POLITECNICO DI TORINO

Master degree course in Computer Engineering
(Ingegneria Informatica)

Master Degree Thesis

HTTP and MQTT: A comparison in the context of Industry 4.0

Supervisor:
Prof. Riccardo SISTO

Candidate:
Domenico SPANTI

Internship Tutor:
Thomas FERRERO

Academic year 2019-2020

Summary

Since the beginning of the Internet spread, HTTP has represented and still represents, the most widely used protocol for the exchange of information. However, with the ever-increasing diffusion of IoT networks, some characteristics of HTTP (e.g. the large overhead produced) brought researchers and companies to implement different protocols. Between them, MQTT is the one that is spreading faster.

The aim of this thesis is to compare the various technical aspects of the two protocols, especially for what concerns their performance and security.

The two protocols will be evaluated from a practical point of view through their implementation in the Myna software suite, an open-source energy management system developed by Myna-Project.Org s.r.l., with two industrial use cases in the context of the H.O.M.E. (Hierarchical Open Manufacturing Europe) project.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the last decade, domotics has represented one of the fastest-growing fields of applied science. Terms as Cloud Computing, cyber-physical systems (CPS) and Internet of Things (IoT) have become known to a large part of the population, and the cost of smart devices is decreasing through years in a manner that is inversely proportional to the increase of their performance. In the technical field, words as Industry 4.0 and Smart Manufacturing have become frequent to indicate the use of these new technologies with the aim of automating industry.

Bain & Company, one of the three world's largest management consultancy companies, estimates that the markets for IoT hardware, software, systems integration, and data and telecom services will grow to 520$ billion in 2021, more than double of the market value in 2017 (235$ billion)[1].

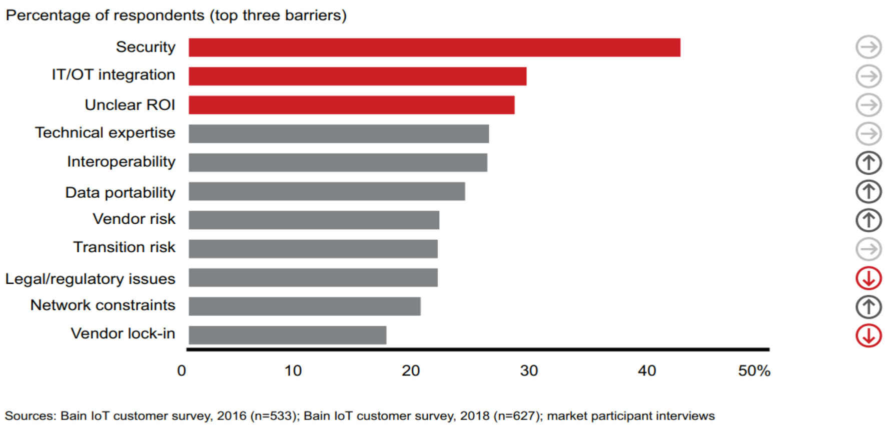


Figure 1.1. What are the most significant barriers limiting your adoption of IoT/analytics solutions? (and change since 2016)[1]

However, despite the growth of investments in the IoT market, as reported in Figure 1.1, it is a common perception that IoT solutions could be insecure and laborious to integrate with the Operational Technology already in use, particularly in the case of IIoT (Industrial IoT). Also, other critical aspects like transition risk and network constraints slowed down the implementation of IoT in the industrial field.

It can be noted that all these critical aspects can be linked to two factors, security and performance, which can result as important enablers for IoT in the context of production. The choice of the application protocol used by the device impact significantly on the performance and security of it.

Between the wide number of messaging protocols used in IoT, MQTT is the one that is becoming more established, resulting as the second most used communication protocol in the 2019 IoT Developer Survey, made by Eclipse Foundation, behind HTTP[5]. It also results as the most used messaging protocol in the 2018 edition of the same survey with a preference of 62% and a positive trend through the years, as shown in Figure 1.2.

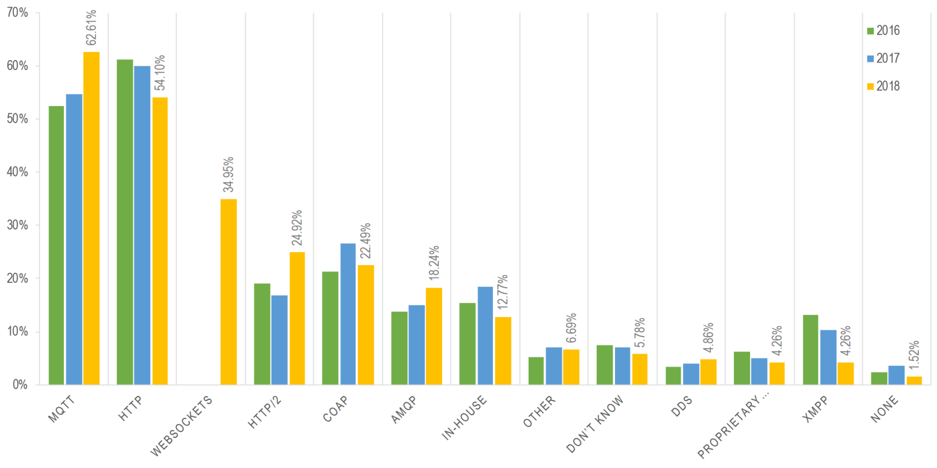


Figure 1.2. Messaging standards, trends on IoT solutions between 2016 and 2018[2]

1.1 Objectives

The aim of this thesis is to study, analyze and implement the two most used messaging protocols in IoT, HTTP and MQTT, compare their architectural aspects, and then compare their security and performance in a real scenario implementation.

The two protocols will be implemented inside an in-development project created by Myna-Project.Org s.r.l. (hereinafter also referred to as Myna-Project or Myna) and known as Myna Software Suite, an open-source energy management system, and tested through different networks and workload conditions.

The result that this thesis wants to achieve is to give a comprehensive overview of HTTP and MQTT and explain, by comparison, in which situation one is better than the other or equivalent.

1.2 Document structure

This thesis begins with a theoretical overview of the current state of the art for HTTP and MQTT as reported in Chapter 2. The energy management system of Myna-Project and its constituent parts are analyzed in Chapter 3.

The deployment of HTTP and MQTT protocols, and how it has been implemented in the energy management system, is depicted respectively in Chapters 4 and 5, and in Chapter 6 the result of their comparison in different environments is reported.

Lastly, the results of the thesis and possible future steps for improvement are discussed in Chapter 7.

Chapter 2

Protocols analysis

2.1 HTTP structure

HTTP (HyperText Transfer Protocol) is the application protocol used to communicate over the World Wide Web. Tim Berners-Lee and his team initially developed it at CERN in 1989, and in the years, also because of its simplicity, it has become the de-facto standard for Web communication.

HTTP is a reliable request-response protocol [6] in a client-server model that stands on top of a TCP/IP connection. In HTTP, a client (e.g., a web browser) sends a request to a web server and receive a response. There are different types of requests; one of the most common is the request to get web content from the server. The source of the web content is called **web resource** that can be an image, a text, or any kind of file. Since the Internet hosts many different data types, HTTP tags each object being transported to the Web with a data format label called a MIME (Multipurpose Internet Mail Extensions) type, designed initially for mail systems and then adopted by HTTP.

Every resource has a name, so clients can point out what they are interested in. The resource name is called URI (uniform resource identifier). The most common form of a resource identifier is the URL (uniform resource locator) that uniquely identifies a resource on the Internet and it composed of three parts: the scheme, the host and the path of the resource.

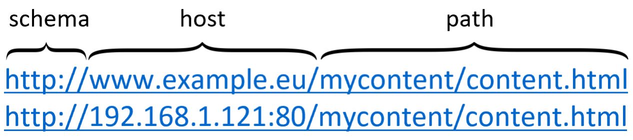

schema
host
path
http://www.example.eu/mycontent/content.html
http://192.168.1.121:80/mycontent/content.html

Figure 2.1. Examples of URLs with a textual domain name and with IP and port

As shown in figure Figure 2.1, an URL can be represented through the IP address of the server and a port number (which can be assumed to be 80 when not specified), or through a textual domain name, or hostname, that is just a human-friendly alias of an IP and it is converted by a facility called Domain Name System (DNS).

Requests & Responses Every HTTP request message has a method, that is a command used to tells the server what action to perform. Some of the most common HTTP methods are **GET**, used by the client request a named resource, **PUT**, used to store data from client into a named server resource, **POST**, used to store data from client to a server, and **DELETE**, used to delete a named resource from the server.

Every HTTP response comes back with a status code. The status code is a three digit numeric coda that tells the client if the request succeeded, or if there are required actions. Some example of HTTP status code are the **200**, request returned correctly (**OK**), **201**, resource **created**, and **404**, resource **not found**.[7]

Connection & Authentication The HTTP connection, as mentioned before, is reliable because it uses the Transmission Control Protocol (TCP) that provides an ordered and error-checked delivery of massages. HTTP is also a stateless protocol, which means that each request/response happens in isolation. For this reason, web sites need a way to distinguish HTTP transactions from different users. There are a few techniques to do so, for example through HTTP headers. The most used techniques are[7]:

- *User login.* A built-in mechanism that uses WWW-Authenticate and Authorization headers to implements logins. A server can explicitly ask the identity of a user replying with the HTTP 401 Login Required status code to an unauthenticated request. An example of this mechanism, represented in Figure 2.2, is the Basic Authentication method, which sends the credentials encoded with the Base64 algorithm. This method is typically used with HTTPS encryption because the encoding alone doesn't provide data confidentiality. Alternatively, the Digest Authentication method allows an encrypted authentication that can be used in untrusted connections;

- *Cookies.* Used for several reasons (including web tracking), they are sent by websites and stored by the web browser, generally in a text file, allowing authentication and can last after browser or computer restart (persistent cookies). Cookies are sent with every user request in the HTTP header Set-Cookie.

Versions HTTP is a protocol with more than 30 years of history. During these years the protocol changed and many new features have been introduced and improved through versions. The released HTTP versions are:

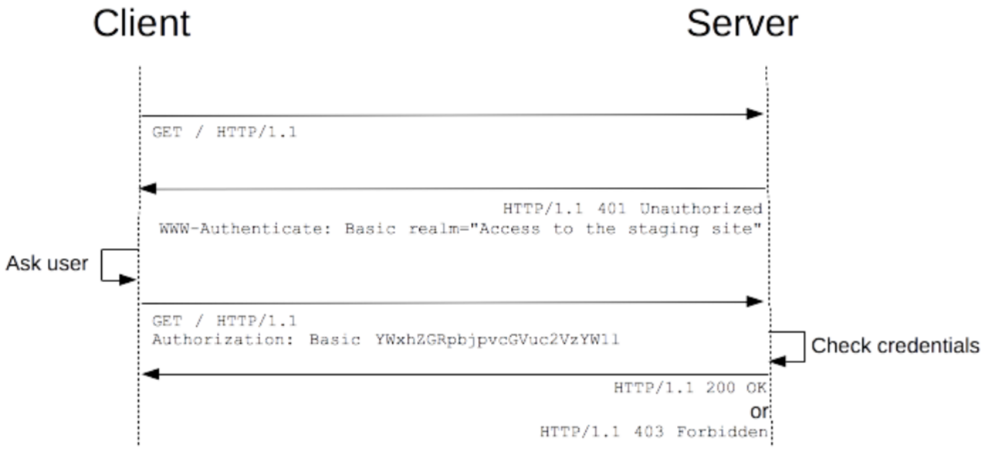

Client
Server
GET / HTTP/1.1
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Basic realm="Access to the staging site"
Ask user
GET / HTTP/1.1
Authorization: Basic YWxhZGRpbjpvcGVuc2VzYW11
Check credentials
HTTP/1.1 200 OK
or
HTTP/1.1 403 Forbidden

Figure 2.2. HTTP User login authentication: credentials are sent to server with the Basic Authentication method
1

- *HTTP/0.9.* The basic and initial version of HTTP, it is very simple, requests are on one line and the only possible method is GET. The responses contain only the file itself;

- *HTTP/1.0.* Due to browsers and server needs, the initial version of HTTP has been expanded with several features including versioning information on each request (HTTP/1.0 is appended to the get line), the status code line at the beginning of the response and the notion of HTTP headers together with the ability to transmit other documents than plain HTML files (thanks to **Content-Type** header). As a result of many different features introduced by browsers and server with a try-and-see approach, the 1.0 versions suffered a lot of interoperability problems;

- *HTTP/1.1.* Version 1.1 has been released just a few months after version 1.0, to standardize the protocol, clarifying ambiguities and introduce improvements such as the possibility to reuse connections, pipelining, chunked responses and cache control. Thanks to its extensibility, e.g. the possibility to create easily new headers or methods HTTP/1.1 had two revisions, in 1999 and in 2014 in prevision of the release of HTTP/2, showing its stability through more than 15 years. In this version of HTTP security aspects such as CORS[2]

[2]Cross-Origin Resource Sharing

and CSP[3] headers and the SSL(TLS) security layer has been introduced and will be detailed in the next section. Another important step in the evolution of the protocol has been made in 2000 with the design of a new pattern for using HTTP in a more extensive way, the representational state transfer (REST). REST allowed any Web application to provide an API for retrieval and modification of its data, not by using new HTTP methods but only by accessing specific URIs with basic HTTP/1.1 methods;

- *HTTP/2.* Standardized in May 2015, HTTP/2 has been created to respond to the increasing needs of resources and multiple connections from high-traffic web applications, necessity covered only partially from version 1.1. HTTP/2 received many contributions to its making from a protocol named SPDY and developed by Google, and has many differences from HTTP/1.1, as it is a binary protocol rather tan text, can handle several parallel requests on the same connection, compress headers and allows a mechanism called server push to populate data in a client cache from the server prior to the request. HTTP/2 has had a rapid spread, especially in high-traffic websites, also because it not require adaption on Web sites (HTTP/1.1 and HTTP/2 are transparent to them).[8]

2.1.1 HTTP Security

As described above, the HTTP protocol has not been designed to handle critical information, such as credit card numbers or personal data, but with its wide adoption, a security improvement became necessary. In 1994 Netscape Communications created HTTPS, a version of HTTP that uses the TLS (previously SSL) protocol to grant confidentiality through encryption and integrity with keyed MAC[4] check.

As defined in RFC 5246[5], TLS has four main goals:

1. *Cryptographic security.* TLS should be used to establish a secure connection between two parties.
2. *Interoperability.* Independent programmers should be able to develop applications utilizing TLS that can successfully exchange cryptographic parameters without knowledge of one another's code;
3. *Extensibility.* TLS seeks to provide a framework into which new public key and bulk encryption methods can be incorporated as necessary. This will also accomplish two sub-goals: preventing the need to create a new protocol (and risking the introduction of possible new weaknesses) and avoiding the need to implement an entire new security library;

[3]Content Security Policy

[4]Message authentication code

[5]`https://tools.ietf.org/html/rfc5246#section-2`

4. *Relative efficiency.* Cryptographic operations tend to be highly CPU intensive, particularly public key operations. For this reason, the TLS protocol has incorporated an optional session caching scheme to reduce the number of connections that need to be established from scratch. Additionally, care has been taken to reduce network activity.

TLS works in two phases, the first phase of handshake, where the cryptographic methods and the keys to encrypt data are negotiated between parts, and a second phase where data are exchanged. During the handshake phase, there is also the exchange and validity check of the certificates. The choice of appropriate cipher suites and keys strongly affects the security of the communication because a weak cyber suite allows an attacker to decrypt the ongoing traffic easily.

Over the years, several vulnerabilities of TLS were found and fixed. On its technical report[9], Jung categorized attacks in three categories, outlining three general channels that can be attacked: the Users, meant as any part involved with TLS (including CAs), the protocol itself and the implementation of the protocol (e.g. bugs in software that allows overcoming the security mechanisms of TLS). Three known cases of these categories are described below.

Heartbleed Discovered by Google researchers, Heartbleed is an attack that targets the implementation of TLS protocol by exploiting a bug of an extension, Heartbeat, used to check if the other party is still available. Through the extension, a client sends a request to the server containing an arbitrary string and its length. The server then answers with the received string, in order to maintain the connection active. This process is also performed in the opposite direction, from server to client.
The vulnerability is due to the length field that is not verified by OpenSSL, enabling an attacker to read up to 64kB of data from victim memory (the length field is 2 bytes long). These memory areas could contain useless binary data but could also contain sensitive information such as private keys. To avoid the attack, TLS must be updated, and all certificates and passwords renewed to ward off previous data leaks.

MitM The Man in the Middle (MitM) attack can be considered as a general technique of attack in computer networks more than a specific attack against TLS. In MitM an attacker is placed between the two parts of communication and relays messages between them, potentially tapping sensitive data. In the specific case of TLS, there are tools that allow attackers to perform a MitM-like attack for an HTTPS session. Combined with an ARP Table poisoning, the sslstrip application can be used by the attacker to set an insecure connection between him and the victim. Many applications do not try to establish unsecured connections, so they are not affected. But in web browser users can accept to continue an untrusted connection making sslstrip a serious threat.
So, it's important to face this attack that users are well aware when they accept a

certificate that is untrusted and when they are connected to a website via HTTPS or not.

BEAST The Browser Exploit Against SSL/TLS (BEAST) is an attack based on a theoretical exploit on the Cipher Block Chaining (CBC) algorithm, which is often used in TLS. In CBC, a plaintext that needs to be encrypted is divided into blocks and every block Is ciphered using the previous ciphered block with an XOR operation. The first block of the chain is encrypted with a random value known as the Initialization Vector (IV). In TLS 1.0, at the end of a message the last ciphered block will be used as IV for the next message, and this could lead to an attack based on guessing the plaintext for the first block on certain messages.
Even if this attack is mostly theoretical, because the possibility to guess a whole block of 8 bytes results in $225^8 \approx 1{,}7 \times 10^{19}$ possibilities, if the first 7 bytes are known as they are predictable in some circumstances (e.g. with a fixed header of the message) the number of possibilities goes down to only 255. TLS 1.1 fixed this issue by adding a unique IV field to every message. So, to avoid the attack, the server administrator should forbid the use of TLS version 1.0 and less.

These examples show that the use of HTTPS does not automatically imply full protection over the Internet and, even if attacks as BEAST are known from 2011, SSL Labs stated that in 2020 nearly the 60% of websites (on a sample of 138,000 websites) still supports TLS 1.0[6], and major browsers like Chrome and Firefox started the procedure to deprecate TLS 1.0 and 1.1 only in 2018[7][8]. Nevertheless, when applied correctly and updated TLS still represent a good way for securing HTTP.

It is important to notice that TLS represents only a part, even if important, of the HTTP security. Other aspects not directly linked to transport security, such as access authorization to resources, are more related to the design of the application and will be addressed in chapter 4.

2.2 MQTT structure

MQTT (Message Queue Telemetry Transport) is an asynchronous and lightweight publish-subscribe protocol, firstly developed in 1999 by Arlen Nipper and Andy Stanford-Clark [9]. Unlike HTTP and other client-server protocols, the publish-subscriber pattern of MQTT makes this protocol architecture quite different from the HTTP architecture. In MQTT workflow there are three main “actors” involved:

[6] `https://web.archive.org/web/20200226204529/https://www.ssllabs.com/ssl-pulse/`

[7] `https://blog.chromium.org/2019/10/chrome-ui-for-deprecating-legacy-tls.html`

[8] `https://bugzilla.mozilla.org/show_bug.cgi?id=1579270`

[9] `https://mqtt.org/2009/07/10th-birthday-party`

- *The Publisher.* It is the device that sends data that wants to send to a specific target of subscribers;
- *The Subscriber.* A Subscriber is a device that wants to receive data regarding specific arguments to which it is interested in;
- *The Broker.* The Broker is the central point of the architecture. Every client (publisher or subscriber) establishes a connection with the broker to receive or dispatch messages. From MQTT version 3.1.1 the MQTT broker is known as the MQTT server and, hereinafter, the two terms will be used indistinctly[10, 11].

The channels where messages are dispatched by publishers and then sent by the broker to relative subscribers are known as **topics**.

Topics A topic is a named logical channel referred to a UTF-8 string separated by the forward-slash "/" symbol, known as the **topic level separator**. Every part of the topic divided by "/" represents a topic level.

Each message sent in MQTT belongs to a topic and the broker applies a **topic-based filtering** to deliver the message to all the subscribers interested in that topic.

Wildcards Wildcards are special characters used in topics to allow subscribers an easier way to subscribe to multiple topics without specifying the exact name of every topic. There are two different wildcards symbols:

- *Single Level Wildcard.* The plus sign "+" represents the Single Level Wildcard, it can be used at any level of the Topic Filter, including first and last levels and it must occupy an entire level of the filter (e.g. "home/+/lamp" is valid while "home+" is not valid);
- *Multi Level Wildcard.* The number sign "#" matches any number of levels within a topic, it must be specified on its own or after a topic level separator but, in both cases, it must be the last character specified in the Topic Filter[11] (e.g. "home/#" and "#" are valid while "home/#/lamp" is not valid);

Connection & Authentication MQTT connections are performed through a series of packets named MQTT Control Packets. The first packet sent by a Client (a publisher or a subscriber) to a server when a connection is established is the CONNECT packet. A Client can only send the CONNECT packet once over a connection, a second CONNECT packet must be processed as a protocol violation by the Server and forces Client disconnection. The payload of CONNECT packet contains one or more encoded fields including the Client Identifier (the only that is mandatory), Will Topic, Will Message, User Name and Password. The Will Message is a message that must be stored by the server and sent to the client's subscribers in case of a forced disconnection.

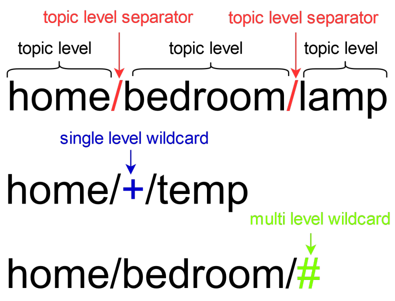


Figure 2.3. Topics and wildcards examples

User Name and Password fields are the built-in authentication method of MQTT, they are optional fields and it is possible to send a Username without a Password (but not vice versa). The CONNECT packet received by the server corresponds to a CONNACK packet sent to the Client. This packet is the first sent from the Server to the Client and if it is not sent in a reasonable amount of time (that depends on the application type) Client should end the connection. CONNACK contains in its variable header a Return Code used to validate Client credentials.

Value	**Return Code Response**
0	Connection Accepted
4	Connection Refused, bad user name or password
5	Connection Refused, not authorized

Table 2.1. Notable CONNACK Return Codes

When the connection is established the Client sends a PUBLISH packet that contains in the payload the message that will be forwarded to the subscriber by the Server. PUBLISH packet has a fixed header and a variable header. The fixed header contains the DUP field, which indicates that the message is a duplicate if set to 1, RETAIN, which indicate that the message must be stored by the MQTT Broker and delivered to future subscribers, and QoS, that will be examined further in a dedicated paragraph, together with other packets that are QoS-specific. The variable header contains the Topic Name and the Packet Identifier, a serial number that identifies the packet when QoS is greater than 0.

The SUBSCRIBE packet is used by Clients that want to subscribe to one or more topics. The payload of the packet contains the list of desired Topic Filters together with the chosen QoS. A SUBACK packet is sent by the Server as acknowledgment and also indicates the maximum QoS available. When a Client wants to unsubscribe one or more topics it sends an UNSUBSCRIBE packet to the Server, it is acknowledged by the UNSUBACK packet.

To check if the connection is active the packets PINGREQ and PINGRESP could be used. At the end of a connection, the Client sends a DISCONNECT packet to the Server which, once received the packet, will discard every Client's Will message. [11]

Quality of Service MQTT presents three levels of Quality of Service (QoS). QoS is decided between sender and receiver (publisher-broker and broker-subscriber), so it's not mandatory to have the same QoS level from publisher to subscriber. The three levels are:

- *QoS 0: At most once delivery.* This is the level of service with less quality. A single PUBLISH packet is sent to the receiver that accepts the ownership without sending responses. No retry is performed by the sender.

- *QoS 1: At least once delivery.* QoS 1 ensures that the message arrives at least once. Every PUBLISH packet has a Packet Identifier in its header and its acknowledged by a PUBACK Packet.

  The specificity of this level of service is the possibility to send further PUBLISH packets with different Packet Identifier, and it could be a problem in systems where duplicates are not acceptable. Once a PUBACK is received, the Packet Identifier could be reused for new packets.

- *QoS 2: Exactly once delivery* This is the service with the highest quality, it prevents duplication and loss of the packet. A QoS 2 message has a Packet Identifier in its header like a QoS 1 and uses a four-part handshake in the delivery of messages. As for other levels, the first packet is the PUBLISH packet.

  After that message there are two methods to handle the packet: with Method A the message is only stored by the broker, while with Method B only the Packet Identifier is stored and the message is forwarded to subscribed clients. Then the broker sends a PUBREC packet to the Sender (and a PUBREC packet for any PUBLISH packet with the same Packet Identifier) and waits for a PUBREL packet from the publisher. The broker must not deliver any duplicate message with different Packet Identifiers that could be sent by the publisher.

  After receiving the PUBREL packet with Method A the broker will forward the message to subscribers and delete the message, while with Method B broker simply discard the Packet Identifier. Finally, the broker sends a PUBCOMP packet to the publisher that will discard the initial message (it can also reuse the Packet Identifier) and start a new publication.

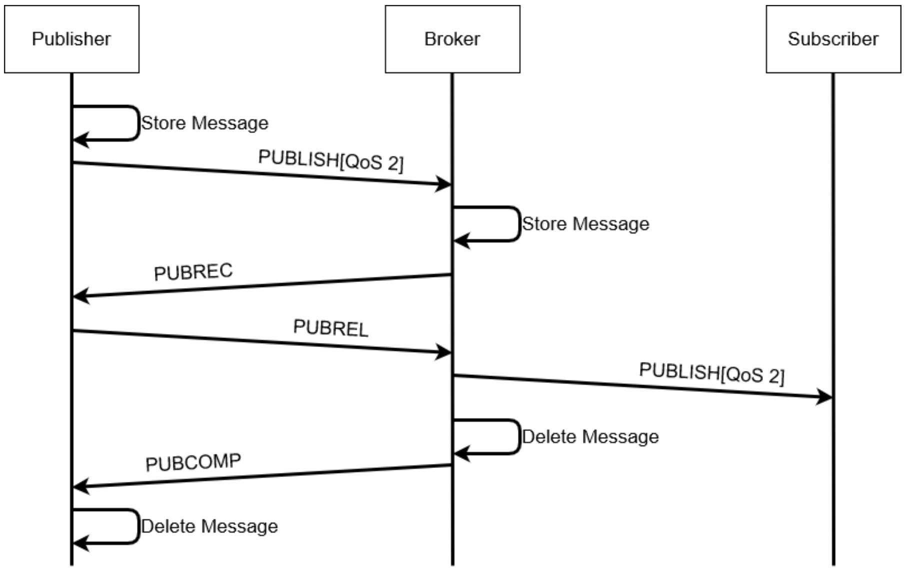


Figure 2.4. An MQTT QoS 2 delivery with Method A

2.2.1 MQTT Security

Security in a lightweight and IoT-oriented protocol as MQTT is an aspect with several challenges to face, due to the necessity of a balanced trade-off between a good level of security and the limited computing power and memory of IoT devices. It should also be considered that IoT platforms, in most cases, are not deployed and configured with a focus on security features, so, protocols and application design must cover this gap.

MQTT, as described in the previous section, provides a built-in authentication mechanism, but this mechanism is pretty basic and unsafe, because credentials are sent to the Broker in clear text and, depending on the Broker settings, subscription without authentication could be allowed by the Broker. There is also other information provided by MQTT that can be used for authentication such as the client identifier, a unique ID assigned by the broker, and X.509 client certificates, used with TLS for the handshake between client and server and reused for authentication purpose. Moreover, it is possible to implement authentication with database support. This is the methodology implemented in this work and will be addressed in Chapter 5.

For what concerns confidentiality, MQTT stands on top of a TCP connection. As for HTTP, TCP connections do not use encryption by default. For this reason, many MQTT brokers allow the possibility to use of TLS, TLS protects credentials if the built-in authentication is used, allow the use of certificates (even for authentication purposes) but in particular encrypt the payload of the packet giving confidentiality, and integrity, to the protocol.

Instead of the standard MQTT port, 1883, it is strongly recommended that Server implementations that offer TLS should use TCP port 8883, which IANA service name is `secure-mqtt`. It is implied that all the aspects related to TLS vulnerabilities and possible attack scenarios addressed at 2.1.1 may also be experienced in MQTT.

Authorization is another important aspect that should be taken into account. Without proper authorization, each authenticated client can publish and subscribe to available topics. MQTT 3.1.1 specifications advise to adopt authorization mechanisms in the implementation[11]. Read and write permissions could be implemented on the Broker side. There are different approaches to do so, mainly based on Access Control Lists(ACLs), that vary with the Broker. As for other implementation-specific aspects, security details on the authentication approach chosen will be described in Chapter 5.

MQTT security risks One of the main threats in MQTT remains the misperception of IoT devices' risk by users and misconfigured MQTT servers. Studies on users' privacy risk awareness found that risk assessment on IoT devices vary varies according to the manufacturer, the device functions, collection of bio-metrics data, and, more generally, with the context, as indicated by Naeni et al. [12]. Risk related to some devices, such as domotic lamps or temperature sensors, tend to be underrated, but those devices can be exploited and used as attack vectors in DDoS attacks.

In August 2018, Avast reports that it is possible to find, using the Shordan IoT search engine, almost 49,000 MQTT servers exposed on the internet due to a misconfigured MQTT protocol, including more than 32,000 servers that had no password encryption[10].

2.3 Related works

Other comparatives studies over HTTP and MQTT have been made during the years.

Yokotani and Sasaki[13] analyzed protocols overhead, transmitted bytes with increasing payloads, required server resources and transmitted bytes with increasing topic length (only for MQTT) for HTTP and MQTT. The study finds that MQTT is better in almost all cases excluding the case in which topic length is greater than 680 Bytes. In the study, it is also provided a solution to enhance MQTT performance for long topics through topics compression.

[10]`https://blog.avast.com/mqtt-vulnerabilities-hacking-smart-homes`

Naik, in its 2017 study[14], compares four IoT protocols including MQTT and HTTP. The criteria of the analysis highlight better results for MQTT in terms of overhead, power consumption and resource requirements, bandwidth and latency, reliability and M2M usage. HTTP resulted in better interoperability, security, provisioning and standardization.

Wukkadada et al.[15] focused their comparison on battery consumption and message delivery reliability concluding that MQTT is better of HTTP for power consumption to keep the connection open.

Chapter 3

The energy management system

The energy management system (hereinafter also named EMS) of Myna-Project consists of an open-source software suite and a distributed hardware network to collect, analyze and manage energy data in IIoT contexts.

The system aims to make aggregate energy data available to customers through a web interface in order to have a better understanding of corporate assets consumption and production(e.g. machines, solar panels, offices, etc.) and gives the possibility to find energy waste and excessive consumption.

Those data can also be used to estimate the cost of the energy component in the various production stages, enabling cost engineering operations to calculate production processes efficiency.

3.1 Network & Hardware Components

The physical part of the EMS is formed by the hardware devices and the network that connects them.

Hardware Components The hardware devices are:

- *Energy Meters.* Meters are directly connected to the monitored assets and detect input and output consumption;
- *Bridges.* Optional components, bridges are used for several reasons. Whenever a change of protocol is needed (e.g. from Modbus RTU, which is one of the most common protocols used by meters in the architecture, to Modbus

TCP) and when the number of interfaces in the microserver is not sufficient to connect the meters directly.

For Modbus RTU meters there is a third reason to use a Modbus RTU/TCP bridge: when assets are very distant from the microserver, which is a common scenario in large factories, long serial cables are subject to strong signal attenuation.

Ethernet cables are instead at least reliable for distances below 100 meters[1], they can also be extended through switches and, in most cases, it is more convenient for companies to have a larger Ethernet infrastructure than a serial one;

- *Microserver.* It is the device where all or part of the software infrastructure is installed and its located in one of the customer's facilities;
- *Data Center.* It could be owned by the customer or connected to a server in DMZ located in Myna headquarter (if the customer cannot preserve data on its own properly), the Data Center is where energy information is kept.

Network As mentioned in the previous paragraph, mostly of meter-bridge connections are in Modbus RTU, whereas connections between bridges and microservers are in Modbus TCP.

Each customer's microserver is directly reachable from Myna network via a VPN (Virtual Private Network) created with the open-source application OpenVPN[2]. The VPN is also present between Myna network and Customers Data Centers, allowing them to reach the Data Center.

External Cloud Data Centers receive data from their microservers through a secure internet connection and standing behind a reverse proxy, which functioning will be further discussed in section 3.2.

[1]`https://www.se.com/ww/en/faqs/FA269550/`

[2]`https://openvpn.net/`

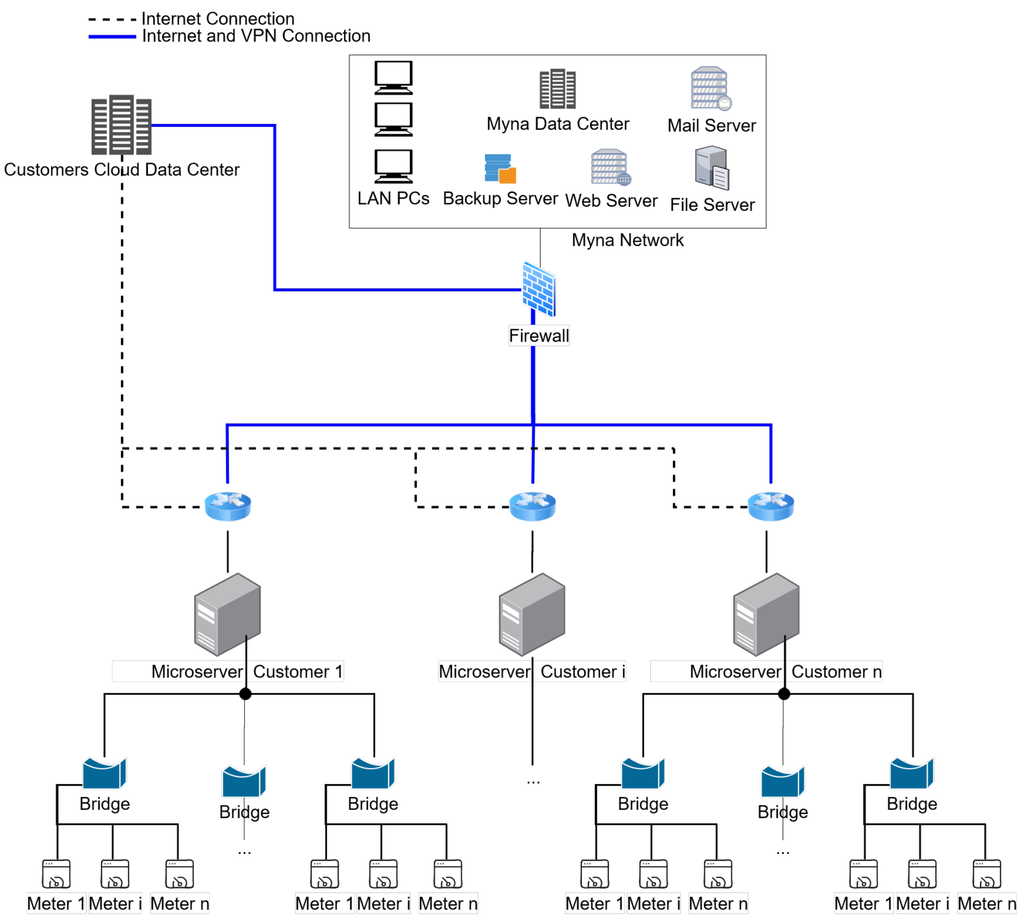


Figure 3.1. Network & Hardware Architecture Schema

3.2 Software Components

Every server in the architecture uses Linux Debian as O.S. and every application installed on is open-source, as well as the applications developed by Myna-Project, the core of the EMS.

The application required for the EMS are:

- *PostgreSQL*[3]. Relational DBMS (DataBase Management System) used by the back-end for data persistence;
- *Apache Tomcat*[4]. HTTP Application Server where back-end and front-end application are deployed;
- *OpenJDK*[5]. open-source implementation of the Java SE Platform, required for the functioning of IEnergyDa (the back-end);
- *Apache HTTP Server*[6]. The web server deployed in plants where a Reverse Proxy is needed.

Anyway, not every required application is needed in every plant and, generally, for plants where only the IoT Gateway is needed (e.g. certain customer's Micorservers) none of these applications is needed.

Software Suite The core of the EMS is represented by the three software components developed in Myna. These components are:

- *Wolf.* It is the **IoT Gateway** of the architecture, it can retrieve or receive (depending on the input protocol) measures from meters and bridges, saves data in a queue, and then convert and send them in output. It is written in Python and its structure is organized in input and output modules;
- *IEnergyDa.* IEnergyDa is the central part of the architecture. It is the **back-end** of the application and performs data analysis (from which letters "Da" in the name are derived) with data received from Wolf. It is developed in Java with Spring and Hibernate frameworks;
- *IEnergyUtils.* It is the front-end application used to request and visualize aggregated data from IEnergyDa with plots and tables. It is based on AngularJs.

[3]`https://www.postgresql.org/`

[4]`https://tomcat.apache.org/`

[5]`https://openjdk.java.net/`

[6]`https://httpd.apache.org/`

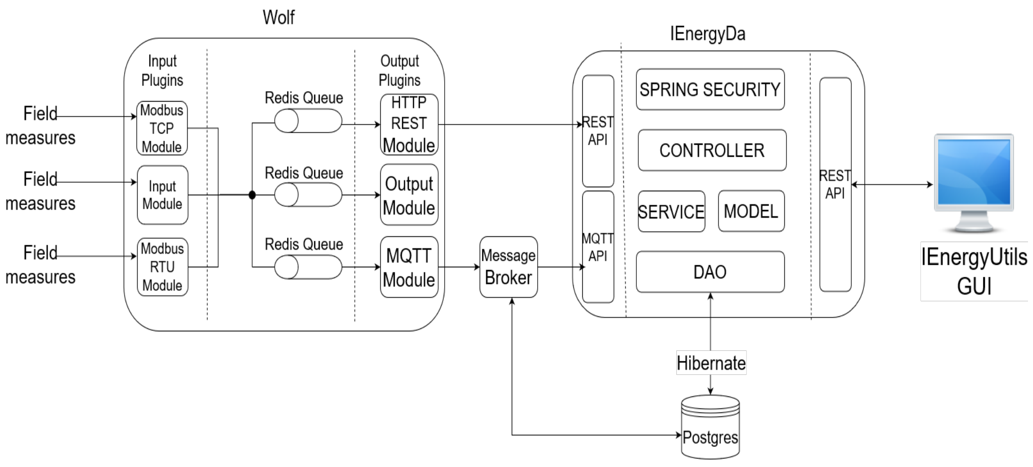


Figure 3.2. Software Architecture Schema

The two components that will be part of the trial, Wolf and IEnergyDa, will be detailed in the next Sections and, for protocols implementation, in chapter 4 and chapter 5.

3.3 Wolf

Wolf[7] is a lightweight and modular IoT gateway written in Python. Wolf receives or retrieves data, depending on the protocol used, from various industrial electronic devices and then stores it into InfluxDB[8] and/or sends it to other Information Systems through the output plugins.

Wolf also exposes REST services to get, set, or update the configuration, and to retrieve the measures data from InfluxDB, these REST services are used by WolfUI (Wolf User Interface) to provide the web-based graphical interface. Wolf design is lightweight to run low-performance environments such as embedded systems.

Plugins & Configuration Wolf supports the following input plugins:

- AMQP - JSON and XML data format
- EnOcean

[7]`https://github.com/myna-project/Wolf`

[8]`https://www.influxdata.com/`

- IO-Link - JSON
- Modbus RTU
- Modbus TCP
- MQTT - JSON, Raw and XLM data format
- TPLink Smart plugs

and the following output plugins:

- AMQP
- InfluxDB (for local storage)
- **IEnergyDa MQTT**
- **IEnergyDa HTTP REST**

Wolf searches and loads plugins in the plugins directory and use them if they are configured in the configuration **wolf.ini** file. Every plugin requests his own configuration parameters and only REST and MQTT output plugin configuration will be detailed in chapter 4 and chapter 5, respectively.

Unlike plugins configuration, Wolf configuration is always required. The fields of Wolf configuration are:

- *loglevel.* Describe the verbosity of the log file;
- *interval.* An integer parameter that describes the time interval between a pooling cycle and the next one. When the interval specified is greater than 60, its value is rounded to the closest multiple of 60;
- *clientid.* It is a parameter needed in IEnergyDa, where Wolf is recognized as a specific case of an asset (client).

Redis cache Wolf uses a back-end database on Redis[9] to implement caching and data persistence. Cache implementation use *<plugin_name>.<instance_id>* as **key** and the *msgpack* of the JSON, made by an input plugin, as **value**. To make use of Redis cache a configuration inside wolf.ini is required. The parameters needed are:

- *host.* IP address of redis server;
- *port.* Port of redis instance;
- *db.* Redis database name;
- *expire.* Max number of days for data retention (default value is no expire).

[9]https://redis.io/

Data from input plugins are cached until they are requested from the output plugin or sent in a pooling cycle. In case of failure, data remain on the cache to be sent back in the next pooling cycle until the transmission succeeds. The parameter expires is usually not needed, but it can be useful for installations with low disk space, such as embedded systems.

3.4 IEnergyDa

IEnergyDa (also called I-Da) is a Java back-end and data-analysis software for industrial energy management. The project uses Spring[10] framework and its modules for the application design structure (Spring MVC) and security (Spring Security). Application's build and dependencies management are delegated to Apache Maven[11].

Interaction between I-Da and the Postgres database is handled with Hibernate[12], an ORM (object-relational mapping) framework which maps Java classes to database tables and Java types to Postgres data types.

Hibernate also provides a query language called Hibernate Query Language (HQL) which uses an SQL-like syntax to create queries with parameters. HQL queries are checked and escaped by the engine to mitigate SQL-injection attempts.

Even if mitigated, SQL-injections are still possible if HQL is used in an incorrect way[13], so developers should take other measures to secure the application, such as avoid improper usage of *native* HQL queries[14].

Database structure The database structure is mainly based on eight entities. Three of them are related to user authorization and authentication:

- *Role.* This entity indicates the user's permissions. There are three types of roles: ROLE_ADMIN, can access to every resource in both reading and writing, ROLE_USER, a user with this role can access every resource in the organizations for which it has a Job, and ROLE_USER_RO, which is the same of ROLE_USER but with read-only access to resources;
- *User.* A personal account used for authentication;
- *Job.* It is the link between Users and Organizations, every user can access one or more organizations according to its Jobs.

[10]`https://spring.io/projects/spring-framework`

[11]`https://maven.apache.org/`

[12]`https://hibernate.org/`

[13]`https://owasp.org/www-community/Hibernate`

[14]`https://web.archive.org/web/20170227185238/https://owasp.org/index.php/Hibernate-Guidelines/#Important`

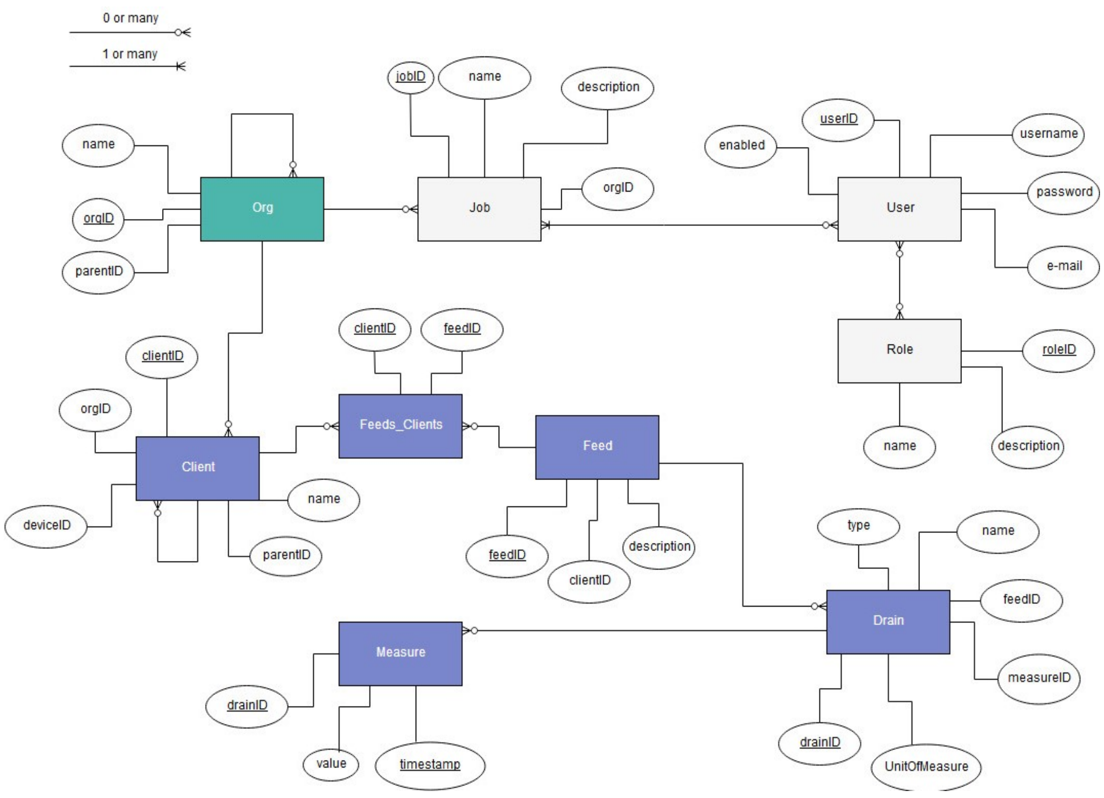


Figure 3.3. Entity-relationship model of IEnergyDa database (intermediate entities excluded)

Other relevant entities involve energy measures and are organized in a hierarchical structure. These entities are:

- *Organization.* An organization (or org) corresponds to a company, a plant or an industry. Organizations are recursive, so a single org can be only a part of a business, like a branch or a subsidiary of a company, and thus be a "child" organization of the main one;
- *Client.* It is a measured asset for the organization. A client can be a building, a machine, a solar panel, or any other device that produces or consume energy. Every client is associated with an organization and like organization can be recursive;
- *Feed.* It represents the physical dimension of a measure. Examples of Feed can be Energy, Power or Voltage.
- *Drain.* A drain is the detailed element measured, as many clients, such as

compressors and other heavy machinery, are three-phase systems is needed to distinguish a phase from another or the sum of them;

- *Measure.* It is the raw data, represented by a float value, a timestamp and an associated drainID.

Structure The structure of IEnergyDa is strongly based on Spring MVC as the project has been initially created to be a part of RESTful architecture. MVC stands for Model-View-Controller and it is a well known and popular software design pattern for the development of web applications. The first elements, for the definition and startup of the application, are **servlet** (particularly DispatcherServlet) and **applicationContext**.

Servlets & Controllers A servlet is a Java technology-based web component, managed by a container, that generates dynamic content. The servlet container, sometimes called the servlet engine, is a part of a Web server or application server (for I-Da the application server is Tomcat) that provides the network services over which requests and responses are sent. A servlet container also contains and manages servlet through their life cycle[16]. It also improves exception handling through HTML pages generated and returned to the user after exception processing.

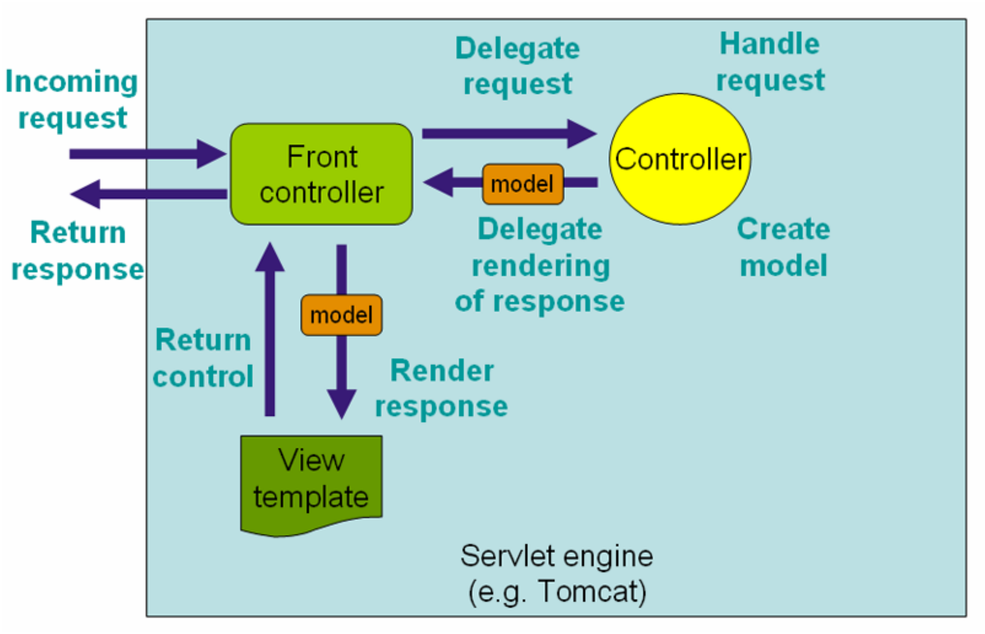


Figure 3.4. The request processing workflow in Spring Web MVC (high level)[3]

Spring's `DispatcherServlet` is a type of servlet that is more integrated with other Spring's features. As shown in Figure 3.4, `DispatcherServlet` delegate requests to a specific Controller, depending on the URL. Controllers, which represent the C in the word MVC, provide access to an application behavior typically defined through a service interface. Controllers interpret user input and transform it into a model that is represented to the user by the view. Spring implements controllers in a very abstract way, which enables to create a wide variety of controllers[3].

With Spring 2.5 annotation-based programming model for MVC controllers has been introduced. Java **annotations** are syntactic metadata, preceded by "@" symbol, that can be added to classes, method, fields and other meta-objects. They are used as an alternative or in conjunction with XML for Spring configuration, and can also be used in several cases, such as for Inversion of Control (IoC) design pattern, serialization and ORM (they are also used by Hibernate). Controller annotations will be dealt with other HTTP implementation aspects in chapter 4.

ApplicationContext To introduce the concept of application context it is first necessary to explain the concept of IoC and introduce the concept of bean. Inversion of control, also known as dependency injection (DI), is a process in which objects define their dependencies through constructor arguments or properties that are set on the object instance after it is constructed from a factory method.

This process is the inverse of what happens with a direct construction of classes, or with mechanisms such as the Service Locator pattern, where the bean itself verify the installation or location of its dependencies. A **bean** is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container. In other words, it is simply an object within the application, such as a service or a DAO (Data Access Object) in IEnergyDa.

To manage beans Spring utilize an advanced configuration mechanism: the BeanFactory interface. ApplicationContext is a sub-interface of BeanFactory that adds an easier integration with Spring's AOP (Aspect Oriented Programming) features, message resource handling (for internationalization, event publication and application-layer specific contexts[3]. In IEnergyDa the main bean components can be categorized as follows:

- *Services*;
- *Dao*;
- *AuthenticationHandler*;
- *CronJobs.*

Service components are used to implement the business logic separated from the REST Controller and their classes have the @Service annotation. Dao is strictly linked to services and implements the part of operations related to Hibernate database transactions. In Dao classes, HQL is used on entity classes (defined by the @Entity annotation) which reflect the database entities.

AuthenticationHandler is used for application login. Details on its implementation are given on subsection 4.2.1. CronJobs instead, is a set of triggers to activate periodically the classes that check interruptions or errors in the flow of energy measures from Wolf and, in one of these cases, send a report to the administrator. Triggers activation time indicated through cron expressions.

Model & View The Model, which is the M in word MVC, is how a controller communicates data to the view. The format chosen to send this data is JSON (JavaScript Object Notation) format, for its better readability compared to XML. Finally, the View (the V in MVC), instead of a classical JSP (JavaServer Pages) view approach, is externalized on the client-side and it is represented by the AngularJs User: Interface IEnergyUtils.

Chapter 4

HTTP implementation

HTTP implementation, as mentioned in the previous chapter, is based on REST communication between Wolf and IEnergyDa. REST (REpresentational State Transfer) is an architectural style, defined in 2000 by Roy Fielding, for providing standards between systems on the web with the aim of facilitating their communications.

REST has 6 (5 plus an optional one) architectural constraints, or guiding principles, that must be respected to create a REST compliant interface, also defined RESTful. These principles are:

1. **Client-Server**. The first constraint is about the separation of user interfaces from the data storage concerns, to improve the portability of user interface across multiple platforms and to improve scalability by simplifying server components. The separation also allows an independent evolution of the components.
2. **Stateless**. Session state is kept entirely on the client and each request from the client to server must contain all of the information necessary to understand the request. This constraint improves visibility because a monitoring system does not have to look beyond a single request to determinate its nature, reliability, because its easier to recover partial failure, and scalability because the server does not need to store state from every user and quickly free resources.
3. **Cache**. To improve network efficiency it is required that response data will be implicitly or explicitly labeled as cachable or not. If a response is cachable, then a client can reuse it for later, equivalent requests.
4. **Uniform interface**. By applying the software engineering principle of generality to the component interface, the overall system architecture is simplified and the visibility of interactions is improved. In order to obtain a uniform interface, multiple architectural constraints are needed to guide the behavior of components. REST is defined by four interface constraints: identification of

resources, manipulation of resources through representations, self-descriptive messages hypermedia as the engine of application state (HATEOAS).

5. **Layered System**. This principle allows an architecture to be composed of hierarchical layers by constraining component behavior such that components cannot be visible beyond the immediate layer with which they are interacting.
6. **Code on demand (optional)**. REST allows client functionality to be extended by downloading and executing code in the form of applets or scripts. This simplifies clients by reducing the number of features required to be pre-implemented.

The key abstraction of information in REST is a **resource**. Any information can be a resource and REST uses a resource identifier to target a particular resource involved in an interaction between components. The state of a resource in a particular moment is known as resource representation and consists of data, metadata that describes the data and hypermedia links to help clients in transition to the next desired state [17].

Resources are linked to resource methods used to access and modify a specific resource. In HTTP, the pre-defined methods (GET, POST, PUT and DELETE) can be used as resource methods, even in Fielding's work, there aren't recommendations on which method use and in which condition. The only requirement is to maintain a uniform interface as declared by the fourth guiding principle.

In the next sections, the HTTP REST implementation in Wolf REST output plugin and IEnergyDa input REST API will be detailed.

4.1 Wolf REST output plugin

Wolf REST output plugin is searched and loaded from the main component of the gateway on startup, together with other installed plugins.

The configuration of REST output plugin has four parameters:

- *baseurl*;
- *username*;
- *password*;
- *retries*;
- *backoff*.

Baseurl is the REST API endpoint that Wolf use to contact I-Da, represented by an HTTPS URL. **Username** and **Passoword** are credentials for authentication purposes. **Retries** is an integer number that indicates the number of retries that will be attempted in case of error and it is related to **Backoff** which is a float number that represents an exponentially increasing time in seconds that will be waited for every retry in order to mitigate response lack due to a server overload. The last two parameters for retry attempts are used as part of a specific Retry object which is included in the **urllib3** library, an HTTP client for Python used in Wolf.

Once the plugin parameters are loaded, on the application start-up, the function `__post_config()` is activated from the main thread. This function runs only once when Wolf starts, and it is used to send the configuration of Wolf's clients, feeds and drains to I-Da, which will add or update its previous configuration (if necessary), in order to avoid mismatch of entities between the gateway and the back-end.

Every n seconds (where n is the parameter *interval* defined in Wolf configuration) the function `post` is called by the main module. In this function raw data are picked up from redis queue and parsed to form the JSON file that will be send.

The function `__post()` is called by the two functions described previously to concretely send both configuration and field data to I-Da through a POST request. By going into detail of this function, it can be noticed that, before the POST request, if the **csrf** header is missing, function `__get_token()` is called.

In function `__get_token()` it is performed a GET request to I-Da in order to obtain a X-CSRF-TOKEN which is used to protect the application from **Cross-site request forgery** (CSRF) attacks. This type of protection, covered in subsection 4.2.1, is unnecessary for non-browser clients[18] like Wolf, but it is needed for Web UI, so it is mandatory for communications with I-Da and leaves also possibilities for future Wolf-based UIs.

When X-CSRF-TOKEN is set, the `auth` variable is filled with username and password and the POST request is sent. If the HTTP response code is equal to **403**, it could be possible that X-CSRF-TOKEN expired, so a recursion flag is set and another request with a new token will be tried. If the recursion variable is already flagged, the request operation will not be repeated to avoid recursion and an error message will be displayed. For any other error case, the request will be repeated according to the Retry variable.

4.2 IEnergyDa REST API

4.2.1 Authentication & Authorization

As mentioned previously, I-Da security features are provided with Spring Security, a powerful and highly customizable authentication and access-control framework which represents the de-facto standard for securing Spring-based applications [18]. The class where security features are implemented and configured is called `SecurityConfig`.

`SecurityConfig` starts with two Spring annotations, `@Configuration` and `@EnableWebSecurity`. The first one is used to declare one or more beans that need to be dealt on run-time, while the second one is a marker annotation and, combined with @Configuration, allows Spring to find and automatically apply the class to the global WebSecurity and switch-off the default web security configuration.

For the authentication part, `UserDetailsService` interface is used. This component is a DAO interface for loading data that is specific to a user account. Once an `AuthenticationManager` is created, the `UserDetailsService` is assigned to it together with a password encoder. The password encoder originally chosen for the application was BCrypt but, after the introduction of MQTT in the architecture with the go-auth plugin (details on subsection 5.1.2), the algorithm has been changed in favor of PBKDF2, with a custom encoder to meet the go-auth specifics.

The main method of the class is the `configure(HttpSecurity http)` method. This method accomplish several security functions:

- Defines a CORS filter;
- Defines a CSRF Token filter;
- Specifies authorization rules;
- Ensures that any request to the application requires the user to be authenticated;
- Sets the authentication details.

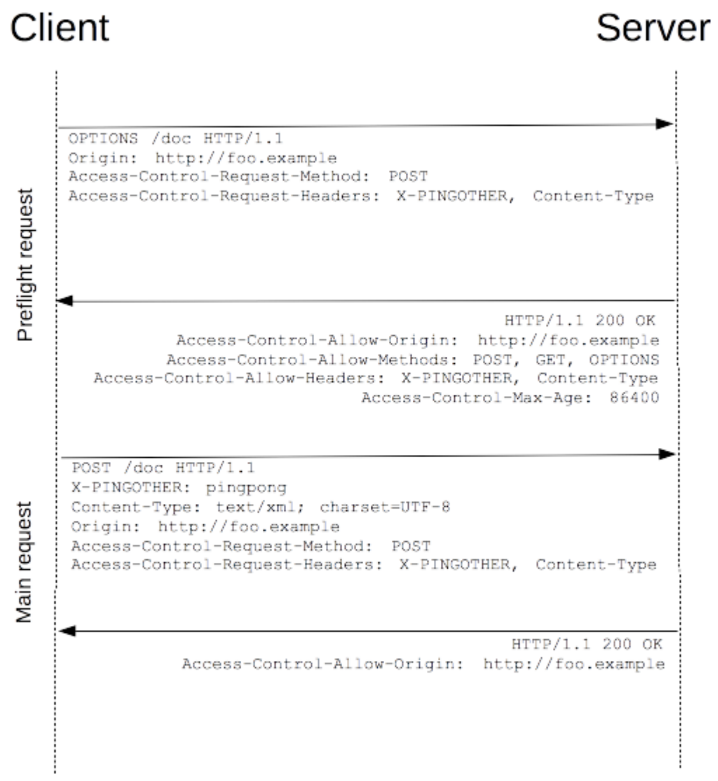
The CORS filter CORS is a mechanism that uses additional HTTP headers to allows a web application that runs on a certain origin[1] to access selected resources from a different origin. For security reasons, browsers restrict cross-origin HTTP requests initiated from scripts, such as `XMLHttpRequest`.

CORS mechanism supports secure cross-origin requests and transfers between browser and server. If a request is considered a **simple request**, e.g. a GET or a HEAD without custom header, the request is performed without additional actions. Other requests, that may have implications to user data, are called **preflighted**, since before the actual request a call with `OPTIONS` method is performed by the client to determinate if the cross-origin request is allowed [4].

In I-Da, CORS requests are managed through a dedicated class which import from a property file the list of `originsAllowed`.

This list varies according to the configuration file of each installation. If a prelighted request origin is in the list, I-Da will send a response with a **202 - Accepted** status code together with `Access-Control-*` headers of allowed credentials, headers and methods.

[1]An origin is formed by a schema and a host.



```
OPTIONS /doc HTTP/1.1
Origin: http://foo.example
Access-Control-Request-Method: POST
Access-Control-Request-Headers: X-PINGOTHER, Content-Type
```

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: http://foo.example
Access-Control-Allow-Methods: POST, GET, OPTIONS
Access-Control-Allow-Headers: X-PINGOTHER, Content-Type
Access-Control-Max-Age: 86400
```

```
POST /doc HTTP/1.1
X-PINGOTHER: pingpong
Content-Type: text/xml; charset=UTF-8
Origin: http://foo.example
Access-Control-Request-Method: POST
Access-Control-Request-Headers: X-PINGOTHER, Content-Type
```

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: http://foo.example
```

Figure 4.1. Example of a request with a previous preflight[4]

CSRF A CSRF attack force a logged victim's browser to execute unwanted actions on a web application. This is due to the fact that, for most sites, browser requests automatically include any credentials associated with the site, including session cookies and IP addresses.

An attacker can send a hyperlink to an end-user and convince him to open it through social engineering techniques. Once the user clicks on a mock form or, with an automated request, simply opens the malicious link a forged HTTP request is sent by the attacker's site to the attacked web site[19]. Even if the attacker can't access to the response of that request, a state-changing request sent with this method can be a serious threat, allowing operations such as money transfers.

Spring offers several countermeasures against CSRF attack: in Spring MVC, with `@EnableWebSecurity`, a default CSRF token protection is provided. This solution, known as Synchronizer Token Pattern, is based on a randomly generated token as an HTTP parameter, in addition to the session cookie. When a request is submitted the server will compare the generated token with the one in the request. If the value doesn't match, the request will be refused.

Nevertheless, as I-Da use JSON requests, it is not possible to submit the CSRF token with an HTTP parameter. It is instead possible to use HTTP headers. On the initial visit to the web site, and without an associated session, a user retrieves with a safe method a cookie containing a randomly generated token. This token is sent in the `X-CSRF-TOKEN` and will be used for every state-changing request. The attacker can only guess the value of the token and this strongly mitigates the attack because a right guess is highly unlikely.

`CsrfFilter` class handles the CSRF token mechanism as discussed above, creates the `X-CSRF-TOKEN`, check and filters the requests with a correct token. `CsrfFilter` token retrieve use a GET request, as this method is considered safe. This assumption is true only if HTTP verbs are used properly and GET request is used only to retrieve information and works without a CSRF token. Otherwise, GET request cannot be considered safe as private information in an HTTP GET could be leaked[18].

Authorization I-Da authorization policy is handled with a **RBAC** (Role-Based Access Control) approach. As described for the database structure, there are three active roles within the application:

- *ADMIN*;
- *USER*;
- *USER_RO*.

Every role has the prefix `ROLE_` and can access only certain URLs. Custom requirements for URLs shall be specified by adding multiple `antMatchers()` to the `http.authorizeRequests()` method.

Children of `authorizeRequests()`, also called matchers, are considered in the order of their declaration and can use regular expression pattern, starting from the most specific to the most generic pattern. The *Admin* user can access every resource:

```
.antMatchers("/**").hasRole("ADMIN")
```

while the access to the token for the **CSRF** protection and to the swagger interface is allowed to anyone by the `.permitAll()` method.

Requests authentication The last part of the security configuration sets authentication details. In this section of code is specified that every request to the application must be authenticated and specify the URL to validate username and password. There are also specifications of handlers for successful and failed Login, successful logout, and for authentication method which is HTTP Basic authentication. As already addressed in chapter 2, Basic authentication is not a safe authentication method unless the communication is under HTTPS, which must be used (the mandatory redirection from HTTP to HTTPS is handled in the Apache Web Server configuration).

4.2.2 Measures handling

The controller dedicated to managing the incoming measures from Wolf is named `WolfController`. It is divided into two sections, one for the admin, and one for the user roles, which perform similar functions but map different URLs.

For every section, there are two methods `POST`, one to handle a JSON file of measures for a certain client in a certain timestamp called `PostMeasures`, and one to handle a matrix of measures for the same or for several clients with a different timestamp and called `PostMeasuresMatrix`.

During the trial, `PostMeasuresForUser`, which is the single client POST method for users, has been used for the sake of simplicity, but there is no substantial difference in the choice between the two methods ahead of trial's goals.

`postMeasure` method is preceded by a `@RequestMapping` annotation. This annotation is used to map web requests onto specific handler classes or methods. A `@RequestBody` annotation is used to indicate a method parameter that should be bound to the body of the web request. For I-Da, the method parameter is a JSON file containing measures and mapped with the class `CsvMeasuresJson`.

In the body of the method, the authenticated user will be retrieved through `SecurityContextHolder` class, with `getAuthentication()` method. Even if the RBAC authorization approach filters user roles, `postMeasure` performs another check on the user to verify if it has the right to access that client (through its jobs). If the user cannot access that client a **404 - Not Found** status code will be returned to the user. The response is a 404 and not a **401 - Unauthorized** because this message could give information not needed to the user on the presence (or not) of certain clients onto the database.

If the user can access the client, measures will be collected into the Postgres database through service and DAO interfaces. After a correct insertion of measures, a response with a **201 - Created** HTTP status code will be returned.

Chapter 5

MQTT implementation

The implementation of MQTT in Myna architecture has been designed to be an alternative to the REST implementation. It has been tried to integrate as much as possible MQTT with the existing architecture, complying with its principles.

An example of this approach (that will be further discussed on section 5.1) is the name of the topics where measures are sent: a topic is created for every user linked to a specific installation, and that user can only send measures on its topics.

An MQTT broker has been placed between Wolf and IEnergyDa to allow a unidirectional communication from the publisher (Wolf), and the subscriber (IEnergyDa).

The chosen MQTT broker is Mosquitto, which offers not only message broker capabilities, but also security functions, like TLS communication, and Access Control Lists. The broker, by means of the Go Auth plugin, is also directly connected to the same Postgres database used by IDa.

In the next Mosquitto will be analyzed in the details, while Wolf and IEnergyDa parts will be analyzed in section 5.2 and section 5.3 respectively.

5.1 Mosquitto Broker

Mosquitto[1] is an open-source message broker developed by Eclipse Foundation.

In Debian, Mosquitto can be installed from its repository on APT[2]:

```
apt-get install mosquitto
```

[1] `https://mosquitto.org/`

[2] Advanced Packaging Tool, a command-line package manager for Debian

Once installed, Mosquitto must be configured. The configuration file is located on **/etc/mosquitto/mosquitto.conf** and in our case it will have two parameters:

```
acl_file /etc/mosquitto/auth/acls
allow_anonymous false
```

the first parameter specifies the location of the ACLs file, while the second parameter blocks anonymous users. In fact, Mosquitto allows anonymous users so it must be specified that this option is unwanted to make the authentication mandatory.

5.1.1 ACLs

In ACLs file the access rights on a specific topic are defined with the **topic** parameter in the following way:

```
topic [read|write|readwrite] <topic>
```

where the access type is one of the three defined in the brackets and `<topic>` is the topic name.

The first set of topics in the file are applied to anonymous clients, if `allow_anonymous` is set to true in Mosquitto configuration. User-specific topics can be defined after the **user** parameter.

It is also possible to define ACLs based on pattern substitution within the topic. In this case, the form is the same of **topic** parameter which is substituted with the word **pattern** and with **%c** and **%u** to indicate the client or the user in the topic[20].

An example of a working configuration for Myna architecture is the following:

```
pattern read wolf/%u/#
pattern write wolf/%u/#
user admin
topic write #
topic read #
```

In this example, the user **admin** (which is defined in I-Da MQTT configuration) can write (publish) and read (subscribe) any topic, while every other user could write and read on its wolf/`username`/# topics.

It is worth noting that, while an anonymous user cannot connect to the broker, an authenticated user can publish and subscribe to any topic. In fact, the authorization policy of Mosquitto is not explicit. If an authenticated user tries to publish on a topic without the right authorization, the message will be discarded by Mosquitto without notifications to the user. Similarly, if a user subscribes to a topic for which is not authorized, it will never receive messages.

5.1.2 Authentication plugin

Mosquitto, in addition to its authentication features, has basic built-in authentication mechanisms. These mechanisms are based on textual password files configured by the user. However, we preferred an authentication system that can connect a back-end database.

Mosquitto functions can be integrated with plugins and for authentication the Mosquitto Go Auth plugin has been chosen. **Mosquitto Go Auth**[3] is a mosquitto plugin almost written in Go which implements several back-end authentications, including:

- *Files*;
- *PostgreSQL*;
- *JWT (with local DB or remote API)*;
- *HTTP*;
- *Redis*;
- *Mysql*;
- *SQLite3*;
- *MongoDB*;
- *gRPC.*

The Postgres back-end has been used to integrate the preexisting IDa authentication with mosquitto. With this connection, it is possible to use I-Da users also on the broker.

An example of a Go Auth configuration (located on **/etc/mosquitto/conf.d/ auth.conf**) is the following:

```
auth_plugin /usr/lib/mosquitto-auth-plugin/go-auth.so
auth_opt_backends postgres
auth_opt_pg_host localhost
auth_opt_pg_port 5432
auth_opt_pg_dbname ienergy
auth_opt_pg_user ienergy
auth_opt_pg_password ienergy_pwd
auth_opt_pg_userquery select password from users where
                   username = $1 and enabled = 1 limit 1
auth_opt_log_dest file
auth_opt_log_level debug
```

[3]https://github.com/iegomez/mosquitto-go-auth

```
auth_opt_log_file /var/log/mosquitto/auth.log
```

Besides database configuration (host, port, database name etc.) and logs parameters, the *userquery* parameter is used to retrieve the user's password from the Postgres database.

Go Auth plugin requires PBKDF2 hash. PBKDF2 (Password-Based Key Derivation Function 2), is a deterministic algorithm used t derive cryptographic keys from a secret value (e.g. a password). The key derivation is obtained from the secret value itself, a salt and an iteration count.

A **salt** is a non-secret binary value used as an input to the key derivation function. Salt should be large and sufficiently random to assure that the generated key is difficult to compute and unlikely to be selected twice. An **iteration count** is a method to increase the cost of producing keys from a password through algorithm iterations. From a mathematical point of view, an iteration count will increase the security of the password of $\log_2(c)$. Another important parameter for PBKDF2 is the underlying **hash function** used by the algorithm (e.g. SHA-256 and SHA-512)[21, 22].

The PBKDF2 hash required by Go Auth contains the word PSKDF2, the hash function, the iteration counter, the salt and the password hash separated by the $ symbol. An example of the hash is the following:

```
PBKDF2$sha512$100000$znG9iOH+a2oOSgoSyec56A==$4+GzKfvFd3cYszjwTesu
DYbIiPh5GUCVpl/2Nbq8y+97eSocqWj5t6IF4xbyiZgC60Fe1GdctZ/QBfLd0starA==
```

Debian installation The plugin does not provide an installation packet for Debian (or other distribution/architecture). Only the source code of Go Auth is available to be compiled. To compile the plugin, mosquitto sources are needed. In Debian 10 it is possible to obtain these sources, after enabling them on ATP **sources.list**, with the command:

```
apt-get source mosquitto
```

Then, to compile (for mosquitto v. 1.5.7):

```
export CGO_CFLAGS = -fPIC -I../mosquitto-1.5.7 -I../mosquitto-1.5.7/lib
export CGO_LDFLAGS = -shared
make
```

The file obtained, **go-auth.so**, is the plugin, and will be moved to **/usr/lib/mosquitto-auth-plugin** folder.

Myna created a fork of the plugin[4] which presents two patches. One is for is to solve the unavailability of the accessory functions `mosquitto_client_id()`

[4] `https://github.com/myna-project/mosquitto-go-auth`

and `mosquitto_client_username()`. The other patch execute a formal control on password hashes to avoid plugin's exemptions and crashes.

5.1.3 TLS on Mosquitto

Besides authentication and authorization functionalities, a TLS configuration (on **/etc/mosquitto/conf.d/ssl.conf**) has been added. A secure TLS communication must be used for a public message broker. The configuration parameters are:

- *port.* The default port used by mosquitto listener;
- *listener.* This parameter can be used multiple times, indicates other listener ports. The port 8883 is the one used for MQTTS;
- *cafile.* Used to define the path to a file containing the PEM[5] encoded CA certificates that are trusted;
- *keyfile.* Used to define the path to the PEM encoded keyfile;
- *certfile.* Used to define the path to the PEM encoded server certificate;
- *tls_version.* Configure the version of the TLS protocol to be used for the listener[20].

5.2 Wolf MQTT output plugin

The Wolf MQTT output plugin an REST output protocol work in a similar way. Both protocols send measures to I-Da with an interval set in Wolf configuration. The plugin is developed with Eclipse Paho library and supports the following parameters:

- *host.* The IP address or the hostname of the MQTT broker;
- *port.* The port of MQTT broker (default is 1883, 8883 when it is on TLS;
- *username.* The username of the sender;
- *password.* The password of the sender;
- *transport.* The transport protocol between tcp and websocket;
- *protocol.* Select the protocol version between MQTTv31 and MQTTv311;
- *keepalive.* Used to keep a session with the broker open for a certain period of time (default is 60 seconds);
- *topic.* The name of the topic in which the plugin publish (topic should possibly be authorized on mosquitto for the publisher user);

[5]PEM (Privacy-Enhanced Mail) is the de-facto standard file format used to store and send ciphered data.

- *qos.* The quality of service. Values are 0, 1 and 2 with 0 default;
- *cacert.* The CA certificate, required only with TLS;
- *tlsversion.* Minimum TLS version. Available versions are: TLS1.0, TLS1.1 and TLS1.2 (default);
- *tlsverify.* Server certificate validation (default disabled);
- *retain.* Flag MQTT for message retain (default disabled).

The body of Wolf MQTT plugin is very simple. It use functions `post_config()` and `post` to send client's configuration an measures to IDa, similarly to its REST counterpart.

There are also four functions that manage the connection with the broker:

1. `on_connect()`, notify a successful connection in the log and an associated **rc (result code)**;
2. `on_disconnect()`, notify a disconnection when the rc is not 0;
3. `on_pubblish()`, notify that a message has been successfully published on broker and the **mid (message identifier)**;
4. `stop()`, disconnect Wolf from the broker.

There are six defined result codes in Eclipse Paho:

- *0.* Connection successful;
- *1.* Connection refused - incorrect protocol version;
- *2.* Connection refused - invalid client identifier;
- *3.* Connection refused - server unavailable;
- *4.* Connection refused - bad username or password;
- *5.* Connection refused - not authorised.

Result codes from 6 to 255 are currently unused[23].

If QoS>0, the plugin also logs message delivery, but there is only a guarantee that the message arrived to the broker. There is no way to know if the message also arrived to I-Da.

5.3 IEnergyDa MQTT API

I-Da MQTT API has been introduced with the class `MqttStarter` and the related component. The `MqttStarter` component is executed through Application Context on IEnergyDa startup.

As for Wolf, also I-Da implements MQTT with the Eclipse Paho library, in its Java version. When the components starts, it requires from the **config.properties** file all the necessary parameters:

- *mqttTopics.* Optional topics that can be manually added to be subscribed ;
- *mqttServerURI.* the URI of the server and the port, in the form `tcp://address:1883` for insecure connections and `ssl://address:8883` for TLS connections;
- *mqttCaCert.* The path of CA certificate (mandatory for TLS connections);
- *mqttCert.* The path of user certificate (optional for TLS connections);
- *mqttKey.* The path of user private key (optional for TLS connections);
- *mqttQos.* The QoS level for messages (default is 0);
- *mqttUser.* The username of the administrator;
- *mqttPassword.* The password of the administrator.

As "administrator" is intended the user, defined in mosquitto ACLs, with the right to read and write on any topic.

When properties parameters are set the method `afterPropertiesSet()` is executed. In this method, all the users with the role **ROLE_USER** are collected and for every user a topic **wolf/`username`/config** and a topic **wolf/`username`/measures** are added to the topics list.

At this point, an `MqttClient` is created with all the related properties (username, password, and TLS configuration, if any). Then, the client connects to the broker and all the topics in the topics list are subscribed.

The method `destroy()` is used to close the connection to the I-Da from the broker if it is connected.

The methods `subscribeUser(User u)` and `unsubscribeUser(User u)` are used respectively to create and unsubscribe config and measures topics when a used is created or deleted. The method `unsubscribeUser(User u)` is also called when the role is changed for a user because topics are created only for the **ROLE_USER**.

If the connection between IDa and mosquitto is interrupted, the method `connectionLost(Throwable throwable)` is executed. This method tries, if it is possible, to disconnect the client and then tries to connect it again and to subscribe to the topics list.

Incoming messages To handle the messages received from the subscribed topics, the method `messageArrived(String topic, MqttMessage mqttMessage)` is used.

This method has two input parameters: the topic of the received message and an `MqttMessage` object, that contains the payload and the options of the message.

The message received has the same JSON file sent with REST as payload. This file has, for measures messages, the following form:

```
{"device_id": "SN20200411084541", "at": "2020-04-17T15:03:00+0200",
"measures": [{"value": 123.4, "measure_id": "M1"},
            {"value": 123.4, "measure_id": "Mi"},
            {"value": 123.4, "measure_id": "Mn"}],
"client_id": 282}
```

In this JSON, **device_id** is the asset measured, **at** is the timestamp, **client_id** is the Wolf client that handles the asset and **measures** is an array of measures identified by **measure_id** which is the identifier for the drain of the measure and **value** which is the measured value.

The config JSON file, is instead an array of configuration for every drain, with the form:

```
[{
'client_id': '1234', 'device_id': '1234', 'measure_id': 'M1',
'plugin_id': 'modbus_tcp.1', 'device_descr': 'solar_panel07',
'measure_descr': 'RMS sum active power', 'measure_unit': 'kW',
'measure_type': 'f'},
...
'client_id': '1234', 'device_id': '1234', 'measure_id': 'Mn',
'plugin_id': 'modbus_tcp.1', 'device_descr': 'solar_panel07',
'measure_descr': 'RMS sum apparent energy', 'measure_unit': 'kWh',
'measure_type': 'f'
}]
```

In addition to the already described **client_id**, **device_id** and **measure_id**, the config JSON file contains a **plugin_id** which is the input plugin used to retrieve that measure, the **device_desc** which is the client name, the **measure_desc** that is the name of the drain and the **measure_unit** that corresponds to the the unit of measure. The parameter **measure_type** is the type of data which, in the example, is (f)loat. This field is actually unused in IDa.

As described in the REST part, JSON files are mapped on `CsvMeasuresJson` class (for measures) and `ConfigMeasureJson` (for config) through the Spring annotation `@RequestBody`. For MQTT, where Spring is not used, the open-source JSON parser Jackson[6], with its `ObjectMapper` class, has been used.

The method `readValue()` parse and convert the JSON into a Java object that, for measures, will be used in the method `createMeasuresFromJson()`. This method is the same method called from the REST controller of IDa. It takes in input the JSON file and the user who sent the measures, to ensure that it has the right to

[6]`https://github.com/FasterXML/jackson`

save those measures. The user is obtained from the topic name, and the identity of the user is proved by the ACLs mechanism of mosquitto.

5.3.1 IEnergyDa MQTT Security

The TLS connection between mosquitto and IDa is handled with the Java `SSLSocketFactory` class and the security API **Bouncy Castle**[7].

`SSLSocketFactory` is used to load the CA certificate from the file-system. When client certificates and keys are needed, to read the PEM format of the private keys, BouncyCastle methods are used.

After the setup of CA certificates and, eventually, client certificates and keys, a `SSLSocketFactory` object is created with a specified TLS version (default is TLS1.2).

[7]`https://bouncycastle.org/`

Chapter 6

Protocols comparison

To compare behaviors and performances of the two protocols with the implementations described in previous chapters, several tests has been executed.

Starting from a basic configuration, which reflect an hypothetical real case scenario, variations on **payload size** and **clients number** have been tried. Those tests have been executed over the Internet, both with a normal HTTP or MQTT connection (without security) and with TLS.

In this chapter will be also shown two 24 hours tests on two particular industrial use cases involved in the H.O.M.E.[1] project. H.O.M.E. is an open-source project to interconnect and automate factories following Industry 4.0 principles.

One of these use cases is characterized by a very **limited bandwidth**, while the other uses **websockets** to establish the MQTT connection due to the impossibility to use a public MQTT broker on the customer data center.

Time synchronization Time synchronization between the sender and the receiver represented one of the main issues in order to report correctly the performance of the protocols. To evaluate those performances, a timestamp is reported on the sender machine logs just before the `post()` operation, and another timestamp is stored on the receiver machine just after the message's arrival.

Most of the packets sent in the various tests required from 100 milliseconds to a few seconds to be completed so, the delay between the two clocks must be minimum. To do so both sender and receiver, in every configuration tested, have been connected to the same NTP server.

[1]`https://www.home-opensystem.org/index.php/en/home-3/`

Through the use of the command `ntptime` it is also possible to evaluate the **estimated error** between a client and the NTP server. In every machine involved the maximum estimated error between the client and the NTP server has been 3120 μs $\approx$ 3.1 ms. This value represents only half of the uncertainty.

Considering the time t taken by a measure to go from a sender s to a receiver r as:

$$t = t_r - t_s$$

The uncertainty of the difference is given by the formula:

$$\delta t = \sqrt{(\delta t_r)^2 + (\delta t_s)^2}$$

It is also possible to consider an approximation of the formula above, to maintain larger the confidence interval:

$$\delta t = \delta t_r + \delta t_s = 3.1ms + 3.1ms \approx 6.2ms$$

So, every measure in the test is considered with an absolute uncertainty δt of $\pm 6.2ms$.

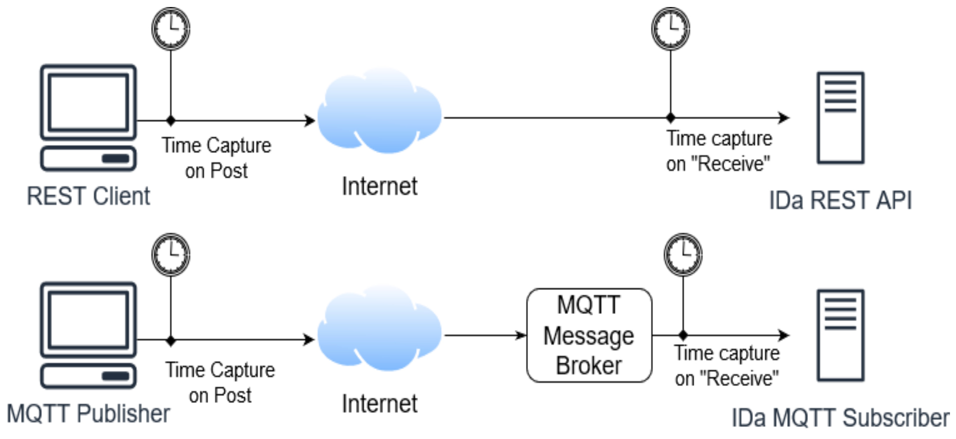


Figure 6.1. Time captures schema.

6.1 Payload tests

In this section will be reported the results of 16 payload tests, 8 tests on HTTP and MQTT without a secure connection, and 8 tests with TLS.

The average payload size of a packet sent from Wolf to I-Da is about 2-3 kB (kilobyte). However, this average packet size reported is merely indicative, because it varies widely in the various installation depending on the number of sensors connected to Wolf.

The starting payload for every test is 10 times an average packet (25 kB) and is doubled for any subsequent test. For each test, Wolf sends a packet every 5 seconds to I-Da for a total of about 300 packets in each trial. MQTT tests used a QoS of 1 which ensures that almost a packet arrives. This QoS level does not protect from duplicated packets which, however, can be handled from I-Da.

6.1.1 Payload tests without TLS

As shown from Figure 6.2, with the increasing of the payload an increase in the delivery time can be observed both for MQTT and HTTP.

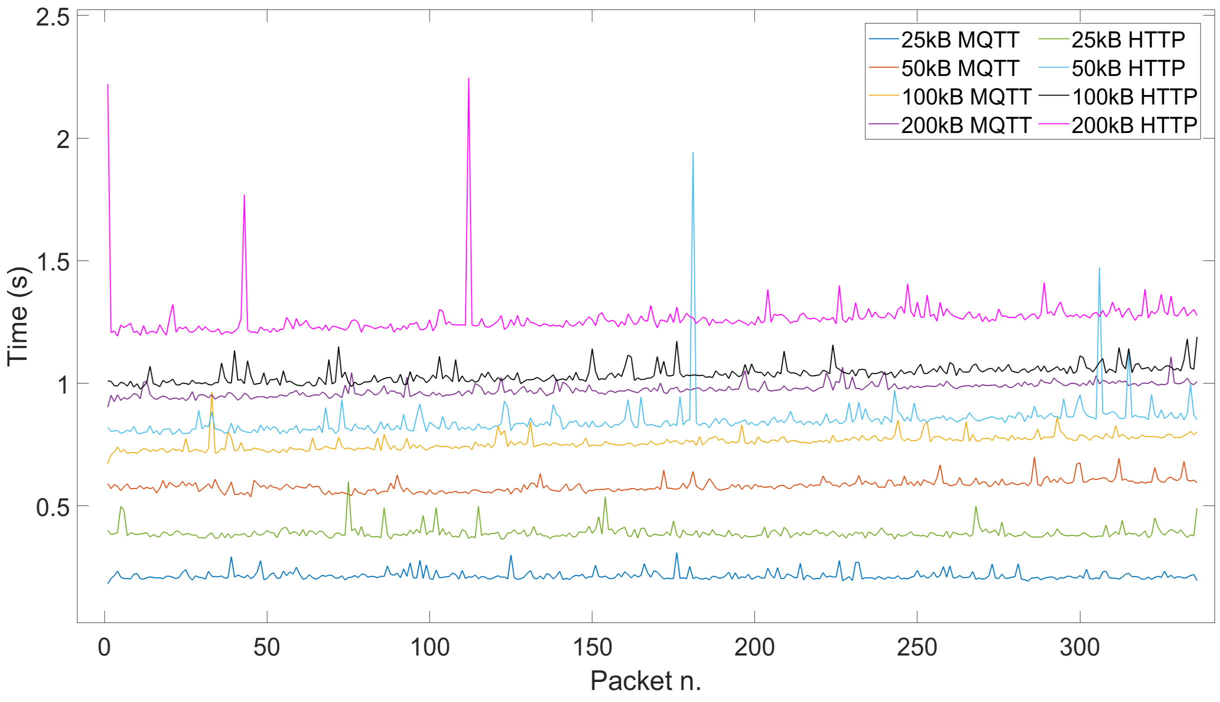
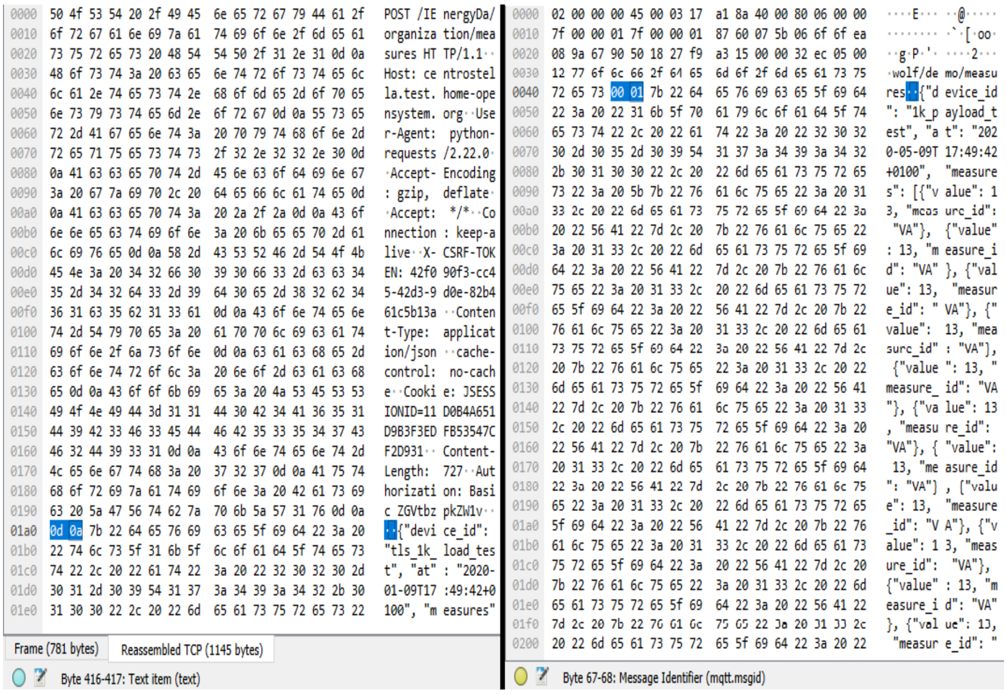


Figure 6.2. Payload tests without a secure connection

Is it also possible to observe that MQTT performance better than HTTP for every payload. The reason for this difference can be sought in the architectural differences between the two protocols.

Message overhead is one of these differences, as can be noticed from Figure 6.3, which shows the two protocols headers captured with the [2], HTTP header is more than 6 times bigger than MQTT header.



```
0000  50 4f 53 54 20 2f 49 45  6e 65 72 67 79 44 61 2f   POST /IE nergyDa/
0010  6f 72 67 61 6e 69 7a 61  74 69 6f 6e 2f 6d 65 61   organiza tion/mea
0020  73 75 72 65 73 20 48 54  54 50 2f 31 2e 31 0d 0a   sures HT TP/1.1··
0030  48 6f 73 74 3a 20 63 65  6e 74 72 6f 73 74 65 6c   Host: ce ntrostel
0040  6c 61 2e 74 65 73 74 2e  68 6f 6d 65 2d 6f 70 65   la.test. home-ope
0050  6e 73 79 73 74 65 6d 2e  6f 72 67 0d 0a 55 73 65   nsystem. org··Use
0060  72 2d 41 67 65 6e 74 3a  20 70 79 74 68 6f 6e 2d   r-Agent:  python-
0070  72 65 71 75 65 73 74 73  2f 32 2e 32 32 2e 30 0d   requests /2.22.0·
0080  0a 41 63 63 65 70 74 2d  45 6e 63 6f 64 69 6e 67   ·Accept- Encoding
0090  3a 20 67 7a 69 70 2c 20  64 65 66 6c 61 74 65 0d   : gzip,  deflate·
00a0  0a 41 63 63 65 70 74 3a  20 2a 2f 2a 0d 0a 43 6f   ·Accept:  */*··Co
00b0  6e 6e 65 63 74 69 6f 6e  3a 20 6b 65 65 70 2d 61   nnection : keep-a
00c0  6c 69 76 65 0d 0a 58 2d  43 53 52 46 2d 54 4f 4b   live··X- CSRF-TOK
00d0  45 4e 3a 20 34 32 66 30  39 30 66 33 2d 63 63 34   EN: 42f0 90f3-cc4
00e0  35 2d 34 32 64 33 2d 39  64 30 65 2d 38 32 62 34   5-42d3-9 d0e-82b4
00f0  36 31 63 35 62 31 33 61  0d 0a 43 6f 6e 74 65 6e   61c5b13a ··Conten
0100  74 2d 54 79 70 65 3a 20  61 70 70 6c 69 63 61 74   t-Type:  applicat
0110  69 6f 6e 2f 6a 73 6f 6e  0d 0a 63 61 63 68 65 2d   ion/json ··cache-
0120  63 6f 6e 74 72 6f 6c 3a  20 6e 6f 2d 63 61 63 68   control:  no-cach
0130  65 0d 0a 43 6f 6f 6b 69  65 3a 20 4a 53 45 53 53   e··Cooki e: JSESS
0140  49 4f 4e 49 44 3d 31 31  44 30 42 34 41 36 35 31   IONID=11 D0B4A651
0150  44 39 42 33 46 33 45 44  46 42 35 33 35 34 37 43   D9B3F3ED FB53547C
0160  46 32 44 39 33 31 0d 0a  43 6f 6e 74 65 6e 74 2d   F2D931·· Content-
0170  4c 65 6e 67 74 68 3a 20  37 32 37 0d 0a 41 75 74   Length:  727··Aut
0180  68 6f 72 69 7a 61 74 69  6f 6e 3a 20 42 61 73 69   horizati on: Basi
0190  63 20 5a 47 56 74 62 7a  70 6b 5a 57 31 76 0d 0a   c ZGVtbz pkZW1v··
01a0  0d 0a 7b 22 64 65 76 69  63 65 5f 69 64 22 3a 20   ··{"devi ce_id":
01b0  22 74 6c 73 5f 31 6b 5f  6c 6f 61 64 5f 74 65 73   "tls_1k_ load_tes
01c0  74 22 2c 20 22 61 74 22  3a 20 22 32 30 32 30 2d   t", "at" : "2020-
01d0  30 31 2d 30 39 54 31 37  3a 34 39 3a 34 32 2b 30   01-09T17 :49:42+0
01e0  31 30 30 22 2c 20 22 6d  65 61 73 75 72 65 73 22   100", "m easures"

Frame (781 bytes)   Reassembled TCP (1145 bytes)
Byte 416-417: Text item (text)
```

```
0000  02 00 00 00 45 00 03 17  a1 8a 40 00 80 06 00 00   ····E··· ··@·····
0010  7f 00 00 01 7f 00 00 01  87 60 07 5b 06 6f 6f ea   ········ ·`·[·oo·
0020  08 9a 67 90 50 18 27 f9  a3 15 00 00 32 ec 05 00   ··g·P·'· ····2···
0030  12 77 6f 6c 66 2f 64 65  6d 6f 2f 6d 65 61 73 75   ·wolf/de mo/measu
0040  72 65 73 00 01 7b 22 64  65 76 69 63 65 5f 69 64   res··{"d evice_id
0050  22 3a 20 22 31 6b 5f 70  61 79 6c 6f 61 64 5f 74   ": "1k_p ayload_t
0060  65 73 74 22 2c 20 22 61  74 22 3a 20 22 32 30 32   est", "a t": "202
0070  30 2d 30 35 2d 30 39 54  31 37 3a 34 39 3a 34 32   0-05-09T 17:49:42
0080  2b 30 31 30 30 22 2c 20  22 6d 65 61 73 75 72 65   +0100",  "measure
0090  73 22 3a 20 5b 7b 22 76  61 6c 75 65 22 3a 20 31   s": [{"v alue": 1
00a0  33 2c 20 22 6d 65 61 73  75 72 65 5f 69 64 22 3a   3, "meas ure_id":
00b0  20 22 56 41 22 7d 2c 20  7b 22 76 61 6c 75 65 22    "VA"},  {"value"
00c0  3a 20 31 33 2c 20 22 6d  65 61 73 75 72 65 5f 69   : 13, "m easure_i
00d0  64 22 3a 20 22 56 41 22  7d 2c 20 7b 22 76 61 6c   d": "VA" }, {"val
00e0  75 65 22 3a 20 31 33 2c  20 22 6d 65 61 73 75 72   ue": 13,  "measur
00f0  65 5f 69 64 22 3a 20 22  56 41 22 7d 2c 20 7b 22   e_id": " VA"}, {"
0100  76 61 6c 75 65 22 3a 20  31 33 2c 20 22 6d 65 61   value":  13, "mea
0110  73 75 72 65 5f 69 64 22  3a 20 22 56 41 22 7d 2c   sure_id" : "VA"},
0120  20 7b 22 76 61 6c 75 65  22 3a 20 31 33 2c 20 22    {"value ": 13, "
0130  6d 65 61 73 75 72 65 5f  69 64 22 3a 20 22 56 41   measure_ id": "VA
0140  22 7d 2c 20 7b 22 76 61  6c 75 65 22 3a 20 31 33   "}, {"va lue": 13
0150  2c 20 22 6d 65 61 73 75  72 65 5f 69 64 22 3a 20   , "measu re_id":
0160  22 56 41 22 7d 2c 20 7b  22 76 61 6c 75 65 22 3a   "VA"}, { "value":
0170  20 31 33 2c 20 22 6d 65  61 73 75 72 65 5f 69 64    13, "me asure_id
0180  22 3a 20 22 56 41 22 7d  2c 20 7b 22 76 61 6c 75   ": "VA"} , {"valu
0190  65 22 3a 20 31 33 2c 20  22 6d 65 61 73 75 72 65   e": 13,  "measure
01a0  5f 69 64 22 3a 20 22 56  41 22 7d 2c 20 7b 22 76   _id": "V A"}, {"v
01b0  61 6c 75 65 22 3a 20 31  33 2c 20 22 6d 65 61 73   alue": 1 3, "meas
01c0  75 72 65 5f 69 64 22 3a  20 22 56 41 22 7d 2c 20   ure_id":  "VA"},
01d0  7b 22 76 61 6c 75 65 22  3a 20 31 33 2c 20 22 6d   {"value" : 13, "m
01e0  65 61 73 75 72 65 5f 69  64 22 3a 20 22 56 41 22   easure_i d": "VA"
01f0  7d 2c 20 7b 22 76 61 6c  75 65 22 3a 20 31 33 2c   }, {"val ue": 13,
0200  20 22 6d 65 61 73 75 72  65 5f 69 64 22 3a 20 22    "measur e_id": "

Byte 67-68: Message Identifier (mqtt.msgid)
```

Figure 6.3. Header differences in message delivery between HTTP (left) and MQTT (right). The blue highlight indicates the end of the header

However, overhead by itself does not completely explain why this difference increase with message size increasing. In this case, the difference could be not in the protocols themselves but in the actual implementation. JSON deserialization is performed manually for MQTT and through Spring MVC for REST. Even if in the first case the deserialization requires 10 ms, with Spring mapping it could take longer for huge JSON files, justifying the increasing difference in the arrival time measurement.

[2]`https://www.wireshark.org/`

It can also be noticed that, from time to time, HTTP had some spikes in the measures, particularly for 50 kB and 200 kB tests. These anomalies could be caused by temporary network instabilities but, overall, given the high number of sample packets, there is not a significant influence (the difference is about 10-20 ms) of these spikes in the average delivery time reported (Table 6.1).

Payload size	MQTT avg	HTTP avg
25 kB	0.214 s	0.389 s
50 kB	0.586 s	0.857 s
100 kB	0.773 s	1.042 s
200 kB	0.979 s	1.29 s

Table 6.1. Payload tests average delivery times

6.1.2 Payload tests with TLS

The same payload tests executed with a secure connection, represented in Figure 6.4, show that the average message delivery in every test, compared to the previous connection tests, is almost halved.

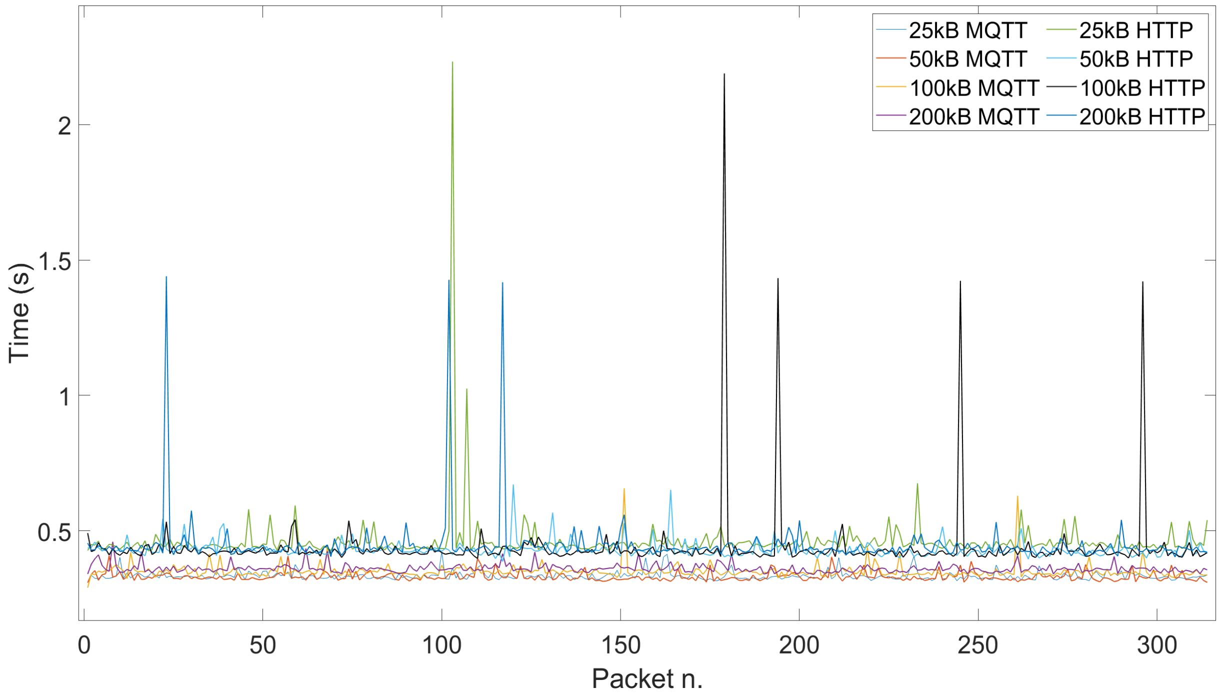
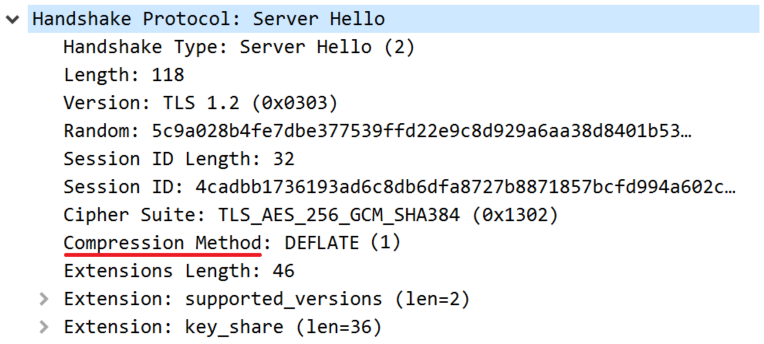


Figure 6.4. Payload tests with TLS

This result appears counter-intuitive, as TLS handshake and message encryption require additional time to complete a transaction.

However, there are some cases in which a TLS connection could perform better than a normal one. For example with HTTP/2, browser optimization[24] or with TLS 1.3 proposed mechanisms, such as 1-RTT and 0-RTT[3]. The performance improvement can also be given by the compression that TLS could apply to the messages. In fact, in addition to the encryption provided by TLS, the security layer also provide data compression with an algorithm indicated in the Compression Method field.



```
v Handshake Protocol: Server Hello
    Handshake Type: Server Hello (2)
    Length: 118
    Version: TLS 1.2 (0x0303)
    Random: 5c9a028b4fe7dbe377539ffd22e9c8d929a6aa38d8401b53…
    Session ID Length: 32
    Session ID: 4cadbb1736193ad6c8db6dfa8727b8871857bcfd994a602c…
    Cipher Suite: TLS_AES_256_GCM_SHA384 (0x1302)
    Compression Method: DEFLATE (1)
    Extensions Length: 46
  > Extension: supported_versions (len=2)
  > Extension: key_share (len=36)
```

Figure 6.5. Encryption settings and the agreed Compression Method indicated in the Server Hello message of HTTPS handshake

Analyzing the TLS handshake messages with Wireshark (Figure 6.5), it can be observed that the `Compression Method` field is set to 1 for both algorithms, which means that DEFLATE compression algorithm has been used, leading to the better performance collected.

Even for these tests, MQTT performed slightly better (Table 6.2) and some time spikes can be observed in the HTTP curves. Even in this case, the average is poorly affected by these peaks.

Payload size	**MQTT-S avg**	**HTTPS avg**
25 kB	0.333 s	0.462 s
50 kB	0.329 s	0.43 s
100 kB	0.35 s	0.437 s
200 kB	0.359 s	0.446 s

Table 6.2. TLS payload tests average delivery times

[3]https://hpbn.co/transport-layer-security-tls/

6.2 Multithread tests

Multithread tests have been executed in conditions similar to those applied for payload tests. In this case, a standard payload of around 2 kB has been used for every test. Even in this case, 16 tests have been executed, 8 with TLS and 8 without the security layer.

The other 8 tests have been performed and reported on subsection 6.2.3, to prove a suspected architectural bottleneck observed during previous tests.

The simultaneous delivery has been performed with 3, 5, 8, and 10 concurrent users, every 15 seconds for approximately 1 hour on each test. To measure the performance the "time window" between the dispatch of the first message and the arrival of the last message of the thread pool.

6.2.1 Multithread tests without TLS

The first group of tests (Figure 6.6 reports the trace of time windows without TLS.

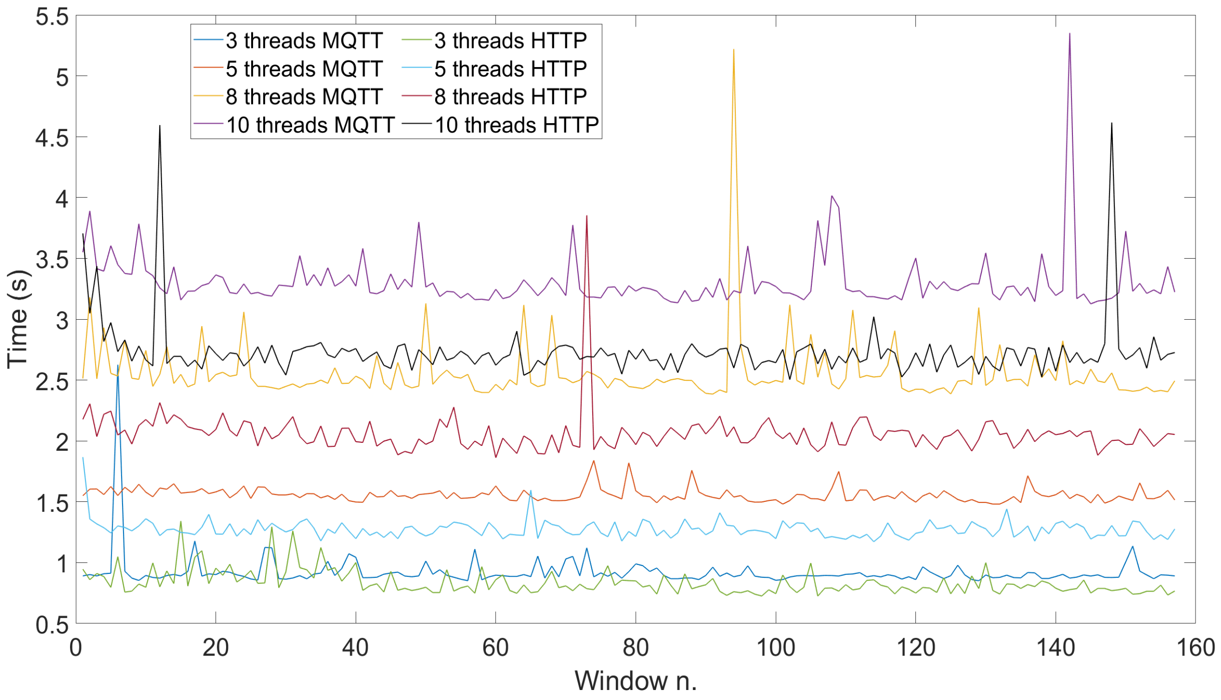


Figure 6.6. Multithread tests without TLS

Unlike single-thread tests, it can be observed that, excluding the tests executed with 3 threads that show comparable results, there is a significant gap between MQTT and HTTP, but this time in favor of the second one.

HTTP performs a 17% better than MQTT on average, with 5 threads, 20% better with 8 threads, and 25% better with 10 threads. Time peaks are present for both protocols. Table 6.3 shows in detail the average of the trials.

Threads number	**MQTT avg**	**HTTP avg**
3 threads	0.922 s	0.81 s
5 threads	1.555 s	1.291 s
8 threads	2.542 s	2.038 s
10 threads	3.639 s	2.715 s

Table 6.3. Multithread tests average time windows results

6.2.2 Multithread tests with TLS

TLS multithread tests show a similar trend of the equivalent tests without security (Figure 6.7.

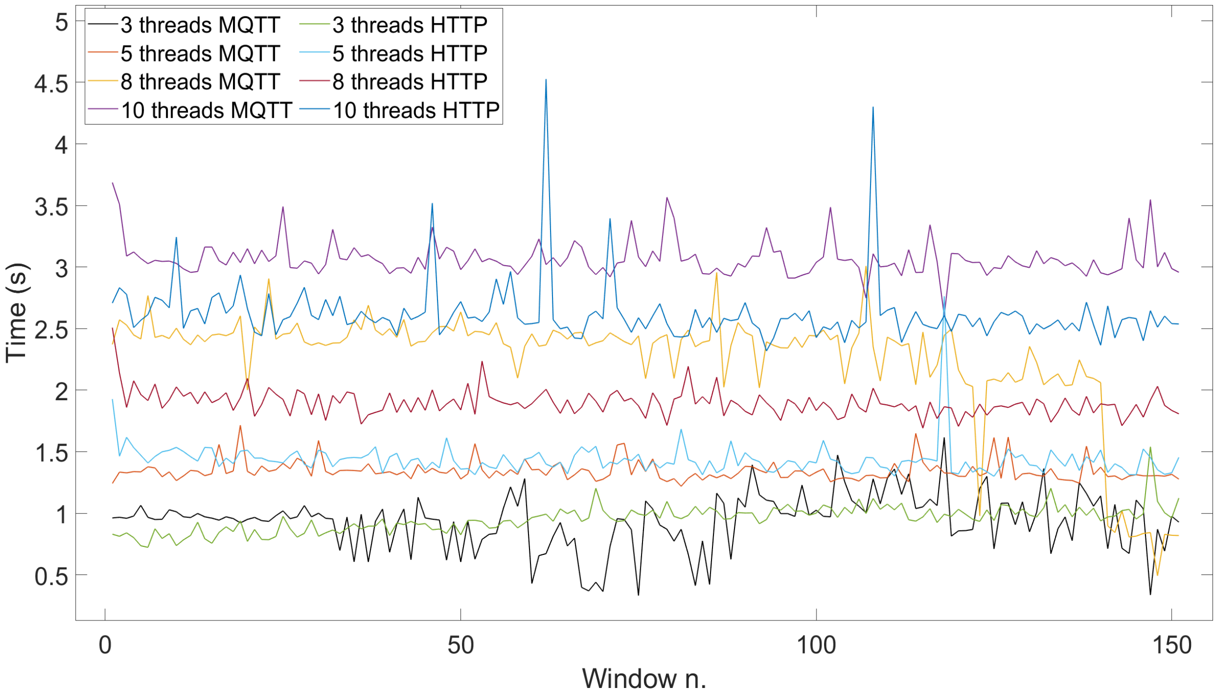


Figure 6.7. Multithread tests with TLS

Also in this case TLS compression improved performances but without a great time range, as packet size is very small if compared with payload tests.

The differences between the two protocols are less pronounced in these tests and, even with a notable floundering, MQTT performs slightly better for 3 threads an 5 threads tests.

With 8 threads there is an important gap between MQTT performance and HTTP, but a fall in the last part of the test affected the average value which resulted to be almost the same for the two protocols (Table 6.4). 10 thread tests show instead of a better performance of HTTP of about 14%.

Threads number	**MQTT-S avg**	**HTTPS avg**
3 threads	0.923 s	0.995 s
5 threads	1.333 s	1.419 s
8 threads	1.863 s	1.87 s
10 threads	3.013 s	2.591 s

Table 6.4. TLS multithread tests average time windows results

6.2.3 Authentication plugin bottleneck

After the result reported in the previous two chapters, multithread tests have been repeated several times and analyzed in-depth both with Wireshark and file logging.

Given the better MQTT performances in single-thread tests, it could be supposed that multithreads tests would have confirmed these findings, on equal multithread handling, as also reported in previous multithread studies[13, 14].

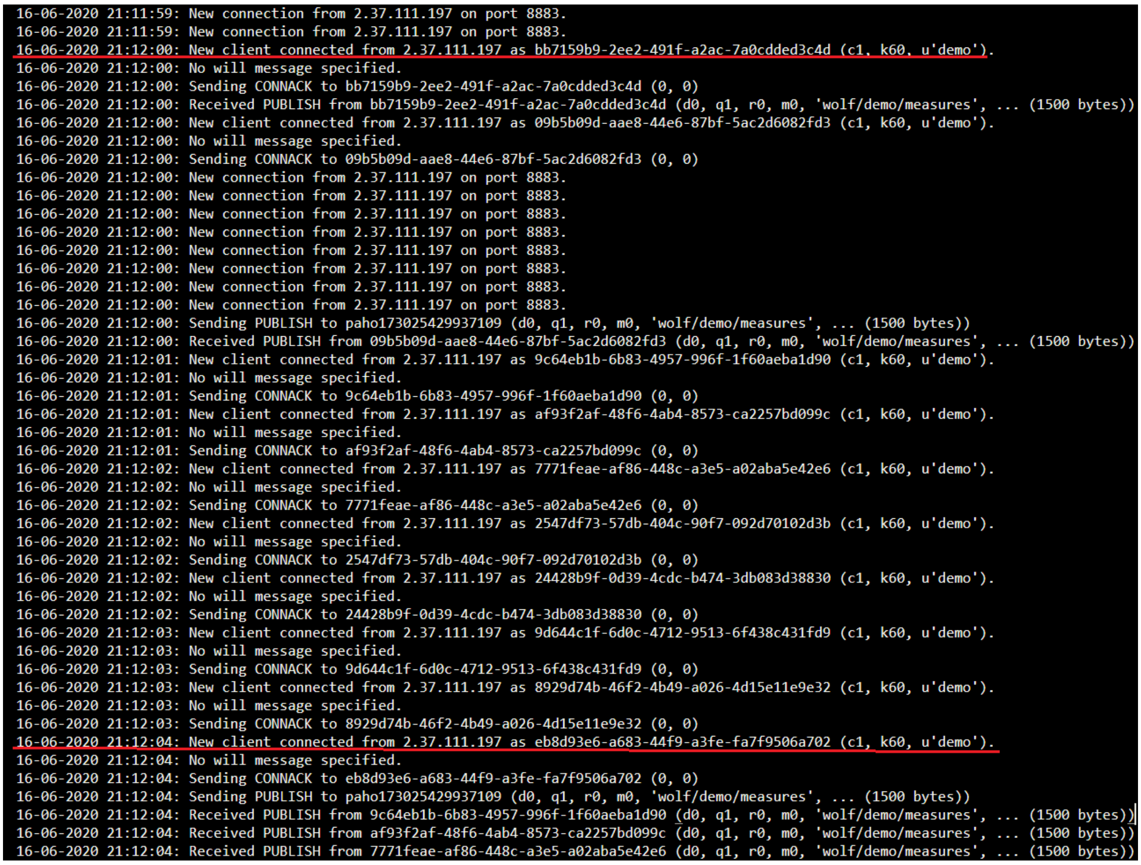
However, subsequent tries showed the same results of the previously reported data, with better performance for MQTT and worse swinging behavior for MQTT.

For both protocols, the multithread handling is managed by Tomcat servlets so the only possible difference should have been in the Mosquitto Broker.

Even if, compared with other broker, MQTT can be slower in multithread message dispatching, the study conducted by Pipatsakulroj et al.[25] shows that Mosquitto can handle a huge number of subscriber connected.

However, after an in-depth analysis of mosquitto log, a huge bottleneck in dispatch time has emerged. Even if for **n** threads in a test all the connections take place almost simultaneously, the authentication of each user requires an important amount of time. The authentication mechanism is managed by the external Mosquitto auth plugin, covered on subsection 5.1.2, and implemented to use the same authentication database for both protocols, which was not possible with normal Mosquitto authentication.

In Figure 6.8 is reported a part of the Mosquitto log for 10 threads test, which shows how long it took the plugin to authenticate all clients.



```
16-06-2020 21:11:59: New connection from 2.37.111.197 on port 8883.
16-06-2020 21:11:59: New connection from 2.37.111.197 on port 8883.
16-06-2020 21:12:00: New client connected from 2.37.111.197 as bb7159b9-2ee2-491f-a2ac-7a0cdded3c4d (c1, k60, u'demo').
16-06-2020 21:12:00: No will message specified.
16-06-2020 21:12:00: Sending CONNACK to bb7159b9-2ee2-491f-a2ac-7a0cdded3c4d (0, 0)
16-06-2020 21:12:00: Received PUBLISH from bb7159b9-2ee2-491f-a2ac-7a0cdded3c4d (d0, q1, r0, m0, 'wolf/demo/measures', ... (1500 bytes))
16-06-2020 21:12:00: New client connected from 2.37.111.197 as 09b5b09d-aae8-44e6-87bf-5ac2d6082fd3 (c1, k60, u'demo').
16-06-2020 21:12:00: No will message specified.
16-06-2020 21:12:00: Sending CONNACK to 09b5b09d-aae8-44e6-87bf-5ac2d6082fd3 (0, 0)
16-06-2020 21:12:00: New connection from 2.37.111.197 on port 8883.
16-06-2020 21:12:00: New connection from 2.37.111.197 on port 8883.
16-06-2020 21:12:00: New connection from 2.37.111.197 on port 8883.
16-06-2020 21:12:00: New connection from 2.37.111.197 on port 8883.
16-06-2020 21:12:00: New connection from 2.37.111.197 on port 8883.
16-06-2020 21:12:00: New connection from 2.37.111.197 on port 8883.
16-06-2020 21:12:00: New connection from 2.37.111.197 on port 8883.
16-06-2020 21:12:00: New connection from 2.37.111.197 on port 8883.
16-06-2020 21:12:00: Sending PUBLISH to paho173025429937109 (d0, q1, r0, m0, 'wolf/demo/measures', ... (1500 bytes))
16-06-2020 21:12:00: Received PUBLISH from 09b5b09d-aae8-44e6-87bf-5ac2d6082fd3 (d0, q1, r0, m0, 'wolf/demo/measures', ... (1500 bytes))
16-06-2020 21:12:01: New client connected from 2.37.111.197 as 9c64eb1b-6b83-4957-996f-1f60aeba1d90 (c1, k60, u'demo').
16-06-2020 21:12:01: No will message specified.
16-06-2020 21:12:01: Sending CONNACK to 9c64eb1b-6b83-4957-996f-1f60aeba1d90 (0, 0)
16-06-2020 21:12:01: New client connected from 2.37.111.197 as af93f2af-48f6-4ab4-8573-ca2257bd099c (c1, k60, u'demo').
16-06-2020 21:12:01: No will message specified.
16-06-2020 21:12:01: Sending CONNACK to af93f2af-48f6-4ab4-8573-ca2257bd099c (0, 0)
16-06-2020 21:12:02: New client connected from 2.37.111.197 as 7771feae-af86-448c-a3e5-a02aba5e42e6 (c1, k60, u'demo').
16-06-2020 21:12:02: No will message specified.
16-06-2020 21:12:02: Sending CONNACK to 7771feae-af86-448c-a3e5-a02aba5e42e6 (0, 0)
16-06-2020 21:12:02: New client connected from 2.37.111.197 as 2547df73-57db-404c-90f7-092d70102d3b (c1, k60, u'demo').
16-06-2020 21:12:02: No will message specified.
16-06-2020 21:12:02: Sending CONNACK to 2547df73-57db-404c-90f7-092d70102d3b (0, 0)
16-06-2020 21:12:02: New client connected from 2.37.111.197 as 24428b9f-0d39-4cdc-b474-3db083d38830 (c1, k60, u'demo').
16-06-2020 21:12:02: No will message specified.
16-06-2020 21:12:02: Sending CONNACK to 24428b9f-0d39-4cdc-b474-3db083d38830 (0, 0)
16-06-2020 21:12:03: New client connected from 2.37.111.197 as 9d644c1f-6d0c-4712-9513-6f438c431fd9 (c1, k60, u'demo').
16-06-2020 21:12:03: No will message specified.
16-06-2020 21:12:03: Sending CONNACK to 9d644c1f-6d0c-4712-9513-6f438c431fd9 (0, 0)
16-06-2020 21:12:03: New client connected from 2.37.111.197 as 8929d74b-46f2-4b49-a026-4d15e11e9e32 (c1, k60, u'demo').
16-06-2020 21:12:03: No will message specified.
16-06-2020 21:12:03: Sending CONNACK to 8929d74b-46f2-4b49-a026-4d15e11e9e32 (0, 0)
16-06-2020 21:12:04: New client connected from 2.37.111.197 as eb8d93e6-a683-44f9-a3fe-fa7f9506a702 (c1, k60, u'demo').
16-06-2020 21:12:04: No will message specified.
16-06-2020 21:12:04: Sending CONNACK to eb8d93e6-a683-44f9-a3fe-fa7f9506a702 (0, 0)
16-06-2020 21:12:04: Sending PUBLISH to paho173025429937109 (d0, q1, r0, m0, 'wolf/demo/measures', ... (1500 bytes))
16-06-2020 21:12:04: Received PUBLISH from 9c64eb1b-6b83-4957-996f-1f60aeba1d90 (d0, q1, r0, m0, 'wolf/demo/measures', ... (1500 bytes))
16-06-2020 21:12:04: Received PUBLISH from af93f2af-48f6-4ab4-8573-ca2257bd099c (d0, q1, r0, m0, 'wolf/demo/measures', ... (1500 bytes))
16-06-2020 21:12:04: Received PUBLISH from 7771feae-af86-448c-a3e5-a02aba5e42e6 (d0, q1, r0, m0, 'wolf/demo/measures', ... (1500 bytes))
```

Figure 6.8. Part of mosquitto.log file which shows the authentication bottleneck

As shown, the last thread is authenticated in more than 3 seconds, which is more than the 80% of the average time elapsed to send messages from Wolf to IDa in MQTT.

To have more consistent prove of the slowdown caused by mosquitto authentication plugin, MQTT tests have been repeated again without authentication. Some adjustment of mosquitto configuration has been necessary to allow unauthenticated users to publish messages, such as the `allow_anonymous` flag set to true and specific write rule in the ACLs file.

Figure 6.9 and Figure 6.10 show the performances improvement of MQTT without authentication.

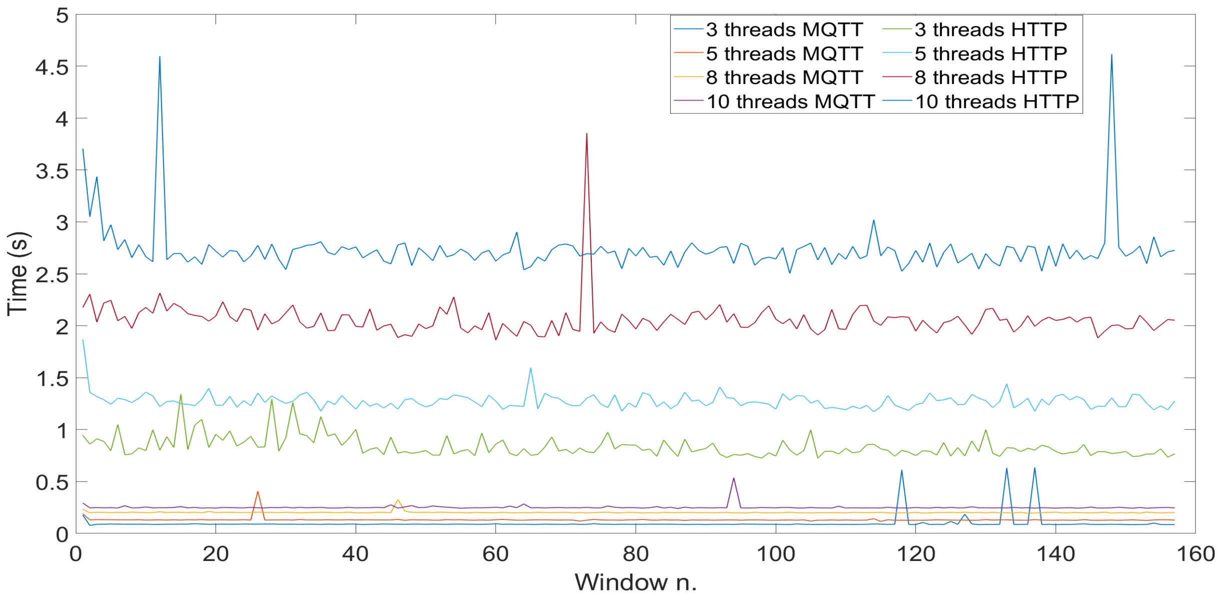


Figure 6.9. Multithread tests without MQTT authentication

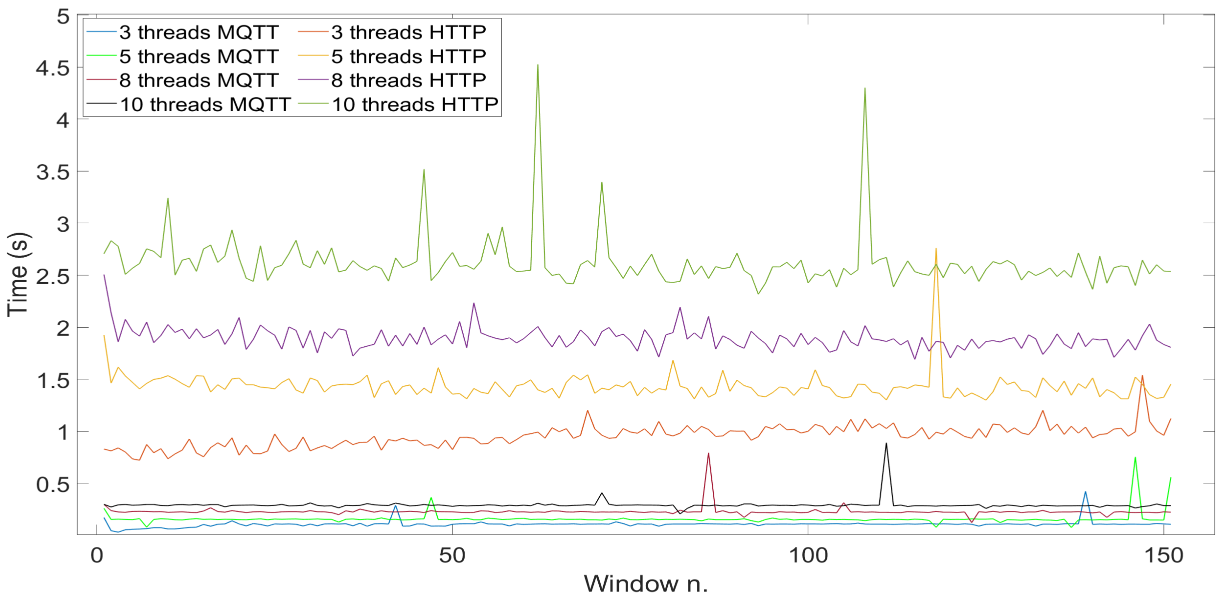


Figure 6.10. TLS multithread tests without MQTT authentication

In the previous figures, MQTT sessions are compared with the previous HTTP just to emphasize the performance gain without the bottleneck, as to perform a correct comparison between the two protocols also HTTP sessions should be unauthenticated.

Table 6.5 compares all MQTT multithread tests, highlighting a better performance of MQTT without authentication of about 10 times the authenticated version.

Threads n.	MQTT avg	MQTT no-auth avg	MQTT-S avg	MQTT-S no-auth avg
3 threads	0.922 s	0.09 s	0.923 s	0.102 s
5 threads	1.555 s	0.131 s	1.333 s	0.159 s
8 threads	2.542 s	0.204 s	1.863 s	0.226 s
10 threads	3.639 s	0.246 s	3.013 s	0.289 s

Table 6.5. MQTT tests average time windows results with and without authentication

6.3 Industrial use cases tests

6.3.1 Limited bandwidth

The first use case represents a factory local in a rural area of Piedmont, with several bandwidth problems, due to the high distance from the nearest ADSL cabinet and cables abrasion.

The factory sends data through the Internet to Myna Data Center. Using the `Ping` command from the factory to the Data Center it can be observed latency values that could be higher than **150ms**. Upload speed from the factory to the Data Center is highly variable, with values ranging from 7-8 kB/s to 60-65 kB/s.

Every three minutes the Wolf installed on the factory microserver sends energy measures to the IEnergyDa placed on Myna Data Center and reachable through Internet through Apache reverse proxy. The connection uses TLS, to protect confidential data over the web.

Figure 6.11 shows the two curves. HTTP delivery times vary from a minimum of 141 ms to a maximum of 324 ms, aside from a peak of 1.6 seconds. The average value, indicated with the green line, is **172 ms**.

MQTT minimum delivery time is 97 ms, and the plot displays some time spikes with a maximum value of 1 second.

The MQTT average, indicated with the red line, is **112 ms**, which is a 34.9% better of the HTTP average.

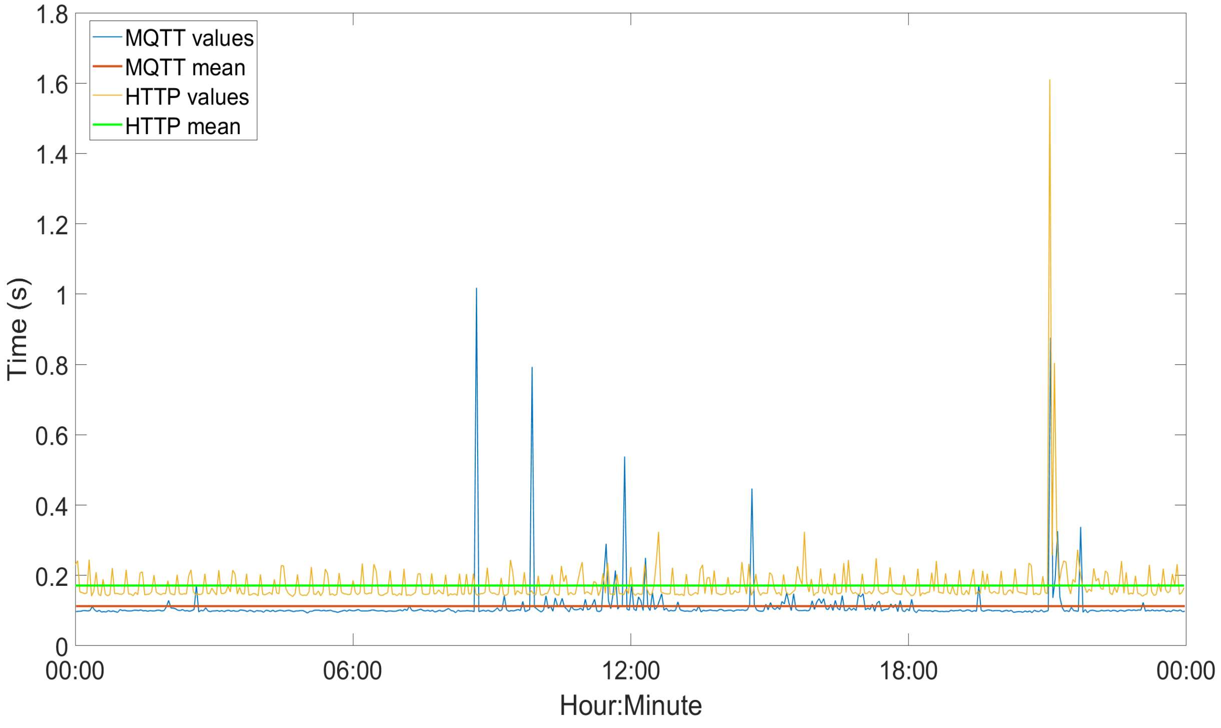


Figure 6.11. Limited bandwidth 24 hours test

6.3.2 MQTT over websockets

The second use case, and the last trial analyzed in this work, is a connection between a Wolf microserver on a paper industry connected to a remote Customer Data Center with IDa.

However, it was not possible to expose Mosquitto broker over the internet in the Customer Data Center, because, for company policy, all incoming Internet traffic must pass through the firewall reverse proxy.

To overcome this issue, as it was not possible to directly send MQTT packets through the reverse proxy, MQTT over Websockets has been used.

Websocket is a communication protocol that allows to create a full-duplex communication on top of a single TCP connection[4].

MQTT offers the possibility to send messages encapsulated in Websocket also with

[4]`https://en.wikipedia.org/wiki/WebSocket`

TLS, and these messages are able to pass through the customer firewall reverse-proxy. The initial connection with this approach uses HTTP to establish the communication and then MQTT messages are packed and send over websocket.

To use websockets in Mosquitto it is needed to download the source code of the broker, enable `WITH_WEBSOCKETS` option in **config.mk** and build it separately, as websocket is disabled by default at compile time[20]. The library `libwesockets` is also needed to support websockets. Finally, to enable websockets the configuration has been modified as follows:

```
listener 1883
listener 9001
protocol websockets
```

The first listener is on the standard MQTT port and it is used to the IEnergyDa subscriber to receive data, as for other cases. The second listener on port 9001 is instead for websocket, which should be also declared with the `protocol` option.

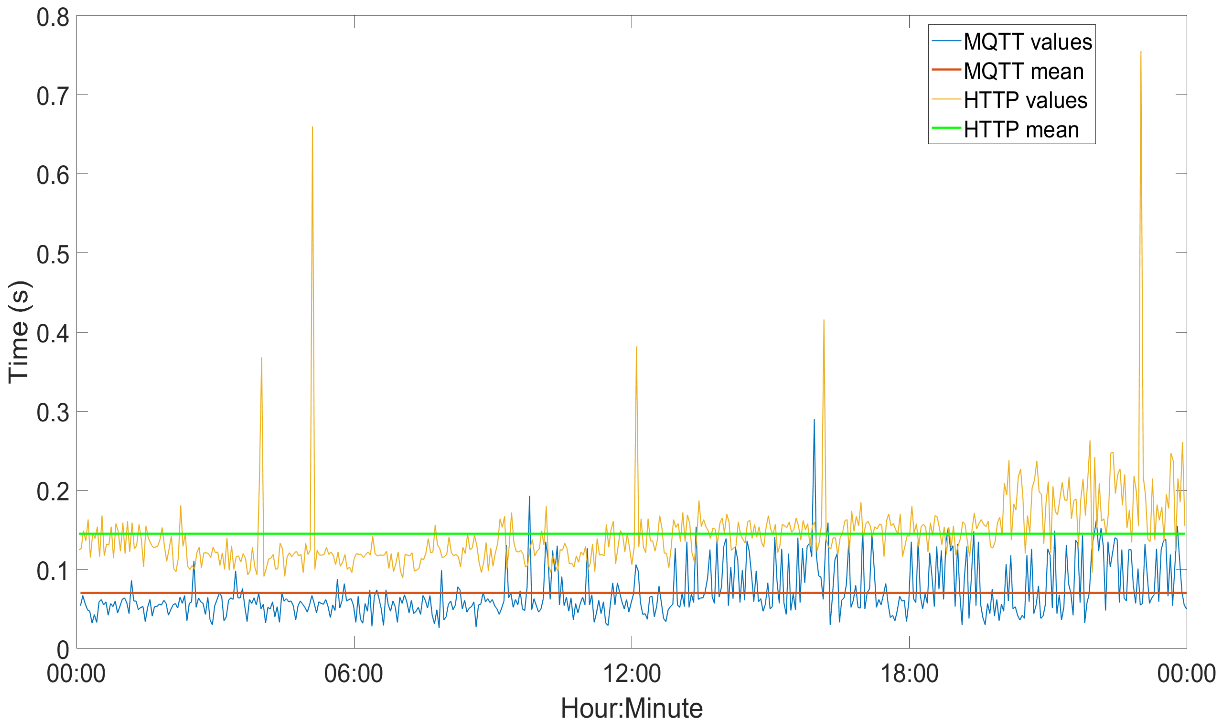


Figure 6.12. Use case with MQTT over Websockets 24 hours test

From the client point of view instead, the only difference to the MQTT configuration has been made on Wolf output plugin, by setting the new endpoint and port indicated for the Customer Data Center (instead of the broker address which is remapped from the reverse-proxy) and the protocol parameter as `protocol=websockets` to enable websockets communication.

Figure 6.12 shows the result of a 24 hours test with the same parameters of the first use case, one measure every 3 minutes, a payload of around 2 kB and TLS connection. This time, between the sender and the receiver the latency is low and stable, with an average of **30 ms**.

MQTT average is **70 ms** and it is indicated with the red line on the plot. MQTT values range from a minimum of 30 ms to a maximum of 290 ms-

HTTP average is **145 ms**, about twice the MQTT average, and it is indicated whit the green line on the plot. HTTP minimum value is 89 ms and the peak value is 755 ms, while over the 98% of HTTP values are under 270 ms.

Both values curves show an increment in the last part of the day, potentially due to a connection slowdown.

Chapter 7

Conclusion and future works

The aim of this work was to study and compare HTTP and MQTT, the two most used IoT application protocols, and to implement them in a industry-oriented software architecture.

The differences in the structure of the two protocols have been highlighted in the theoretical part, with an in-depth analysis of the security aspects and the most common vulnerabilities. The different architectures have exposed their positive and negative aspects.

While the lean implementation of MQTT makes this protocol less affected by overhead and performance issues, its publish-subscribe approach with message broker adds a third component in the communication, leading complexity in some aspects such as the awareness of the publisher on what message is received by the subscriber on certain levels of QoS.

On the other side, HTTP with its important overhead can lead to worse performances but offers better feedback with its detailed status codes on response.

These differences have been validated in the implementation and tests part, where MQTT performances confirmed the results already reported in literature both for payload tests and with a use case affected by high latency problems.

MQTT, with the websockets implementation, has also proved to be a flexible protocol, able to fit particular cases such as the impossibility to communicate directly with the message broker as addressed in subsection 6.3.2 use case.

The main performance difference has been observed in the multithread tests. After an in-depth analysis of the communication, it has been found that this difference has been caused by an implementation bottleneck, due to the authentication plugin for mosquitto broker, used to maintain the authentication part on the Postgres database.

The normal authentication provided by the broker is however inapplicable to the architecture used in the Myna suite, as it is based on a plain text password file that, besides the related security risks, results too static to be used, because every time a new user is registered it should be reported on the password file, and a similar procedure should be done to delete a user when it is removed from the database.

For these reasons, Myna-Project will maintain the authentication plugin in its architecture, reporting the issue to the plugin developer and cooperating with him to provide multithread handling to the application, also through the company's GitHub fork of the project.

In future developments, MQTT will be further integrated into the Myna suite, also in other parts of the architecture, to use it for a certain context or in certain moments of the day, during network stressing operations (e.g. remote backups) and in the interface between IEnergyDa and the GUI through websockets.

Bibliography

[1] A. Bosche, D. Crawford, D. Jackson, M. Schallehn, and C. Schorling. "Unlocking Opportunities in the Internet of Things", 2018. `https://www.bain.com/contentassets/5aa3a678438846289af59f62e62a3456/bain_brief_unlocking_opportunities_in_the_internet_of_things.pdf`.

[2] Eclipse Foundation. "IoT Developer Surveys 2018", 2018. `https://iot.eclipse.org/resources/iot-developer-survey/iot-developer-survey-2018.pdf`.

[3] R. Johnson et al. "Spring Framework Reference Documentation", 2016. `https://docs.spring.io/spring/docs/4.2.x/spring-framework-reference/pdf/spring-framework-reference.pdf`.

[4] Mozilla Foundation. "Cross-Origin Resource Sharing (CORS)", 2019. `https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS`.

[5] Eclipse Foundation. "IoT Developer Surveys 2019", 2019. `https://iot.eclipse.org/resources/iot-developer-survey/iot-developer-survey-2019.pdf`.

[6] J.C. Mogul. "Clarifying the fundamentals of HTTP". *Software Practice and Experience*, pages 103–134, 2004.

[7] D. Gourley and B. Totty. *"HTTP: The Definitive Guide"*. O'Reilly, 2002.

[8] Mozilla Foundation. "Evolution of HTTP", 2019. `https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Evolution_of_HTTP`.

[9] P. Jung. “On the Security of the TLS Protocol”. *Software Practice and Experience*, 2015.

[10] G.C. Hillar. *“MQTT Essentials - A Lightweight IoT Protocol”*. Packt Publishing, 2017.

[11] OASIS Standard. “MQTT Version 3.1.1”, 2014.
`http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.pdf`.

[12] P. E. Naeini, S. Bhagavatula, H. Habib, M. Degeling, L. Bauer, L. Cranor, and N. Sadeh. “Privacy Expectations and Preferences in an IoT World”. *Symposium on Usable Privacy and Security*, 2017.

[13] T. Yokotani and Y. Sasaki. “Comparison with HTTP and MQTT on Required Network Resources for IoT ”. *International Conference on Control, Electronics, Renewable Energy and Communications*, 2016.

[14] N. Naik. “Choice of Effective Messaging Protocols for IoT Systems: MQTT, CoAP, AMQP and HTTP”. *IEEE International Systems Engineering Symposium*, 2017.

[15] B. Wukkadada, K. Wankhede, R. Nambiar, and A. Nair. “Comparison with HTTP and MQTT In Internet of Things (IoT)”. *International Conference on Inventive Research in Computing Applications*, 2018.

[16] S. W. Chan and E. Burns. “Java™Servlet Specification”, 2017.
`https://javaee.github.io/servlet-spec/downloads/servlet-4.0/servlet-4_0_FINAL.pdf`.

[17] R.T. Fielding. *“Architectural Styles and the Design of Network-based Software Architectures”*. PhD thesis, University of California, Irvine, 2000.

[18] B. Alex, L. Taylor, R. Winch, and G. Hillert. “Spring Security Reference”, 2015.
`https://docs.spring.io/spring-security/site/docs/4.2.11.RELEASE/reference/pdf/spring-security-reference.pdf`.

[19] The Open Web Application Security Project (OWASP). “OWASP Top 10 2013”, 2013.
`https://owasp.org/www-pdf-archive/OWASP_Top_10_-_2013.pdf`.

[20] R. A. Light. "mosquitto.conf man page", 2013.
`https://mosquitto.org/man/mosquitto-conf-5.html`.

[21] K. Moriarty, B. Kaliski, and A. Rusch. "PKCS #5: Password-Based Cryptography Specification Version 2.1", 2017.
`https://tools.ietf.org/html/rfc8018`.

[22] M. Sönmez Turan, E. Barker, W. Burr, and L. Chen. "NIST Special Publication 800-132 - Recommendation for Password-Based Key Derivation Part 1: Storage Applications ", 2010.
`https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-132.pdf`.

[23] Eclipse Foundation. "Eclipse Paho Python Client - documentation ".
`https://www.eclipse.org/paho/clients/python/docs/`.

[24] B. Jackson. "Analyzing HTTPS Performance Overhead".
`https://www.keycdn.com/blog/https-performance-overhead`.

[25] W. Pipatsakulroj et al. "muMQ: A Lightweight and Scalable MQTT Broker". *2017 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*, 2017.