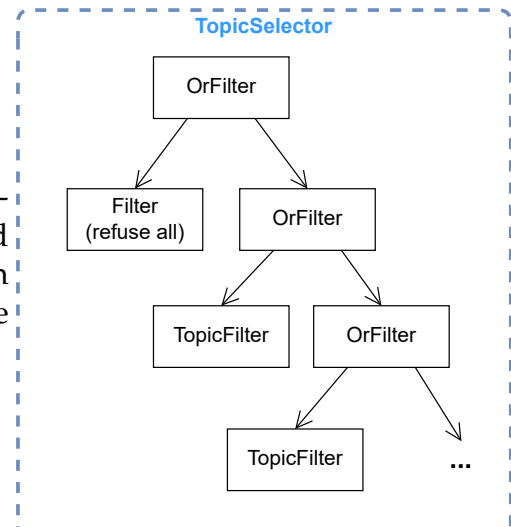


Figure 24: The filter tree to compose complex filters by the user.

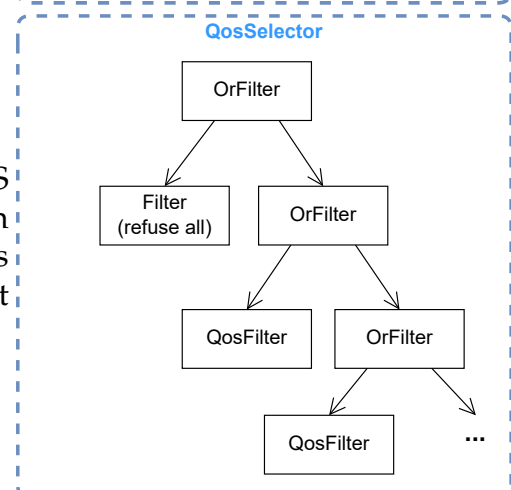
In this component, the user will select the set of clients who should have received the messages he wants to see. This chain of AndFilters with ReceiverFilters will only keep the messages where all the selected clients are in the set of receivers of each message.



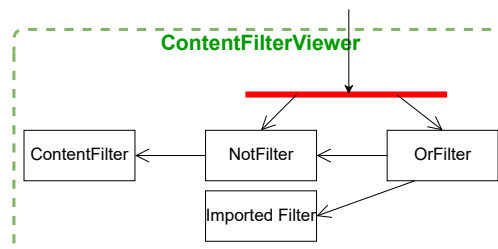
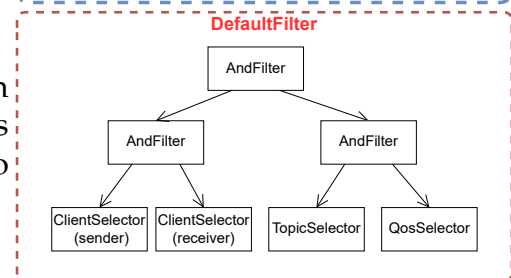
In this component, the user will select the set of topics on which the messages he wants to see should have been published. This chain of OrFilters with TopicFilters will only keep the messages that have been published on a topic selected by the user.



In this component, the user will select the set of QoS values with which the messages should have been transmitted to the broker. This chain of OrFilters with QoSFilters will only keep the messages that have been transmitted with the selected QoS values.



This component will mix the forwarded filters with AndFilters to take into account all the requirements made by the user and will forward the new filter to its parent component.



Remember that we provide a mount point for the plugins to build filters for a spe-

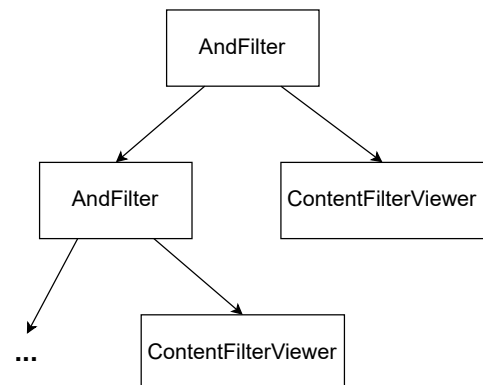
cific payload format. `messageFilters` is the mount point to mount a component where the user will be able to build a filter (for instance, all the JSON messages where the key value "foo" equals "bar") and this component will forward the produced filter to `ContentFilterViewer`. However, this produced filter should only be applied to the messages from which the payload format is managed by this plugin (the XML filters will not be applied to a JSON message). The plugin then provides a function to differentiate payloads that are managed by this plugin from those that are not. This function is given by the plugin with the `messageContentVerifier` mount point. This function will be placed in a `ContentFilter` which will accept the messages of this format only.

Let's take a look at the filter built by `ContentFilterViewer`. The red line represents the decision by the user to show or not the messages with a payload format managed by the plugin (the user decides to show or hide the JSON messages). The left branch is the decision to hide and the right branch is the decision to show them.

If the user wants to see this type of message (the right branch), the produced filter will accept all the messages that are not managed by this plugin (thanks to the `NotFilter(ContentFilter)`: if this filter is true for a message, the filter provided by the plugin will not be executed for this message thanks to the property of `OrFilter` to check only the first filter if this one is true) or will apply the filter returned by the plugin if this message content is managed by this plugin.

If the user does not want to see the messages of this type, the left branch will be taken and only the `NotFilter(ContentFilter)` will be applied so that only messages other than those managed by the plugin will be accepted.

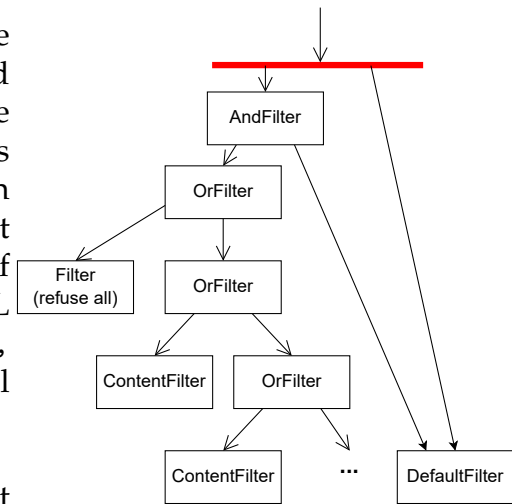
The produced filter by `ContentFilterViewer` for each plugin that provides filters will be chained with `AndFilters` to take into account all the filters produced by the plugins.



Finally, the user will have to decide to display or not the messages from which the payload formats are not managed by any filter plugins.

If the user chooses to not see those messages (the user chooses to take the left branch under the red line), a chain of `OrFilters` will be generated with the set of `ContentFilters` that only accepts the messages that are managed by the set of filter plugins. With this chain, the messages with formats that are not managed by plugins will be rejected. For instance, if the only available plugins manage JSON and XML formats, the chain will be `OrFilter(JSON format, XML format)` and the messages in another format will be ignored.

If the user chooses to see those messages (the right branch), we do nothing and we just forward the filter produced by the default filter. The messages that are not managed by filter plugins are accepted by default.



5 Development methodology

The first step of the development was the research on what is the MQTT protocol and an overview of this protocol. The second step was to get the requirements needed for this project. The first required feature for this project was to display the messages that go through an MQTT broker as POSThere does with the HTTP posted messages. An additional requirement was the broker independence, O had to find a way to differentiate the MQTT broker environments with which we are working.

Having this in mind, I had to find a technique to do this. To solve this problem, a further understanding of the MQTT protocol was needed and I had to understand the fields present in the MQTT packets. The solution was finally found with the username used to differentiate the environments (see 4.3.1). I had then to find an appropriate technology to handle and modify the events that occur in a broker. The high-level architecture was also imagined so that a prototype can be built quickly enough.

With the decided high-level architecture, the development of the broker was a priority. The architecture of the broker was then decided and the technology to communicate with the viewer too (the API). In parallel, the proxy between the API, the broker, and the database was made. In the beginning, no databases were present. The data was only stored in memory in the proxy temporally for testing purposes. With that development, the reflection of what information about an MQTT message should be displayed to the user was made so that we have a good understanding of what should be recorded. Once this broker was built, we were able to test this prototype and successfully fetched the published messages through the API (without the viewer, the data was fetched using an HTTP client). The MQTT client used to test this prototype (and even the final project) is an online MQTT client which can be found at this URL: <https://testclient-cloud.mqtt.cool/>.

After this, the viewer needed to be developed. Technology for the viewer was chosen so that the application code was executed client-side to not overload the broker. The first feature to appear in the viewer was the feature to display the ID of newly created broker environments. When building the interface between the broker and the viewer, the data representation in the viewer was created with the decisions on how to store this data on the viewer. Once it was done, I had to make a decision on how it should be displayed with the development of the feature of the list of messages that have been published on the broker.

Once this prototype of the viewer was in place, a decision about how data should be stored on the broker side was made. The choice about which database to use was made based on the fact that we want a document-oriented database. The proxy was also modified to handle this newly installed database. The same online MQTT client was used to test manually the corner cases of what was already in place (the usage of non-valid broker

IDs in the username field, the will messages, the usage of several clients with the same ID, etc).

To complete the tests, a real MQTT flow from solar panels was sent to a broker environment. This highlights the fact that the viewer was not really scalable with the number of messages. Counter-measures were taken to make the viewer more scalable: the viewer should not render messages that are not visible and should not fetch all the messages on the broker at each update. The ACID properties were also added with transactions in the database as the consistency was not preserved with the high rate of messages (several recorded messages had the same serial number).

In parallel, one new feature was imagined where it would be possible to visualize the state of a broker and the flows of messages in it. With this idea started the development of the broker view. It started with the creation of the `topics` and `clients` collection in the database, the recording of the clients and topics in the broker and the proxy, and the new routes added to the API to fetch those clients and topics. From the viewer side, the data representation of the clients and the topics were made and the visualisation of the broker view was made.

Two additional requirements appeared: a better visualisation of messages with well-known payload formats like JSON or XML and a tool to display the telemetry of the flow of messages. As the first requirement was very general and may include a lot of payload formats, it has been decided to implement a plugin system where the viewer can be extended to handle future message payload formats. This came with research on how plugins are implemented in programs. Once the plugin system was in place with at least the `messageContentViewer` and `messageContentVerifier` mount points (4.4.3), two plugins were developed: one to better visualise JSON messages and the other for the XML format. The second requirement was not really an issue: the accounting and the statistics are computed client-side in the viewer and displayed in a graph.

With the real MQTT flow coming from solar panels, it has been noticed that it was hard to visualise the messages because the flow rate was too high. It then has been decided to add a feature to filter messages in the list of MQTT messages. Those filters would have to take into account the data outside and inside the payload. As we had an already-in-place plugin system, it would be a good idea to create a mount point to extend the filter system. That is with those ideas that were created the whole filter structure with the implementation of filter plugins 4.4.6.

Finally, the `additionalFeature` mount point for plugins appeared to be able to integrate any features needing the broker state that have not been imagined for this project. At the same time, the project was deployed on AWS. It has been noticed that the project was hard to deploy due to the big amount of different possible environments so it has

been decided to use Docker to deploy the project more easily in a controlled environment. Additionally, configuration files were added to be able to modify easily important constants like IPs, port numbers, etc.

During the whole process of this development, Gitlab repositories were used to save the code of the project and keep track of the modifications. Additionally, the documentation of each part of the code was made during its writing and special attention was made to make it as complete as possible. At the end of the project, the whole specification part of the documentation was copied in markdown files to be more easily accessible (this documentation can be found at 10.1). The testing was done manually using an online MQTT client by testing corner cases. Several times during this project, the code was refactored to be more extensible and the user interface of the viewer was improved. A final path was made to make the viewer more user friendly with some explanation.

6 Implementation details

6.1 The Database

6.1.1 Technology used

MongoDB has been chosen as the database for this project. MongoDB is a cross-platform document-oriented database [6]. This database has been chosen because it is a document-oriented database.

6.1.2 Indexes

MongoDB allows the integration of hashed and sorted indexes. A hashed index has then be created with the field `brokerID` for the collections `clients` and `topics`. We do not need to create an index for the collection `counters` as MongoDB already does it by default with the field `_id`. Since version 4.4, this is possible to mix hashed and sorted indexes with MongoDB as we want for the collection `published` [4]. We then create a compound hashed index with the field `brokerID` as hashed and the field `sn` which will be sorted.

6.1.3 Atomicity

The ACID properties were needed in this project. Hopefully, transactions with ACID properties were introduced in MongoDB the version 4.0 [5]. However, the database needs to be started in replica set to enable this feature. A MongoDB a replica set is a feature with a set of MongoDB nodes that replicate the stored data in several nodes and distribute the load among the nodes. The replica set feature is not especially needed but is required to use the transactions. We can use a replica set with only one node.

6.2 The Broker

6.2.1 Technologies used

NodeJS NodeJS is a JavaScript cross-platform environment which is oriented for event handling in server-side applications and is designed to handle highly concurrent programs [8]. This environment is perfect for our broker because it will handle the events which happen in the MQTT broker environments created by the users and it will handle the events of the viewers which will fetch the data of the broker environments.

Aedes Aedes is a NodeJS library which implements an MQTT broker but also provides an API to handle and modify the events during the execution of the broker [1]. This library is perfect because we can listen to the events that occur in the MQTT broker to record them and see what happened inside an MQTT broker. As we have to modify the MQTT packets that arrive and leave the MQTT broker, this is also perfect because Aedes

does not only provide listeners but also handlers where we can modify the events that occur in the MQTT broker.

Fastify Fastify is a client-side web framework for NodeJS [3]. It will be used to create the API between the broker and the viewer so that the viewer can fetch the recorded MQTT data via HTTP requests.

6.2.2 How to generate a broker ID

As we said in 4.3.1, we need to generate an ID that will be used as username to differentiate the MQTT broker environments. This ID must be unique. This is will the role of the MongoDB database to generate this ID. Indeed, MongoDB always generates an additional field per default when we create a document which is the `_id` field. This ID is unique and is used to identify uniquely a document. The first thing that we do when we create an MQTT broker environment is to create a `counters` collection. This is the generated `_id` of the `counters` document that will be used as ID.

6.2.3 Broker Architecture

When we come back to our broker architecture in 4.3.2, we see that we have to implement an MQTT broker with an event modifier, an API, and a proxy between the broker, the API, and the database.

The proxy will be implemented `records.js` and will use the NodeJS API of MongoDB to save and fetch data which is recorded on the MQTT broker. The `mqttBroker.js` file will run the MQTT broker and use the handlers and listeners provided by the Aedes API to modify and record the events that occur in the broker. Finally, the API of the broker will be implemented in the `snifferAPI.js` file which will import the routes in the `routes` directory. Each of the routes corresponds to a specific type of request from the viewer: the route to request the creation of a broker environment, one to fetch the published messages, another to fetch the MQTT clients, and the last one to request the set of topics of the broker. Each of those routes will call the appropriate function in the proxy `records.js` to fetch data on the MongoDB database.

6.2.4 MQTT Events

As said in 4.3.3, we need to catch the arriving and exiting MQTT packets to record the events and modify some of the fields. Hopefully, this can be done with the handlers and listeners provided by the Aedes library. The `CONNECT` packets will be intercepted and modified thanks to the `authenticate` handler of Aedes, the `SUBSCRIBE` packets will be modified and recorded by the `authorizeSubscribe` handler, the incoming `PUBLISH` packets by the `authorizePublish` handler, and the exiting `PUBLISH` packets by the `authorizeForward` handler. The `DISCONNECT` packet does not need to be modified

but only recorded so we can use the listener `clientDisconnect` for that (the difference between handlers and listeners is that handlers allow modifying an event unlike the listeners). Unfortunately, Aedes does not provide a handler when an UNSUBSCRIBE packet occurs but only a listener. However, we need to modify it so that the broker stops to send PUBLISH packets to the unsubscribed clients. We will then generate another corrected UNSUBSCRIBE packet in the listener to unsubscribe the modified topic.

6.3 The Viewer

6.3.1 Technology used

React is a declarative component-based platform for NodeJS used to create client-side user interface for web applications [10]. It will be used by our viewer to display the recorded data on the broker. This technology has been used because of its maturity and the huge amount of libraries made by the community. The React code is compiled at the server-side but is executed on the client-side which will be useful to not overload the server.

6.3.2 React components tree

The main goal of React is to create simple user interface components and to mix them to make complex interfaces as we built our viewer. This is then implemented in the exact same logic as the component dependency tree which is shown in figure 22.

6.3.3 Plugins

The plugins are used to add content to a given feature in the viewer. For now, plugins can add 4 features: an improved display of the content of an MQTT message, a user interface to create better filters on the content of messages, a function to classify if the payload of a message is managed by this plugin or not, and an additional panel which allows a plugin to implement any feature given the current state of the broker.

To be loaded, the plugins have to register themselves in file `plugins.json`. At every server start, the plugins will be loaded and the bunch of features that the plugin provides will be listed thanks to the `package.json` file in every plugin directory. This file lists the npm dependencies of this package and the list of features this plugin provides with an entry point for each feature. Those "features" are the mount points identified by a string that we explained in 4.4.3.

Every part in the viewer which can use a feature provided by a plugin (each mount point) will call `loader.js` and give it the name of the mount point (identified by a string) to notify the loader that this part of the plugin needs to be loaded. This loader will then return an array of plugins which implement this feature with the imported feature. The

importation inside the loader is made thanks to dynamic imports in JavaScript.

You can find more information on how to implement a plugin in 10.1.

6.3.4 Filters

In 4.4.6, we discussed that the filters were built recursively in a callback cascade. Each React filter component at the leaf of the React tree will allow the user to make simple filters. When the filter in one React component is modified, it will be forwarded to its parent component thanks to the `onChange` callback. The parent component will get this modified filter, mix it with the filters of the other child components to create a more complex filter, and forward it to its parent React component, etc.

6.3.5 Scalability

During the testing phase with a real MQTT flow, it has been noticed that the viewer cannot display an MQTT communication with hundreds of published messages per minute. Indeed, React is not able to render tens of thousands of components (which are needed to render the list of MQTT messages). A library called `react-virtualized` allows us to render components only when they are visible to the user. The render of the components is done when the user scrolls down or up in the list of messages. When the user arrives in front of the message, it is rendered. Otherwise, the messages are not rendered and do not use resources (except storage resources).

7 Deployment

The project has to be deployable in the cloud so that the application can be publicly available. The main difficulty was finding an easy way to deploy the project with the same facility independently of the environment. Hopefully, MongoDB and NodeJS are cross-platform and the execution of the code of our project will not depend on the environment. However, given versions of NodeJS and npm (the dependency resolver of NodeJS) are needed for this project. The big problem is that NodeJS is not always up to date on the package providers of the environments. This is then hard to script the deployment of the project.

The found solution was to use Docker. Docker allows us to have a controlled environment where we can install the image version of NodeJS and MongoDB we want. Furthermore, Docker provides scripts that will be executed during the deployment. This is then the main solution that was kept. However, we will have to initiate the replica set and indexes of MongoDB manually as this is not scriptable. It also has been noticed that Docker can use a lot of storage capacity to store the containers and a system with a small disk capacity may run out of storage quickly.

7.1 MQTT Broker

7.1.1 Manual installation

The first requirement is to install the latest version of MongoDB and NodeJS.

MongoDB has to be launched in a replica set [2]. The MongoDB service has to be stopped before if it is already started. The simplest way is to deploy a replica set with only one node and let it run in background.

```
mongod --bind_ip localhost --replSet rs0
```

Then, initiate the replica set in only one node.

```
mongo
rs.initiate()
quit()
```

You can then clone the gitlab repository of the broker, install the npm dependencies, creates the collections and indexes, and start the broker.

```
git clone https://gitlab.com/mqtt-broker-sniffer/sniffer.git
cd ./sniffer
npm i
npm run init
npm run dock
```

This will start the broker with the ports specified in 7.1.3.

7.1.2 Docker

Copy the following instructions and place it in a `docker-compose.yml` file.

```
version: "3.3"

services:
  mqtt-broker:
    image: martindetienne/mqtt-broker-recorder:latest
    container_name: mqtt-broker
    ports:
      - "4040:4040"
      - "1883:1883"
    links:
      - mqtt-db
    depends_on:
      - mqtt-db

  mqtt-db:
    hostname: mqtt-db
    container_name: mqtt-db
    image: mongo:latest
    command: "--quiet --bind_ip_all --replSet_rs0"
    volumes:
      - mongo_db:/data/db

volumes:
  mongo_db:
```

Then, run the following command:

```
sudo docker-compose up -d
```

You have then to initiate the replica set how this is explained in the following section.

Build in Docker This is possible to build the broker in the docker environment. First, clone the gitlab repository, and build the broker in docker compose.

```
git clone https://gitlab.com/mqtt-broker-sniffer/sniffer.git
cd ./sniffer
sudo docker-compose up -d
```

Once it's done, initiate the MongoDB replica set and create the indexes with the following commands:

```

sudo docker exec -i -t mqtt-db /bin/bash
mongo
rs.initiate()
quit()
exit
sudo docker exec -i -t mqtt-broker /bin/bash
npm run init
exit

```

7.1.3 Configuration

Several options are configurable in the broker. The configuration file location is `config/production.json`. If one of the parameters is not specified, it will be replaced by its default value in `config/default.json`. The configuration options are listed in table 1.

Table 1: Broker configuration options

Parameter	Default	Description
<code>cors</code>	<code>{"origin": "*"}</code>	The CORS policy for the API.
<code>clientPort</code>	4040	The port at which the viewer will connect to access the API.
<code>mqttPort</code>	1883	The port for the MQTT clients.
<code>dbURL</code>	<code>"mongodb://localhost:27017"</code>	The URL of the MongoDB database to store data.
<code>dbName</code>	<code>"mqtt_proxy"</code>	The name of the MongoDB database to store data.
<code>bucket</code>	None	The specification of the token bucket for one MQTT broker. Type: <code>{"maxToken": number, "rate": number}</code> . <code>maxToken</code> is the maximum number of tokens in the bucket. <code>rate</code> is the number of tokens pushed into the bucket per minute. Does not apply the token bucket algorithm by default.
<code>recordMax</code>	None	The maximum number of messages kept in the database for one MQTT broker. Type: number. Does not delete the messages by default.

7.2 MQTT Viewer

7.2.1 Manual installation

The first requirement is to install the version `v16.15.0` of NodeJS with `npm 8.5.5`.

You can then clone the gitlab repository, install the npm dependencies, build the app, and start it.

```

git clone https://gitlab.com/mqtt-broker-sniffer/client.git
cd ./client
npm install --force
npm install -g serve
npm run build
serve -s build -l <port>

```

The <port> is the TCP port the viewer will use.

Plugin installation You can install plugins for the viewer through an npm command.

```
npm run install_plugin <url>
```

where the <url> is the URL of the zipped plugin.

7.2.2 Docker

You can run the latest deployed Docker image with the following commands:

```
docker pull martindetienne/mqtt-viewer
docker run -p 80:4030 --name mqtt-viewer -d martindetienne/mqtt-viewer
```

This image will fetch the messages from `http://localhost:4040`.

Build The viewer can be built in the Docker environment. First, clone the gitlab repository, and build the viewer in docker compose.

```
git clone https://gitlab.com/mqtt-broker-sniffer/client.git
cd ./client
sudo docker-compose up -d
```

7.2.3 Configuration

The viewer can be configured thanks to the `src/config/production.js` file. The configuration parameters are listed in table 7.2.3. If the parameter is not specified, its default value from `src/config/default.js` is used.

Table 2: Viewer configuration options

Parameter	Default	Description
brokerAddress	"localhost"	The IP of the broker from which the MQTT messages are recorded.
brokerAPIPort	4040	The port used by the broker for the API.
maxFetchPublication	100	The number of MQTT messages that are fetched at the beginning of a session in the viewer.

8 Results

After explaining the goal of our project, its architecture, the development methodology, the implementation details, and the deployment of our project, let's check the results by reviewing the performances of the broker and what the viewer looks like.

8.1 Performances of the broker

The broker is a publicly available tool which records the messages that go through an MQTT communication. As a typical MQTT communication is composed of hundreds of messages per minute, this is important to have an efficient broker. We will review the latency of the broker to handle the publication of MQTT messages and to respond to queries on his API.

All the tests were made on a Debian computer with an Intel Core i5-4690 (3.5 GHz) CPU and 8 gigabytes of RAM (1600 MHz). The database is running on the same system and data is stored on a hard disk drive. The requests are sent since the same computer to not be influenced by the network bottleneck.

The scripts used for the tests can be found in this Gitlab repository: <https://gitlab.com/mqtt-broker-sniffer/test>.

8.1.1 MQTT Publish latency

For this test, we first choose an interval of time between each publication of MQTT messages. After this, we create a new MQTT broker environment (we fetch a new broker ID to use as username). We then connect on the MQTT broker with the given username and we subscribe to the topic "test". Once it is done, we publish 100 random MQTT messages on the topic "test" separated by the interval of time we chose and we compute the average delay between the publication of an MQTT message and its reception by the client (as it is subscribed on the topic). The latencies are listed in table 8.1.1.

Table 3: MQTT latencies compared to the interval between each MQTT publication.

Intervals (ms)	1000	100	60	30	20	15	10	5	1
Delays (ms)	24.9	20.23	23.99	42.58	64.7	288.48	1017.6	1427.12	1932.42

It seems that the broker cannot handle more than one published MQTT message every 15 milliseconds (which is 4000 messages per minute). If this "burst" is longer (with 1000 messages for instance), the value of the average delay blows up. With one message every 20 milliseconds (3000 messages per minute), the value is contained and does not blow up with the size of the burst (the average delay was 76.884ms with a burst of 1000 messages at 3000 messages per minute and 111.77ms with a burst of 100,000 messages).

8.1.2 API latency

With this test, we will measure the latency to fetch a given number of MQTT messages with the API given by the broker. We will create more than 100,000 MQTT messages on an environment and fetch a given number of them every 100 milliseconds and ten times to record the average delay to fetch the messages. The latencies are listed in table 8.1.2.

Table 4: The latency of the API to fetch a given number of messages.

Number of messages	1	10	100	1000	10,000	20,000	30,000	40,000	50,000	100,000
Delays (ms)	38.8	36.8	42.4	56.1	160.5	791.6	1673.2	2449.4	3729	8322

We can see that the latency begins to be big when we fetch more than 10,000 messages at the same time.

8.2 Viewer screenshots

We will see in this section what the user will see on the viewer and which feature it provides.

In the first image 25, we can see the view where the user creates new MQTT environments and gets a broker ID that he will have to use as username in his MQTT clients.

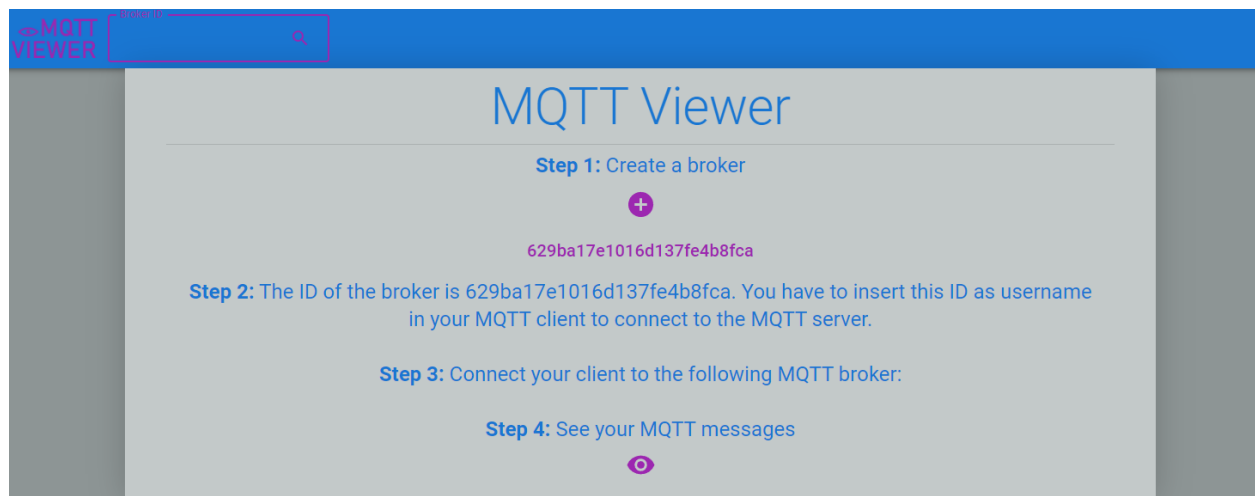


Figure 25: The Home component

Image 26 shows the list of messages that go through the MQTT broker. Several parts of information are displayed like the sender, the topic where the message has been published, the set of receivers, the time at which it has been published, and the QoS. The JSON payloads are displayed with the appropriate plugin which displays JSON. We can

see in this image tabs where we can show the broker view and the telemetry. This is also in those tabs that the additionalFeature plugins will be added.

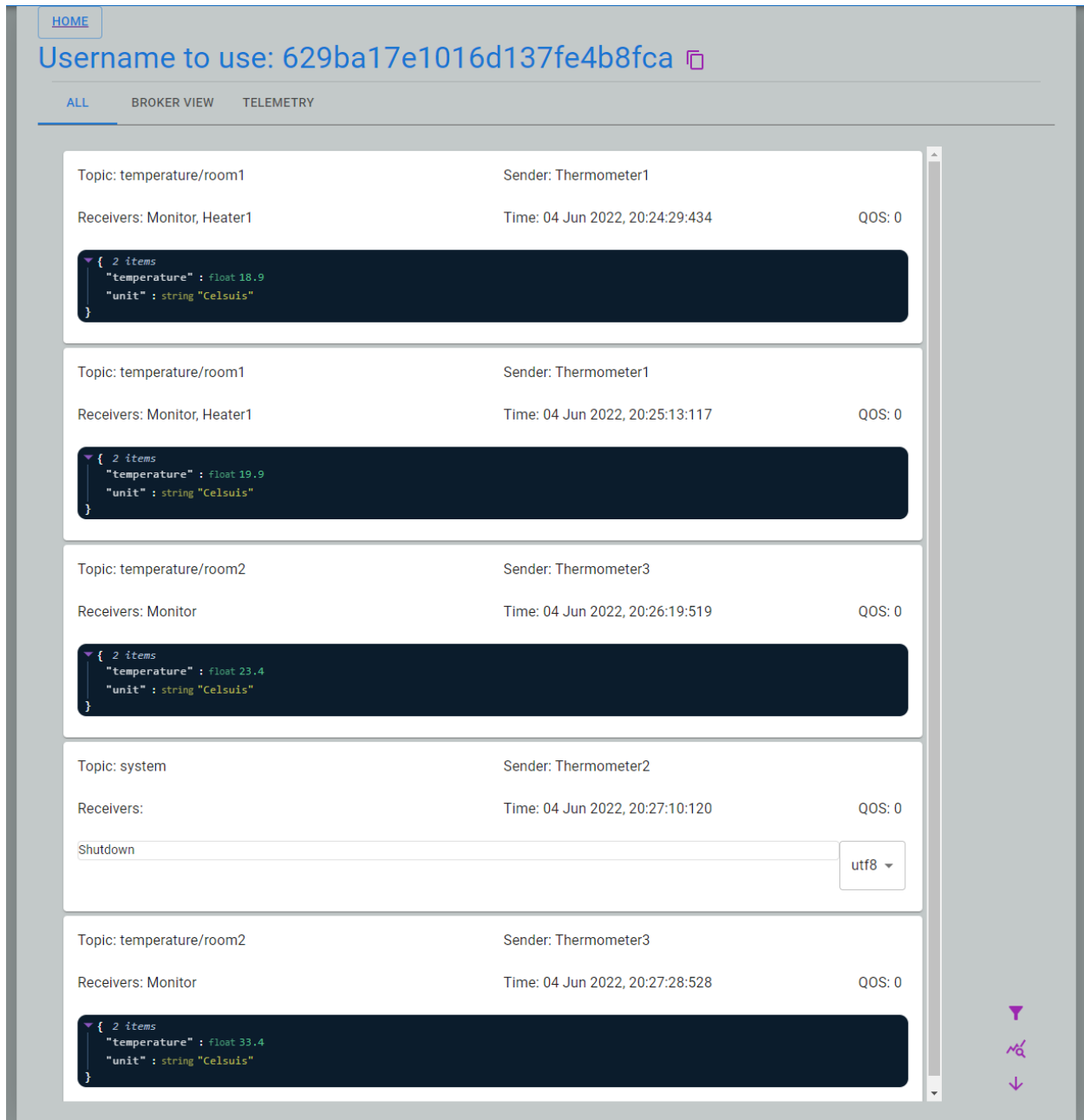


Figure 26: The MessageList component

Image 27 shows the current state of the broker. We can see which clients have published on which topics, which clients are subscribing to which topics, the flows of messages and the connection status of the clients with their IP addresses.

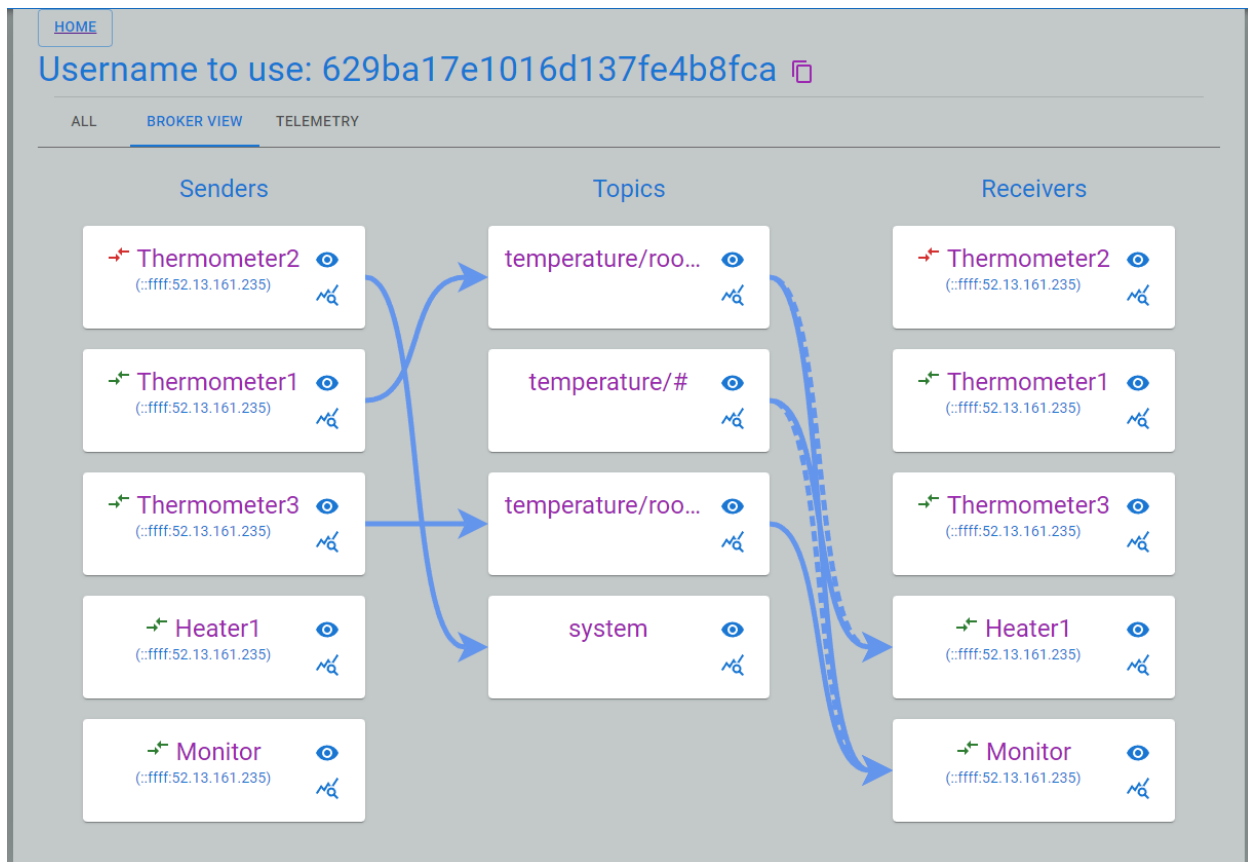


Figure 27: The BrokerView component

In image 28, we can see the number of messages that have been published during each minute where a message has been published.

Finally, in image 29, we can see the filter panel where the user will be able to build complex filters. We can see at the bottom of this filter panel the filters provided by the plugins where we can choose to see or not the type of message payloads.

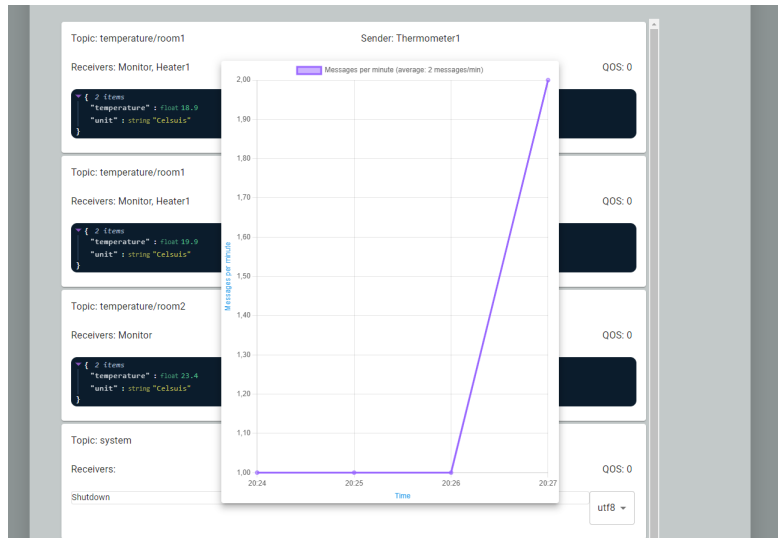


Figure 28: The Telemetry component

The FilterView component interface provides configuration options for data filtering. It includes the following sections:

- Select senders:** A list of senders with checkboxes:
 - Thermometer2 (checked)
 - Thermometer1 (checked)
 - Thermometer3 (checked)
 - Heater1 (unchecked)
- Select receivers:** A list of receivers with checkboxes:
 - Thermometer2 (unchecked)
 - Thermometer1 (unchecked)
 - Thermometer3 (unchecked)
 - Heater1 (unchecked)
- Select topics:** A list of topics with checkboxes:
 - temperature/room1 (checked)
 - temperature/# (checked)
 - temperature/room2 (checked)
 - system (checked)
- Select QOS:** A list of QOS values with checkboxes:
 - 0 (checked)
 - 1 (checked)
 - 2 (checked)
- JSON:** A checkbox that is checked, with a dropdown menu set to "utf8".
- Other types of content:** A checkbox that is checked.

Figure 29: The FilterView component

9 Possible improvements and limitations

9.1 The MQTT username

In 4.3.1, we were describing how to make the different MQTT broker environments independent from each other. The found solution was to set the username field of the CONNECT packet with the ID of the MQTT broker at which we want to connect. However, not all MQTT clients and devices implement this username (as this field is not mandatory) or some devices do not give access to the username. Those devices will then not be able to connect to the broker.

Unfortunately, as the usage of the username to place the broker ID is part of the core of the architecture of the broker, it would be hard to solve this issue. The accessibility of the username field in the MQTT devices which will be used with our tool is then a requirement for the usage of this project.

9.2 Message modification

The goal of the broker is to record and transmit transparently the events that occur in an MQTT environment. Once connected, an MQTT client will not make the difference between a normal MQTT broker and our broker. The messages that are published by a client will be forwarded to the subscribed client with an unchanged payload. However, the modification of messages between the moment when they are published and the moment when they are forwarded may be useful for testing purposes. For instance, in the situation where a thermometer influences the actions of heaters, we may want to modify the messages sent by the thermometer to observe the reaction of the heaters.

This could be done by adding routes in the API of the broker that provides the feature to modify messages on the fly. This API would have to catch the PUBLISH messages sent by a client of interest and modify the payload of the messages (this can be done with Aedes and the `authorizePublish` handler). The viewer would also need to be modified to implement a tool to modify MQTT messages.

9.3 Plugins

The plugins can currently implement four types of features 4.4.3. A possible improvement of the plugin system in the viewer could be the addition of features (mount points) that the plugins could handle.

A possible additional feature that the plugins could handle is a telemetry feature for instance. For now, the telemetry only indicates the number of messages per minute. But it could display other metrics provided by the plugins or even do analytics on the content of the messages. For instance, a telemetry plugin could show the number of occurrences

of a given word in the JSON messages per minute.

Another possible feature could be the modification of the user interface of an MQTT message or the broker view.

All those limitations are mitigated by the `additionalFeature` mount point. This mount point of plugins allows the plugins to add a panel with a completely new feature. This feature is given the current state of the broker then there is a large range of possibilities for new features.

9.4 Replica sets

The replica set is a technique to deploy a program or a database in several nodes. Currently, only the MongoDB database of the broker is used in replica set mode and can be deployed in several nodes. But this is also possible to do it with the broker program. Aedes provides additional extensions which provide clusters of brokers [1].

9.5 Embedded MQTT clients in the viewer

An additional interesting tool could be an integrated MQTT client in the broker. For now, we need external MQTT clients or devices to publish on the broker. If we want to publish a message on the broker to analyse the reaction of a device, we need to use an external tool. The integration of embedded MQTT clients could be then useful.

If we want to go further, specific device emulators could be implemented by plugins and the imported MQTT clients could transmit from the viewer. For instance, if we do not have a thermometer yet during the debug phase of an MQTT development, this emulator could be useful to simulate the presence of this thermometer and see the reaction of other devices.

A possible solution to mitigate this lack of built-in MQTT clients is to implement an MQTT client inside a plugin mounted on the `additionalFeature` mount point. However, those clients will need to be on an additional panel.

10 Additional documents

10.1 Documentation

The documentation about the broker can be found at the URL: <https://gitlab.com/mqtt-broker-sniffer/sniffer/-/tree/main/doc>. This documentation contains the specification of the API to fetch MQTT data from the broker and the documentation of the proxy between the MQTT broker, the API, and the database.

The documentation about the viewer can be found here: <https://gitlab.com/mqtt-broker-sniffer/client/-/tree/main/doc>. This documentation contains the specification of the used React components, the documentation about the model of the application, the request functions used to fetch MQTT data from the broker, and the documentation necessary to build plugins.

10.2 Deployment example

This project is currently deployed on the following website: <http://mqtt.netips.org/>.

10.3 Docker images

Docker images of the deployed project are available of Docker Hub:

1. The broker:
<https://hub.docker.com/repository/docker/martindetienne/mqtt-broker-recorder>
2. The viewer:
<https://hub.docker.com/repository/docker/martindetienne/mqtt-viewer>

10.4 Gitlab repositories

The code is publicly available in those public repositories:

1. The broker: <https://gitlab.com/mqtt-broker-sniffer/sniffer>
2. The viewer: <https://gitlab.com/mqtt-broker-sniffer/client>

11 Conclusion

To conclude, the built app meets all the requirements and planned features that we had. The set of messages that go through a broker is easily identifiable and understandable. We can see the state of the broker at a given time and understand the flows in it. We can see the telemetry of the flows in the broker and filter the set of messages. Additionally, we succeeded to run several independent MQTT broker environments in the same system by using the username field of the MQTT protocol. However, this brings the limitation that the IoT devices that will have to be used with this tool must give access to the username field to the user. The viewer can be extended with the plugin system to add a better visualisation and filtering for some message payload formats or to add a feature that has not been imagined in the core application. A big part of the possible today limitations of our tool can be mitigated thanks to this plugin system. For instance, a built-in MQTT client in the viewer may be useful for testing purposes and this could be implemented via a plugin. However, the broker cannot be extended with plugins. Then a limitation like the impossibility to modify an MQTT message on the fly cannot be resolved by only a plugin. One of the possible improvements of the project could be the implementation of a plugin system inside the broker (by adding mount points in the API and the listeners of Aedes for instance) to be more extensible. Another possible improvement of the broker can be its replication so that the client requests can be managed on several nodes. However, with all those limitations in mind, the project meets all the requirements and has been judged satisfactory.

References

- [1] Aedes github repository. <https://github.com/moscajs/aedes>.
- [2] Deploy a Replica Set. MongoDB. <https://www.mongodb.com/docs/manual/tutorial/deploy-replica-set/>.
- [3] Fastify github repository. <https://github.com/fastify/fastify>.
- [4] MongoDB hashed index. <https://www.mongodb.com/docs/manual/core/index-hashed/>.
- [5] MongoDB Multi-Document ACID Transactions. May 2020. https://webassets.mongodb.com/MongoDB_Multi_Doc_Transactions.pdf.
- [6] MongoDB Website. <https://www.mongodb.com/>.
- [7] MQTT Version 3.1.1. Edited by Andrew Banks and Rahul Gupta. 29 October 2014. OASIS Standard. <http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html>. Latest version: <http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html>.
- [8] NodeJS github repository. <https://github.com/nodejs/node>.
- [9] Postel, J., ed., "Transmission Control Protocol - DARPA Internet Program Protocol Specification", RFC 793, Information Sciences Institute, September 1981.
- [10] React github repository. <https://github.com/facebook/react>.
- [11] Tim Bray. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 7159, March 2014.