
Master Thesis Dissertation
Master Degree in Computer Engineering (Cybersecurity)

# MQTT-based Infrastructure for Distributed Secure Energy Communities

By

**Lorenzo Sebastiano Mathis**

**Supervisor(s):**
Prof. Alessandro Savino, Supervisor
Dr. Vahid Eftekhari Moghadam, Co-Supervisor

Politecnico di Torino
2025

Alla mia famiglia, per il loro infinito incoraggiamento, la pazienza e la fiducia che hanno sempre riposto in me, e a Rebecca, per il suo amore, il suo aiuto e il suo costante sostegno lungo questo percorso.

——— ~ ———

To my family, for their endless encouragement, patience, and unwavering belief in me, and to Rebecca, for her love, support, and constant encouragement throughout this journey.

# Acknowledgements

I would like to express my gratitude to my supervisor, Prof. Alessandro Savino, for their continuous guidance, constructive feedback and invaluable expertise, which have been fundamental to the development of this thesis. A special appreciation goes to my co-supervisor, Dr. Vahid Eftekhari Moghadam, for their encouragement, patience and insightful advice, as well as their continuous support throughout this research and its prototyping.

I extend my thanks to CINI for providing the resources and environment that made this work possible, as well as to Politecnico di Torino for the journey that take me to this point.

# Abstract

This thesis addresses the design of a secure-by-design framework for energy communities, with a particular emphasis on communication and the components needed to ensure its security. The highly interconnected nature of this ecosystem, combined with the criticality of the infrastructure involved, necessitated the development of a communication model capable of meeting performance and environmental constraints, while also addressing fundamental security and safety requirements.

Renewable Energy Communities are innovative and participatory models for local production, sharing and consumption of renewable energy. Their objective consists in delivering environmental, economic and social value together with the generation of clean energy. The work starts from the IEC 62443-Based Framework for the RECs to analyze the message brokers and the MQTT protocol, highlighting their main characteristics, strengths and vulnerabilities, defining the technological context within which the research is situated.

The core of the thesis focuses on the design of the components, functionalities and requirements deployed in the Energy Management System (EMS) platform and in members premises. In this phase, the high-level requirements defined by the reference framework are mapped onto the characteristics of the MQTT protocol and its implementations, integrating aspects related to communication architecture and the security of exchanged data.

Alongside the theoretical development, a simulator prototype was developed to verify the validity and applicability of the proposed solutions through testing on a selection of specific brokers and real-world device abstractions.

In conclusion, the MQTT protocol and its implementations satisfy the various security and operational requirements. The outcome of the work is an MQTT-based architecture capable of carrying the communication across the various components of the REC. The simulations performed on the prototype have confirmed the theoretical framework.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 RECs - Renewable Energy Communities

Renewable Energy Communities (RECs) are innovative and participatory models for the local production, sharing and consumption of renewable energy. They operate as autonomous entities, based on open and voluntary participation, and are controlled by their members — citizens, small and medium-sized enterprises and local authorities — who cooperate to harness the energy resources available in their area.[18][4]

The objective of RECs is not only to generate clean energy, but also to deliver environmental, economic and social value to both members and the wider community. By integrating technologies such as photovoltaics, energy storage systems, microgrids and digital energy management platforms, RECs promote a more sustainable, resilient and distributed model of energy production.

At their core there are prosumers, who generate energy (typically from photovoltaic plants) and can consume or share it. Surplus energy is stored in household or community batteries and redistributed during peak demands or when renewable generation is low. Energy flows are managed by Energy Management Systems (EMS), which analyze real-time consumption and production data, optimize energy use and balance distributed resources. EMS can also integrate smart loads, enabling appliances or industrial systems to operate when renewable energy is most abundant, reducing waste and minimizing grid stress.

Local microgrids further enhance the resilience of RECs, allowing communities to function as semi-autonomous energy systems, even during disruption of the main grid. Within these microgrids, peer-to-peer (P2P) energy exchange systems can be deployed, enabling direct transactions between members, governed by digital platforms and algorithms.

The adoption of innovative technologies plays a key role in the functioning of RECs. The Internet of Things (IoT) enables the interconnection of devices and monitoring systems (e.g. sensors), while artificial intelligence supports forecasting and optimizing both energy production and consumption, as well as managing safety (e.g. disaster recovery) and security challenges. Finally, distributed ledger technologies (DLT), such as blockchain, ensure transparency and traceability in energy transaction, further reinforcing the autonomy of these communities.[2]

This work focuses on analyzing the communication model of IoT networks and defining a framework that ensures both resilience and security. By investigating the interactions between devices, data flows and network protocols, the goal is to develop solutions that can maintain reliable operations even in the presence of fault, cyber threats or disruptions, thus strengthening the overall performance and robustness of RECs.

## 1.2 IEC 62443-Based Framework for Secure-by-Design Energy Communities

This section presents a survey of the article *An IEC 62443-Based Framework for Secure-by-Design Energy Communities*[10]. The study proposes a set of guidelines to be respected during the design and deployment phase of the REC; the recommendations take into account both the operational requirements of RECs and the security requirements of the IEC 62443 standard, with the aim of minimizing the cybersecurity risk from the design phase.

### 1.2.1 Reference architecture for RECs

The paper highlights the necessity of software platforms that support the main functional requirements of RECs. These platforms typically provide:

- **Administrative dashboard**: supports REC managers in handling member data, including admission and exit of participants.
- **Energy dashboard**: enables operators and end-users to monitor, in real time, community-level energy flows as well as individual consumption and production.
- **Energy Management System (EMS)**: optimizes the operation of community-owned assets and member-integrated resources.
- **Financial overview**: accessible to all participants, reporting community earnings from grid sales and benefits from internal energy sharing.

These functionalities require the installation of a *smart gateway* at each member's premises. The gateway acts as a local controller, interfacing household devices with the EMS platform over a secure IP-based connection. It communicates with meters and sensors, collects data, and can forward control commands from the platform.

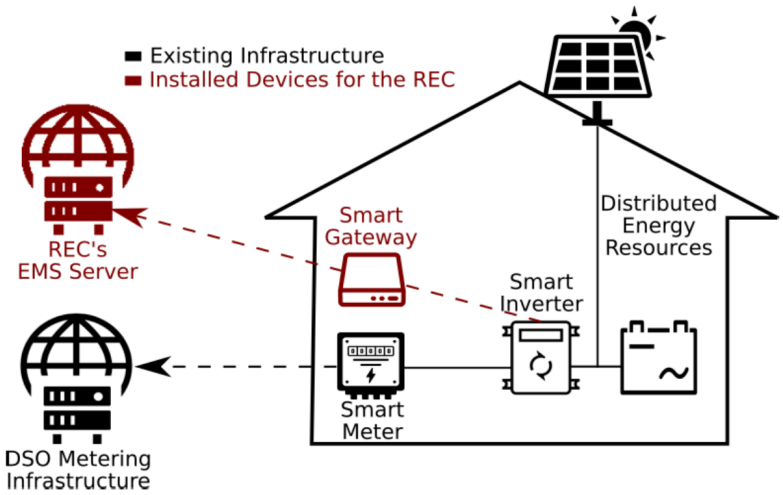


Fig. 1.1 Communication architecture of a REC's member [10].

The authors also present three commercial platforms that already implement this reference architecture:

**Regalgrid platform [17]**

The Regalgrid platform integrates a cloud-based system for the management of individual RECs with dedicated smart gateways installed at members' premises. Its main functionalities include community monitoring, real-time remote control of smart loads, energy balancing with the optimization of plant efficiency, as well as the distribution of incentives and the reporting of environmental impact. To enable the connection of different types of users, Regalgrid offers two smart gateway devices:

- the *SNOCU DIN*, designed to integrate prosumers into the REC,
- the *SNOCU Plug&Play*, intended for simple consumers.

**ROSE by Maps Energy [9]**

ROSE, developed by Maps Energy, is a cloud-based solution aimed at large-scale REC management. It is organized into three modules: *ROSE Designer*, which simulates the economic and energetic performance of both existing and planned communities; *ROSE Promoter*, which supports the management and evaluation of new candidate members; and *ROSE Manager*, which provides integrated tools to optimize REC operations from administrative, economic, and energetic perspectives.

**ER-Libra CE by Algowatt [1]**

ER-Libra CE, proposed by Algowatt, is another cloud-based platform designed for the comprehensive monitoring and management of RECs. The platform acquire data from smart meters using protocols such as MQTT or Modbus. The retrieved metrics about energy production and consumption, together with generation performance indications, support a detailed analysis process that provides insights both at the points of delivery (PODs) level and at the community level. Additional features include forecasting and statistical comparison of REC performance, administrative and financial management tools, and the supervision of physical community assets together with other advanced control functionalities.

### 1.2.2 Attack model

The authors distinguish two main attack surfaces in the reference REC architecture. On the one hand, local attacks can target the communication channels or the smart gateways deployed at members' premises. These attacks generally affect only a small subset of devices and mainly result in economic damage. On the other hand, centralized attacks may compromise the EMS platform, which supervises the entire REC. In this case, the consequences are much more severe, since the platform operates as a SCADA system for distributed resources; a successful breach could propagate through the whole community and even endanger the stability of the distribution grid.

More specifically, the authors classify three categories of attacks. The first concerns the *communication stack*, which includes both short-range links between smart gateways and field devices, and long-range links connecting gateways to the EMS.

Short-range communication relies on local protocols that are vulnerable to jamming, protocol weaknesses or physical tampering. The impact of these exploits is generally limited to a single prosumer and remains mostly economic.

Long-range communication, instead, is established via LAN or cellular networks; its vulnerabilities depend on the generation of mobile technology in use, and may cause denial of service if availability between gateway and EMS is disrupted.

A second category involves direct attacks on the *smart gateway*. Being an IoT device with constrained resources, and often equipped with a web interface for configuration, the gateway is exposed to common cyber threats and physical tampering. Two risks are especially relevant:

- the possibility of sending malicious commands to local devices, with limited but tangible economic consequences for a single prosumer
- and the possibility of injecting false or manipulated data into the EMS, which could compromise the decision-making process of the entire platform.

The third and most critical category is represented by attacks on the *EMS platform*. As the central controller of the REC, the EMS has an extensive attack surface, since it exposes web services and dashboards to operators and members. A successful intrusion would allow the adversary to alter control commands for a large set of

generators and loads, potentially causing not only economic losses and denial of service at community level, but also widespread disruption of the distribution grid.

### 1.2.3 Mapping RECs architecture to IEC 62443

The paper[10] applies the IEC 62443 methodology to the previously defined REC reference architecture. Following the standard, the first step is to identify zones and conduits and the second is to assign the appropriate security level targets (SLTs). The mapping also considers the Purdue Model, although in this case communication often relies on public networks, making traditional segmentation less applicable.

#### Zones

Two main zones are defined:

- *Zone 1* corresponds to the local network of a single prosumer, which includes the smart gateway, the smart meter, and the inverters controlling local generators and loads. All devices in this zone share the same security requirements since they collectively contribute to the control of a single member.
- *Zone 2* represents the centralized EMS platform that supervises the REC. This platform is physically separate from prosumer devices, and its violation would have more serious consequences than the breach into a single member.

Communication between these zones occurs through the public internet, which is modeled as the conduit connecting them.

#### Security Levels

- For Zone 1, the authors assign *SL-2*, which corresponds to protection against intentional misuse by attackers with limited resources, general skills, and low motivation. The rationale is that attacks at this level would primarily have economic consequences and remain limited to a single prosumer.
- For Zone 2, the EMS platform, the assigned level is *SL-3*, which protects against more sophisticated attackers with moderate resources and domain-specific knowledge. A compromise of this zone could affect the entire REC

and potentially destabilize the wider distribution grid, thus requiring stronger guarantees.

The public internet acts as the conduit between the zones, and its security level is determined by the protections enforced at the endpoints.

### 1.2.4 Practices and Guidelines

The authors adopt a defense-in-depth strategy, structured around four key pillars: **segmentation and perimeter defense**, **communication security**, **host security**, and **cybersecurity monitoring**.

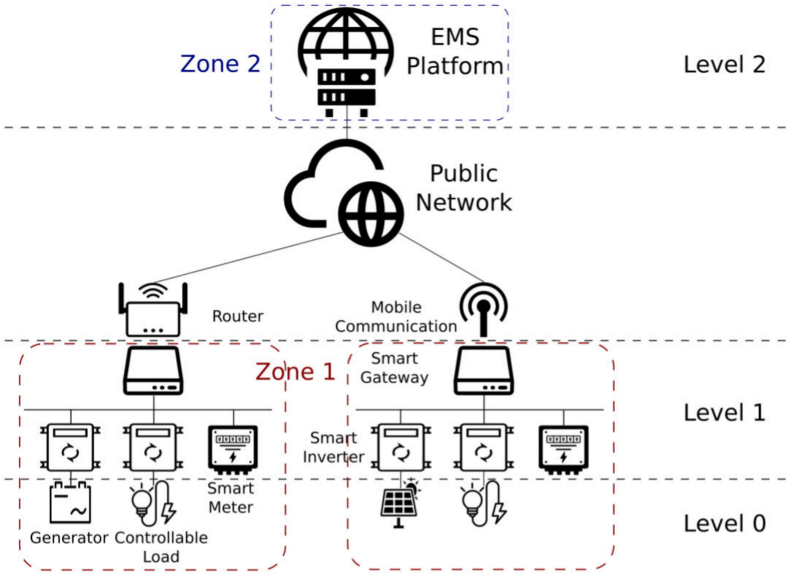


Fig. 1.2 Mapping of the REC reference architecture to the Purdue model [10].

### A. Segmentation and Perimeter Defense

1. **Dedicated sub-network for prosumer control devices**: Assets at levels 0 and 1 often provide built-in connectivity. In such cases, it is recommended to create a dedicated control sub-network that encompasses both levels. If the control network relies on the user's home network, two logically separated sub-networks should be deployed, restricting user access to the control sub-network.

2. **Limit wireless communication**: Wireless links in the control network should be minimized, monitored, and used only when strictly necessary. Wired communication is preferred.

3. **Firewalling**: The local control network must be protected by a firewall. All external connections should be denied by default, except for authenticated traffic originating from the EMS platform.

### B. Communication Security

1. **Mutual authentication between smart gateway and EMS platform**: Secure communication requires robust identity verification. Recommended mechanisms include:
   - Public Key Infrastructure (PKI) with X.509 certificates for strong authentication.
   - Transport Layer Security (TLS) for encrypted communication, with TLS/PSK as a fallback for constrained devices.
   - OAuth2 with token-based authentication to enable temporary, revocable access credentials.
   - Authenticated firmware and software execution on IoT devices to prevent tampering and unauthorized code deployment.

2. **End-to-end encryption**: All communications between smart gateways and the EMS platform must be encrypted from origin to destination, ensuring confidentiality and integrity.

3. **Authentication on the web server**: Access to the EMS platform should be governed by role-based access control (RBAC). The authors identify three primary user profiles:
    - *Member*: limited to their own consumption and production data.
    - *Manager*: able to access profiles and data of all community members.
    - *Operator*: responsible for technical operations and with broader system privileges.

### C. Host Security

1. **Separation of control and user functionalities**: The EMS engine (responsible for control) and the user-facing portal must run on distinct servers, communicating through well-defined interfaces.
2. **Implementation of a DMZ**: A demilitarized zone (DMZ) must isolate the EMS control server (OT domain) from the web server (IT domain). This architecture ensures strict segregation of operational and user-facing components.

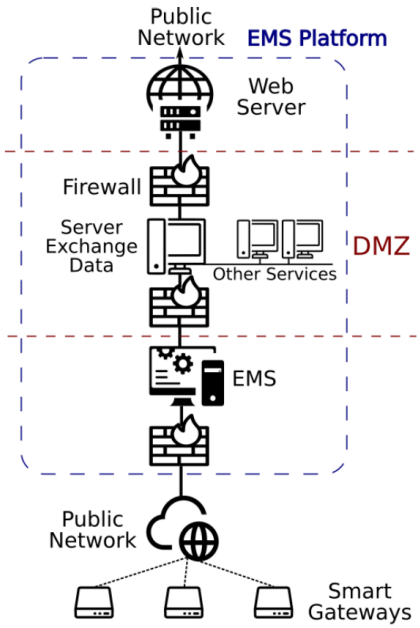


Fig. 1.3 Logical scheme of the EMS platform [10].

3. **Restrict access to the control network**: The control network — including the EMS engine, smart gateways, and smart devices — must be completely isolated. Access should be limited to administrators and automation engineers through strong multi-factor authentication (MFA).

4. **Secure software development for smart gateways**: Software deployed on smart gateways should follow established best practices (e.g., NIST SP 800-53, NISTIR 8259A, IEC 62443). This includes secure coding, threat modeling, vulnerability testing, access control, and firmware signing.

### D. Cybersecurity Monitoring

Both the smart gateways and the EMS platform must generate detailed security logs. A centralized **Security Information and Event Management (SIEM)** system, located in the DMZ, should collect, analyze, and correlate these logs for auditing and real-time detection of anomalies.

The required logs include:

- Web server logs (EMS platform).
- Application logs (EMS platform and smart gateway).
- System logs (EMS platform and smart gateway).
- Authentication and authorization logs (both sides).
- Network activity logs (both sides).
- Security event logs (antivirus, IDS/IPS, EDR/XDR alerts).
- Audit logs (configuration changes and administrative actions).

This monitoring infrastructure enables continuous visibility, automated alerting, and forensic investigation in the event of a security incident.

# Chapter 2

# Background

## 2.1 Message Brokers

In distributed systems, message brokers have become a cornerstone for reliable communication between decoupled components. Acting as intermediaries, they receive, store and forward messages between producers and consumers, ensuring efficient and resilient data exchange.

The use cases of message brokers span from microservices backbones and event-driven architectures to real-time data processing pipelines.

Conceptually, a broker provides a middleware layer for asynchronous communication. Producers publish messages without knowing the consumers, while consumers subscribe to messages of interest without concern for their origin or timing. This decoupling enables scalability, resilience and flexibility in system design.

### 2.1.1 Messaging Patterns

Message brokers support different communication patterns that address distinct interaction needs in distributed systems.

### Publish–Subscribe

In the publish–subscribe paradigm, producers publish messages to well-defined channels, often referred to as *topics*, while consumers subscribe to those topics in order to receive relevant messages. This model provides strong decoupling: producers remain unaware of how many consumers exist or who they are, while consumers can independently decide which topics to follow. The pattern is widely used when multiple components must react to the same event, for example in notification systems, monitoring pipelines or broadcasting updates to heterogeneous services. The strength of publish–subscribe lies in its flexibility and its ability to scale to many consumers with minimal coordination.

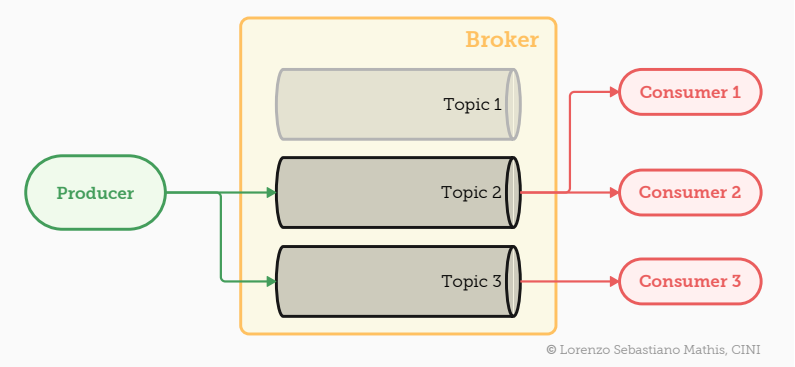


Fig. 2.1 Publish–Subscribe messaging pattern

### Event streaming

Event streaming extends publish–subscribe by treating the stream of messages as a durable, ordered log. Producers append new events to this log, and consumers read them sequentially, typically tracking their own position within the stream. Unlike transient messaging, event streaming emphasizes persistence and replayability: consumers can process events in near real-time, re-read past events for recovery or reprocess data to build new views and analytics. This paradigm underpins event-driven architectures and stream processing systems, where reproducibility, ordering and temporal analysis are crucial. Systems such as Apache Kafka are built directly around this model.

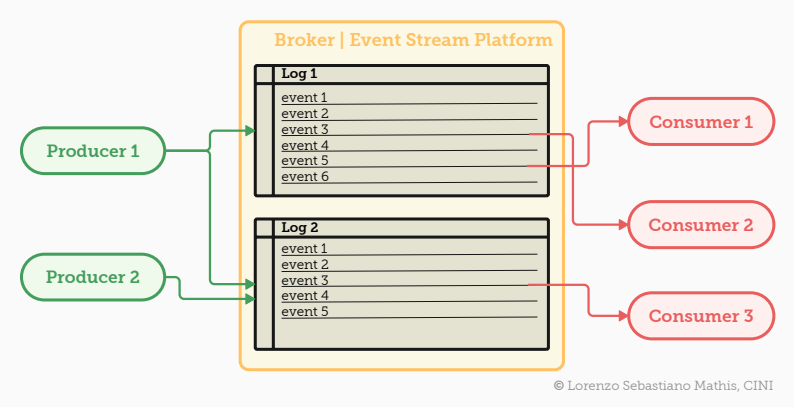


Fig. 2.2 Event Streaming messaging pattern

### Request–Reply

The request–reply pattern represents a conversational synchronous-like interaction rather than a one-way dissemination of events. A producer (the requester) sends a message expecting a specific response, while a consumer (the replier) processes it and returns the result. The broker mediates this exchange, often by using a temporary or dedicated reply queue and correlation identifiers to match requests with responses. This model resembles *remote procedure calls* but retains the benefits of messaging middleware, such as reliability, decoupling and resilience to partial failures. It is useful when synchronous workflows must be integrated into asynchronous systems, for example to query service state or delegate a task requiring a direct answer.

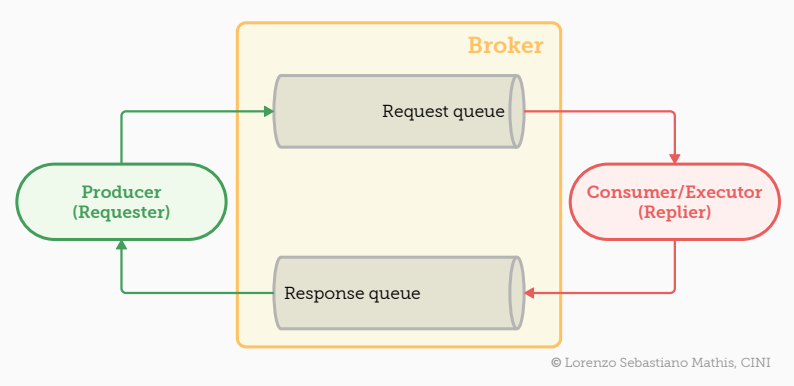


Fig. 2.3 Requrest–Reply messaging pattern

### 2.1.2 Delivery Strategies and Mechanisms

Beyond high-level patterns, brokers differ in how they distribute and deliver messages.

**Distribution models:**

- *Point-to-point* – each message is delivered to a single consumer.
- *Broadcast/cooperative* – multiple consumers receive distinct copies of the same message and process them independently.

**Retrieval models:**

- *Pull-based* – consumers explicitly poll the broker for new messages at their own pace. This makes back-pressure explicit and supports heterogeneous consumers, while the broker takes a passive role.
- *Push-based* – the broker delivers messages proactively to the consumers as they arrive. Back-pressure is managed by higher-level mechanisms such as QoS levels and in-flight message limits. This approach favors immediate delivery without polling.

**Delivery guarantees:**

- *At-most-once*: messages may be lost, but are never redelivered. This is a fire-and-forget model with minimal overhead.
- *At-least-once*: each message is redelivered until acknowledged by the consumer. Messages are guaranteed to arrive but may appear multiple times if acknowledgments are lost. Consumers must handle duplicates if operations are not idempotent.
- *Exactly-once*: each message is delivered once and only once. This requires two-phase acknowledgment and coordination, ensuring no duplicates but at the cost of computational and network overhead.

### 2.1.3 Message semantic

Message semantics describe the intent of exchanged messages. While not strictly defined by the broker, the broker's features influence how effectively these semantics are supported. Broadly, messages fall into two categories: commands and events.

**Command.** A command is a directive issued by a producer to one or more consumers, instructing them to perform a certain operation. Its essence is *something that should happen in the future*.

Commands are often transmitted via point-to-point channels (sometimes multicast). Push-based delivery with exactly-once semantics, if not idempotent, is preferred to ensure minimal latency and avoid inconsistencies.

**Event.** An event is a notification indicating that something has occurred. It does not mandate a specific reaction but informs receivers of a past occurrence. Its essence is *something that has already happened in the past*.

Events are typically broadcast and generally tolerate looser requirements in terms of latency and delivery guarantees, depending on the application.

### 2.1.4 Advanced Message Queuing Protocol – AMQP

The Advanced Message Queuing Protocol (AMQP) is an open, standardized messaging protocol designed to enable reliable and interoperable communication between distributed systems. AMQP defines both the wire-level format and the messaging semantics, making it protocol-agnostic with respect to programming languages and platforms.

#### Main Characteristics [5]

- *Standardization:* AMQP is governed by the OASIS consortium and has a stable specification.
- *Interoperability:* Applications developed in different programming languages and running on different middleware can exchange messages seamlessly.

- *Flexibility:* Supports multiple messaging patterns, including publish–subscribe, request–response and work queues.
- *Reliability:* Provides built-in delivery guarantees depending on broker configuration and client cooperation.
- *Security:* Natively supports TLS (encryption) and SASL (authentication).

### AMQP Model

AMQP defines a logical messaging model composed of:

- *Producers:* send messages to the broker.
- *Exchanges:* route messages based on defined rules.
- *Queues:* store messages until they are consumed, with support for durability and persistence.
- *Bindings:* define the relationship between exchanges and queues for message routing.
- *Consumers:* retrieve and process messages.

### AMQP Routing [3]

Messages are first published to an exchange, which can be viewed as a post office. The exchange then routes each message to one or more queues, which can be viewed as mailboxes.

The routing behavior of an exchange depends on its type:

- *Direct exchange:* the message is sent to the queue(s) whose binding key exactly matches the routing key (`queue.binding == message.routing_key`).
- *Fanout exchange:* the message is broadcast to all queues bound to the exchange.
- *Topic exchange:* the message is sent to queues whose binding key matches a pattern defined by the routing key (`queue.binding.match(message.routing_key)`). Wildcards allow flexible matching (e.g., `logs.*` matches all keys starting with `logs`).

- *Headers exchange:* the message is routed based on header attributes rather than the routing key, enabling custom routing mechanisms.

### RabbitMQ [16]

RabbitMQ is one of the most widely adopted open-source message brokers and the most popular implementation of AMQP. Originally developed by Rabbit Technologies in 2007 and now maintained by VMware, RabbitMQ supports the AMQP 0.9.1 specification as its core protocol. Over time it has also introduced support for additional protocols, such as MQTT and STOMP, through plug-ins.

RabbitMQ provides:

- A robust and battle-tested implementation of AMQP, with strong community and enterprise adoption.
- Rich routing capabilities via exchanges and bindings, including support for direct, fanout, topic, and headers exchanges.
- High availability and fault-tolerance through clustering and queue mirroring.
- Extensibility via a plug-in architecture (e.g., protocol adapters, monitoring, federation, shoveling).
- Management and observability features, including an HTTP API and a web-based management interface.

RabbitMQ is widely used in cloud-native architectures, microservices communication and enterprise integration scenarios where reliability and interoperability are critical.

## 2.2 MQTT protocol [12] [13]

MQTT is an asynchronous, lightweight messaging protocol that utilizes TCP/IP to connect and control remote small, constrained devices in unreliable network environments. It supports both data sharing and device management/control.

MQTT employs a publish–subscribe model that meets the IoT systems requirements: as the client does not need updates, it remains in idle saving both computational and network resources.

### 2.2.1 MQTT Messaging model

MQTT messaging model consists of two major components:

- The *message broker* – it is a program or device based on MQTT that acts as a message hub, receiving published messages and delivering them to subscribed clients. Its responsibilities encompass the management of the client connection, client subscription, message exchange (publishing, storing and delivering).
- The *clients* – they can be any IoT device that sends messages or uses applications to process the received messages. A client transmitting messages is called *publisher*, while one receiving messages is called *subscriber*; note that the two roles are not mutually exclusive, a client may act as both publisher and subscriber.

MQTT is a bi-directional communication protocol where messages are organized in *topics* and allows both data acquisition (data flows from the IoT devices) and device management and control (data flows to the IoT devices). The main concepts are:

- *Message* – the data that moves through the service. A message has standard format composed by a fixed header, an optional variable header and an optional payload.
- *Topic* – a named entity that denotes a feed of messages. It functions like a mailbox.
- *Subscription* – it represents the interest by a subscriber in receiving messages from a specific topic.
- *Publisher* – it is a client that produces messages and publish them into a topic.

- *Subscriber* – it is a client that subscribes to topics and receives messages from them.

The MQTT client initializes a MQTT session opening a TCP/IP connection with the broker, the standard uses two ports:

- `TCP/1883` for unencrypted traffic,
- `TCP/8883` for TLS-encrypted traffic.

### 2.2.2 MQTT Packet format

The MQTT packets reside on top of the plain or encrypted TCP protocol in the OSI model – application level. The packets are composed by three parts:

- **2-bytes fixed header** – The first byte identifies the packet type and its control flags, while the second byte indicates the remaining length of the packet.
- **Variable header** (optional) – Provides additional information specific to the packet type and supports protocol extensions.
- **Payload** (optional) – Contains the application data being transmitted.

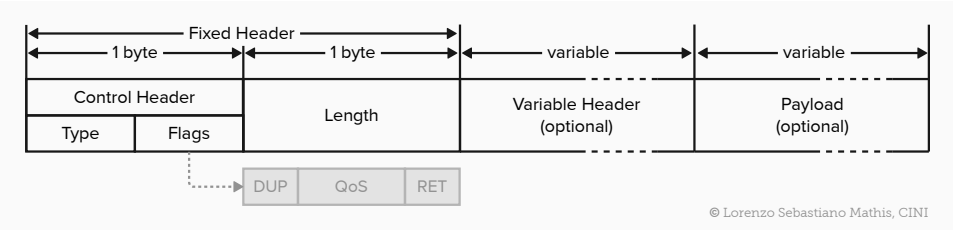


Fig. 2.4 Structure of an MQTT packet.

### 2.2.3 MQTT Features

Beyond its lightweight design, MQTT provides several features that enhance reliability and flexibility in message delivery, such as Retained Messages, Will Messages, QoS Levels, Clean Session flag and Keep-Alive parameters. These features can be configured both when establishing the connection to the broker and during publish/-subscribe actions.

**Retained Messages.** By default, the MQTT broker discards messages once they are delivered or if no subscriber is present. The retained messages feature instructs the broker to store the most recent retained message along with its topic. If available, the broker transmits this message immediately to any client that subscribes to the topic. To retain a message, the publisher must set the retain flag (`RET`) in the fixed header of the *PUBLISH* packet.

**Will Messages.** Known as the Last Will and Testament (LWT), this feature allows a client to specify a special message by setting the Will flag in the header of the `CONNECT` packet, including the message as payload and indicating the target topic. If the client disconnects unexpectedly – i.e., without sending a `DISCONNECT` message – the broker broadcasts the LWT message to all clients subscribed to the topic.

**QoS Levels.** MQTT defines three levels of Quality of Service (QoS): **QoS 0** provides at-most-once delivery, **QoS 1** ensures at-least-once delivery by retransmitting the message until acknowledged and **QoS 2** guarantees exactly-once delivery through a four-step handshake. Higher QoS levels increase both network and computational overhead.

**Clean Session.** The MQTT specification defines the Clean Session flag, which allows a client to instruct the broker on whether to maintain its state across multiple connections. It is specified in the header of the `CONNECT` packet. When set to 1, the client requests a persistent session: the broker retains its subscriptions and all unacknowledged and future QoS 1 or 2 messages, delivering them upon reconnection. When set to 0, the broker treats each connection as independent and discards any stored state.

**Keep-Alive Parameters.** MQTT includes a Keep-Alive mechanism to detect inactive clients and avoid half-open TCP connections. When connecting, the client may specify a keep-alive interval in seconds in the header of the `CONNECT` packet. This value defines the maximum allowed interval between two messages. If no data is exchanged within this period, the broker and client use special messages — `PINGREQ` and `PINGRESP` — to confirm that the connection remains active.

### 2.2.4 MQTTv5.0

So far, we have referred to MQTT version 3.1.1. In 2018, MQTT 5.0 was released, bringing improvements in performance, error reporting, small client support, and authentication, among others. Key new features include Flow Control, Shared Subscriptions, Server Redirection, User Properties, Session Expiry Interval, and Enhanced Authentication.

For this work, after considering the benefits of the new features, the support provided by the state of the art and the potential security risks associated with an expanded attack surface, I have chosen to use MQTT version 3.1.1.

### 2.2.5 Comparison with other IoT application protocols

In the literature, several works have compared MQTT with other IoT application-layer protocols. Among them, Lakshminarayana et al. [12] provide a detailed comparison with CoAP, AMQP, XMPP, DDS, WebSockets, and RESTful HTTP.

**CoAP.** The Constrained Application Protocol (CoAP) is a lightweight request–response protocol designed for constrained environment and operates over UDP and supports only two levels of reliability (confirmable and non-confirmable messages). Security is not natively integrated into the protocol and relies on DTLS to provide confidentiality, integrity and authentication. However, the combination of UDP and DTLS introduces significant limitations: unstable packet delivery, fragmentation overhead and expensive handshakes. With respect to MQTT, it provides less QoS options and less protection with higher overhead: performance evaluations show that CoAP over DTLS incurs a bandwidth overhead of approximately 1000%, whereas MQTT over TLS requires only 74–200%.

**AMQP.** The Advanced Message Queuing Protocol (AMQP) was developed to support reliable, enterprise-level messaging, offering various communication patterns and authentication with SASL and encryption via TLS. Studies shows that AMQP performs poorly in low-bandwidth environment due to its significant computational and memory overhead.

**XMPP.** The Extensible Messaging and Presence Protocol (XMPP) is a XML-based standard primarily designed for instant messaging, online gaming and telepresence applications. Security is built into the specification and is guaranteed using SALS for authentication and TLS for encryption. The protocols lacks of QoS, requires higher processing costs due to XML parsing, with respect to the lighter approach of MQTT that does not impose any payload format.

**DDS.** The Data Distribution Service (DDS) is a data-centric middleware standard developed for real-time and embedded systems. It offers advanced QoS settings and can be carried on both UDP and TCP. Security is enforced via DTLS/TLS and it provides a Security Model and Service Plugin Interfaces. However, the protocol is heavyweight and introduces high overhead, increasing power consumption and latency.

**WebSockets.** WebSockets, part of the HTML5 standard, enable real-time, bidirectional communication over a single TCP connection. Security is provided via TLS or SSL. However, WebSockets were not designed specifically for constrained environments, do not offer native QoS and generally consume more resources then MQTT.

**RESTful HTTP** RESTful HTTP is an architectural style based on standard HTTP methods and is widely supported on modern devices, making it easy to implement. It lacks of native QoS mechanisms and requires polling from clients to maintain updates, which increases latency, energy consumption and network overhead.

### Conclusions

Table 2.1 summarizes the main features of IoT application protocols, with the best-performing values highlighted in bold.

MQTT v3.1.1 stands out as the best-performing protocol across most of the considered metrics. Its major limitation lies in the security mechanisms defined by the standard: it only specifies username–password authentication and does not include authorization policy enforcement. These shortcomings of the specification require the use of TLS and other security mechanisms to make the protocol suitable.

| Feature | MQTT | CoAP | AMQP | XMPP | DDS | WebSockets | RESTful HTTP |
|---|---|---|---|---|---|---|---|
| **Technical Features** | | | | | | | |
| *Communication Model* | Publish-Subscribe | Request-Response | Publish-Subscribe | Publish-Subscribe Request-Response | Publish-Subscribe | Client-Server | Request-Response |
| *Underlying Transport Protocol* | TCP | UDP | TCP | TCP | TCP/UDP | TCP | TCP |
| *Header Size* | 2 bytes | 4 bytes | 8 bytes | Variable | Variable | 2-14 bytes | 8 bytes |
| *QoS provided* | **Yes** | **Yes** | **Yes** | No | **Yes** | No | No |
| **Technical Features on Security** | | | | | | | |
| *Security Provided* | SSL/TLS | DTLS | SASL/TLS | SASL/TLS | TLS/DTLS/SPIs | SSL/TLS | HTTPS/TLS |
| *Authentication* | Username and Password / Client Certificates | IPsec/DTLS | SASL/TLS | SASL/TLS | Auth Plugin / TLS / DTLS | HTTP / TLS-based | Token-based / API-tokens / TLS |
| *Authorization Policies* | No | No | Yes | Yes | Yes | No | No |
| *Encryption* | SSL/TLS | DTLS | TLS | TLS | TLS/DTLS | SSL/TLS | TLS |
| *Level of Security Provided* | Moderate | Moderate | **High** | **High** | Moderate | Moderate | Moderate |
| **Performance Metrics** | | | | | | | |
| *Power Consumption* | **Lowest** | Medium | High | High | High | Medium | High |
| *Bandwidth usage* | **Lowest** | **Low** | Medium | High | High | Medium | High |
| *Communication overhead* | **Low** | **Low** | High | High | High | Medium | High |
| *Protocol Complexity* | **Low** | **Low** | High | High | High | Medium | High |
| *Reliability* | **High** | Moderate | **High** | **High** | **High** | Moderate | Low |
| *Scalability* | **High** | Moderate | **High** | Low | **High** | Low | Low |

Table 2.1 Comparison of Communication Protocols [12].

However, when analyzing MQTT broker implementations, discussed in Section 2.4, many of these security lacks can be mitigated. Most brokers support TLS with X.509 certificates and mutual authentication, where the Common Name is used as the client identity (see Section 3.5).

Authorization policies are typically enforced at the broker level, often through ACLs. State-of-the-art MQTT brokers also provide more advanced authorization mechanisms (see Section 3.5.2).

### 2.2.6 MQTT Bridges [19]

An MQTT broker manages its own independent namespace of topics, creating an isolated MQTT network. For example, in a smart home setup with multiple central nodes (brokers), each broker can have the same topic structure and logical functions, but the topics of one broker are not visible to another. As a result, clients connected to different brokers cannot exchange messages directly.

In many cases, this isolation is either irrelevant or even desirable, as it provides a clear boundary between systems. However, in some scenarios — such as when multiple brokers need to collaborate, share data, or provide redundancy — it can be useful to interconnect them. To address this need, many MQTT implementations provide a mechanism called bridging, which allows brokers to exchange messages across their topic spaces.

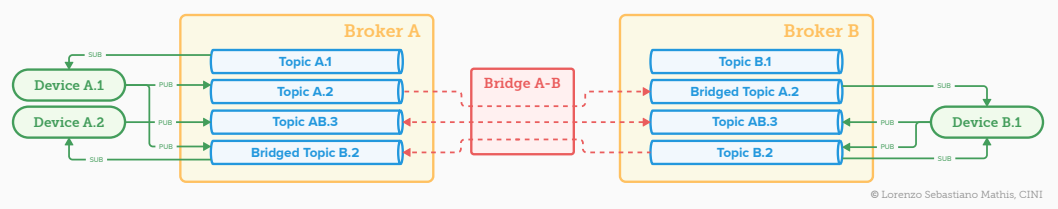


Fig. 2.5 MQTT bridge architecture

Bridging can be achieved either through the built-in mechanisms offered by several brokers or through custom implementations. In this way, each broker can serve as an isolated MQTT sub-network, while bridges interconnect these sub-networks to enable higher-level data exchange and to improve both reliability and scalability.

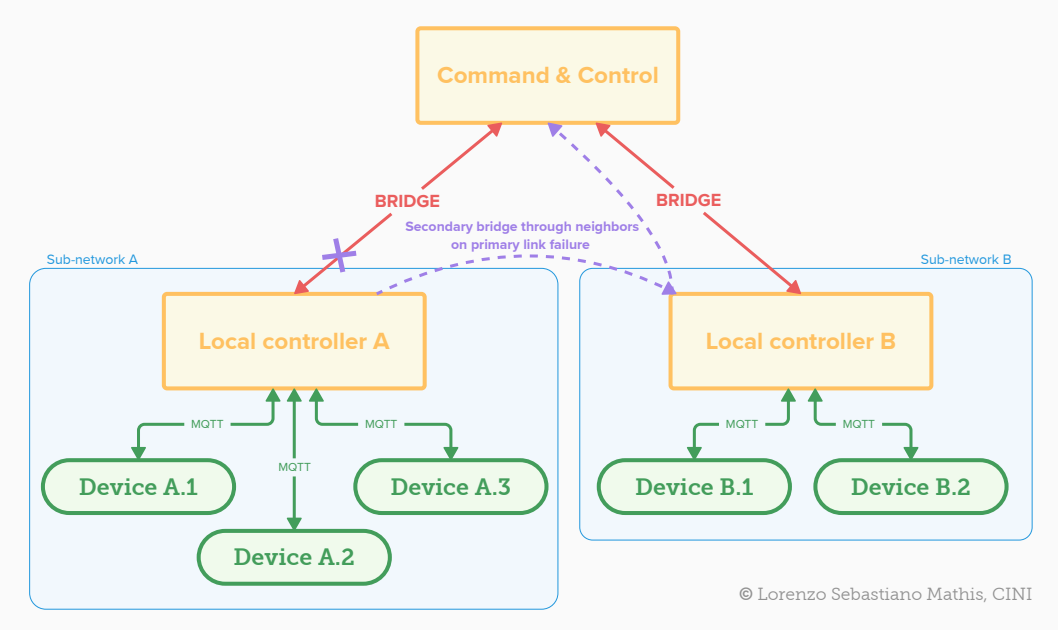


Fig. 2.6 Network partitioning with MQTT bridges.

## 2.3 MQTT Vulnerabilities and attacks

Lakshminarayana et al. [12] work has produced a classification of both MQTT protocol vulnerabilities and attacks against MQTT communication. The following section tries to summarize the outcome of that work.

### 2.3.1 MQTT Vulnerabilities

The MQTT vulnerabilities, as highlighted in Figure 2.7, can be broadly classified into two families according to their root cause:

- **Weak protocol specification**: vulnerabilities that originate from limitations or missing features in the protocol design itself.
- **Weak implementation practices**: vulnerabilities arising from insecure configurations or flawed software implementations of the protocol.

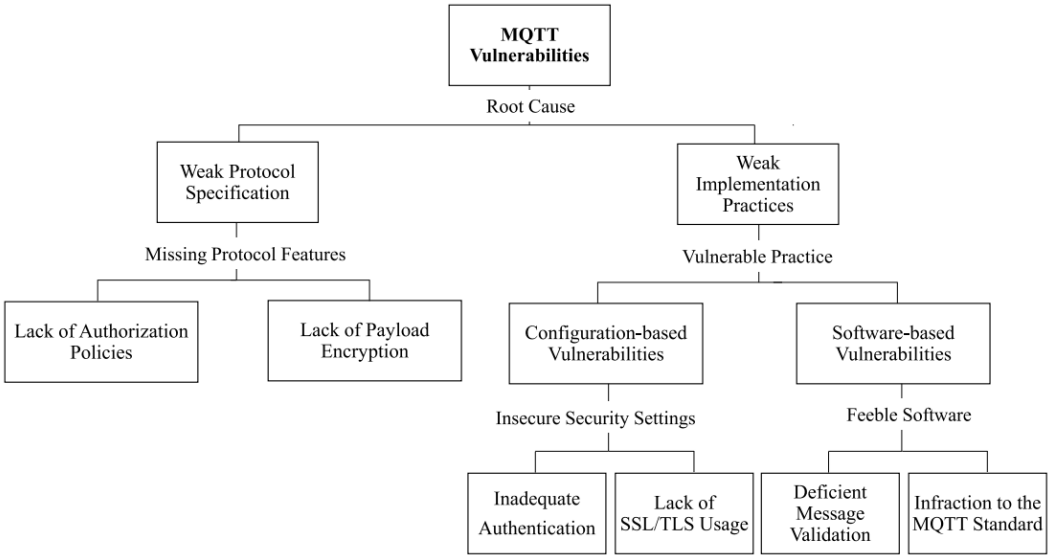


Fig. 2.7 A taxonomy of MQTT vulnerabilities [12].

### 2.3.2 Attacks against MQTT

MQTT-based IoT systems are exposed to multiple attack vectors. Attacks can either exploit known vulnerabilities in the protocol or its implementation, or leverage legitimate protocol features in ways that compromise the system. While the former largely overlaps with vulnerabilities discussed in Section 2.3.1, the latter represents attacks that can be executed even in well-configured and up-to-date systems.

### Exploiting MQTT Vulnerabilities

Attacks exploiting known vulnerabilities often target weak protocol specifications or poor implementation practices. Examples include unauthorized access due to weak authentication, eavesdropping or message tampering when payload encryption is not enforced, and denial-of-service (DoS) or command injection attacks exploiting software flaws. These attacks typically require the presence of unpatched vulnerabilities or misconfigurations, and are largely mitigated when recommended security practices are applied. Nevertheless, they remain a primary threat in many real-world deployments.

### Exploiting MQTT Protocol Features

Several attacks exploit the intended features of the MQTT protocol to disrupt services, compromise data confidentiality or exhaust system resources. Notable examples include:

**Flooding Attacks.** MQTT control packets, such as CONNECT, PUBLISH or SUBSCRIBE messages, can be sent in large volumes to overwhelm the broker, causing denial-of-service (DoS) conditions. High-quality-of-service (QoS) messages or large payloads further increase computational and memory load on brokers and clients.

**Abuse of Retain Flag.** Attackers can publish retained messages on topics, forcing brokers to store messages indefinitely. This can be exploited to exhaust storage or CPU resources or to deliver malicious messages to future subscribers.

**Exploitation of Will Messages.** The MQTT Last Will and Testament (LWT) feature can be manipulated to trigger unexpected actions or floods when clients disconnect unexpectedly. This allows attackers to indirectly control IoT devices or initiate service disruption.

**Misuse of Keep-Alive and Clean Session Flags.** By setting extreme keep-alive times or preventing session cleanup, attackers can maintain persistent connections

that consume broker resources, leading to slow DoS attacks or allowing revoked clients to remain connected and receive messages.

**Payload-based Attacks.** Maliciously large payloads can overwhelm clients and brokers, potentially causing buffer overflows, DoS or degraded performance. Attackers can craft messages that appear legitimate, making detection difficult.

### 2.3.3 Securing MQTT

As previously discussed, the MQTT protocol is not free from security flaws. While certain attacks that exploit inherent protocol features cannot always be prevented directly, their behavioral patterns can be monitored and strong authentication and authorization mechanisms can significantly reduce the attack surface. By contrast, protocol-level weaknesses can be mitigated by integrating MQTT with complementary security solutions.

MQTT can be actively hardened in three main areas, which are addressed by the proposed solution in Section 3.5:

1. Authentication,
2. Authorization, and
3. Encryption.

## 2.4 MQTT Implementations

A MQTT network is composed by two kind of entities:

- broker(s),
- clients.

A broad overview of available MQTT implementations can be found on the Wikipedia page Comparison of MQTT implementations[20], while more detailed comparative analyses, focusing on performance and features, have been provided by Lakshminarayana et al.[12] and Mishra and Kertesz [13].

### 2.4.1 MQTT Broker

MQTT brokers may adopt either centralized or distributed architectures.

- *Centralized brokers* operate as a single instance responsible for all message routing and topic management. While they offer simplicity in deployment, they may present a single point of failure and may introduce bottlenecks; thus are less suitable for mission-critical or large-scale environments.
- *Distributed brokers* implement clustering or bridging techniques to improve scalability, fault tolerance and geographical redundancy. Cluster-based solution provide higher resilience and horizontal scalability, whereas bridging approaches reduce traffic overhead and resource usage but may introduce increased latency and reduced resiliency.

The evaluation of MQTT brokers considered various factors, such as:

- Licensing model – in this work we considered only open-source products.
- Supported MQTT version and protocol-specific features.
- Scalability and clustering capabilities.
- Security features.

The comparative analysis is presented in Table 2.2

| Broker | Version | OS Support | QoS | RF | PS | SS | LWT | EL | GW | WSS | AuthN | AuthZ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *Mosquitto* | 3.x, 5.0 | L/W/M | 0 - 1 - 2 | Y | Y | N | Y | N | N | Y | TLS/SSL | ACL, Plugins |
| *EMQ X* | 3.1.1 | L/W/M + BDS | 0 - 1 - 2 | Y | Y | Y | Y | N | N | Y | TLS/SSL | Plugins |
| *HiveMQ CE* | 3.x, 5.0 | L/W/M | 0 - 1 - 2 | Y | Y | Y | Y | N | N | Y | No certificates | Java Plugin |
| *VerneMQ* | 3.x, 5.0 | L/M | 0 - 1 - 2 | Y | Y | Y | Y | Y | N | Y | TLS/SSL | ACL, Plugins |
| *RabbitMQ* | 3.1.1 | L/W/M + UNIX + BSD | 0 - 1 | P | Y | N | Y | Y | Y | Y | TLS/SSL | HTTP/REST |
| *Apache ActiveMQ* | 3.1 | L/W + UNIX + Cygwin | 0 - 1 - 2 | Y | Y | N | Y | Y* | N | Y | JAAS | XML config |
| *Apache ActiveMQ Artemis* | 3.x | L/W + UNIX + Cygwin | 0 - 1 - 2 | Y | Y | N | Y | Y* | N | Y | JAAS | XML config |
| *FlashMQ* | 3.x, 5.0 | L | 0 - 1 - 2 | Y | Y | N | Y | Y* | N | Y | TLS/SSL | Plugins |

| Legend | |
|---|---|
| RF | Retain Flag |
| PS | Persistent Session \| Clean Session Flag |
| SS | Shared Subscription (MQTTv5.0) |
| LWT | Last Will and Testament |
| EL | Error Log |
| GW | Built-in Gateway support |
| AuthN | Authentication mechanisms |
| AuthZ | Authorization mechanisms |
| WSS | Secure WebSocket transport support |

Table 2.2 Comparison of open-source MQTT brokers

Among the examined implementations, Mosquitto is distinguished by its lightweight footprint and extensive documentation, making it particularly well-suited for resource-constrained devices and development environments.

VerneMQ offers clustering capabilities and horizontal scalability, facilitating its deployment in distributed and high-availability systems.

EMQ X, along with its lightweight variant NanoMQ, provides an enterprise-oriented solution featuring a comprehensive plugin system for integration with external authentication, authorization, and monitoring infrastructures.

HiveMQ CE is primarily designed for large-scale industrial applications, whereas RabbitMQ, ActiveMQ, and Artemis extend their general-purpose messaging frameworks to support MQTT-based communication. FlashMQ represents a minimalistic alternative with a selectively implemented feature set. It is noteworthy that HiveMQ CE, ActiveMQ, and Artemis are Java Virtual Machine (JVM)-based, which may introduce additional overhead and is generally less suitable for deployments in highly constrained environments.

A comprehensive list of currently available MQTT brokers is maintained on GitHub [14].

### 2.4.2 MQTT Client Libraries

Complementary to brokers, client libraries implement the MQTT protocol stack for resource-constrained devices, edge gateways and cloud applications. Table 2.3 outlines a selection of open-source MQTT client libraries, their language support, protocol version compatibility and licensing status.

| Library | MQTT Versions | Languages |
|---|---|---|
| *Adafruit IO* | 3.1.1 | Python, Ruby, Arduino |
| *Eclipse Paho MQTT* | 3.x, 5.0 | Java, Kotlin, Python, JS, Go, C, C++, Rust, Embedded-C |
| *wolfMQT* | 3.1.1, 5.0 | C |
| *Eclipse M2Mqtt* | 3.x | C |
| *Machine Head* | 3.x, 5.0 | Clojure |
| *MQTT-C* | 3.1.1 | C |

Table 2.3 Comparison of open-source MQTT client libraries

The Eclipse Paho MQTT project has emerged as a de facto standard client library in both research and commercial projects, owing to its comprehensive language support, active community and alignment with both MQTT 3.1.1 and 5.0 standards.

# Chapter 3

# MQTT-Based framework for Secure-by-Design Energy Communities

This thesis focuses on the design of a Secure-by-Design framework for communication within Energy Communities based on the MQTT protocol, presented in Chapter 3, and on the development of a simulation framework prototype for testing Energy Community devices, discussed in Chapter 4.

This chapter is structured as follows. Section 3.1 introduces the overall architecture of the proposed framework. Section 3.2 maps the practices and guidelines — defined by Gaggero at al., see Section 1.2.4 — to MQTT features and concepts. Section 3.3 presents the abstraction layer designed to unify the interaction between smart and traditional devices in Renewable Energy Communities (RECs), providing guidelines and defining a common interface. Section 3.4 defines the communication model, while Section 3.5 analyzes the authentication and authorization mechanisms that ensure secure interactions among devices.

## 3.1 Framework architecture

The proposed framework builds upon the reference architecture defined by Gaggero et al.(see Section 1.2.1). In particular, it extends this foundation by designing a REC architecture relying on message brokers and MQTT communication protocol.

In particular, the framework:

- defines a global architecture centered on MQTT-based messaging,
- details the Energy Management System (EMS) platform, its components, roles, and interactions,
- specifies the REC member structure, including its local components and communication mechanisms.

### 3.1.1 Architecture and base components

The REC backbone network, built upon MQTT bridges between the central broker [1.c] hosted in the EMS platform and the local brokers[3.b] deployed with the smart gateways at each REC member premises. This design enables a flexible and scalable communication fabric.

Building on the high-level overview in Figure 3.1, this section discusses the architecture in details, progressively analyzing different deployment scenarios and their control implications.

#### MQTT-bridged network backbone

The backbone of the REC network — shown as yellow links in Figure 3.1 — consists of MQTT bridges that selectively share topics between the central broker and local brokers.

The bridge can be implemented using different approaches:

1. Relying on built-in broker features. Most MQTT brokers available on the market provide varying degrees of bridging capabilities. In this approach, the smart gateway connects only to the local broker.

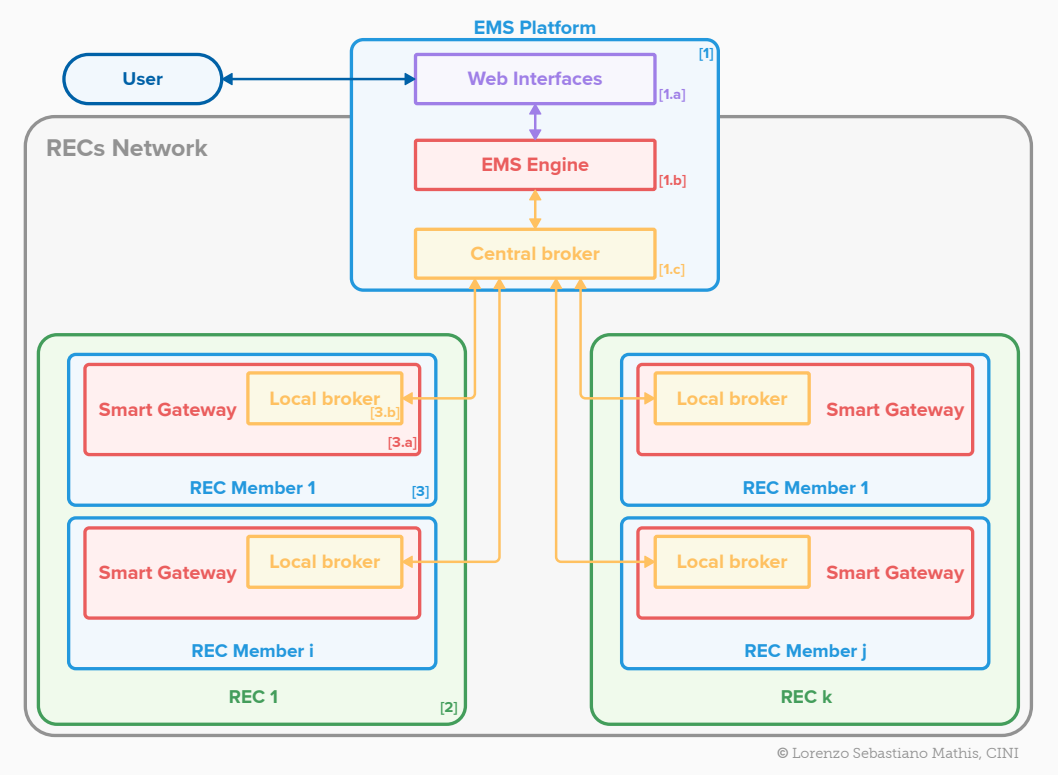


Fig. 3.1 REC network overview. *The numbered labels [n.x] correspond to the components referenced in the text.*

2. Embedding the bridging logic within the smart gateway. The gateway connects to both the local and remote brokers and handles the routing of data flows.

3. Developing a custom, independent bridging module that connects the central and local brokers. The smart gateway interacts only with the local broker.

Each approach has advantages and limitations. Broker-native bridging is lightweight and simplifies deployment, but its capabilities depend on the specific broker implementation, which can reduce transparency and interoperability in heterogeneous environments.

The latter two approaches provide greater flexibility and customization. Embedding the bridge logic in the smart gateway reduces the number of components in the network, decreasing potential points of failure, but increases the gateway's complexity. A standalone bridge module decouples network management from local logic, improving transparency and interoperability, especially when brokers are heteroge-

neous. The trade-off is additional implementation, deployment and configuration effort.

Ideally, the bridging mechanism should support secure multi-hop failover, allowing traffic to be rerouted through a neighboring broker if a direct link fails. Although optional, this feature is critical to ensure reliability in REC communications.

Broker support varies:

- Mosquitto provides a basic mechanism that binds to a single bridge link multiple addresses: the first is used as primary, while the other as fallback. The mechanism does not natively support multi-hop rerouting.
- NanoMQ, instead, offers hybrid bridging with dynamic failover, enabling multi-hop forwarding via bridge-specific topic filters.

No implementation provides a way to notify when traffic is rerouted from a direct link to a multi-hop fallback, causing the central node to infer it from connection identifiers or other metadata. Each hop relies on a separate TLS connection for encryption and authentication. However, the security of messages within intermediary brokers depends on their configuration; therefore, it is recommended to implement end-to-end protections at the protocol level, such as encrypting or signing the MQTT payload to prevent interception or impersonation during multi-hop forwarding.

### Single REC scenario

The simplest case consists of a single REC [2], where the EMS engine orchestrates the community through the central broker:

- The central broker[1.c] aggregates all the data streams originating from the local brokers in the members' premises.
- The EMS engine[1.b] processes these data and issues control commands to manage REC resources.
- Users, according to their assigned roles and permissions, interact with the REC through web-based interfaces.

### Multiple RECs scenario

The same architecture extends naturally to multiple RECs coexisting under the same EMS platform[1]. Two edge scenarios can be identified:

- Independent RECs, where each community operates autonomously, managed by a shared EMS platform.
- Collaborative RECs, where multiple communities interconnect, exchanging information and resources. In this case, the EMS engine also acts as mediator between domains.

This flexibility makes the architecture suitable for both localized communities and larger cooperative networks.

### Aggregation layers

Beyond single- and multi-REC levels, the architecture can incorporate aggregation layers at district, regional or functional levels. Such layers group multiple RECs, enabling hierarchical control strategies and distributed decision-making. This design improves scalability and prepares the platform for large-scale optimizations, potentially extending toward regional or national coordination.

### Main components

The core architectural components are:

- The EMS central broker[1.c], the backbone message hub connecting the local brokers with the EMS engine.
- The EMS engine[1.b], the computational core of the platform responsible for data processing, decision-making and orchestration.
- The EMS web interfaces[1.a], the user-facing modules enabling visualization, monitoring and interaction with the REC environment.
- Smart gateway[3.a] and local brokers[3.b], the edge components deployed at each REC member's site, responsible for local data acquisition and secure communication with the central broker.

### 3.1.2 EMS Platform architecture

The EMS platform is built on a microservice-oriented architecture, illustrated in Figure 3.2. Each service is responsible for a specific subset of functionalities. At the core of the platform lies the EMS engine[3] and its Command & Control web interface[4], supported by the security manager service[2] and the PKI management API[5]. The central broker[1], described in the previous section, serves as the messaging hub of the backbone network.

In addition to these core services, the platform integrates a Log Management System [6.a] responsible for collecting logs from both EMS services and smart gateways. Finally, the platform provides three web-based dashboards[7.x], which allow users to monitor and manage different aspects of REC operations.

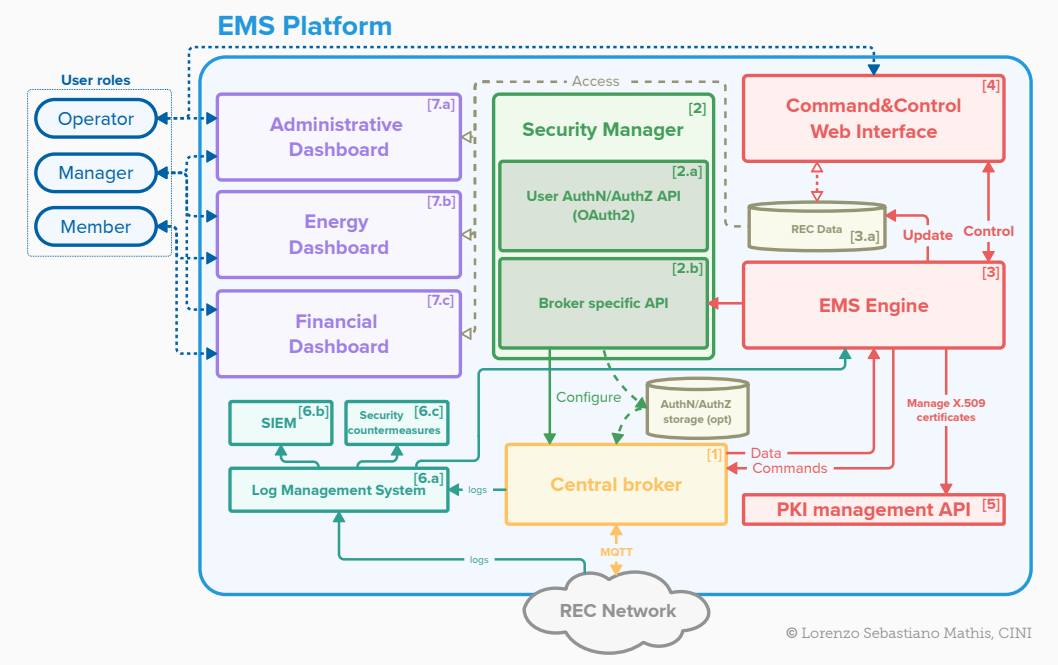


Fig. 3.2 EMS platform overview. *The numbered labels [n.x] correspond to the components referenced in the text.*

#### Central Broker [1]

The central broker is responsible for establishing secure, bidirectional communication between the EMS Engine and the smart gateways deployed at REC members'

premises. The EMS Engine is the only service authorized to interact directly with the gateways, ensuring that all communication is mediated and controlled.

To fulfill this role, the broker must satisfy several security and performance requirements:

- *(Performance)* The broker should provide clustering support to guarantee horizontal scalability and fault tolerance. The risk of central broker downtime must be minimized and disruptions must be mitigated at the protocol level.
- *(Performance)* The broker must be capable of handling high traffic volumes and supporting a large number of concurrent connections.
- *(Security)* The broker must support TLS with X.509 certificates and enforce mutual authentication.
- *(Security)* The broker must implement dynamic authentication and authorization mechanisms, either natively or via verified plugins. Static ACL-based mechanisms are not suitable for the central broker, as they often require broker restarts, which is unsustainable in a production environment.

Among available MQTT brokers, Mosquitto (with its dynamic security plugin) can be adequate for small-scale deployments, making it suitable for prototyping and testing. For production environments, however, more robust solutions such as VerneMQ (offering similar semantics but superior performance) or EMQX are recommended.

#### EMS Engine [3]

The EMS Engine is the computational and decision-making core of the entire platform and thus of the REC(s). It is responsible for:

- Collecting data from REC members via the central broker. These data streams include diagnostic information, device statistics and energy production/consumption metrics.
- Analyzing and aggregating the collected data to support both automatic and semi-automatic decision-making processes.
- Issuing commands according to predefined operational checklists, as well as control actions provided by human operators through the command-and-control interface.

- Managing security information through the *Security Manager* service and handling certificate lifecycles via the PKI management APIs.

The EMS Engine plays a critical role within the REC ecosystem. A successful compromise of this component would not only endanger the privacy of community members but could also severely disrupt REC operations—potentially rendering the entire system instable or inoperative.

### Command & Control Web Interface [4]

The Command & Control (C&C) Interface constitutes the primary access point for human operators to the operational technology (OT) domain of the REC. Through this interface, authenticated operators can interact with the EMS Engine in order to monitor, analyze, and control the operational state of the community.

Beyond monitoring and control, the C&C Interface also supports REC configuration tasks, including:

- registering new components, such as entire RECs, new members within an existing REC or additional devices for a given member;
- interacting with dedicated security services, e.g., updating security configurations of brokers;
- managing the lifecycle of TLS certificates, which form the foundation of authentication and authorization mechanisms.

The C&C Interface is implemented as a web-based service. Nevertheless, given its critical role in system operations, it must not be exposed to the public Internet. Instead, access should be restricted to authorized operators within the local environment. Adequate access control must be enforced both at the *software level* (e.g., via OAuth2) and at the *physical level*. Furthermore, the service should reside on a network segment isolated from externally exposed components, thereby ensuring proper separation of concerns and reducing the overall attack surface of the REC platform.

### Public Key Infrastructure (PKI) [5]

The platform employs a closed, private Public Key Infrastructure (PKI) to manage authentication and authorization within the energy community network. Its purpose is to provide role-specific, trusted identities for three main domains:

- the Energy Management System (EMS) components — including the MQTT central broker and the various MQTT clients deployed within the EMS platform;
- the inter-broker bridging layer, including failover paths;
- the local REC members' components — local brokers, smart gateways, and end devices.

Core PKI functions — such as certificate issuance, lifecycle management, revocation, trust distribution, and identity anchoring using Extended Key Usage (EKU) and Subject Alternative Name (SAN) conventions — are implemented, but their scope is intentionally constrained to this closed ecosystem.

Unlike a traditional PKI designed for open, multi-tenant environments, this architecture is centrally governed by the EMS Engine and does not rely on external trust anchors or public validation mechanisms. The certification hierarchy consists of a single RootCA with three functional branches:

- `EMSCA`, for EMS components,
- `BRIDGECA`, for the inter-broker bridging layer,
- `RECxCA` $\longrightarrow$ `ExCA`, for each REC member.

This segmentation is logical rather than organizational, enforcing network compartmentalization and enabling multi-tier revocation at the level of an individual device, a member or an entire REC.

Several canonical PKI components were deliberately excluded or adapted. A dedicated Registration Authority (RA) is not employed, as all certificates are issued within a single administrative domain based on pre-approved asset inventories, making third-party identity verification unnecessary. Issuance flows follow a hybrid model: EMS and bridge certificates can be provisioned automatically, whereas field device certificates are typically issued manually during scheduled maintenance.

Certificate Revocation Lists (CRLs) are used as the sole revocation mechanism, as they provide robust and predictable behaviour in intermittently connected and

resource-constrained environments; Online Certificate Status Protocol (OCSP) responders were deemed unnecessary overhead.

Finally, the PKI applies minimal Extended Key Usage (EKU) and Subject Alternative Name (SAN) policies — `ServerAuth` for brokers/listeners and `ClientAuth` for clients/bridges — rather than complex policy OIDs, aligning with the authorization capabilities natively provided by the broker implementations and their underlying mechanisms.

This design results in a PKI that is tightly integrated with the MQTT backbone, operationally lean and easily scalable: adding a new REC member requires only the creation of an `ExCA` under the appropriate `RECxCA`, without requiring modifications to EMS or bridging trust anchors.

The platform design includes a Management API [5], responsible for handling operational tasks. In the diagram, it is depicted as a separate entity, but it can also be integrated within the Security Manager 3.1.2.

### Security Manager [2]

The Security Manager is a dedicated service responsible for managing all aspects related to authentication and authorization within the REC platform.

Two distinct domains fall under its scope:

- *User-facing access control* — The platform exposes various dashboards, either externally or internally, each providing a different subset of data. Access to these interfaces must be restricted to authenticated users and granted based on their role within the REC.
- *Broker-level access control* — On the broker must be defined fine-grained rules determining where and what each client — including devices, smart gateways, bridges, and internal services — can publish or subscribe to.

The first domain can be implemented using established identity and access management (IAM) standards and tools, such as OAuth 2.0 in combination with a platform like Keycloak. Strong authentication and role-based access control (RBAC) are required to ensure appropriate data segregation across users and roles.

The second domain is more challenging to standardize, as each broker implementation provides its own mechanisms for authentication and authorization configuration. Examples include dynamically managed local file storage (e.g., the Dynamic Security Plugin for Mosquitto), database-backed systems (e.g., VerneMQ, EMQX) or HTTP-based services (e.g., EMQX, NanoMQ).

To address this heterogeneity, the Security Manager should expose a unified API that allows the EMS Engine to configure broker-specific mechanisms without being aware of their internal details. The service acts as an abstraction layer, translating a generic authorization model into the appropriate configuration format for each broker.

Specifically, the Security Manager should:

- Provide connectors for the authentication and authorization mechanisms of all brokers deployed or deployable within the REC network.
- Maintain an up-to-date database mapping each REC member to its respective broker implementation.
- Expose a secure, authenticated API to the EMS engine. This interface must abstract the broker-specific technology, enabling the EMS Engine to configure a client's authentication and authorization rules in a standardized manner.

A suggested approach for defining client authentication and authorization policies is to embed them within the extension fields of the X.509 certificate. The EMS Engine would generate the certificate signing request (CSR) and issue the certificate via the PKI management interface. The Security Manager would then extract and interpret the information contained in the X.509 certificate according to a well-defined convention, translating it into broker-specific authentication and authorization rules. Finally, the certificate is delivered to the client. Although this approach slightly increases the certificate's size, it ensures that authentication and authorization information is securely and consistently propagated along with the certificate itself.

#### SIEM and Log Extraction System [6.x]

The distributed and interconnected nature of energy communities, comprising numerous heterogeneous devices that continuously exchange critical data, necessitates the implementation of robust monitoring and security mechanisms. The diverse array

of components involved — such as smart devices, gateways, management systems and web servers — generates a continuous and voluminous flow of events and logs. These logs originate from various operational contexts and often differ in format and semantics, posing challenges for centralized analysis and response.

To effectively manage this complexity, a comprehensive Log Management System (LMS) is essential. An LMS serves as the foundational layer for collecting, storing and analyzing log data from disparate sources within the network. Its primary functions include:

- *Log Collection and Aggregation*: systematically gathering log data from multiple sources, including smart devices, gateways and other networked components, to centralize information for analysis.
- *Data Normalization*: standardizing log data into a consistent format to facilitate efficient analysis and correlation across heterogeneous systems.
- *Storage and Retention*: managing the storage of log data in compliance with organizational policies and regulatory requirements, ensuring data integrity and availability for auditing purposes.
- *Analysis and Correlation*: employing analytical tools to identify patterns, detect anomalies and correlate events across different sources to uncover potential security incidents or anomalies.

In the context of energy communities, the architecture involves smart devices generating logs that are collected by smart gateways via MQTT. The gateways process these logs, pushing critical logs in real-time to the LMS, while less critical information logs are stored locally and periodically transmitted to the LMS to reduce network load.

Other components within the network, primarily deployed within the EMS platform, also generate a wide variety of logs. These are integrated into the LMS, providing a centralized repository for all log data within the system.

The LMS does not operate in isolation but is cooperation with additional security mechanisms to enhance overall system security and safety. Integration with a Security Information and Event Management [6.b] (SIEM) system allows for advanced analysis and real-time threat detection by correlating log data with known threat intelligence. Furthermore, the LMS interfaces with the EMS engine to provide

contextual information for energy management decisions and supports other network security mechanisms [6.c] (e.g., IDS/IPS or dynamic firewalls).

This integrated approach ensures a holistic security and safety framework within energy communities, enabling proactive detection and response to both security incidents and operational anomalies while maintaining efficiency.

#### Dashboards and UI [7.x]

This architecture incorporates the web dashboards proposed by Gaggero et al. [10], integrating them with the components defined in the EMS platform. These dashboards include:

- *Administrative Dashboard*: supports REC managers in handling member data and administrative operations, such as the enrollment of a new REC member. Note that this refers only to administrative and legal enrollment; the technical onboarding is handled by the command and control interface.
- *Energy Dashboard*: enables real-time energy monitoring and can provide suggestions or request modifications to the REC energy state. For example, a user may request activation of an EV charging station.
- *Financial Overview*: supports the financial and economic management of the REC.

Since these dashboards are exposed externally, it is essential to protect them from unauthorized access and to isolate them from the rest of the platform to mitigate potential security vulnerabilities. Interactions between their backend and the REC database should be mediated through appropriate protection mechanisms, potentially using a two-tier architecture in which the web server communicates with a service layer that enforces controlled access to the underlying data.

### 3.1.3 REC Member architecture

Each REC Member hosts a local ecosystem of devices and software components that operate in coordination with the wider REC network. The objective of this architecture is to enable local management of energy production, storage and consumption while maintaining seamless interaction with the central REC backbone.

The core components deployed at each member's premises include:

- a local MQTT broker[1],
- the smart gateway[2],
- local data storage for logs and member-specific information,
- the interface to the DSO[3],
- a set of smart controllers[4.a], smart devices[4.b] and energy probes [4.c].

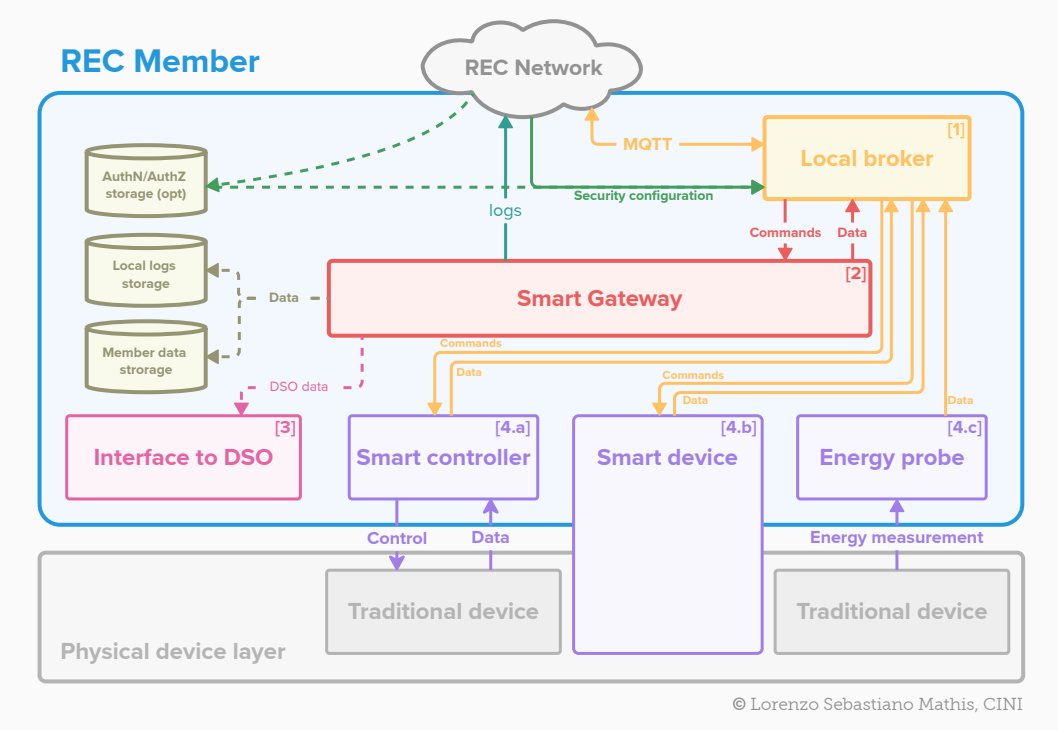


Fig. 3.3 REC member overview. *The numbered labels [n.x] correspond to the components referenced in the text.*

### Smart Gateway

The Smart Gateway[2] is a compact hardware device deployed at the premises of each REC Member. Its primary role is to interconnect the local energy network with the remote REC platform, while at the same time managing the sub-community of devices within the member domain.

From an architectural perspective, the Smart Gateway represents the integration point of multiple functions:

- it embeds the core REC management logic, enabling local decision-making aligned with the global control strategy;
- it hosts the local broker, which acts as the internal communication hub for all devices and controllers;
- it may incorporate additional modules or services required for communication, monitoring and security.

From a deployment perspective, the Smart Gateway hardware hosts several of the components highlighted in Figure 3.3, including the local databases, the optional security storage, the local broker and the gateway logic itself. This co-location of services within a single device simplifies integration and deployment, while reducing the number of devices required on the member's premises.

The local broker[1] is a lightweight MQTT broker instance that enables publish and subscribe messaging among smart components. It is also connected to the REC backbone via an MQTT bridge, as detailed in Section 3.1.3. This configuration ensures structured bidirectional communication with the central broker while preserving isolation of local traffic. Depending on the selected bridging strategy, the local broker may also handle message forwarding between the member domain and the central EMS platform and the failover mechanism.

The Smart Gateway's primary function is to coordinate local energy operations in accordance with the directives issued by the central decision-making node — the EMS engine (Section 3.1.2). This includes translating globally scoped controls into device-specific commands, relaying direct instructions, collecting and processing device data, and ensuring the secure integration of the member into the broader REC ecosystem.

Together with the bridging module, the Smart Gateway is also responsible for detecting failures of the primary bridge link and redirecting communication through neighboring gateways within the same REC. To guarantee the confidentiality and integrity of long-range communication, it additionally should provide end-to-end protection of exchanged data.

A further essential feature of the Smart Gateway is its ability to autonomously manage local devices whenever the REC member becomes temporarily isolated from the community, for instance due to public network unavailability. This functionality is fundamental to guarantee both safety and availability: the sub-community must always preserve a stable configuration, even under exceptional conditions such as natural disasters or other disruptive events.

### MQTT bridge implementation

The choice of the MQTT broker, especially for the local broker, is critical for supporting bridge functionality and the failover mechanism described in Section 3.1.1. As previously introduced, two brokers are primarily considered in this work for the local implementation: Mosquitto and NanoMQ.

Neither broker provides a native or robust failover mechanism. Both are capable of rerouting traffic to a secondary MQTT bridge, but this redirection was not originally designed with failover semantics in mind.

Since bridge-level and member-level TLS certificates are issued by different certification authorities (see Section 3.1.2), the broker must configure two distinct listeners, one for each CA. Both Mosquitto and NanoMQ support this configuration, but authorization mechanisms require careful consideration (see Section 3.5.2).

As highlighted in Section 3.1.1, the MQTT bridging mechanism can be implemented either through ad hoc logic embedded within the gateway or via a dedicated module. This discussion focuses on the dedicated module approach, although the same reasoning applies if the functionality is integrated into the smart gateway that provides greater security guarantees.

The bridging module is responsible for establishing the primary bridge[PB] and for managing failover when this link becomes unavailable. If deployed close to the smart gateway logic (either embedded within it or as a service running on the same device), the bridging module can also act as a security terminator, implementing end-to-end communication protection (see Section 3.5.3).

The module uses its certificate, issued by the `BRIDGECA`, to connect to both the local broker[1.a] and the central broker[1.b]. Its core functionality is straightforward: it subscribes to the outgoing side of a topic and republishes the messages on the corre-

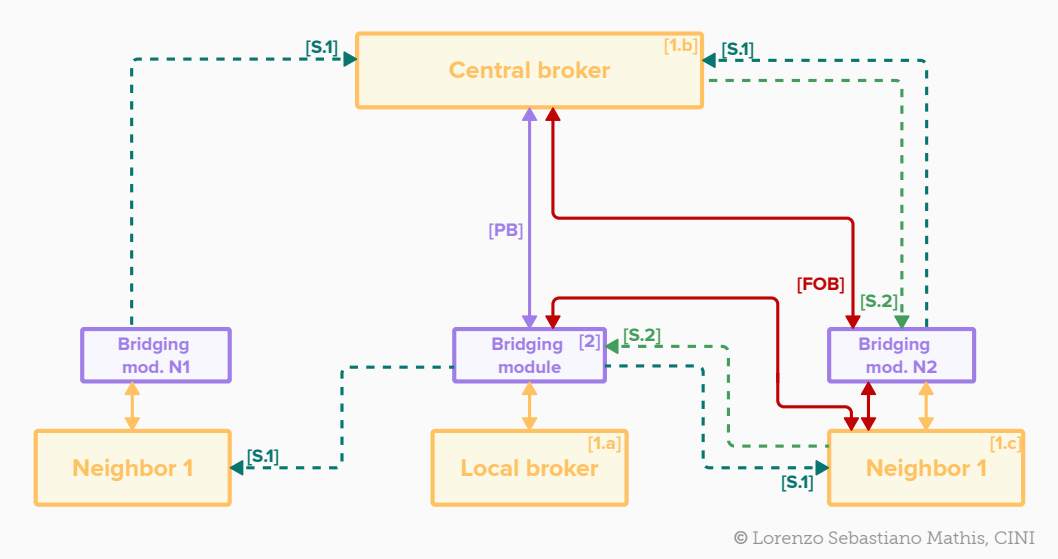


Fig. 3.4 Bridging and failover scheme. *The labels [L] correspond to the references in the text.*

sponding topic of the opposite side. Additional features — such as custom protection mechanisms, topic mapping or message buffering — can also be implemented.

The failover mechanism relies on a preliminary signaling phase triggered by the entity detecting the link failure, which may be either the central broker or the bridging module. Figure 3.4 illustrates the procedure initiated by a member, with a symmetric process followed when the failure is detected on the central broker side. The procedure consists of the following steps:

1. The bridging module[2] detects the failure of the primary link.
2. The bridging module broadcasts a signaling message[S.1] to a predefined list of neighboring modules. This message indicates the failure of the primary link, provides the identity of the requesting member and lists possible alternative paths.
3. Neighboring bridging modules forward the signaling message to the central broker.
4. The central broker receives the signaling message (possibly duplicated due to broadcasting), selects the failover path and responds with a signaling message[S.2]. This message instructs both the member's bridging module and the selected intermediary modules to establish the failover bridge link[FOB].

5. The failover bridge is established and traffic is redirected to it.

When the primary link becomes available again, communication is seamlessly migrated back to it. To prevent abuse, the signaling exchange must implement replay protection and enforce strong message security, including authentication and, where appropriate, encryption. From the perspective of an intermediary node, only two updates are required:

1. the access control configuration of the neighboring local broker[1,c] must be updated to grant both bridging modules access to the failover channel; this update is performed by the Security Manager;
2. the bridging module configuration must be extended to create and maintain the failover bridge; this update is managed by the signaling protocol.

***Note***: bridging modules are MQTT clients, and therefore cannot exchange messages directly. If the signaling protocol is implemented over MQTT, it requires a pre-authorized broadcast channel. Alternatively, it can be deployed over a side channel using protocols such as WebSocket or another lightweight application protocol. Since signaling is only required during exceptional scenarios, the overhead introduced by these mechanisms is minimal and limited to a small number of messages.

### DSO interface

The DSO interface[3] acts as an abstraction layer for the standard DSO hardware installed on the member's premises. Functionally, it corresponds to the energy meter, while the interface extends its capabilities by enabling secure communication with the Smart Gateway through the local broker. In this work, the communication link between the meter and the DSO is considered out of scope and is assumed to be inherently secure.

### Devices, Controllers and Energy Probes

Within the member premises, both smart and traditional devices coexist. To provide a uniform abstraction, three categories of smart connectors can be identified:

- *Native smart devices*[4.b] — these devices are inherently designed for connectivity and remote control. In practice, some implementations may require a lightweight interface to connect with the local broker. However, MQTT support is widely available across most modern smart devices.
- *Smart controllers*[4.a] — these units transform traditional devices into smart ones, enabling both remote control and monitoring (i.e., data acquisition).
- *Energy probes*[4.c] — when a traditional device does not support or does not require remote control, energy probes allow the acquisition of consumption metrics. They serve purely for monitoring purposes.

This categorization is based on the operational role of devices within the REC ecosystem. A more detailed analysis is provided in Section 3.3.

Having outlined the framework architecture and its main components, the next step is to relate this design to the practices and guidelines presented in Section 1.2.4. Although these practices served as the conceptual foundation of the framework, they are introduced here after the architecture, so that the mapping between abstract principles and concrete design choices can be followed more clearly.

## 3.2 Practice and guidelines mapping

The practices and guidelines proposed by Gaggero et al. [10] and discussed in Section 1.2.4 are now systematically mapped onto the architectural elements and protocol features described in the previous section.

### A. Segmentation and Perimeter Defense

1. **Dedicated sub-network for prosumer control devices:** the proposed architecture relies on large-scale public connectivity. Providing a physically dedicated sub-network for control devices would require an unsustainable deployment effort. Instead, the adopted solution introduces logical-level isolation, which is enforced at two levels:

- Between the external world and the REC network, through a TLS-enforced MQTT network.
- Between each local member network and the global REC network, through a two-layer MQTT broker hierarchy and the MQTT bridge backbone (see Section 3.1.1). Segmentation is further reinforced at the TLS level using the PKI (see Section 3.1.2).

2. **Limit wireless communication:** the proposed architecture does not prescribe any specific physical-level connectivity. Instead, it addresses network partitioning at two levels: link failures on the MQTT backbone (see Section 3.1.3) and local link failures between devices and their local broker (see Section 3.3).

3. **Firewalling:** firewalls play a critical role in the architecture. Even though they are not explicitly depicted in the diagrams, every component must implement network protection strategies:
   - EMS platform components must integrate firewalls and IDS/IPS solutions, in line with the architecture proposed by Gaggero et al., as illustrated in Figure 1.3.
   - The device (or set of devices) hosting the smart gateway, local broker and, if present, the bridging module, must be exposed to the public network only behind a firewall. Any connection attempts from entities other than the central broker, security manager or pre-authorized neighboring bridging modules must be blocked.

**B. Communication Security**

1-2. **Mutual authentication between smart gateways and the EMS platform and end-to-end encryption:** by default, the proposed architecture ensures mutual authentication and payload encryption on each individual MQTT link through the TLS protocol. Two deployment scenarios must be distinguished:
   - *Without failover:* mutual authentication and end-to-end encryption requirements may be considered satisfied by TLS. Data are encrypted between the bridging component in use (built-in, embedded in the gateway or handled by a dedicated module; see Section 3.1.3) and the central broker, with both peers mutually authenticated.

- *With failover:* the TLS layer provides mutual authentication and encryption only on a per-link basis. The failover bridge[FOB] spans two or three distinct TLS segments, depending on the chosen bridging strategy. Referring to Figure 3.4, the three-segment case includes: (i) the member's bridging module to the neighbor's local broker, (ii) the neighbor's local broker to the neighbor's bridging module, and (iii) the neighbor's bridging module to the central broker. Since TLS cannot ensure end-to-end security across multiple segments, this scenario requires an additional mechanism that applies end-to-end protection at the MQTT payload level.

Section 3.5 discusses security-related concerns in more detail.

3. **Authentication on the web server:** authentication of both external and internal web interfaces follows a standard pattern commonly adopted in modern architectures. Section 3.1.2 analyzes the Security Manager and introduces an authentication and authorization mechanism based on IAM and OAuth2.

**C. Host Security**

The EMS platform is designed as a microservice-based architecture, which enables a high level of decoupling between services. The separation of control and user functionalities (1) is achieved through compartmentalization, with each function encapsulated into dedicated services that interact via well-defined interfaces (APIs or intermediary services).

Each microservice can be deployed either on customer-premises physical machines or in a cloud environment. Both deployment models support network segmentation and the implementation of a DMZ (2).

Segmentation is organized into three main domains:

- An isolated internal network hosting the central broker, the security manager, PKI management APIs and the EMS engine.
- An internal, human-accessible network hosting the log management system and the command & control interface, available only through dedicated operator terminals.
- An externally exposed network providing dashboards accessible to authenticated users.

The MQTT network provides unidirectional channels for both data acquisition (device → gateway → EMS engine) and remote control (EMS engine → gateway → device). Read/write capabilities on these channels are strictly regulated by the authentication and authorization mechanisms of the architecture, enforcing a logical restriction on the control network (3).

Secure software development for smart gateways (4) is delegated to the final product developers. Implementations must rely on a proper MQTT client library and ensure full support for all required features (see Sections 3.4 and 3.5).

#### D. Cybersecurity Monitoring

The EMS platform includes a log management system (see Section 3.1.2) responsible for collecting, storing and analyzing logs generated by the various devices. This system is integrated with a SIEM for security-related events and interacts with the EMS engine to enable rapid responses to operational anomalies.

## 3.3 Smart and traditional devices abstraction

The Energy Communities ecosystem comprises a heterogeneous landscape of devices located within member premises. The specific nature, capabilities and implementation details of these devices fall beyond the scope of this work.

The platform must therefore accommodate this heterogeneity, where smart and traditional devices not only must coexist, but in many cases already do. Some devices are natively equipped with smart capabilities and may expose an MQTT-compatible interface, over which the communication model guidelines can be directly applied. Others may expose different interfaces, requiring adapters in order to be integrated into the ecosystem. Furthermore, the platform must be able to interoperate with a wide range of traditional devices that are already deployed or lack smart capabilities, relying on ad hoc interfaces for monitoring and, where possible, control.

Such devices may include brand-new commercial products, legacy products adapted by the manufacturer or devices integrated through third-party solutions.

To establish a baseline, this work categorizes devices, as introduced in Section 3.1.3, into three groups:

- native smart devices,
- smart controllers interfacing with controllable traditional devices,
- energy probes connecting to passive traditional devices.

This classification reflects the intrinsic nature and capabilities of the devices.

Independently of this categorization, all devices must implement the communication model described in Section 3.4 and comply with the security requirements specified by the security model in Section 3.5.

From an operational perspective, devices fall into three main superclasses which define the operations they can perform and the data they are expected to share:

- *Generators*: devices capable of producing and supplying energy. They are regarded as pure producers. In addition to standard logs, diagnostic information and state data, they provide metrics related to their energy production.
- *Loads*: devices that consume energy. They are regarded as pure consumers. Alongside logs, diagnostic information and state data, they provide metrics on their energy consumption. This abstraction covers both proactive (controllable) and passive (non-controllable) devices. In both cases, devices must behave deterministically and predictably: controllable devices must react consistently to valid commands, while non-controllable devices must reject any command with an appropriate error notification, without leading to crashes or undefined behaviors.
- *Accumulators*: devices designed to store energy. They can operate both as generators and as loads. In addition to logs, diagnostic information, and state data, they provide metrics describing the battery's status and the amount of energy absorbed or delivered, depending on whether they are in charging or discharging mode.

#### Network partitioning and device isolation

In the proposed REC architecture (see Section 3.1.3), devices connect to the local broker via network links that may vary in nature and reliability. Wired connections, e.g. IEEE 802.3 (Ethernet) or power-line communications (PLC), are generally preferred, as they provide greater stability, resilience, isolation and security. However,

for practical reasons or contextual constraints, wireless connections - such as IEEE 802.11 (WiFi), or low-power technologies - may be used to implement the links, increasing the risk of disconnection due to environmental or malicious interference.

MQTT-SN (MQTT for Sensor Network) is a variant of MQTT designed for devices relying on non-IP/non-TCP network stacks, such as IEEE 802.15.4 - ZigBee, XBee, and Bluetooth Low Energy - BLE. The design of the proposed architecture concentrates on standard MQTT over TCP/IP, the adoption of MQTT-SN represents an interesting case study and a potential extension of the framework.

Given the instability of wireless links, devices must be able to handle temporary isolation from the broker—and, by extension, from the wider network. Each device should therefore maintain a consistent and reliable local state, ensuring that its operation remains safe and autonomous. This requirement prevents disconnections from leading to failures or anomalous behaviors that could harm the overall performance or stability of the energy community.

## 3.4 Communication model — Protocol and Conventions

The framework adopts a communication model structured around MQTT topics, which encompasses both monitoring and control data flows. Four high-level categories of data flows have been identified:

- Device and Gateway Status Signaling,
- Energy Metrics,
- Logs Propagation and Extraction,
- Member and Device Remote Control.

Each data flow is mapped on a specific MQTT topic or pattern convention. This approach also improve model extensibility: a new data flow can be added simply by introducing a new convention.

The devices should connect to the broker with the `Clean Session` flag set to `false`. This instructs the broker to establish a stateful connection, maintaining the client's subscriptions and storing QoS 1 and QoS 2 messages directed to the device. Although

this introduces some overhead on the local broker, it ensures that control messages are not lost during short-lived disconnections.

### 3.4.1 Device and Gateway Status Signaling

The status signaling convention is designed to provide an early alert mechanism for both operational and infrastructural status changes. It is composed of three MQTT topics:

1. the *status flag*,
2. the *status information channel*,
3. the *diagnostic information channel*.

#### 1. Status Flag

The status flag indicates the state of the MQTT connection between a client and its broker. By convention, a device should publish the value `online` immediately after a successful connection and `offline` just before disconnecting.

Messages must be published with the `RETAIN` flag enabled and a QoS level 1 or higher. This guarantee that the status flag remains available on the topic after the initial delivery and that subscribers receive the message reliably.

In addition, devices should register a *Last Will and Testament* (LWT) message with the value `offline` on the flag topic during connection setup. This mechanism guarantees that, if the client disconnect abruptly (e.g., crash or network failure), the broker automatically updates the status flag, enabling control logic to detect and react to the failure.

The topic conventions[1] are defined as follows:

- Devices publish to the local broker on the topic:
  `/device/<device-id>/status`
- Smart gateways publish to the local broker on the topic:
  `/smart-gateway/status`

[1] If the bridge functionalities are embedded within the smart gateway, the smart gateway status convention is slightly different, the flag can be directly published on both brokers.

- Bridge clients publish to both the local broker and the central broker respectively on the topic:
  `/bridge/status`
  `/<rec-id>/<member-id>/bridge/status`
- The local flag information are automatically propagated to the central broker by the bridge:
  - from `/device/<device-id>/status`
    to `/<rec-id>/<member-id>/device/<device-id>/status`
  - from `/smart-gateway/status`
    to `/<rec-id>/<member-id>/smart-gateway/status`

**2. Status Information**

The status information channel serves as an additional channel for clients to provide additional details regarding status changes or operational sub-states.

Devices may use this channel to communicate information such as:

- the reason for a voluntary disconnection (before disconnecting),
- details of an involuntary disconnection,
- information about connection retries (after reconnection),
- other relevant status details.

The semantics of this channel can be extended; however, the convention recommends avoiding the use of this channel for issues related to energy-related operational status.

Similar to the status flag, messages on this channel should be published with the `RETAIN` flag enabled and a QoS level of 1 or higher, to guarantee persistence and reliable delivery. Since the information published may lose relevance over time, devices must include temporal metadata or explicitly clear outdated data by publishing an empty message with the `RETAIN` flag set.

The topic conventions adopt the same structure as the status flag, with messages published under the sub-topic `/<...>/status/info`.

### 3. Diagnostic Information

The diagnostic information channel is intended for retrieving diagnostic and system-related data. Devices publish messages on this topic in response to specific commands or events. Due to the critical nature of the information, messages on this channel must be published with a QoS level of 1 or higher and the `RETAIN` flag must never be set.

The specific content of the messages is defined case-by-case, depending on the physical device and the deployment context. The convention recommends using this channel for system information, such as software version, memory state (and optionally memory hashes), CPU state and other relevant diagnostic metrics.

This channel is automatically propagated from the local broker to the central broker via the bridge mechanism; smart gateways do not need to access this information.

The diagnostic procedure is triggered by two types of events:

- A direct, authenticated diagnostic command issued by the EMS engine.
- Following an update or reset of the device, such as during an OTA update procedure (if applicable).

The topic conventions are defined as follows:

- Devices publish to the local broker on the topic:
  `/device/<device-id>/diagnostic`
  bridged to
  `/<rec-id>/<member-id>/device/<device-id>/diagnostic`
- Smart gateways publish to the local broker on the topic:
  `/smart-gateway/diagnostic`
  bridged to
  `/<rec-id>/<member-id>/smart-gateway/diagnostic`
- Bridge clients publish to the central broker on the topic:
  `/<rec-id>/<member-id>/bridge/diagnostic`

### 3.4.2 Energy Metrics

The energy metrics convention provides a guideline for the standardized sharing of energy-related information. The model is based on two distinct data exchanges:

- the device → smart gateway data flow,
- the smart gateway → EMS platform data flow.

This channel is used to propagate energy production, consumption and accumulation metrics from devices, through the smart gateway, up to the EMS engine enabling informed decision-making.

Devices share information with the smart gateway via the local broker. The smart gateway aggregates and processes this information, enabling local decision-making, and then publishes the relevant energy state of the member to the EMS engine through the bridge and central broker. The exact choice of metrics and the appropriate level of detail are determined by domain experts.

The convention prescribes the following topic hierarchy for organizing metrics:

- Devices publish to the local broker on the topic:
  `/device/<device-id>/metrics/#`
- Smart gateways subscribe to:
  `/device/+/metrics/#`
  to receive metrics from all devices.
- Smart gateways publish to the local broker on the topic:
  `/platform/metrics/#`
  bridged to:
  `/<rec-id>/<member-id>/metrics/#`

### 3.4.3 Logs Propagation and Extraction

Section 3.1.2 highlighted the need to deploy a Log Management System (LMS) integrated with security components and the EMS engine. The communication model integrates the LMS with the MQTT functionalities.

The proposed architecture relies on MQTT to collect the logs from each member device to the smart gateway. The smart gateway is responsible to store locally the

logs (i.e. maintain a database where these logs are stored) and forward them to the LMS on the remote platform.

The smart gateway can be considered a less-constrained device with respect to the other member devices and it is already exposed to the public network. Log propagation to the EMS platform can be performed using the MQTT network or other solutions. To avoid overloading the MQTT network, this solution analyze non-MQTT alternatives, relying eventually on the LMS-specific features.

A implementation-agnostic solution deploys a parallel message broker — oriented to event processing (i.e. Apache Kafka) — on the EMS platform. The smart gateway connects to it pushing the logs, then the LMS consumes them. This approach completely decouples the gateway and LMS implementations but requires the deplyoment of an additional message broker on the EMS platform.

Alternatively, log collection can rely on LMS-specific mechanisms. For example, Splunk provides a lightweight agent — the Universal Forwarder — or an agent-less option over HTTP(s), both secured with mTLS. Similarly, the ELK stack and Graylog2 support both gateway-based agents and agent-less configurations.

The communication model addresses the collection of logs from member devices through the local MQTT network. Each smart device should implement a logger capable of publishing the log message on MQTT topics, optionally mirroring local log solution (e.g. file based). The model recommends to maintain small log buffer on the device to handle temporary disconnections and facilitates auditing after incidents.

The convention prescribes the following topic hierarchy for organizing logs:

- Devices publish to the local broker on the topic:
  `/device/<device-id>/log/<level>`
- Smart gateways subscribe to:
  `/device/+/log/+`
  to receive the logs from all devices.

Log levels should be standardized. A suggested hierarchy is:

1. `DEBUG` — Troubleshooting and debugging, collected only within the member if enabled for maintenance or deployment phases.

2. `INFO` — General information, collected by the gateway and forwarded asynchronously to the LMS.

3. `WARN` — Low-importance anomalies, collected and processed by the gateway. Logs of this level are downgraded to `INFO` if resolved locally, otherwise upgraded to `ERROR`.

4. `ERROR` and `CRITICAL` — Medium- and high-importance anomalies, collected by the gateway and forwarded in real-time to the LMS.

5. `FATAL` — Critical anomalies causing operational interruption, collected by the gateway and forwarded in real-time to the LMS.

### 3.4.4 Member and Device Remote Control

The remote control scheme is based on the exchange of messages — commands and command acknowledgments — over the MQTT network. Three representative scenarios are identified:

1. The EMS platform issues commands to control the behavior of an entire member.

2. The EMS platform issues commands targeting a specific device within a member's ecosystem.

3. The smart gateway issues commands to one of its member's devices.

Scenarios (2) and (3) can be considered as specializations of scenario (1). Consequently, the first scenario is analyzed in detail, while the other two are discussed as its variations.

#### Remote Control Procedure

The control convention defines an acknowledged command exchange based on two topics allowing the EMS engine to issue member level controls[1] to the smart gateway, which are processed and translated in opportune device level commands[2], then sent to the specific device. Upon receiving a command and executing the

requested action, devices respond to the smart gateway with a command response[3]. The smart gateway aggregates all the command responses and produces a member-level control response[4].

Both member-level and device-level command responses are subject to timeouts to avoid the sender to wait indefinitely for missing acknowledgments. The timeout values must be carefully configured: tight timeouts may break network latency and operational constraints, while bigger values may delay reaction times and retard the platform's ability to respond promptly to critical events.

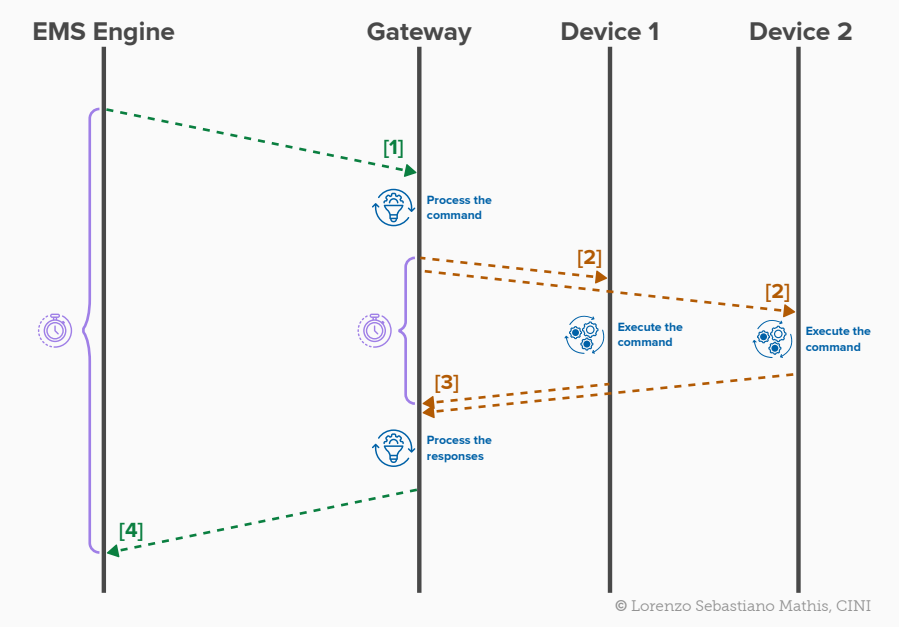


Fig. 3.5 Message Sequence Chart for Member Remote Control

This message exchange is mapped onto specific MQTT topics as follows:

- The EMS platform publishes the member level controls on the central broker to the topic:
  `/<rec-id>/<member-id>/control`
  bridged to:
  `/platform/control`
- Smart gateways subscribe on the local broker to the topic:
  `/platform/control`
  to receive controls from the platform.
- Smart gateways publish the device level commands on the local broker to the topic:
  `/device/<device-id>/control`
- Devices subscribe on the local broker to the topic:
  `/device/<device-id>/control`
  to receive commands from the smart gateway.
- Devices publish the command responses on the local broker to the topic:
  `/device/<device-id>/control/response`
- Smart gateways subscribe on the local broker to:
  `/device/+/control/response`
  to receive command responses from the devices.
- Smart gateways publish on the local broker to the topic:
  `/platform/control/response`
  bridged to:
  `/<rec-id>/<member-id>/control/response`
- The EMS platform subscribes to:
  `/+/+/control/response`
  to receive control responses from the smart gateways.

Scenario (2) represents a specialization of the general control scheme. In this case, the EMS engine, within the member-level control[1], specifies low-level operations that must be executed by a particular device in the member ecosystem. The overall message exchange remains unchanged; however, the smart gateway simply adapts the control message and forwards it to the designated device.

Scenario (3) can be represented by excluding the EMS engine from the exchange. Here, the smart gateway issues a command directly to one of its devices by specifying a device-level command[2]. Devices respond to the smart gateway in the usual manner, but in this case the response propagation terminates at the smart gateway and is not forwarded to the EMS platform.

### Member and Devices Control Protocol

The following protocol describes, step by step, the exchanges between the parties and defines a JSON-like message format to support the remote control procedure presented in Section 3.4.4.

The protocol relies on four message types:

- `member control message`[1] — the member-level control,
- `device control message`[2] — the device-level command,
- `device control response`[3] — the device's response to a command,
- `member control response`[4] — the gateway's response to a control.

The responses indicate to the message issuer the status of the processing at each stage.

The protocol proceeds as follows:

1. The EMS engine constructs a `member control message`.
2. The EMS engine publishes[1] the `member control message` to the central broker to:
   `/<rec-id>/<member-id>/control`
   and, via the bridge, to:
   `/platform/control`

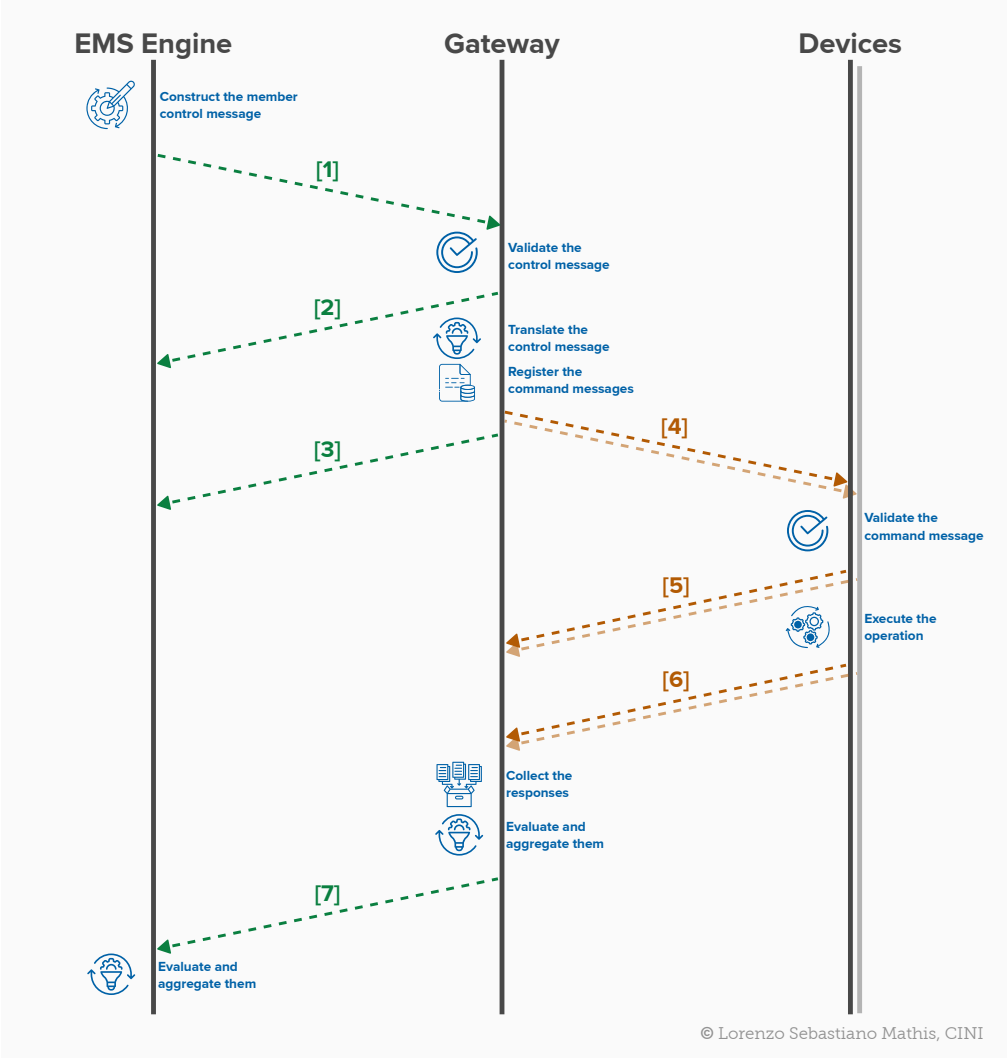


Fig. 3.6 Message Sequence Chart for Member and Devices Control Protocol

3. The smart gateway receives and validates the `member control message`, then publishes[2] a `member control response` with status `RECEIVED` to:
   `/platform/control/response`

4. The smart gateway translates the `member control message` into one or more `device control messages` and records them in its local command log.

5. The smart gateway publishes[3] a `member control response` with status `EXECUTING` to:
   `/platform/control/response`

and sends[4] the `device control messages` to the respective devices topic:
`/device/<device-id>/control`

6. Each device receives and validates the `device control message`, then publishes[5] a `device control response` with status `RECEIVED` to:
   `/device/<device-id>/control/response`
   The smart gateway updates its local command log with the new status.

7. Each device executes the requested operation and, then, publishes[6] a `device control response` with status `COMPLETED` or `FAILED` to:
   `/device/<device-id>/control/response`

8. The smart gateway waits for all `device control messages` to complete, or for their timeout to expire. If any command ends with `FAILED`, the gateway applies its retry strategy.

9. After processing all commands, the smart gateway aggregates the individual `device control message` outcomes into a single `member control response` and publishes[7] it to:
   `/platform/control/response`
   and, via the bridge, to:
   `/<rec-id>/<member-id>/control`

10. The EMS engine waits for the final `member control response`, or for its timeout to expire. If the received status is `COMPLETED`, the operation was successful; otherwise, the EMS engine reacts accordingly.

### Message format and structure

```
{
    "control_id": UUID,
    "target": {
        "member_id": string,
        "device_ids": [string]
    },
    "type": string,
    "intent": "idempotent" | "non-idempotent",
    "params": object,
    "issued_by": string,
    "issued_at": timestamp,
    "expire_at": timestamp,
    "sequence": int,
    "retry_policy": {
        "max_retries": int,
        "backoff": {
            "initial": int,
            "exponential": bool
        }
    }
    // authentication data
}
```

Listing 1 — `member command message` JSON-like format.

```
{
    "control_id": UUID,
    "command_id": UUID
    "device_id": string,
    "type": string,
    "params": object,
    "issued_by": string,
    "issued_at": timestamp,
    "expire_at": timestamp,
    "sequence": int,
    // authentication data
}
```

Listing 2 — `device command message` JSON-like format.

```
{
    "command_id": UUID,
    "device_id": string,
    "status": "RECEIVED" | "EXECUTING" | "COMPLETED" |
    "FAILED",
    "exit_code": int,
    "issued_at": timestamp,
    "details": object | string
    // authentication data
}
```

Listing 3 — `member command response` JSON-like format.

```
{
    "control_id": UUID,
    "gateway_id": string,
    "status": "RECEIVED" | "EXECUTING" | "SUCCESS" |
    "PARTIAL" | "FAILED",
    "details": {
        "general": string | null,
        "per_device": [
            {
                "device_id": string,
                "status": "SUCCESS" | "FAILED",
                "exit_code": int,
                "details": object | string,
                "executed_at": timestamp
            },
            // ...
        ]
    },
    "issued_at": timestamp,
    // authentication data
}
```

Listing 4 — `device command response` JSON-like format.

### Controls and Commands

The communication model developed in this work places its main emphasis on the infrastructural, communication and security layers of RECs. The precise set of controls and commands to manage a REC are not addressed here and should be determinate by domain experts.

The model explicitly introduces only a single command, the `diagnostic` command. The semantic and technical details of this command are described in Section 3.4.4.

In the proposed protocol, the terminology distinguishes between two types of messages: those issued by the EMS engine, referred to as *controls*, and those sent by the smart gateway, referred to as *commands*. This distinction is intended to underline the different logical scopes at which they operate:

- *EMS engine controls*: these are typically directed to REC and REC member scopes, providing controls on that level, such as *increasing energy production*, *reducing network load* or *preparing for specific operating conditions* (for instance, if a period of cloudy or rainy weather is expected, solar power production may be temporarily increased to charge the accumulators in advance). There are also cases where the EMS engine needs to address a single device directly (e.g., activating an electric vehicle charging station, causing a localized consumption peak); such interactions are supported by the protocol.
- *Smart gateway commands*: these operate on the member's devices, providing commands such as *switching off a specific device*, *enabling energy generation on an individual photovoltaic unit* or *setting a given accumulator to discharge mode*. They may be issued as a consequence of local decision-making or derived by a member-level control.

### Diagnostic Command

The `diagnostic` command is a special control issued exclusively by the EMS engine specifying the device. The command is used to initiate the diagnostic procedure and obtaining, through the diagnostic information channel (see Section 3.4.1), low level system state diagnostic information (software versions, firmware version and hash, memory maps, CPU snapshots, etc).

Given the highly sensitive nature of data published by the device in response to this command. Data authentication must be increased, also considering as untrusted the smart gateway. The command payload must be extended to include end-to-end authentication and integrity from the EMS engine to the end device. Each smart device, smart controller or energy probe should be able to handle this command providing the capability of verifying the authenticity of the command: it must be signed by the EMS engine.

## 3.5 Security model — Authentication, Authorization and Encryption

This section analyses the core security mechanisms of the proposed framework. The design chooses Transport Layer Security (TLS) as the baseline protection for all communication links. This choice is supported by three main considerations:

- TLS provides strong, well-tested security features and is the de facto standard for securing TCP-based communication.
- It is supported by the majority of devices operating over the TCP/IP stack.
- It is natively integrated with the MQTT specification and is supported by most of all MQTT brokers.

At this point, a brief digression is warranted to clarify the applicability of TLS to highly constrained devices. In Section 3.3, the possible use of UDP instead of TCP or even non-IP network stacks, has been discussed, with reference to the MQTT-SN protocol. However, its applicability within this framework is not straightforward and would require dedicated extensions. Since TLS is not included in its specification, the adoption of such protocols necessarily implies the design of alternative mechanisms capable of ensuring an equivalent level of security.

Furthermore, as highlighted from multiple perspectives, there are communication paths where TLS alone does not provide sufficient protection. In particular, the link between the EMS platform and the Smart Gateway — especially when failover paths are enabled — requires stronger guarantees: an end-to-end encryption mechanism is included at application layer to ensure confidentiality and integrity beyond individual communication hops.

### 3.5.1 Mutual Authentication

Since the REC scenario uses MQTT only for machine-to-machine (M2M) communication, mutual authentication becomes a primary security requirement, as both the client and the broker must be able to prove their identity before any data is exchanged. Otherwise, an attacker could impersonate either the client or the broker, gaining unauthorized access the REC data and remote control exchanges.

MQTT offers different authentication mechanisms: the simplest method relies on including client username and password in the `CONNECT` message; while, the more robust mechanisms employ Transport Layer Security (TLS) and mutual TLS (mTLS) with X.509 certificates.

Username and password method transmits the credentials in plain text within the `CONNECT` message, providing only client authentication. Even leaving aside the security risks associated with this kind of credentials, this approach does not provide server authentication and exposes sensitive information over the network. Consequently, this mechanism sill requires the usage of TLS to authenticate the broker and protect the transmission of credentials.

However, password-based authentication suffers from well-known limitations, including the secure password storage and rotation challenges, and the vulnerability to brute-force attacks. As consequence of this issues and considering that TLS is required anyway, the transition to mutual TLS (mTLS) results as a natural evolution: offering robust, bidirectional authentication through the use of X.509 certificates.

#### Securing REC communications with mTLS

The framework adopts mTLS as principal security mechanism which provides mutual authentication, per-link encryption and a solid base over which authorization and access control can be enforced.

The network is segmented in various authentication domains, each one served by a dedicated certification authority (see. Section 3.1.2). The MQTT brokers indulge the segmentation providing a dedicated listener for each network segment, accepting connections only from clients providing a valid X.509 certificate issued by the competent certification authority.

The client identity is derived from the Common Name and, possibly, other information specified inside the certificate. The access control mechanism described in Section 3.5.2 relies on this information, considering them as trusted.

All broker implementations considered in this work — Mosquitto and NanoMQ for local brokers, and VerneMQ and EMQ X for the central broker — support the configuration of multiple listeners, each associated with its own certification authority, and can extract the client identity from the presented X.509 certificate.

#### Authentication using TLS/PSK

TLS-PSK is an alternative authentication mechanism supported by the TLS protocol, based on pre-shared symmetric keys. This approach should be considered only for devices that are unable to manage X.509 certificates.

While the framework does not include TLS-PSK by default, its integration can be contemplated in environments where it is strictly required. In such cases, TLS-PSK should be regarded as a fallback option, used only when no other authentication method is feasible.

This approach has several security and operational limitations:

- Pre-shared keys must be securely delivered to and stored on the device.
- According with the constrained nature of the client, key rotation and secure delivery over an online channel are generally impractical, often requiring out-of-band distribution (e.g., physical delivery with a defined update procedure).
- Not all brokers support TLS-PSK, and some others may require a dedicated listener for this authentication mode.

Mosquitto supports TLS-PSK but does not allow it to be mixed with X.509-based mTLS and requires a separate listener. PSKs and the corresponding identities are stored in the same password file used for password-based MQTT authentication; however, unlike passwords that can be hashed, PSKs must be stored in plain text or, ideally, in encrypted storage — a feature that Mosquitto does not provide.

NanoMQ does not support TLS-PSK. For brokers lacking native support, a proxy can be deployed to provide TLS termination in front of a dedicated plain listener.

However, separating who performs authentication from the one handling the communication protocol introduces significant security risks and must be carefully managed.

In conclusion, TLS-PSK mechanism can be considered in scenarios with highly constrained operational environments. Its use cannot be standardized across the various implementations and may introduce serious security vulnerabilities. Its adoption is strongly discouraged except as a last-resort fallback.

### 3.5.2 Authorization and Access Control

Within a set of authenticated users, the authorization mechanism enforces access control by defining the set of actions (permissions) that a client can perform at the topic or pattern level.

A first layer of authorization is provided by TLS. The compartmentalization of network segments into different authority domains allows the broker to reject TLS connections, without arriving to the access control level, from clients that belong to the REC ecosystem but are only authorized to communicate with a different listener (e.g., bridge side vs. internal side) or with a different broker (e.g., isolation between members).

Each MQTT broker considered in this work provides heterogeneous authorization mechanisms. The proposed solutions differ in both their approach and the amount of information that can be used to identify the client.

The discussion begins with the common aspects of topic-based authorization. Regardless of the implementation, access control consists of binding a client's identity to:

- a topic filter — a specific topic or a pattern containing wildcards and, possibly, parameters; and
- a permission — i.e., read or write.

The basic mechanism to implement access control in an MQTT broker is the definition of one or more ACL files. Many brokers also support per-listener ACLs. In the context of this framework, this approach presents several security and operational limitations:

1. Access control information is stored without strong protection on the filesystem.
2. Updating the configuration requires file modification, that complicates centralized management by the Security Manager.
3. Some brokers, such as Mosquitto, require a service restart to reload the ACLs.

To address these limitations, three principal approaches have been identified:

#### MQTT-based access control configuration

This approach relies on the MQTT protocol itself to configure access control through the existing communication network. It does not require additional services or dedicated storage: the Security Manager uses special topics to remotely configure broker ACLs. Access to these topics must be pre-authorized and restricted to the manager client, which should connect directly to the broker to avoid interference from intermediary nodes in the bridged path.

The reference implementation is the Mosquitto Dynamic Security Plugin [7], available in Mosquitto v2.0+. This plugin eliminates the need to restart the broker when updating security configurations, enabling dynamic user and ACL management during runtime.

The configuration is stored in a JSON file. At present, this file is stored in plain text at a configurable path in the filesystem, meaning limitation (1) is not resolved. Since the plugin is open source, integration with OpenSSL or similar libraries is feasible and could be considered in the future.

The key concepts of Dynamic Security are clients, roles and groups:

- *Role*: a set of ACLs identified by the `rolename` attribute.
- *Client*: any entity able to connect to the broker and perform operations. Each client is identified by its `username`, as provided in the `CONNECT` packet, or, if TLS is enabled and configured, by the Common Name. Optionally, a client may include a `password`; when TLS is enabled this can be empty. The `clientid` may also be used to prevent multiple connections with the same credentials. Each client maintains its own list of roles and groups.

- *Group*: a set of clients that share a common subset of roles, identified by the `groupname`.

Each ACL represents a single permission and consists of the `acltype` (permission type), the `topic`, a `priority` value and the `allow` value (allow or deny). Supported permissions include:

- `publishClientSend` — allows the entity to publish on the topic or pattern. *Default: deny*.
- `publishClientReceive` — allows the entity to receive published messages on the topic or pattern. *Default: deny*.
- `subscribe` — allows the entity to subscribe to the topic or pattern. *Default: deny*.
- `unsubscribe` — allows the entity to unsubscribe from the topic or pattern. *Default: allow*.

The `publishClientSend` permission defines the capability of a client to send a `PUBLISH` packet to the broker, i.e., to send application data on a topic.
The `publishClientReceive` and `subscribe` permissions relate to the client's ability to receive data from a topic. Their semantics overlap: the first grants permission to receive a `PUBLISH` packet, while the second allows the client to send a `SUBSCRIBE` packet. To effectively receive data from a topic, both permissions must be granted.
The `unsubscribe` permission defines the capability of a client to send an `UNSUBSCRIBE` packet to the broker.

The `subscribe` and `unsubscribe` permissions are further divided into four subtypes: `subscribeLiteral`, `subscribePattern`, `unsubscribeLiteral` and `unsubscribePattern`. The `-Literal` types enforce literal matching between the ACL-defined topic filter and the one present in the packet, while the `-Pattern` types allow wildcard-based pattern matching.

The Dynamic Security Plugin satisfies most authentication and authorization requirements defined by the framework, with one exception: it does not support per-listener configuration. This creates a potential vulnerability: an attacker compromising one certification authority could forge a certificate with a Common Name that collides with ACLs intended for another CA's listener. For example, a malicious certificate with the bridge's Common Name could gain the same permissions as the bridge,

bypassing the smart gateway. Similarly, if the plugin is used on the central broker, a malicious bridge certificate could obtain EMS engine permissions, resulting in a severe access control breach.

In the framework scenario, (1) the CA and signing service are located in the EMS platform, completely isolated from the external world, and (2) the assignment of Common Names is controlled by the EMS engine or Security Manager, without device involvement. The described attack becomes realistic only if the CA or signing service is compromised, or if weak keys or algorithms are used — all of which would break the entire PKI. The recommendation is to adopt a standardized format for Common Names, e.g.:

- `rec_<rec_id>_<member_id>_<device_id>`,
- `bridge_<rec_id>_<member_id>`, or
- `ems_<service_id>`.

#### HTTP-based access control

This approach delegates access control decisions to an external service that allows or denies client operations via HTTP requests. If the service does not reside on the same host (requests only on `localhost`), communication must be protected with TLS.

When supported, this mechanism is suitable for both central and local broker deployments. For the central broker, the HTTP endpoints can be integrated into the Security Manager or exposed by a dedicated middleware service.

For local brokers, it is recommended to host a dedicated REST API within the smart gateway, alongside the broker. This reduces network overhead, delays and mitigate network disconnections. The Security Manager can then push configuration updates through a secure HTTPS endpoint.

The REST API design includes a small set of HTTP endpoints accessible only from localhost, and one external HTTPS endpoint where the Security Manager pushes updates, all the endpoint should implement authentication and authorization. Local configurations can be stored in lightweight databases such as SQLite and in-memory caching can be used to reduce the database accesses.

This solution is supported by NanoMQ and EMQ X and is suggested as the primary authorization (and potentially authentication) mechanism for NanoMQ [15].

NanoMQ requires two endpoints: one for authentication and one for ACL requests. For authentication, the broker can provide a subset of information, including client ID, username, password, IP address, protocol, port, Common Name and subject. Particularly relevant to this framework are the port (to identify the listener), and the Common Name and subject from the X.509 certificate (to retrieve client identity). For authorization, the broker shares client ID, username, access type, IP address, protocol, topic and mountpoint. Authorization is based on the topic, access type and client ID, while the mountpoint can be used to identify the network segment (e.g., EMS internal, bridge, or local member network).

#### Database-backed access control

This approach stores ACL configuration in a database and relies on the broker querying the database whenever a client needs authorization. Both communication and database access must be protected, with write permissions restricted to the Security Manager.

Each broker supports specific DBMS options (e.g., MySQL, MongoDB, PostgreSQL, Redis). The broker queries predefined tables to retrieve ACLs.

This solution is supported by EMQ X and VerneMQ and is primarily recommended for implementing authorization at the central broker. The DBMS can either be a stand-alone deployment within the EMS platform or integrated into the Security Manager.

Using this approach on smart gateways is heavier than the HTTP-based solution. If the HTTP microservice implements effective in-memory caching, it can provide faster responses while relying on lightweight databases such as SQLite.

EMQ X provides a table-based structure where each database entry corresponds to a specific action on a topic filter, identified by an ID and indexed by username. VerneMQ, instead, offers per-user ACLs, each record identified by the triplet (`mountpoint`, `clientid`, `username`), containing both publish and subscribe ACLs.

EMQ X allows mapping the username to the common name, the subject or even the entire certificate (comprehensive but quite large). VerneMQ, however, only

supports mapping the username to the Common Name, mountpoints can be used to enforce isolation in such cases. If improperly handled, the limited support to X.509 information mapping may lead to security risk caused by possible collisions between clients with certificates with the same Common Name, but issued by different CAs.

#### Access control rules convention

The proposed communication model defines structured communication channels mapped to specific topic patterns. Each pattern represents a unidirectional data flow, with exactly one producer and one consumer. The producer requires only write permissions (`publish` action), while the consumer requires only read permissions (`subscribe` action, and possibly `publishClientReceive`).

Topic structures follow a hierarchical scheme:

- The EMS platform identifies channels to a specific REC member using the prefix: `/<rec-id>/<member-id>`.
- The Smart Gateway identifies channels to a specific device using the prefix: `/device/<device-id>`.
- The Smart Gateway identifies channels from the EMS platform using the prefix: `/platform`.
- Data flows from devices directly bridged to the platform are mapped by the bridging module by adding the prefix `/<rec-id>/<member-id>`.

Access control rules are applied according to this structure, following the principle of least privilege. Exact matches are preferred over wildcards whenever possible.

### 3.5.3 Data-in-Motion Protection

The ecosystem described by the framework generates significant network traffic across various MQTT paths. The data transmitted — whether device metrics, member consumption and production statistics or commands issued by the EMS engine — is highly sensitive and requires guarantees of both integrity and confidentiality.

This section focuses on the need to extend the guarantees provided by TLS implementing end-to-end protection and on what should be the ends of such protection.

The adoption of TLS ensures data-in-motion security — in terms of mutual peer authentication, data integrity and confidentiality — over a single inter-node connection. Up to now, the various network components have been considered trusted, but several aspects require closer attention:

1. The security of the central broker, that, even if located near the EMS engine, remains a distinct service to which the engine connects.
2. The security of the bridging mechanism and the access control concerns that its nature involves.
3. The multi-hop nature of failover paths.
4. The security of connections between devices and the smart gateway through the local broker.

The use of end-to-end encryption extends TLS guarantees across the entire communication path between the EMS engine and each member, relying on a per-member secret.

The failover mechanism described in Section 3.1.3 is based on the cooperative behavior of REC members. Its primary goal is to improve reliability; however, it also introduces potentially insecure nodes into the communication path. For instance, an attacker who compromises a node could force other members to send data through it by disrupting their network connection. This risk introduces a second requirement: communication over a failover path must remain equivalent — both functionally and in terms of security — to a direct connection between the member and the central broker.

From a security perspective, end-to-end encryption addresses these concerns and reduces the trust required of intermediate components:

- On the EMS engine side, all outgoing traffic to a member is encrypted with its symmetric key, independently of the communication path. Incoming traffic is decrypted using the symmetric key of the member expected to publish on that channel.
- On the member side, the bridging module is responsible for decrypting the traffic addressed to it and, if failover is active, forwarding the encrypted traffic to the designated recipient via that path.

This mechanism lighten the central broker of responsibility, since all traffic passing through it is encrypted. The EMS engine represents the natural terminator of the end-to-end communication.

On the other side, the smart gateway and the local broker can be treated as a single entity, as they are hosted on the same device. Communication with local devices is always established directly through TLS and is considered secure.

Nonetheless, the mechanism has limitations. While end-to-end encryption guarantees that messages remain opaque to neighboring nodes, it cannot fully prevent vulnerabilities such as denial of service (e.g., a non-cooperative neighbor refusing to forward traffic) or side-channel attacks (e.g. traffic analysis or inference attacks).

The bridging module is a particularly critical element of this ecosystem. Due to its role, the bridge requires extensive permissions to move messages between brokers. In this framework, reference is often made to a dedicated bridging module so that the various functions can be analyzed independently. However, this architectural separation introduces an intermediate node that may itself become a source of vulnerabilities.

The key question is therefore where the end-to-end protection should terminate:

- If the mechanism is embedded within the smart gateway or implemented through broker functionalities, the terminator is unambiguously the gateway.
- If delegated to a dedicated bridging module, the terminator may be either the smart gateway or the module itself.

The most robust approach is to always terminate protection at the smart gateway. Deploying a dedicated bridging module remains feasible to manage failover paths and perform MQTT-level bridging in a transparent manner; however, designating the module as the terminator introduces significant security weaknesses.

Having discussed the need for an end-to-end protection layer on top of MQTT and its functional boundaries, it is now necessary to outline how such protection can be concretely realized. The design involves two aspects: symmetric key establishment and the traffic encryption.

**Key Exchange and Delivery.** Establishing a symmetric key between the EMS engine and each member can be achieved in more than one way. A first strategy could

be to let the EMS engine itself generate the key and deliver it to members. In this case, the key is encrypted with the member's public key and signed with the engine's key, so that authenticity can be verified. This solution places the responsibility for key generation entirely within the platform and can be integrated directly into the Security Manager.

A second approach is to negotiate the key through ephemeral Elliptic Curve Diffie–Hellman (ECDHE). Here, the smart gateway and the EMS engine derive the secret during the exchange. To protect against man-in-the-middle attacks and to support both forward secrecy and structured key rotation, each side should also maintain a long-term key pair used to sign the ephemeral parameters.

**Traffic Encryption.** After that, the actual traffic can be protected with an Authenticated Encryption with Associated Data (AEAD) scheme. Among the practical options are AES-GCM and AES-CCM, both of which combine confidentiality and integrity in a single primitives. In some deployments, however, these may not be ideal: if the hardware lacks efficient AES support or if the gateway require lighter implementations, ChaCha20-Poly1305 is often the preferred alternative because of its speed and portability.

It is also possible to avoid treating key exchange and traffic encryption as two distinct phases. Hybrid constructions such as the Elliptic Curve Integrated Encryption Scheme (ECIES) wrap together elliptic-curve key exchange, a key derivation step, symmetric encryption and message authentication. The result is a unified mechanism that provides confidentiality and integrity while at the same time establishing the session key.

# Chapter 4

# K8s-based simulation framework

## 4.1 Introduction

During the framework design phase, it became necessary to test and prototype some of the theoretically defined features discussed in Chapter 3, such as the bridging mechanism and, more simply, get in touch with the MQTT functionalities.

Docker containerization quickly emerged as the most effective way to integrate the various components and maintain a stable, isolated testing environment. This choice was further motivated by the need to deploy multiple instances of the same program — with different parameterizations — by packaging it into a Docker image.

Several existing solutions for simulating IoT environments were evaluated, including MIMIC IoT Simulator, HiveMQ Swarm and IoTIFY. However, their capabilities and operational constraints were too restrictive or offered only marginal benefits compared to a custom-built approach.

HiveMQ Swarm, for instance, enables the creation of a simulation environment running on Kubernetes, based on a master/slave architecture. A central node (the commander) distributes workload blocks, defined via XML or YAML, to its agents. These agents emulate MQTT clients that connect to the existing system under test. While effective for performance testing, this solution does not provide real support for implementing agent or device logic.

IoTIFY, on the other hand, is a cloud-native IoT simulation platform also designed to run on Kubernetes. It is capable of simulating IoT devices but supports only a

limited subset of the framework components. Furthermore, device definitions are highly constrained, as they must be expressed in a JavaScript-like language.

Similarly, MIMIC IoT Simulator provides an abstraction layer for simulating end devices and testing external infrastructures. Each device is represented as an "agent", described using the TCL language.

In summary, these frameworks focus exclusively on simulating IoT device networks and do not offer support for infrastructure simulation. Additionally, their device definition mechanisms are highly limited, e.g., lacking support for real firmware.

Since a simulator is still required for both the EMS platform and the backbone network up to the smart gateways, extending it to include device support is straightforward and represents a logical step.

The initial development relied on Docker Compose, which remains useful for testing new configurations or feature subsets. For larger-scale deployment and orchestration, the simulator now leverages Kubernetes. The current prototype, deployed on Kind using Helm, is already capable of representing the ecosystem —comprising the EMS platform and RECs — by implementing a selection of the theorized features.

The development of the simulator itself is planned as part of a future project, with the goal of extending functionality, supporting comprehensive EMS platform services and enabling plug-and-play testing of diverse devices, both through emulated firmware and higher-level abstractions.

## 4.2 Kubernetes components and dependencies

When setting up the simulation environment using the `docker-compose.yaml` file, services are defined by fetching or compiling container images, mounting volumes and configuration files, and exposing ports on one or more internal networks. In contrast, Kubernetes requires a more verbose and structured configuration. Its functionalities are divided into dedicated resources such as services, deployments and configmaps, each of which plays a distinct role in application orchestration.

This section first covers some background aspects of Kubernetes and its resources, then introduces Kind, Helm and Cert-Manager.

### 4.2.1 Kubernetes

Kubernetes [11] is an open-source orchestration platform designed to manage containerized applications across clusters of machines. It provides mechanisms for automating deployments, scaling and maintaining applications. In the context of this work, Kubernetes simplifies the orchestration of the various EMS platform components and supports the deployment of each REC member element (including gateways, devices and other required services).

In short, the Kubernetes platform is built around five core components:

**1. API Server.** The API server is the central management component of Kubernetes. It exposes the Kubernetes API, which interfaces the platform to administrators, users and cluster components. All queries and update requests are processed through the API server.

**2. etcd.** The etcd is a distributed, highly available key–value store that maintains all cluster data, such as configurations, states and metadata. It guarantees consistency and reliability, representing the *source of truth* for the cluster.

**3. Scheduler.** The scheduler assigns newly created pods to available nodes within the cluster. The decision considers the resource requirements, policies and workload constraints to balance the performances across the cluster.

**4. Controller Manager.** The manager runs a set of controllers that monitor in real-time the state of the cluster maintaining it aligned with the desired configuration (desired state). E.g., if a pod crashes, it ensures that a replacement pod is deployed automatically.

**5. Kubelet and Kube Proxy.** The kubelet is an agent running on each node ensuring that the containers defined in the pod specifications are running and healthy. The Kube proxy manages the network rules facilitating the communication between services and pods, and enforcing service discovery and load balancing.

### 4.2.2 Kubernetes Resources and Concepts

Kubernetes organizes deployments into a set of resources, each corresponding to a specific concept. The following overview briefly introduces the most relevant ones, clarifying the terminology used throughout this work.

**Pods and Containers.** Pods are the smallest deployable units in Kubernetes. A pod can host one or more containers that share resources such as storage and networking. In practice, pods act as an abstraction layer above containers, integrating them with the broader Kubernetes environment.

**Services and Ingresses.** Services and ingresses are the two main resources that address networking in Kubernetes. A service provides a stable network endpoint for accessing pods, which may be short-lived. Ingresses extends this concept by managing external access to services, typically through HTTP or HTTPS, and adding routing as well as load-balancing capabilities.

**Persistent Volumes and Persistent Volume Claims.** Kubernetes provides abstractions for data persistence, allowing information to survive beyond the life-cycle of a container. Persistent Volumes (PVs) represent storage resources made available within the cluster, while Persistent Volume Claims (PVCs) allow applications to request storage in a decoupled way, increasing flexibility.

**ConfigMaps and Secrets.** Configuration management is demanded to dedicated resources, separating configuration data from the application code. ConfigMaps store key–value pairs, which can be exposed as environment variables or mounted as configuration files within a pod. Secrets extend this mechanism to sensitive information — such as passwords, TLS certificates or API keys — ensuring secure storage and access.

**Deployments, ReplicaSets and StatefulSets.** Deployments describe how applications should be deployed, updated and scaled. ReplicaSets guarantee that a specified number of pod replicas are always running. StatefulSets extend this providing the necessary functionality to manage stateful applications reliably.

### 4.2.3 Kind, Helm and Cert-Manager

**Kubernetes in Docker (Kind)** is a tool that allows running a Kubernetes cluster using Docker containers as nodes. It is particularly useful for testing and small-scale deployments. Depending on the needs, the framework or its portions are tested on clusters with 1 to 5 nodes.

Kind does not support natively LoadBalacer services, there are various add-on to support them. This setup chooses MetalLB.

**Helm** is a package manager for Kubernetes that simplifies the deployment, management and life-cycle operations of applications running on it. A Helm Chart represents a collection of files that describe a related set of Kubernetes resources. Each chart is organized as a folder with the following structure[6]:

- `Chart.yaml` — YAML file containing information about the chart.
- `values.yaml` — Default configuration values for this chart.
- `charts/` — Directory containing any chart upon which this chart depends.
- `templates/` — Directory containing templates that, when combined with values, will generate valid Kubernetes manifest files.

The RECs ecosystem is represented by a Chart. The `values.yaml` file defines high-level constants and parameters (e.g., DNS names, CA issuers or flags enabling/disabling specific services). Each service or group of services belonging to the EMS platform has its own Chart inside the `charts/` folder, as does each REC member. Each subchart's `values.yaml` define the specific component configuration. The separation was made at the member level rather than the REC level to reduce the complexity of each subcharts.

**Cert-Manager** is an X.509 certificate controller for Kubernetes. It supports a variety of public issuers and provides the functionality required to create a private PKI, which is the feature needed by this framework. Once installed, new resources become available in the cluster. Among them, the simulator uses Certificates and Issuers to build the PKI.

## 4.3 Simulator architecture

The simulator is deployed on the Kubernetes cluster using a Helm chart. Each component is defined as a subchart and, within the simulator release, represents an isolated deployment unit. The release of the simulator should be carried out through the main chart, where global configurations are defined. The main `values.yaml` file contains flags to enable or disable each module; these boolean values are used to conditionally enable the corresponding templates.

The modules of the cluster can be grouped into three logical blocks: the EMS platform services, the PKI and the RECs. Since Kubernetes does not support nested namespaces, isolation in the cluster is enforced at a lower level: each service defines its own namespace, the PKI uses the standard `cert-manager` namespace and each REC member has its own namespace shared among all its devices.

### 4.3.1 EMS Platform

In the current version of the simulator, the EMS platform supports:

- A central broker implemented with Mosquitto, with authorization managed through the Dynamic Security Plugin.
- The Security Manager, implemented as the EMS Dynamic Security Admin, a stateless API capable of remotely configuring Mosquitto access control via the Dynamic Security Plugin.
- A CLI providing low-level capabilities to test the communication (e.g., tracing devices and sending commands).

#### EMS master broker

The EMS master broker is the instance of the central broker, implemented as a single Mosquitto service. The image used for this deployment is based on Alpine Linux and uses Mosquitto 2.0.21. The broker is first built from source, then copied into the runtime image, which includes only the broker and the dynamic security library. This process removes unnecessary components and reduces the attack surface.

The image entrypoint is a Python script responsible for the initial setup and for starting the broker service. During startup, the script prepares the default dynamic security configuration and waits for the Security Manager to become available (to prevent inconsistencies in the cluster). These operations are not related to the framework itself but are necessary to ensure a consistent cluster startup.

The Mosquitto configuration is provided through a ConfigMap, and general configuration parameters can be modified in the `values.yaml` file. The configuration enables both persistence and logging by default. The broker exposes two listeners:

1. The EMS Platform listener on port `8883`, which uses TLS with EMSCA certificates and reachable through a service at `ems-master-broker-ems.ems-master.svc.cluster.local:8883`.
2. The Bridge Backbone listener on port `8884`, which uses TLS with BridgeCA certificates and reachable through a service at `ems-master-broker-bridge.ems-master.svc.cluster.local:8883`.

The deployment uses three volumes to store broker data: `ems-master-broker-data` for the Mosquitto persistence database, `ems-master-broker-log` for storing log files and `ems-master-broker-dynsec` for saving the dynamic security plugin configuration.

The default access control configuration denies all permissions to clients, except for `unsubscribe`. To allow remote configuration, the file includes a pre-authorized role and client (CN: `ems-dynsec-admin`) for the Security Manager. This enables it to publish configuration messages to `$CONTROL/dynamic-security/v1` and read responses from `$CONTROL/dynamic-security/v1/response`.

### EMS Dynamic Security Admin

The EMS Dynamic Security Admin is the first prototype of the Security Manager. Its main purpose is to expose a REST API that allows each device and gateway startup script to automatically configure its access control rules.

Its current use is to speed up and automate the configuration of the simulation environment by delegating to each component the setup of its own access control rules. This procedure does not reflect the intended role of the Security Manager,

which is to provide such features only to EMS platform administrators; however, the underlying functionalities remain the same.

The API is implemented using Kotlin and Ktor. It provides both direct mappings to Dynamic Security commands and higher-level abstractions for standard devices. Requests are handled as POST requests, where the path specifies the broker URL and port, the resource and the operation, while the body contains the specific details of the operation. The API relies on a background service responsible for managing the request/response exchange with the broker.

For example: `POST /api/v1/central.broker.xyz/8883/clients/create`

```
{
    "username": string,
    "client_id": string,
    "text_name": string | null,
    "text_description": string | null,
    "groups": [
        {"group_name": string, "priority": int}
    ],
    "roles": [
        {"role_name": string, "priority": int}
    ]
}
```

High- and mid-level requests provided through the API are translated into Dynamic Security Plugin commands (described on GitHub [8]). These commands are then passed to the background service, which maintains a command queue for each broker. At the end of each queue, a consumer implements the MQTT client for the target broker. The consumer sends messages, links them to their corresponding MQTT response messages and propagates the responses upward, transforming them into HTTP responses at the API level. Each consumer is created together with its queue; if the queue does not receive any requests for more than 60 seconds, the client closes the MQTT connection, terminates the queue and shuts down.

### 4.3.2 RECs and REC members

As introduced earlier, the deployment of a REC is carried out by deploying its members one by one. The main `values.yaml` file provides flags to enable either the entire REC or individual members.

The `rec-pki` subchart extends `ems-pki` by creating a dedicated certification authority (CA) for each REC. This CA is responsible for signing the per-member certification authorities.

The definition of a REC consists of creating subcharts for each of its members. The simulator allows members to be defined with any structure, as long as they follow the conventions established by the framework. It also provides a template for a standard member, which can be parameterized or extended as needed.

The standard REC member structure is outlined as follows:

**Namespace.** Each member defines its own namespace, conventionally composed as `<rec-id>-<member-id>`.

**Local PKI and certificates.** Each member creates its own certification authority, signed by its RECXCA, and uses it to sign all device and smart gateway *local* certificates. The bridge certificate, however, is signed by the BridgeCA.

**Smart Gateway.** In the current version of the simulator, the smart gateway is an instance of Mosquitto, configured similarly to the central broker.

**Devices.** Devices are defined in the local `values.yaml`. The `clients` section specifies common settings such as certificate parameters, environment variables and Kubernetes resources, followed by the list of devices with their image details and optional additional environment variables. The clients are automatically deployed by iterating over this list.

### Smart Gateway

The smart gateway implementation consists of two components: the local broker and the local control logic. The logic provides only limited functionality, aimed at testing communication, and does not include decision-making capabilities.

**The local broker** is implemented with Mosquitto, exposing a single TLS-enforced listener. Bridging relies on the internal Mosquitto mechanism, configured to accept incoming commands from the central broker and to forward member information to the platform, in accordance with the conventions defined in Section 3.4.

**The smart gateway logic** is responsible for receiving standard commands, which may target either all devices of a specific category (e.g., generator, load or accumulator) or a specific device. Commands are processed sequentially and include a basic acknowledgment mechanism: if the gateway receives a command before completing the previous one, it responds with `NACK-BUSY`; on success it replies with `ACK-<cmd-id>`; and on failure with `NACK-<cmd-id>`.
Devices send data every 5 seconds, while the gateway collects this data and produces aggregated reports for the platform every 30 seconds.

**Startup script** execution consists of two configuration phases:

1. Phase one – create the default configuration for the dynamic security plugin and wait for the master broker. Then configure the bridge access control rules on the master side using the Security Manager APIs.
2. Start the local instance of Mosquitto.
3. Phase two – configure the bridge access control rules on the gateway side, again using the Security Manager APIs.
4. Finally, start the smart gateway logic.

As with the master broker, these configurations are required to set up the cluster consistently but are not part of the framework itself.

### Smart Devices

The simulator provides three default abstractions for smart devices: the smart generator, the smart load and the smart accumulator. Each abstraction has its own set of commands and shared metrics. The diagnostic command is only partially implemented.

**Startup and common aspects.** All devices share a common startup script, which waits for the local broker to become available before configuring access control rules through the Security Manager APIs. Each device is implemented as a Python3 class, extending the base Device class. Devices have a 15% probability of crashing, represented by the `UNHEALTHY` state. The `reboot` command is used to recover from this state.

The Device class implements core functionalities such as creating the MQTT client, establishing the broker connection, handling the `on_message` logic, decoding commands and providing abstractions for publishing and logging operations. The device abstraction also includes three auxiliary classes: `Command`, `State` and `Topic`. These classes can be extended or used directly. `Command` abstracts device commands, `State` represents the device operational state and `Topic` defines the set of topics used by the device.

**Smart generators and smart loads** support four commands: `power-on`, `shutdown`, `reboot` and `diagnostic`. Their operational state refers to energy production. When powered off, they enter an idle mode while remaining connected to the broker. Shared metrics include `status`, `voltage`, `current` and `temperature`.

**Smart accumulators** support six commands: `shutdown`, `reboot`, `idle-mode`, `charge-mode`, `discharge-mode` and `diagnostic`. Shared metrics include `status`, `voltage`, `current`, `soc` (State of Charge) and `temperature`.

### Smart Meter

The smart meter represents the interface to the DSO. The framework does not prescribe a concrete implementation or strict guidelines for this device, leaving

flexibility for its design. In the simulator, it is implemented as an MQTT client that gets data from the smart gateway through the local broker on dedicated topics and stores them in a JSON file.

### 4.3.3 Public Key Infrastructure – PKI

The public key infrastructure follows the structure prescribed by the framework and detailed in Section 3.1.2. In Kubernetes, it is entirely managed by Cert Manager. The upper part of the certification chain is defined in dedicated charts and confined to the `cert-manager` namespace, while member-specific certification authorities are defined together with the member in the same namespace.

To enforce isolation requirements, the design leverages a combination of `Issuers` and `ClusterIssuers`: the former can issue certificates only within their own namespace, while the latter can issue certificates across the entire cluster.

The certification levels are structured as follows:

* The `RootCA` certificate is signed by a *selfSigned* `Issuer` and serves as the root of trust of the PKI. This certification authority provides an `Issuer` capable of signing certificates within the `cert-manager` namespace.
* The `EMSCA` is signed by the `RootCA` issuer and exposes a `ClusterIssuer`, which is used to sign EMS platform certificates.
* The `BridgeCA` is signed by the `RootCA` issuer and exposes a `ClusterIssuer`, intended to sign the certificates used on the bridged backbone.
+ The `RECxCA` is signed by the `RootCA` issuer and exposes a `ClusterIssuer`, which is intended to sign the member certification authorities (`ExCA`).
- The `ExCA` and their `Issuers` are used to sign local smart gateway and device certificates.

The `ems-pki` chart defines the upper levels of the PKI (*), the `rec-pki` chart defines the various `RECxCA` (+), and the `ExCA` are defined within the corresponding member charts.

## 4.4 Evolution and Future Work

The simulator was initially developed as a test playground for the solutions theoretically defined in the framework. It is now evolving into a structured simulator project, providing a comprehensive set of features. Future work focuses on expanding its functionalities to include different brokers, both in the EMS platform and in the smart gateways, allowing each member to choose its implementation. The EMS platform simulation evolves into a testing framework, where implementations can be compared and integrated as modular building blocks, based on a common interface.

The current implementation includes a Docker image capable of simulating non-Python devices inside QEMU and integrating them into the cluster, but it testing is limited. The device model is structured around Docker images that provide a standard set of configuration data through environment variables.

# Chapter 5

# Conclusions

The thesis aimed to design a framework for the creation of secure-by-design energy communities. After analyzing the main message brokers, the MQTT protocol was identified as a promising candidate for communication. The work then focused on further exploring the protocol's limitations and its implementations.

Once the MQTT's compliance with the expected operational and environmental constraints was verified, the attention shifted to identifying its weaknesses. The critical issues concerned the predefined authentication and authorization mechanisms. Following the guidelines and practices outlined in the framework developed by Gaggero et al. [10], the work mapped MQTT's functionalities and design choices with respect to these specifications, simultaneously analyzing the various components and placing them within the proposed platform.

Isolation was ensured through two complementary strategies: on the one hand, the compartmentalization of functionality into microservices with appropriate authentication and access control mechanisms; on the other, the separation of MQTT network segments using TLS-enforced bridges and the adoption of a hierarchical PKI structure. Within each segment, the theoretical access control model was implemented and validated on concrete solutions. Furthermore, the use of an end-to-end protection mechanism between the members' on-premise smart gateways and the EMS engine hosted on the central platform was suggested.

The proposed framework is based on an MQTT network infrastructure created through bridges between local brokers and the central one, capable of decoupling the local management of individual members from the global management delegated

to the central platform. The MQTT backbone was designed to support failover mechanisms, which – by leveraging neighboring gateways belonging to the same REC – enable the activation of an alternative communication path in the event of a failure of the direct connection.

Finally, a simulation framework prototype was developed to test some of the proposed solutions. This prototype used containerization via Docker and, depending on the needs, Docker Compose or Kubernetes for component orchestration.

In conclusion, the thesis demonstrated that the MQTT protocol, thanks to the features offered by its implementations, is fully compatible with the operational and security requirements of the framework, while ensuring broad support and high performance.

# Chapter 6

# Future work

The framework's design tried to cover the main aspects and components present in the Renewable Energy Communities ecosystem. However, several lines of research and implementation activities remain open and constitute natural developments from this work.

Of particular interest are the implementation of the Log Management System, including the mechanism for extracting logs from smart gateways and their integration with the platform, and the development of the Security Manager service, the requirements of which have been analyzed but not yet implemented. At the same time, the failover mechanism, based on the MQTT bridge network between community gateways, requires further refinement, with particular attention to analyzing the signaling protocol and associated requirements. Furthermore, the implementation of the various services conceptualized in the EMS platform, their integration within the infrastructure and the definition of the related internal and external security measures (e.g., firewalls, IDS/IPS, etc.) remain to be explored in greater depth.

Another research perspective concerns the use of MQTT on different network stacks, such as UDP or non-IP protocols, as well as the integration of MQTT-SN. These approaches, still little explored, could significantly expand the communication possibilities between local brokers and devices installed at community members.

Other key features include the design of over-the-air (OTA) update procedures for smart gateways and, possibly, smart devices. This feature, particularly critical from a security perspective, requires in-depth study and dedicated implementation.

Finally, the prototype framework described in Chapter 4 offers considerable scope for expansion. It could eventually evolve into a full-fledged simulator, capable of supporting experimentation and comparison between different implementations and usage scenarios. Of particular interest is the ability to directly test device firmware by emulating it with QEMU via appropriate containers.

# References


[1] Algowatt. *ER-LIBRA CE*. https://algowatt.com/librace/.

[2] Bokolo Anthony Jnr. "Distributed energy prosumer communities and the application of emerging technologies: A systematic literature review". In: *Sustainable Futures* 9 (2025), p. 100794. ISSN: 2666-1888. DOI: https://doi.org/10.1016/j.sftr.2025.100794. URL: https://www.sciencedirect.com/science/article/pii/S2666188825003594.

[3] Apache Software Foundation. *Apache Qpid – AMQP Implementations*. https://qpid.apache.org/components/. Accessed: 2025-08-26. 2025.

[4] Clearwatts. *What is a Renewable Energy Community?* https://cleanwatts.energy/insight/what-is-a-renewable-energy-community/. 2023.

[5] OASIS Advanced Message Queuing Protocol (AMQP) Technical Committee. *OASIS Advanced Message Queuing Protocol (AMQP) Version 1.0*. Tech. rep. Accessed: 2025-08-26. OASIS, 2012.

[6] Helm Documentation. *Charts*. https://helm.sh/docs/topics/charts/. Accessed: 2025-09.

[7] Eclipse. *Dynamic Security Plugin*. https://mosquitto.org/documentation/dynamic-security/. Accessed: 2025-07/08/09.

[8] Eclipse. *Dynamic Security Plugin commands*. https://github.com/eclipse-mosquitto/mosquitto/tree/master/plugins/dynamic-security. Accessed: 2025-07/08/09.

[9] Maps Energy. *Nuovi strumenti e funzionalità al servizio delle Comunità Energetiche Rinnovabili*. https://energy.mapsgroup.it/rose-la-piattaforma-cloud-per-una-gestione-intelligente-delle-cer/.

[10] Giovanni Gaggero et al. "An IEC 62443-Based Framework for Secure-by-Desing Energy Communities". In: *IEEE Access* PP (Jan. 2024), pp. 1–1. DOI: 10.1109/ACCESS.2024.3492316.

[11] Kubernetes. *Kubernetes Documentation / Concepts*. https://kubernetes.io/docs/concepts/. Accessed: 2025-09.

[12] Sujitha Lakshminarayana, Amit Praseed, and P. Santhi Thilagam. "Securing the IoT Application Layer From an MQTT Protocol Perspective: Challenges and Research Prospects". In: *IEEE Communications Surveys  Tutorials* 26.4 (2024), pp. 2510–2546. DOI: 10.1109/COMST.2024.3372630.

[13] Biswajeeban Mishra and Attila Kertesz. “The Use of MQTT in M2M and IoT Systems: A Survey”. In: *IEEE Access* 8 (2020), pp. 201071–201086. DOI: 10.1109/ACCESS.2020.3035849.

[14] MQTT GitHub contributors. *Brokers*. https://github.com/mqtt/mqtt.org/wiki/brokers. Accessed: 2025-08-28. 2021.

[15] NanoMQ. *HTTP Authorization Configuration*. https://nanomq.io/docs/en/latest/access-control/http.html. Accessed: 2025-07/08/09.

[16] RabbitMQ Developers. *RabbitMQ Documentation*. https://www.rabbitmq.com/documentation.html. Accessed: 2025-08-26. 2025.

[17] Regalgrid. *Regalgrid platform*. https://regalgrid.com/piattaforma-regalgrid/.

[18] Mosè Rossi. *Definition and applications of a Renewable Energy Community (REC)*. https://poweringcitizens.eu/wp-content/uploads/2025/05/5-Renewable-Energy-Communities-RECs-and-their-future-role-in-the-energy-sector-1.pdf. 2025.

[19] V. Thirupathi and K. Sagar. “A Survey on MQTT Bridges, Challenges and its Solutions”. In: *2022 International Conference on Automation, Computing and Renewable Systems (ICACRS)*. 2022, pp. 58–62. DOI: 10.1109/ICACRS55517.2022.10029241.

[20] Wikipedia contributors. *Comparison of MQTT implementations*. https://en.wikipedia.org/wiki/Comparison_of_MQTT_implementations. Accessed: 2025-08-28. 2025.