UNIVERSITÀ DEGLI STUDI DI MESSINA

DIPARTIMENTO DI INGEGNERIA

Corso di Laurea Magistrale in
Engineering and Computer Science (LM-32)

Distributed Systems

Project:

Smart City - Fire Detection System

Students:
Allegra Davide Giuseppe

Miano Alberto

Musmeci Edoardo

ANNO ACCADEMICO 2023-2024

Contents

Introduction

In recent years, the rapid development of technologies for the Internet of Things (IoT) and distributed systems has led to a significant evolution in the field of smart cities. In this context, the integration of intelligent and automated solutions is revolutionising urban infrastructure management, offering new opportunities to improve the safety, efficiency and quality of life of citizens.

The project presented is placed in this scenario and aims to simulate a solution for emergency management in urban areas, with particular attention to the detection and management of risk situations. Through the adoption of advanced technologies, such as reliable communication protocols, containerised infrastructures and distributed architectures, it has been possible to create a scalable system adaptable to the modern needs of smart cities.

Several key technologies were used to realise the system.

- **Raspberry Pi with Pi Camera**: it is used to simulate a sensor capable of detecting a fire, activating the necessary actuators.
- **MQTT Protocol**: hosted on AWS, it has been used to manage communication between devices, ensuring efficiency and reliability.
- **AWS IoT Core**: managed cloud service enabling secure device communication. It uses MQTT, a lightweight messaging protocol, for real-time device-to-cloud communication. Devices publish/subscribe to topics for seamless data exchange. Features include scalability, security (TLS), and rules for data processing. Ideal for IoT applications like monitoring and automation.

- **Docker**: ssed to containerise the various components of the system: Network, Orchestrator, Controller, Database and the Actuators User-Interface.

- **Flask**: used to create two servers for managing a controller and an orchestrator. The controller is responsible for coordinating and managing the connected devices and ensuring that they communicate correctly with the orchestrator; the latter is responsible for collecting and analysing data from the various devices. Both servers are based on an SQL database, which is used to store information about sensors, actuators, and events.

- **NGINX**: configured as a reverse proxy to manage communication between servers. Each communication is interposed through the container network that interacts with the various docker containers in the system.

- **MySQL**: scalable, open-source relational database used to manage structured data. In distributed systems, it supports horizontal scaling, replication for fault tolerance, and high performance for large-scale applications.

- **SQL Stored Procedures**: precompiled set of SQL statements stored in a database, offering better performance, reusability, and maintainability compared to embedding SQL in Python code. It reduces network overhead by executing logic directly on the database server, ensures data security through controlled access, and simplifies complex operations with encapsulated logic. Ideal for repetitive tasks or multi-step queries in applications.

- **Control Access**: used to control access to the system and related calls to endpoints in the API flask set arranged by containers. Each authorized IP address is contained in an "access" table inside the MySQL database.

- **SVM (Support Vector Machine)**: supervised machine learning algorithm used for classification and regression tasks. It finds the

optimal hyperplane to separate data points into classes. In a distributed fire detection system, SVM can classify sensor data (fire, temperature, smoke levels) to identify fire risks. Its ability to handle high-dimensional data and outliers makes it effective in real-time, distributed environments with diverse sensor inputs.

Thanks to this combination of technologies, the proposed system is distinguished by its modularity and the ability to respond in a timely manner to critical events, representing a significant step towards the adoption of innovative solutions for security in smart cities.

Chapter 1

Overview

1.1 Deployment Diagram

The deployment diagram, used in UML, represents the physical architecture of a system, highlighting on which physical nodes the software components are distributed and how they communicate with each other. The Smart City system has therefore been designed as follows:

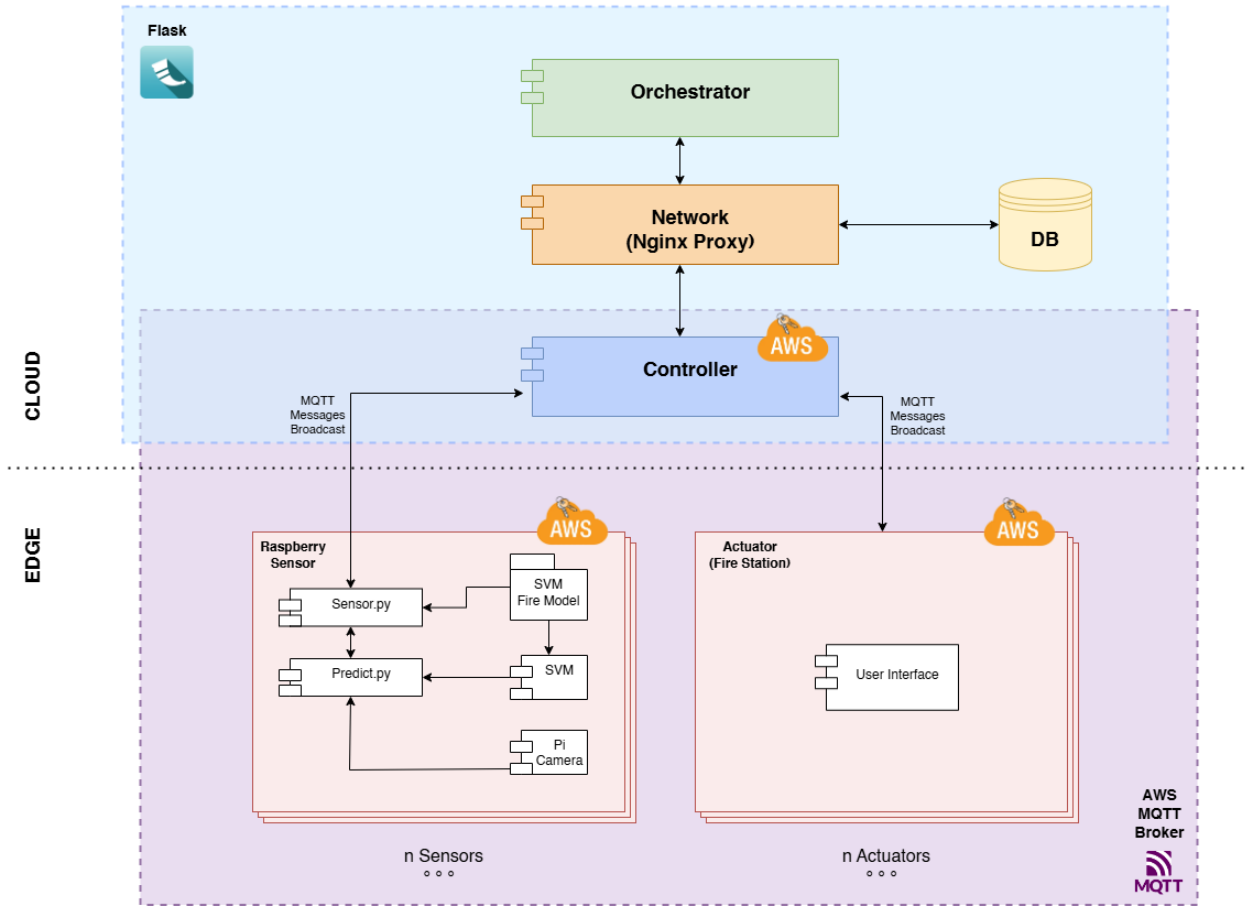


Figure 1.1: Deployment Diagram

The system consists of two macro areas:

- Cloud: where we have a docker container controlling the network (nginx reverse proxy), an orchestrator, a controller, and a shared database. The three components "talk to each other" via the container network; when an API is called, the call is delegated to the network and via a path shunts the request to the container orchestrator or controller.

  Example:

  – PUT /controller/sensor busy: 1

  This request to nginx will be routed to the container controller in this case with PUT method.

- Edge: where we find the layer of physical devices namely sensors and actuators. The devices are connected to the controller via MQTT protocol. Both the controller and all devices talk to each other via shared channel. Both controller and devices remain synchronized with each other in order to avoid disconnections as follows:

  – Controller: sends a PING to the devices every minute to ensure their constant presence. The controller expects within a certain time limit a response return called PING_ACK. If this does not occur the device is considered offline;

  – Device: sends a CHECK_CONTROLLER to the controller every minute to ensure its constant presence. The device expects within a certain time limit a response back called CONTROLLER_ACK. If this does not occur the controller is considered offline. If the controller is offline the device will continue to try to connect until the controller comes back online again.

1.2 Sequence Diagrams

The sequence diagram is a graphical tool used in software engineering, especially in the context of UML (Unified Modelling Language), to represent how objects and components of a system interact with each other over time. The diagram visualises the order and flow of messages exchanged between the different actors and components of the system, clearly showing the sequence of operations.

1.2.1 Sensor Login

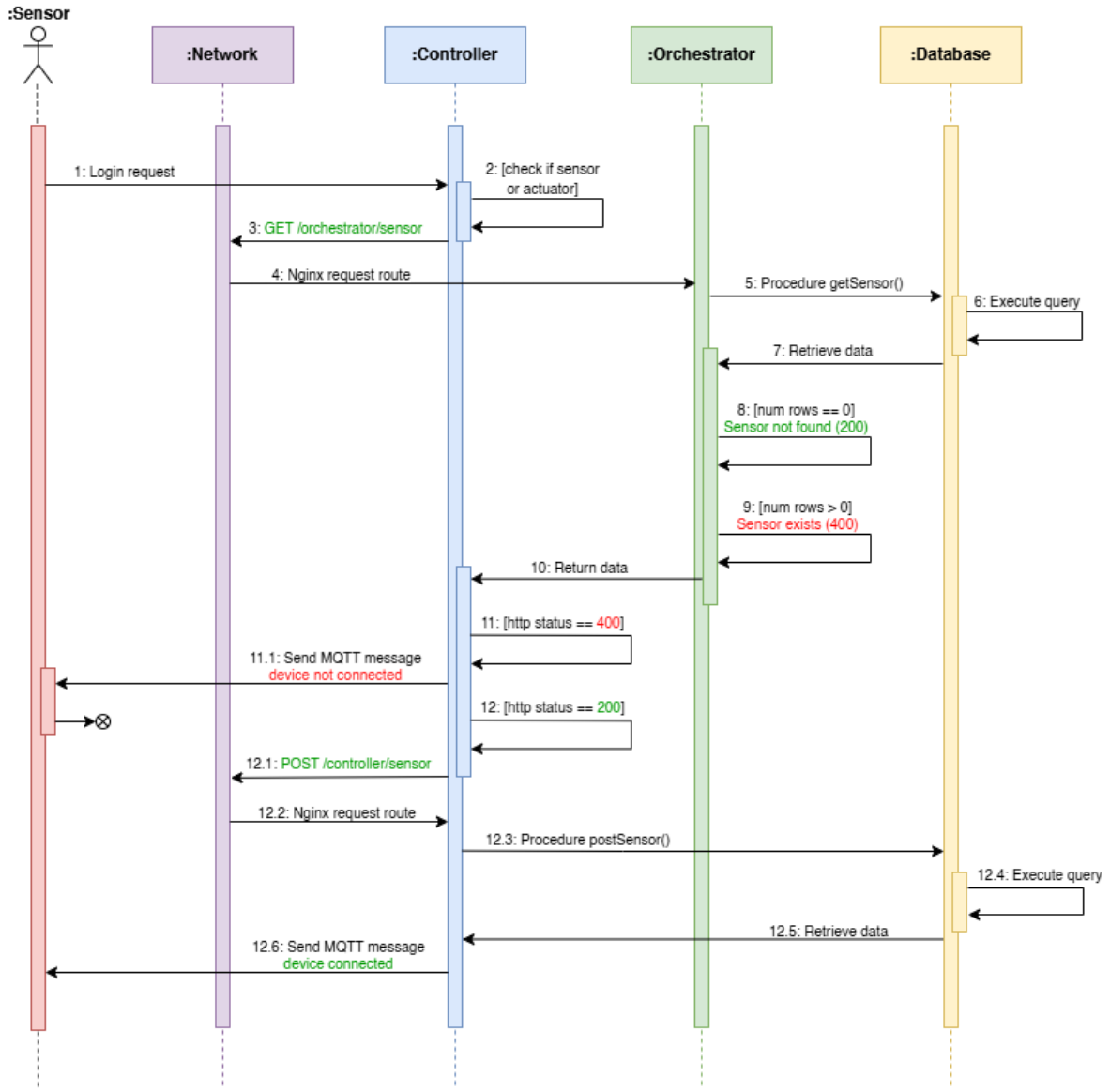


Figure 1.2: Sequence Diagram: Sensor Login

When a device connects to the system it makes an MQTT request to the controller. The controller checks the database to see if the device was already connected, if not, then it is connected to the system via a POST on the sensor. If the login is successful a positive http status value is returned. If the process was successful an MQTT message is returned from the controller to the device.

1.2.2 Event Triggered

In the case of an "Event Triggered" we can see the interactions on the system by the "Raspberry" actor.

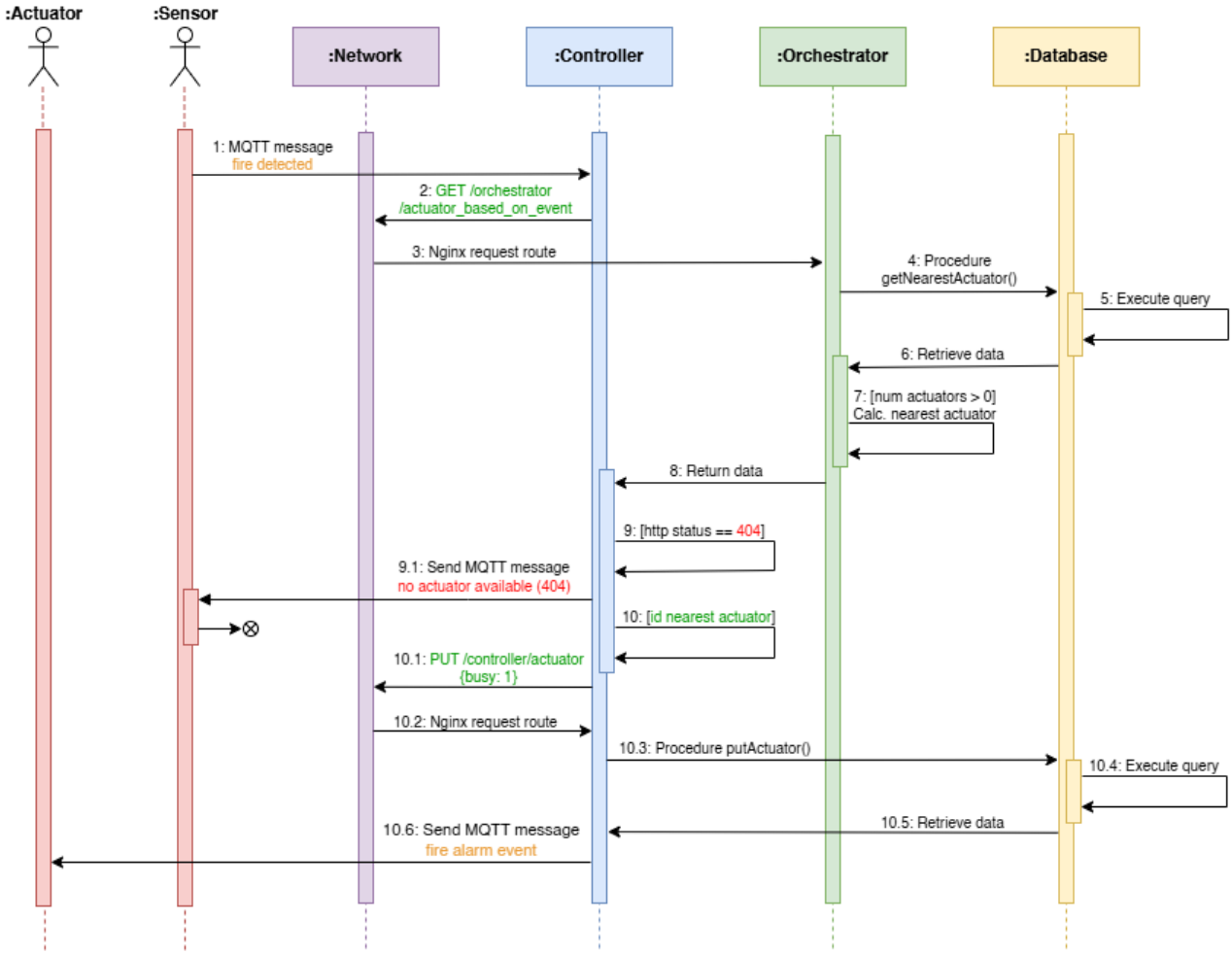


Figure 1.3: Sequence Diagram: Event Triggered

When a sensor, represented as a physical Raspberry, detects a fire a system notification signal is sent by MQTT to the controller, called EVENT

in which it passes as parameters the device identifier and event code E1 (Fire). The controller handles the request by going to make an API call to /actuator_based_on_event the routine that handles the calculation of the nearest actuator that handles that event. The SQL procedure set up to check the available actuators (busy = 0) in the vicinity with respect to the sensor is called. If there are multiple actuators available the closest one is chosen with respect to the sensor coordinates, otherwise if there are no actuators a http status 404 (not found) warning is returned. The chosen actuator is placed as busy (busy = 1) and the actuator is notified to take action.

Chapter 2

Communications Protocols

2.1 Flask

Flask is a microframework written in Python that allows you to create lightweight and scalable web applications.

Provides native support for REST API creation, simplifying the process with built-in features for managing HTTP requests and responses. It is designed to be easily expandible with third-party libraries and plugins; these include some extensions that provide additional features such as database integration, authentication, and token management.
Includes a built-in development server that makes it easy to test and debug APIs during development.
The configuration of Flask is very flexible, as it offers many solutions for every production need.

Example route creation with Flask

```
@app.route('/orchestrator/sensor', methods=['GET'])
def getSensor():
    return SensorAPI.get(request.args, request.remote_addr)

# Class Sensor API Routes
class SensorAPI:

    def get(data, ipAddress):
        ipAddressCheck = db.callProcedure('getIpAddressCheck', (
   ipAddress,))
```

```
10          print(f"Indirizzo {ipAddressCheck}")
11          if len(ipAddressCheck) > 0:
12              if all(key in data for key in ['id_device']):
13                  if Functions.validateType('str', data['id_device']):
14                      id_device = data['id_device']
15                      resultQuery = db.callProcedure('getSensor', (
    id_device,))
16                      return jsonify(resultQuery), 200
17                  else:
18                      return jsonify({"message": "id_device param must be
     string like S1"}), 400
19              else:
20                  resultQuery = db.callProcedure('getSensorAll', ())
21                  return jsonify(resultQuery), 200
22          else:
23              return jsonify({"message": "Ip Address not authorized"}),
    401
24
25
26      def post(data, ipAddress):
27          return jsonify({"message": "Method not implemented"}), 501
28
29      def put(data, ipAddress):
30          return jsonify({"message": "Method not implemented"}), 501
31
32      def delete(data, ipAddress):
33          return jsonify({"message": "Method not implemented"}), 501
```

2.2 API REST

APIs, an acronym for **Application Programming Interface**, are mechanisms that allow two applications, or devices, to connect to each other using a series of protocols. The application that sends the request is called client, the one that sends the response is called server.
APIs can work in four different ways depending on when and why they are created.

1. **API SOAP**: These APIs use the SOAP (Simple Object Access Protocol), in which the client and the server exchange in XML format. This format, however, is less flexible and was more widespread in the past.

2. **API RPC**: In the RPC (Remote Procedure Calls) APIs, the client completes a function, or procedure, on the server and the latter sends the output to the client.

3. **API WebSocket**: These APIs use JSON objects to transfer data and support two-way communication between the client and the server.
4. **API REST**: In these APIs the client sends the requests to the server as data. This data is used by the server to initiate internal functions and return the output data to the client.

Among the four types of APIs described above, REST APIs are the most widespread and flexible to date on the WEB. REST(**Representetional State Transfer**) is a software architecture that establishes the conditions of how an API should work.

The REST APIs follow the six REST design principles, also called architectural constraints:

1. **Uniform interface**
2. **Client-server decoupling**
3. **Stateless condition**
4. **Possibility of storing the cache**
5. **Tiered system architecture**
6. **Code on-demand**

REST APIs use HTTP requests to perform standard database functions, also called CRUD (Create, Read, Update, Delete) operations, within a resource.

In the REST API, CRUD operations are mapped to the appropriate HTTP methods:

- CREATE →**POST**: allows you to create a new resource.
- READ →**GET**: allows you to retrieve a resource.
- UPDATE →**PUT**: allows you to update a resource.
- DELETE →**DELETE**: allows you to delete a resource.

The client sends an HTTP request using a specific format that contains one of the four methods mentioned above and the URL, which in turn

contains the URI, in this case also called "endpoint", as it is the location where the API interacts with the client.

Once the request is received and validated, the host returns the target resource information. Usually, this information is sent in JSON format, which stands for JavaScript Object Notation, as it is a very light and easily readable format by humans.

It is possible, however, to use other formats including HTML, XLT, PHP, Python or plain text.

Client-side API call example

```
# Execute Request
def sendRequest(host, port, method, endpoint, data):

    # Handle request based on method passed
    def handleMethod(method, url, data):
        if method == 'GET':
            return requests.get(url, params=data)
        elif method == 'POST':
            return requests.post(url, json=data)
        elif method == 'PUT':
            return requests.put(url, json=data)
        elif method == 'DELETE':
            return requests.delete(url, params=data)
        else:
            return None

    # Send request
    try:
        url = f"http://{host}:{port}/{endpoint}"
        print(f">> [FLASK REQUEST]: {method} /{endpoint} with data: {
    data}")
        response = handleMethod(method, url, data)

        if response is not None:
            if response.status_code == 200:
                #print(json.loads(response.text))
                return json.loads(response.text)
            elif response.status_code == 404:
                print(f"[ERROR 404]: Resource /{endpoint} not found")
            else:
                print(f"[ERROR {response.status_code}]: {response.text}
    ")
        else:
            print(f"[ERROR 405]: Method {method} not allowed")

    except requests.exceptions.RequestException as e:
        print(f"Errore di connessione: {e}")
# Request example
FlaskClient.sendRequest(ServerProxy.FLASK_HOST, ServerProxy.FLASK_PORT,
    'GET', '/orchestrator/sensor', data)
```

2.3 MQTT

MQTT (Message Queueing Telemetry Transport) is a lightweight communication protocol, based on a publish/subscribe structure. It is designed for environments with limited resources, such as IoT (Internet of Things) devices, or for networks characterised by high latency or intermittent connection. The protocol uses TCP/IP as an underlying transport, ensuring reliability in the delivery of messages.

Its operation is based on three main elements:

- **Publisher**: send messages to specific topics.
- **Subscriber**: receives messages by subscribing to specific topics.
- **Broker**: is the intermediary between publisher and subscriber, manages the delivery and routeing of messages.

AWS (Amazon Web Services), a cloud computing platform that offers a wide range of scalable and flexible compute, storage, and networking services, was used to manage and scale the MQTT-based communication. , taking advantage of the advanced security features offered by the platform. In particular, AWS IoT Core guarantees the protection of data transmitted through MQTT thanks to TLS encryption and a robust certificate-based authentication and authorization system, making communication safe and reliable even in complex environments.

2.3.1 AWS IoT Core

AWS IoT Core is a managed service offered by Amazon Web Services that allows you to connect IoT devices to the cloud in a secure, scalable and simple way. It works as a central platform for receiving, processing and analysing data from connected devices, also allowing commands to be sent to them. Thanks to its integration with other AWS services, AWS IoT Core offers a powerful ecosystem for IoT applications that require real-time communication and advanced data management.

Among the main features there are:

- **Advanced security**: AWS IoT Core uses **TLS (Transport Layer Security)** to ensure encryption of communications and **authentication and authorisation** based on X.509 certificates, AWS IAM credentials or JSON Web Token (JWT) based tokens.
- **Message management**: With the MQTT-based messaging engine, AWS IoT Core can process millions of messages simultaneously, ensuring reliable communications between devices and cloud applications.
- **Scalability and monitoring**: The service is designed to scale dynamically based on the number of connected devices, providing monitoring tools such as AWS CloudWatch to keep track of metrics and performance.

The AWS IoT Core module was used in this project above all to guarantee an advanced (thanks to TLS) and more robust level of security. Each created object (thing) is linked to a certificate and an authorization policy.

Things

Things represent the physical devices you want to connect to AWS IoT. Each Thing can be a sensor, an embedded device or any other IoT object. The objects that have been created are:

- Raspberry: to connect all sensor and actuator devices to the MQTT broker. In fact, in the policy (figure 2.4), we can see that in IOT:CONNECT, the device identifier contains a path "Raspberry" followed by a "*" which allows dynamic client identifiers to access the sensor system: RaspberryS1, RaspberryS2, RaspberryA1, RaspberryA. This way, we have a valid policy for all devices connected to the MQTT broker;
- Controller: to connect the device controller to the MQTT broker. In the design of this system, only one controller is foreseen, so the policy does not allow the connection of multiple controllers with the same client ID.

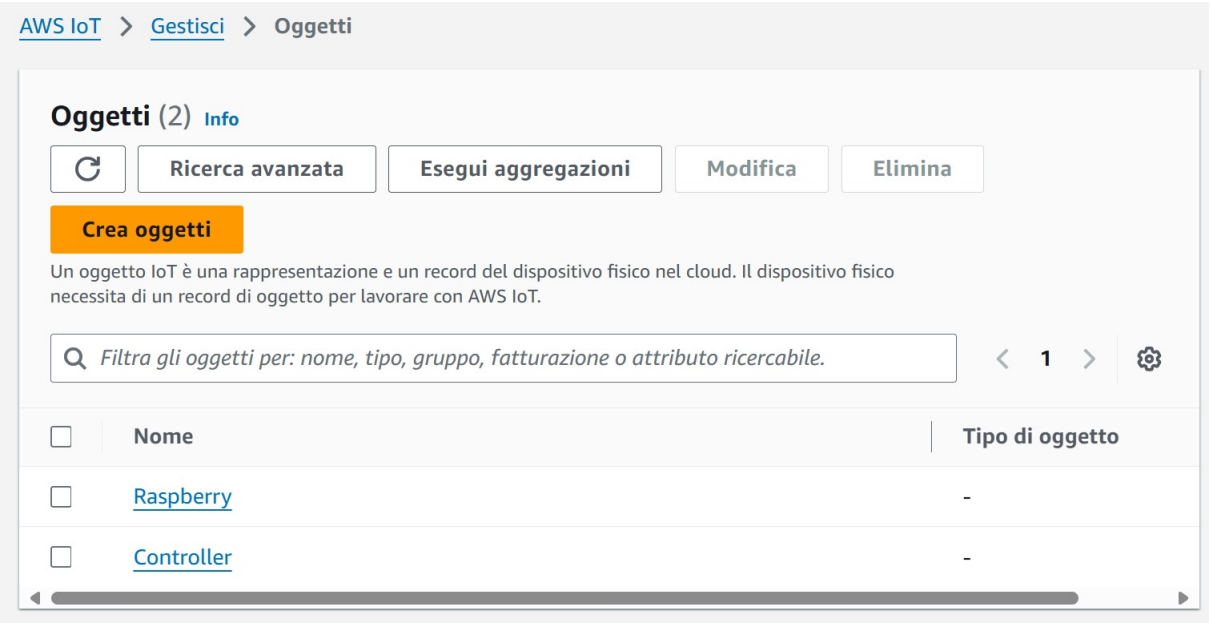


Figure 2.1: AWS Things

Certficates and policies

- Certificates: certificates are used to securely authenticate devices when communicating with AWS IoT Core. Each device requires an X.509 certificate which serves as a unique identifier to establish encrypted connections (TLS).
- Policies: policies define the permissions associated with a certificate. They determine what a device can do, such as publishing or subscribing to messages on specific MQTT topics, or accessing certain AWS APIs.

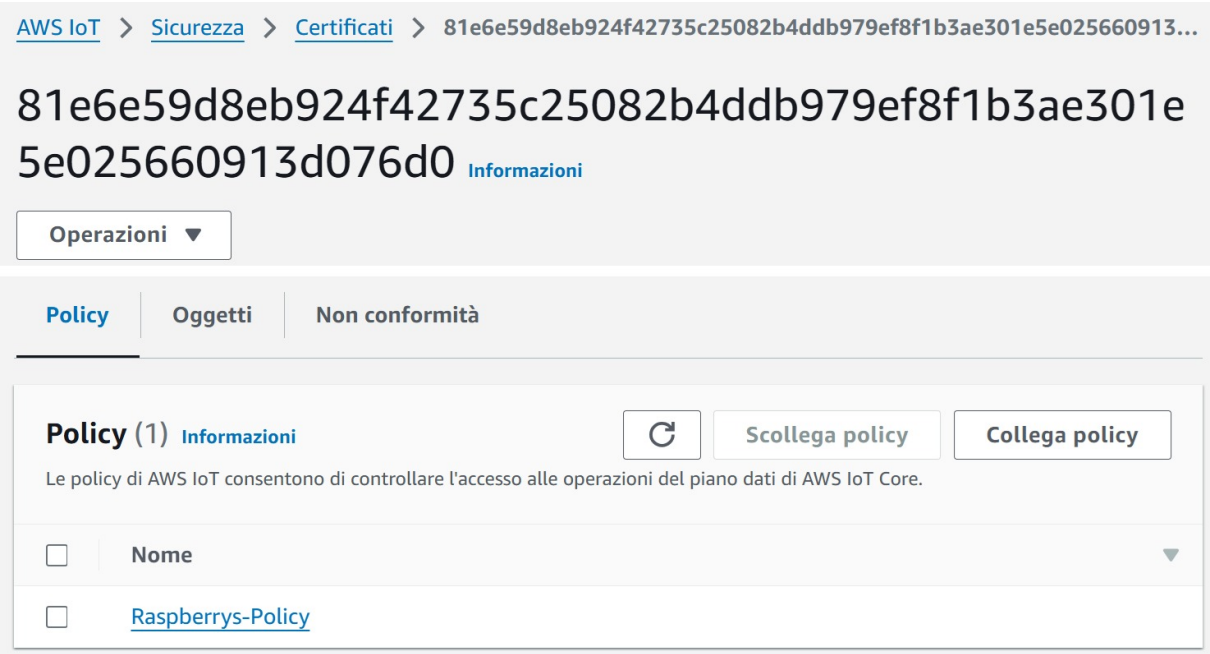


Figure 2.2: AWS Certificate

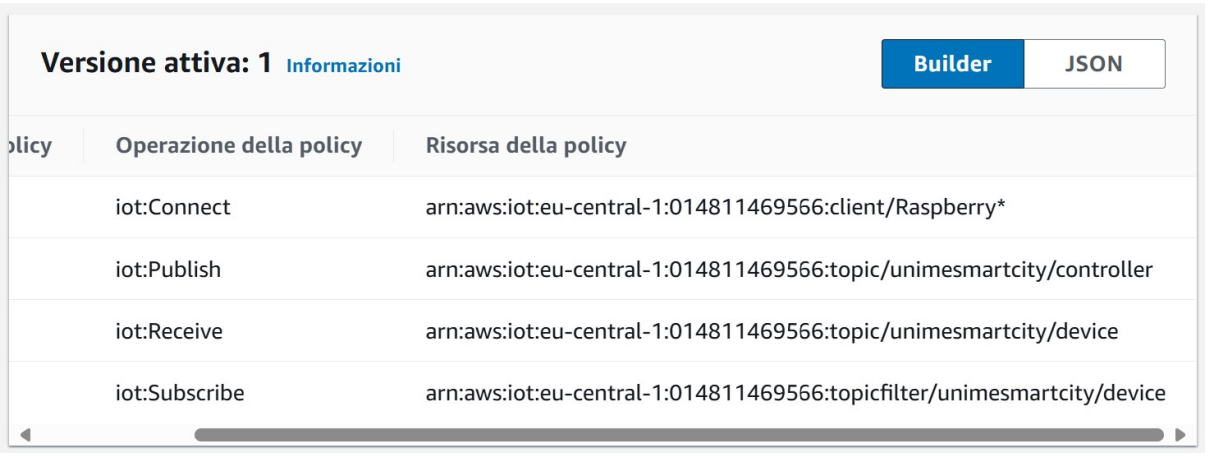


Figure 2.3: AWS Policy

Keys generation

When creating a certificate, a public key and a private key are automatically generated. These keys are essential for asymmetric encryption and are used to establish secure connections between devices and AWS IoT. The private key is stored only by the device, while the public key is part of the certificate registered on AWS.

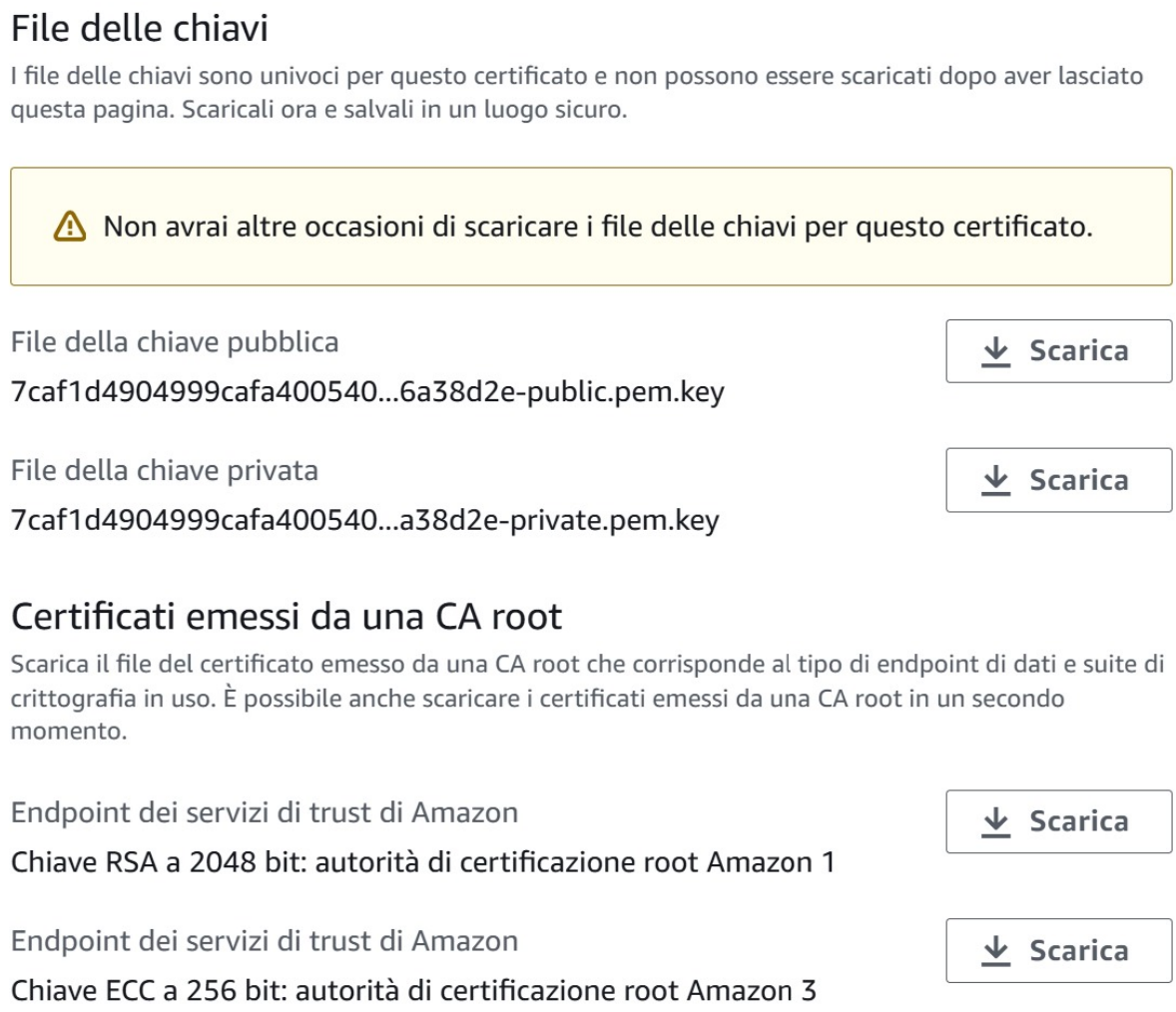


Figure 2.4: AWS Key Generation

2.3.2 Broker subscription

Class AWSConfig.py

```
# Client MQTT configuration

import random

class AWSConfig:
    client_id = "RaspberryS" + str(random.randint(10000, 99999))
    endpoint = "apv9omwb522qd-ats.iot.eu-central-1.amazonaws.com"
    root_ca = "AWS-keys/AWS-RootCA-RSA.pem"
    private_key = "AWS-keys/AWS-raspberry-private.pem.key"
    certificate = "AWS-keys/AWS-raspberry-certificate.pem.crt"
```

Class MQTTbroker.py

```
from datetime import datetime
import json

```

```

# MQTT Config
mqttClient = None
mqttPubTopic = None
QoS = None


class MQTTBroker:
    def __init__(self, mqttClient, mqttPubTopic, QoS):
        self.mqttClient = mqttClient
        self.mqttPubTopic = mqttPubTopic
        self.QoS = QoS

    def sendMessage(self, messageType, messageParams=None):
        currentTime = datetime.now()
        formattedTime = currentTime.strftime("%Y-%m-%d %H:%M:%S")
        message = {
            'time': formattedTime,
            'message_type': messageType
        }
        if messageParams is not None:
            message.update(messageParams)
        self.mqttClient.publish(self.mqttPubTopic, json.dumps(message), self.QoS)
        print(f"[PUB]: {message}")
```

Example client.py

```
from AWSIoTPythonSDK.MQTTLib import AWSIoTMQTTClient
from classes.MQTTBroker import MQTTBroker
from classes.AWSConfig import *

def readMessages(client, userdata, message):
    message = json.loads(message.payload)
    # You can use message here

# Device config
DEVICE_ID = 'S' + str(random.randint(10000, 99999))
DEVICE_NAME = 'Fire Camera'
DEVICE_TYPE = 'SENSOR'
DEVICE_AREA = 'A2'
DEVICE_EVENT = 'E1'

# Civico Via, Citt , Nazione
ADDRESS = "228 Via Cesare Battisti, Messina, Italia"
COORDS = Functions.getCoords(ADDRESS)
DEVICE_COORD_LAT = COORDS[0]
DEVICE_COORD_LON = COORDS[1]

# MQTT config
mqttBroker = None
MQTT_TOPIC_PUB = "unimesmartcity/controller"
MQTT_TOPIC_SUB = "unimesmartcity/device"
QoS = 0

try:
    # AWS MQTT client and connection
```

```
30      my_mqtt_client = AWSIoTMQTTClient(AWSConfig.client_id)
31      my_mqtt_client.configureEndpoint(AWSConfig.endpoint, 8883)
32      my_mqtt_client.configureCredentials(AWSConfig.root_ca, AWSConfig.
        private_key, AWSConfig.certificate)
33      my_mqtt_client.connect()
34      my_mqtt_client.subscribe(MQTT_TOPIC_SUB, QoS, readMessages)
35
36      # Instance this object to use methods of class MQTTBroker to send
        messages
37      mqttBroker = MQTTBroker(my_mqtt_client, MQTT_TOPIC_PUB, QoS)
38
39  message = {
40      'id_sensor': 'S1',
41      'message': 'message here'
42  }
43  mqttBroker.sendMessage('EVENT', message)
```

Chapter 3

Docker

Docker is an open-source platform that enables developers to automate the deployment, scaling, and management of applications within lightweight, portable containers. These containers encapsulate an application and its dependencies, allowing it to run consistently across various computing environments, whether on a developer's local machine, in a data center, or in the cloud.

Benefits of Docker:

- Isolation and Consistency
- Scalability
- Microservices Architecture
- Resource Efficiency
- Cross-Platform Compatibility

Docker serves as a powerful tool in the context of smart cities, particularly for distributed fire detection systems. By providing a robust framework for application deployment and management, it enables developers to create scalable, efficient, and easily maintainable systems that can quickly respond to fire incidents, enhancing public safety and operational efficiency.

3.1 Network Container

The container network is the fulcrux of the communication system of the distributed system. "Nginx" is installed on the container network. Nginx is a high-performance web server that also works as a reverse proxy, load balancer and HTTP cache. In particular, as a reverse proxy, it is useful for forwarding received requests to different servers or containers within a distributed infrastructure.

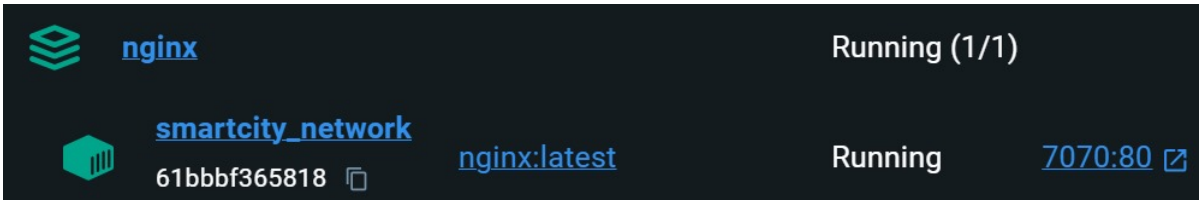


Figure 3.1: Nginx container

In the nginx.conf file we find two routes:

- **/orchestrator**: forwards requests to an Orchestrator container;
- **/controller**: forwards requests to a container Controller.

This is a typical reverse proxy scenario where Nginx acts as an intermediary.

Nginx.conf code

```
http {
    server {
        listen 80;

        # Route per Orchestrator
        location /orchestrator {
            proxy_pass http://10.24.105.214:5050;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for
    ;
            proxy_set_header X-Forwarded-Proto $scheme;
        }

        # Route per Controller
        location /controller {
            proxy_pass http://10.24.104.215:6060;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for
    ;
```

```
            proxy_set_header X-Forwarded-Proto $scheme;
        }
    }
}
```

3.2 Orchestrator Container

The orchestrator in a smart city system is a central component that manages, coordinates and supervises the entire ecosystem of devices and services distributed in the city network, including sensors, actuators, controllers and databases. It is responsible for ensuring that the different parts of the system communicate properly with each other, collecting and analysing data from various devices, and making decisions or sending real-time commands to optimise the efficiency and security of operations.

Figure 3.2: orchestrator container

Orchestrator code

```python
# Server Flask ORCHESTRATOR

from classes.Routes import *
from classes.FlaskConfig import *
from flask import Flask, request, redirect, url_for, jsonify

app = Flask(__name__)


# Sensor API Routing
@app.route('/orchestrator/sensor', methods=['GET'])
def getSensor():
    return SensorAPI.get(request.args, request.remote_addr)

# Sensor API Routing
@app.route('/orchestrator/actuator', methods=['GET'])
def getActuator():
    return ActuatorAPI.get(request.args, request.remote_addr)

@app.route('/orchestrator/actuator_based_on_event', methods=['GET'])
def getActuatorBasedOnEvent():
    return ActuatorAPI.getBasedOnEvent(request.args, request.
   remote_addr)
```

```
23
24
25 # Start Flask Server
26 if __name__ == '__main__':
27     app.run(host=ThisServer.FLASK_HOST, port=ThisServer.FLASK_PORT,
   debug=True)
```

3.3 Controller Container

The device controller is a critical component in a smart city system that acts as an intermediary between sensors, actuators and the central orchestrator. Its main purpose is to coordinate and manage connected devices (such as Raspberry Pi, environmental sensors or actuators) and ensure that they communicate correctly with the orchestrator, facilitating the sending and reception of data and commands.

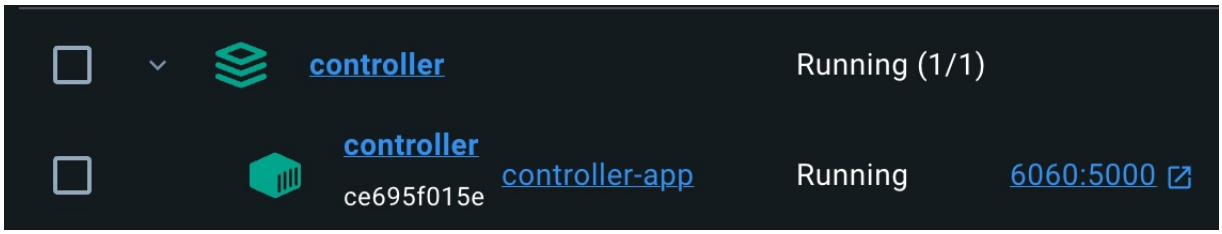


Figure 3.3: container controller

Controller code

```
1  # Server Flask CONTROLLER
2
3  from classes.Routes import *
4  from classes.FlaskConfig import *
5  from flask import Flask, request
6
7  app = Flask(__name__)
8
9
10 # Sensor API Routing
11 @app.route('/controller/sensor', methods=['POST'])
12 def postSensor():
13     return SensorAPI.post(request.get_json(),request.remote_addr)
14 @app.route('/controller/sensor', methods=['PUT'])
15 def putSensor():
16     return SensorAPI.put(request.get_json(),request.remote_addr)
17 @app.route('/controller/sensor', methods=['DELETE'])
18 def deleteSensor():
19     return SensorAPI.delete(request.args,request.remote_addr)
20
21
22 # Sensor API Routing
```

```
23 @app.route('/controller/actuator', methods=['POST'])
24 def postActuator():
25     return ActuatorAPI.post(request.get_json(),request.remote_addr)
26 @app.route('/controller/actuator', methods=['PUT'])
27 def putActuator():
28     return ActuatorAPI.put(request.get_json(),request.remote_addr)
29 @app.route('/controller/actuator', methods=['DELETE'])
30 def deleteActuator():
31     return ActuatorAPI.delete(request.args,request.remote_addr)
32
33 # Start Flask Server
34 if __name__ == '__main__':
35     app.run(host=ThisServer.FLASK_HOST, port=ThisServer.FLASK_PORT,
    debug=True)
```

3.4 Database Container

The Docker container "smartcity_db" is the container that stores and preserves the information from the sensors and actuators of the system.

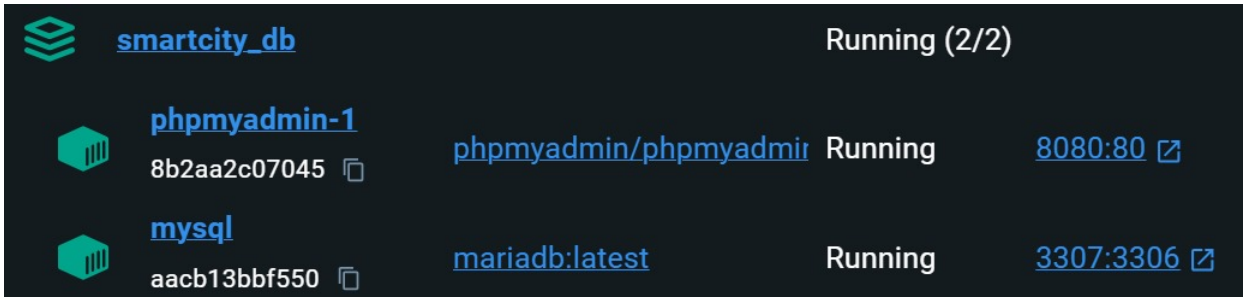


Figure 3.4: database container

The tables that make up the database are:

- **Table: area**
  The "area" table contains the information relating to the geographical areas (A1, A2, ...) that make up the system.

id_area	name
A1	Messina Nord
A2	Messina Centro
A3	Messina Sud

Figure 3.5: Areas list

- **Table: access**
  The "access" table is used to store IP addresses that have permission to make requests to the server. It is used to implement an access control mechanism based on IP addresses.

id	ip_address
1	10.24.105.214
2	10.24.104.215
3	192.168.65.1

Figure 3.6: Access table

- **Table: device_event**
  The "device_event" table associates a device with one or more specific events that it can handle.

id ▲ 1	id_device	id_event
1	S34567	E1
2	S12345	E1
3	A88145	E2
4	S46478	E1

Figure 3.7: Event handled by device list

- **Table: event**
  The "event" table contains the event labels (E1, E2, ...) that can be managed by the system.

id_event	name
E1	Fire
E2	Indoor security
E3	Outdoor security
E4	Emergency first aid

Figure 3.8: Events list

- **Table: mapactuator**
  The "mapactuator" table contains the map of where the actuators are geographically located within the system. In addition, an actuator contains a binary "busy" label with values [0,1] to indicate to the system whether the actuator is currently busy handling an event.

id_device	name	area	coord_lat	coord_lon	busy
A46383	Fire Station	A2	20.12	10.47	0
A68257	Fire Station	A2	23.76	79.27	0
A88145	Police Station	A3	45.88	21.47	1

Figure 3.9: Map of actuators

- **Table: mapsensor**
  The "mapsensor" table - similar to "mapactuator" - contains the map of where the sensors are geographically located within the system. In addition, a sensor contains a "last_alarm" label which, if different from null, indicates the moment in which it sent the alarm.

id_device	name	area	coord_lat	coord_lon	last_alarm
S12345	Fire Camera	A1	1.999	2.4565	*NULL*
S34567	Fire Camera	A1	1.7	2.7	2024-11-05 12:17:21
S46478	Fire Camera	A2	20.1	10	*NULL*

Figure 3.10: Map of sensors

3.4.1 Stored Procedure

Stored procedures are blocks of predefined SQL code stored within a database, which can be executed to perform queries. Among the main advantages of stored procedures is increased performance, as the code is compiled and optimised from the database, reducing network load and improving execution times. They also provide greater security, since the SQL code remains server-side and users can perform the procedures without the need to directly access the data.

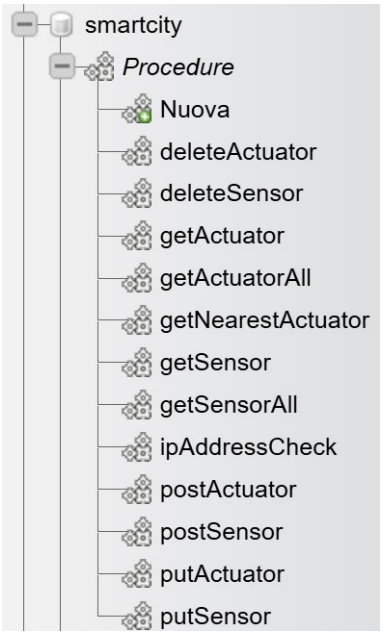


Figure 3.11: Stored procedures list

3.4.2 Diagramma ER

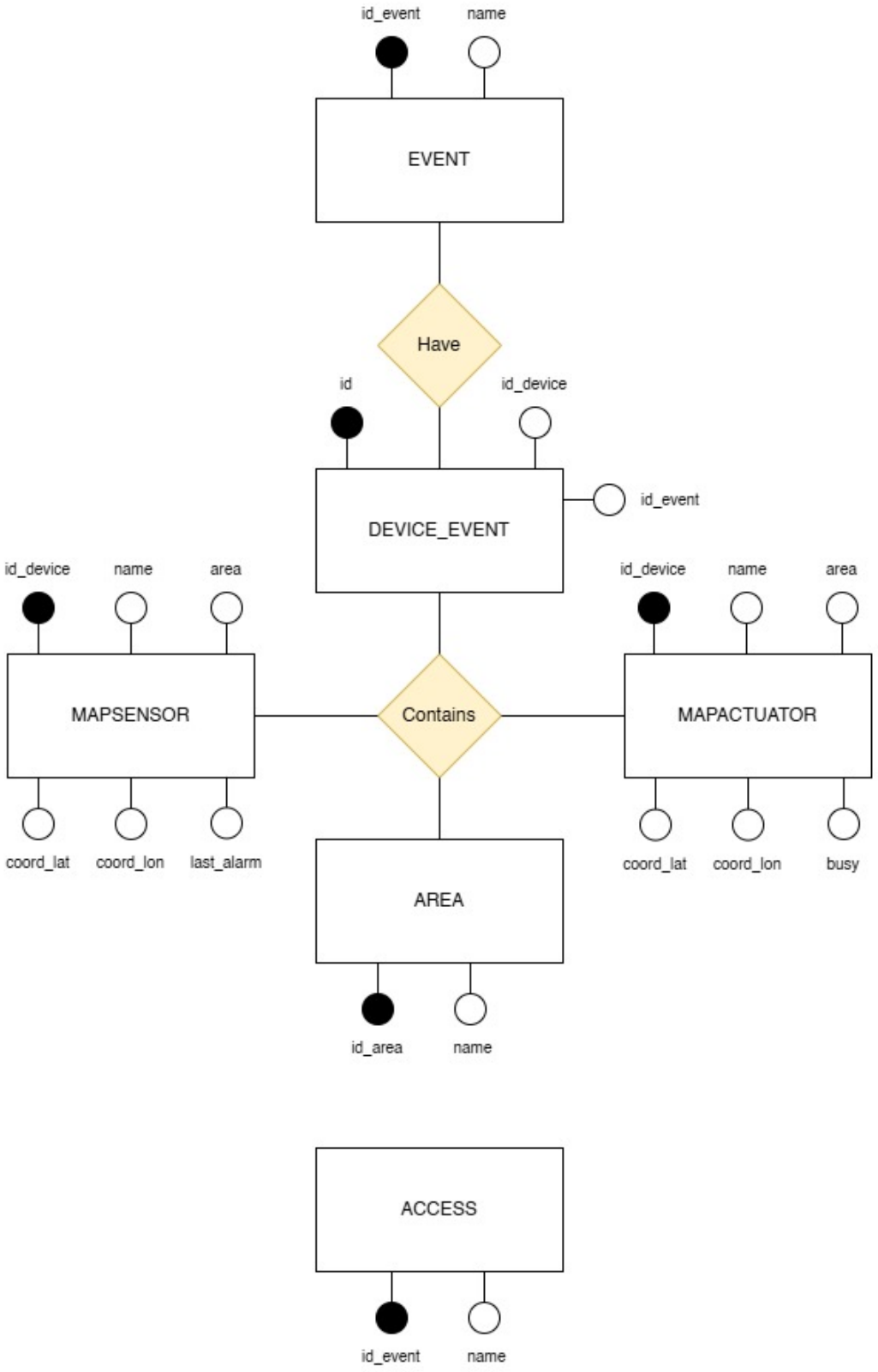
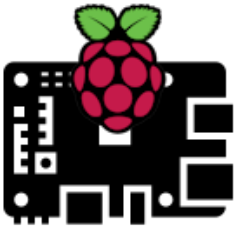


Figure 3.12: ER Diagram

Chapter 4

Raspberry Pi 3



The Raspberry Pi 3B+ is a low-cost and highly versatile mini-computer, equipped with an ARM quad-core processor, Wi-Fi, Bluetooth and various connection ports, which make it ideal for a wide range of applications in the field of the Internet of Things (IoT) and smart cities. Thanks to its sufficient computing power to execute machine learning algorithms, it can be used for solutions that require intelligent local processing, avoiding the need to send large amounts of data to remote servers for analysis.

In the context of our fire detection project in a smart city, the Raspberry Pi 3B+ represents an economical and easily implementable solution for continuous environmental monitoring. Integrated with a Pi Camera, the Raspberry Pi captures real-time images from the surrounding environment. These images are processed locally using a machine learning algorithm SVM (Support Vector Machine) trained to detect the presence of flames or smoke, the first signs of a fire.

Thanks to this local processing capacity, the Raspberry Pi 3B+ can autonomously detect a potential fire, reducing critical response times. When the SVM algorithm detects a dangerous situation, the Raspberry Pi can immediately send alarms and notifications to the central system, facili-

tating a timely response from the emergency services. This architecture makes the system particularly useful in smart cities, where a distributed and intelligent infrastructure is essential to ensure public safety.

4.1 Pi Camera

The Pi Camera is a camera designed specifically for use with Raspberry Pi single-board computers. It is a compact module that connects directly to the Raspberry Pi through the CSI (Camera Serial Interface) connecter, allowing you to capture images and videos in high quality. Compatible with models such as Raspberry Pi 4, 3, Zero and others, it uses a ribbon cable to connect to the CSI connecter present on all modern boards.

Newer versions, such as Pi Camera Module 3, offer high resolutions, autofocus and support for HDR, while previous models, such as Pi Camera Module 2 (8 megapixels) and Camera Module 1 (5 megapixels), maintain good image quality. There are two main types of Pi Camera: the standard version, suitable for capturing images and videos in normal light conditions, and the NoIR (No InfraRed) version, ideal for night vision thanks to the absence of the infrared filter and the possibility of working with IR illuminators.

Software support for the Pi Camera is well integrated into Raspberry Pi operating systems, such as Raspberry Pi OS, and simplifies its use through Python libraries (such as Picamera or OpenCV) or command line tools.

4.2 Raspberry configuration

Installation of the necessary libraries

- pip install opencv-python==4.10.0.84
- pip install opencv-python-headless==4.10.0.84
- pip install joblib==1.4.2
- pip install picamera2==0.3.12
- pip install paho-mqtt==2.1.0
- pip install requests==2.25.1

- pip install scikit-learn==1.1.1
- pip install scipy==1.13.1
- pip install geopy==2.4.1
- pip install AWSIoTPythonSDK==1.5.4
- pip install colorama==0.4.6

4.3 How sensor works

Every minute, the Pi Camera acquires an image of the surrounding environment. The image is then processed locally on the Raspberry Pi through a pre-trained SVM algorithm, whose weights are already loaded into the device. The algorithm analyses the image and returns a binary value:

- 0: indicates that no fire has been detected;
- 1: reports that a potential fire has been detected.

If a fire is detected (1), the system waits for a certain number of minimum attempts to avoid false positives, and only then sends an alarm via the MQTT protocol to the device controller.

Raspberry sensor steps

- the Pi Camera captures a .png image
- process the image via SVM
- the SVM algorithm returns a binary value [0, 1]
- if the value is "1" count the number of detections
- if the counter reaches the minimum number of detections, it sends a message to the MQTT broker
- delete the previously saved photo
- set the system to pause

If a fire is detected during image processing, via SVM, an alert is sent to the system to notify the fire event. To avoid notifying false positives to the system, a counter system has been adopted in which a counter is increased for each fire detected; when the latter reaches the minimum number desired to trigger the fire event, it will send a notification to the system.

How system works after fire detection

From the photo below you can see:

- **MQTT broker connection**: there is confirmation that a connection has been established with an MQTT broker;
- **Controller online**: the device before connecting to the system must ensure that the controller is online. So it sends a message of type CHECK_CONTROLLER. This will be executed asynchronously every 30 seconds even after login;
- **Device connection**: the device sends a message to connect to the CONNECTION type system;
- **Detection of a fire event**: if this number of detections reaches the minimum limit of detections needed, an alert will be sent to the system of type EVENT.

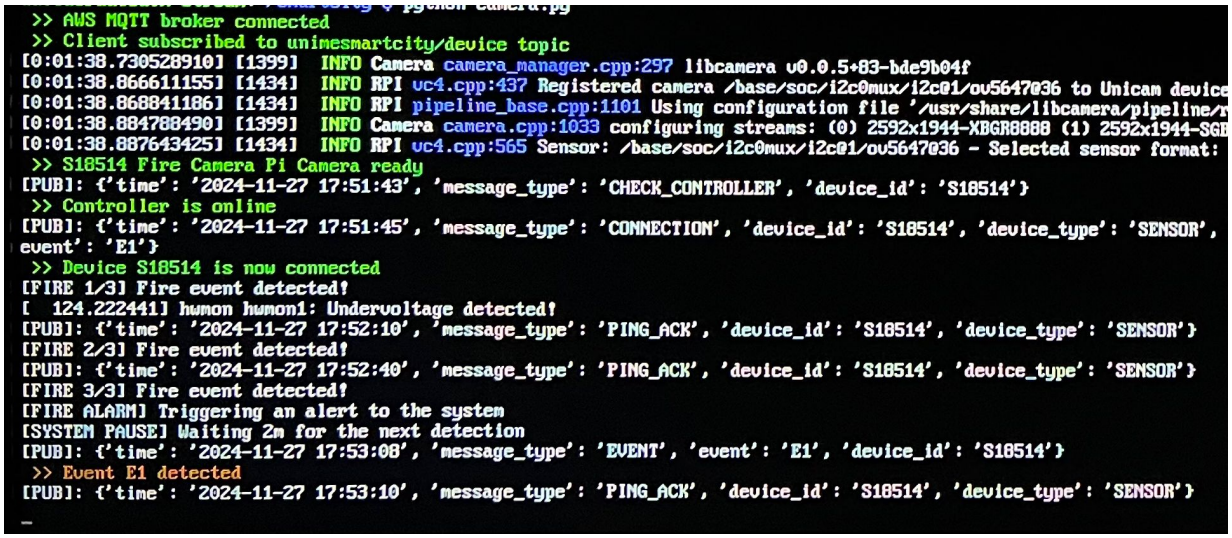


Figure 4.1: Raspberry console

Figure 4.2 shows the logs of the HTTP requests exchanged between the controller and the orchestrator. Specifically: the controller manages and sends data related to sensors and actuators; the orchestrator provides the data required to the controller, including details about specific devices or events.

Details of the requests:

- GET /orchestrator/actuator?id_device=A40861: request to obtain information about an actuator with ID A40861.
- POST /controller/actuator: sending information about an actuator to the controller.
- GET /orchestrator/sensor?id_device=S18514 : eequest to obtain data from a sensor with ID S18514.
- POST /controller/sensor : sending data related to a sensor to the controller.
- PUT /controller/sensor: update of data on the sensor by the controller.
- GET /orchestrator/actuator_based_event: request to obtain information about the actuator, based on a specific event (e.g. id _device=S18514, event=E1, with last alarm recorded at a specific date and time).
- PUT /controller/actuator : update data on an actuator by the controller.

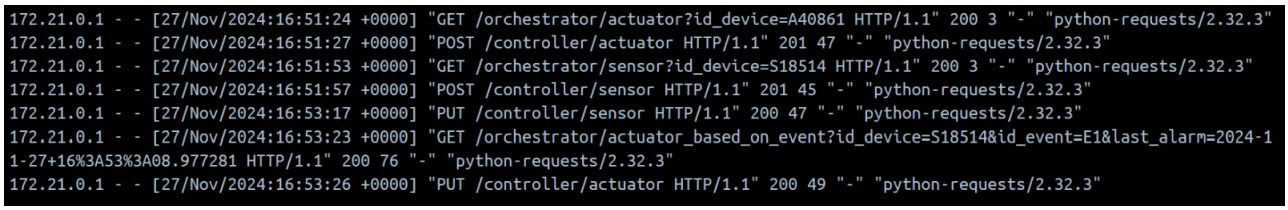


Figure 4.2: Docker console

The actuator connects to the AWS broker, checks the availability of the controller (online controller) and connects to the system. Once connected, the actuator receives the verification messages from the controller

(CHECK CONTROLLER) and responds by sending the confirmation messages (PING ACK).

If a sensor detects an event, for example "Event E1 triggered", it is possible to visualise in the log that it is a fire event, indicating the specific coordinates provided in the image.

```
 >> AWS MQTT broker connected
 >> Client subscribed to unimesmartcity/device topic
[PUB]: {'time': '2024-11-27 17:51:18', 'message_type': 'CHECK_CONTROLLER', 'device_id': 'A40861'}
 >> Controller is online
[PUB]: {'time': '2024-11-27 17:51:20', 'message_type': 'CONNECTION', 'device_id': 'A40861', 'device_type': 'ACTUATOR',
re Camera', 'device_area': 'A3', 'device_coord_lat': 15.5496888, 'device_coord_lon': 38.1835015, 'device_event': 'E1'}
 >> Device A40861 is now connected
[PUB]: {'time': '2024-11-27 17:51:35', 'message_type': 'CHECK_CONTROLLER', 'device_id': 'A40861'}
[PUB]: {'time': '2024-11-27 17:51:45', 'message_type': 'PING_ACK', 'device_id': 'A40861', 'device_type': 'ACTUATOR'}
[PUB]: {'time': '2024-11-27 17:51:51', 'message_type': 'CHECK_CONTROLLER', 'device_id': 'A40861'}
[PUB]: {'time': '2024-11-27 17:52:07', 'message_type': 'CHECK_CONTROLLER', 'device_id': 'A40861'}
[PUB]: {'time': '2024-11-27 17:52:15', 'message_type': 'PING_ACK', 'device_id': 'A40861', 'device_type': 'ACTUATOR'}
[PUB]: {'time': '2024-11-27 17:52:23', 'message_type': 'CHECK_CONTROLLER', 'device_id': 'A40861'}
[PUB]: {'time': '2024-11-27 17:52:39', 'message_type': 'CHECK_CONTROLLER', 'device_id': 'A40861'}
[PUB]: {'time': '2024-11-27 17:52:45', 'message_type': 'PING_ACK', 'device_id': 'A40861', 'device_type': 'ACTUATOR'}
[PUB]: {'time': '2024-11-27 17:52:55', 'message_type': 'CHECK_CONTROLLER', 'device_id': 'A40861'}
[PUB]: {'time': '2024-11-27 17:53:11', 'message_type': 'CHECK_CONTROLLER', 'device_id': 'A40861'}
[PUB]: {'time': '2024-11-27 17:53:15', 'message_type': 'PING_ACK', 'device_id': 'A40861', 'device_type': 'ACTUATOR'}
 >> Event E1 triggered from [lat: 15.5497, lon: 38.1835]
```

Figure 4.3: Actuator console

4.4 Raspberry code

```
# This code runs only into Raspberry

import time
import os
import cv2
import numpy as np
import random
import threading
from picamera2 import Picamera2
from sklearn.svm import SVC
from sklearn.preprocessing import StandardScaler
from joblib import load
from AWSIoTPythonSDK.MQTTLib import AWSIoTMQTTClient
from classes.MQTTBroker import MQTTBroker
from classes.Functions import *
from classes.AWSConfig import *


# Device config
DEVICE_ID = 'S' + str(random.randint(10000, 99999))
DEVICE_NAME = 'Fire Camera'
DEVICE_TYPE = 'SENSOR'
DEVICE_AREA = 'A2'
DEVICE_EVENT = 'E1'

# Civico Via, Citt , Nazione
ADDRESS = "228 Via Cesare Battisti, Messina, Italia"
COORDS = Functions.getCoords(ADDRESS)
DEVICE_COORD_LAT = COORDS[0]
```

```
DEVICE_COORD_LON = COORDS[1]

# MQTT config
mqttBroker = None
MQTT_TOPIC_PUB = "unimesmartcity/controller"
MQTT_TOPIC_SUB = "unimesmartcity/device"
QoS = 0

# Camera sensor config
image_name_png = "photo.png"
image_size = [2592, 1944]
min_detection_triggers = 3
system_pause_min = 2
sleep_time_normal_sec = 30

# Global const
SLEEP_LOOP_SEC = 1
MAX_TIMEOUT_SEC = 15
MIN_NEXT_CHECK_CONNECTION_SEC = 15

# Global vars
busy = False
deviceIsConnected = False
controllerIsConnected = False
lastStatusController = False
counterTimeout = 0
counterCheckConnection = 0
checkForConnection = True
connectionRequested = False


def checkTimeout():
    global MAX_TIMEOUT_SEC, checkForConnection, counterTimeout, controllerIsConnected, deviceIsConnected, lastStatusController, connectionRequested
    counterTimeout += 1
    if checkForConnection:
        if counterTimeout > MAX_TIMEOUT_SEC:
            controllerIsConnected = False
            lastStatusController = False
            checkForConnection = True
            connectionRequested = False
            counterTimeout = 0
            Functions.printLine("RED", ">> Device connection timeout, retrying to connect")
    else:
        if controllerIsConnected and deviceIsConnected:
            checkForConnection = False
            counterTimeout = 0


# Fire detectiong function
def predict_fire(image_path, model, scaler):
    img = cv2.imread(image_path)
    img = cv2.resize(img, (150, 150))
    img = img.flatten()
    img_normalized = scaler.transform([img])
    prediction = model.predict(img_normalized)
```

```
85      return prediction[0]
86
87
88  def readMessages(client, userdata, message):
89      global mqttBroker, controllerIsConnected, deviceIsConnected,
        lastStatusController
90      message = json.loads(message.payload)
91
92      try:
93          if 'message_type' in message:
94
95              # -- If device receive an ACK from the controller
96              if message['message_type'] == 'CONTROLLER_ACK':
97                  if message['device_id'] == DEVICE_ID:
98                      controllerIsConnected = True
99                      if not lastStatusController:
100                         Functions.printLine("GREEN", ">> Controller is
        online")
101                     lastStatusController = True
102
103             # -- If device receive a confirm of connection from the
        controller
104             if message['message_type'] == 'CONNECTION_OK':
105                 if message['device_id'] == DEVICE_ID:
106                     deviceIsConnected = True
107                     Functions.printLine("GREEN", f">> Device {DEVICE_ID
        } is now connected")
108
109             # -- If device receive receive a PING from the controller
        then send an ACK of my presence
110             if message['message_type'] == 'PING':
111                 if controllerIsConnected and deviceIsConnected:
112                     message = {
113                         'device_id': DEVICE_ID,
114                         'device_type': DEVICE_TYPE
115                     }
116                     mqttBroker.sendMessage('PING_ACK', message)
117
118         else:
119             Functions.printLine("RED", ">> Missing 'message_type'
        attribute into the MQTT body message")
120
121     except KeyError as e:
122         pass
123
124
125 try:
126     # AWS MQTT client and connection
127     my_mqtt_client = AWSIoTMQTTClient(AWSConfig.client_id)
128     my_mqtt_client.configureEndpoint(AWSConfig.endpoint, 8883)
129     my_mqtt_client.configureCredentials(AWSConfig.root_ca, AWSConfig.
        private_key, AWSConfig.certificate)
130
131     my_mqtt_client.connect()
132     Functions.printLine("GREEN", ">> AWS MQTT broker connected")
133
134     my_mqtt_client.subscribe(MQTT_TOPIC_SUB, QoS, readMessages)
135     Functions.printLine("GREEN", f">> Client subscribed to {
```

```
MQTT_TOPIC_SUB} topic")

# Instance this object to use methods of class MQTTBroker to send messages
mqttBroker = MQTTBroker(my_mqtt_client, MQTT_TOPIC_PUB, QoS)


# Trained SVM model and Scaler
model_file = 'model/svm_model.joblib'
svm_model = load(model_file)
scaler_file = 'model/scaler.joblib'
scaler = load(scaler_file)

# Initializing camera
picam2 = Picamera2()
config = picam2.create_preview_configuration({"size":(image_size[0], image_size[1])})
picam2.align_configuration(config)
picam2.configure(config)
picam2.start()

# Variables
n_detections = 0
alarm_triggered = False
sleep_time = sleep_time_normal_sec

Functions.printLine("GREEN", f">> {DEVICE_ID} {DEVICE_NAME} Pi Camera ready")



while True:
    if checkForConnection:
        counterCheckConnection += (SLEEP_LOOP_SEC + 1)

        # Step 1 - Check if the controller is online
        if (not controllerIsConnected and counterTimeout == 0) or (counterCheckConnection >= MIN_NEXT_CHECK_CONNECTION_SEC):
            counterCheckConnection = 0
            controllerIsConnected = False
            message = {
                'device_id': DEVICE_ID
            }
            mqttBroker.sendMessage('CHECK_CONTROLLER', message)
        # Step 2 - Trying to join the system
        elif controllerIsConnected and not deviceIsConnected:
            if not connectionRequested:
                message = {
                    'device_id': DEVICE_ID,
                    'device_type': DEVICE_TYPE,
                    'device_name': DEVICE_NAME,
                    'device_area': DEVICE_AREA,
                    'device_coord_lat': DEVICE_COORD_LAT,
                    'device_coord_lon': DEVICE_COORD_LON,
                    'device_event': DEVICE_EVENT
                }
                mqttBroker.sendMessage('CONNECTION', message)
                connectionRequested = True
```

```
189             checkForConnection = False
190         else:
191             # Main code - Detecting events
192             if controllerIsConnected and deviceIsConnected and not busy
    :
193                 # Shot a photo and save the .png
194                 picam2.capture_file(image_name_png)
195                 prediction = predict_fire(image_name_png, svm_model,
    scaler)
196 
197                 if alarm_triggered == False:
198                     if prediction == 1:
199                         n_detections += 1
200                         print(f"[FIRE {n_detections}/{
    min_detection_triggers}] Fire event detected!")
201                         if n_detections >= min_detection_triggers:
202                             print(f"[FIRE ALARM] Triggering an alert to
     the system")
203                             print(f"[SYSTEM PAUSE] Waiting {
    system_pause_min}m for the next detection")
204                             message = {
205                                 'event': 'E1',
206                                 'device_id': DEVICE_ID
207                             }
208                             mqttBroker.sendMessage('EVENT', message)
209                             Functions.printLine("YELLOW", f">> Event E1
     detected")
210                             n_detections = 0
211                             alarm_triggered = True
212                             sleep_time = system_pause_min * 60
213                     else:
214                         print("[OK] No fire event detected")
215                         n_detections = 0
216                 else:
217                     alarm_triggered = False
218                     sleep_time = sleep_time_normal_sec
219 
220                 # Remove the actual photo, allowing generating a new
    photo
221                 os.remove(image_name_png)
222                 time.sleep(sleep_time)
223 
224             checkForConnection = True
225 
226         checkTimeout()
227         time.sleep(SLEEP_LOOP_SEC)
228 
229 except KeyboardInterrupt:
230     my_mqtt_client.disconnect()
231     Functions.printLine("RED", "\n>> AWS MQTT broker disconnected")
```

Chapter 5

Fire Detection

5.1 SVM Training

The training process of the SVM (Support Vector Machine) algorithm to detect fires through images involves several phases:

- Loading data
- Data preprocessing
- Training of the model
- Safeguarding trained weights

The model is trained with images containing fires and photos of forests without any fire.

```
import os
import cv2
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from joblib import dump, load

# Funzione per caricare e preprocessare le immagini
def load_images_from_folder(folder, label):
    images = []
    labels = []
    for filename in os.listdir(folder):
        img_path = os.path.join(folder, filename)
        img = cv2.imread(img_path)
        if img is not None:
            img = cv2.resize(img, (150, 150))  # Ridimensiona le
    immagini
            img = img.flatten()  # Converte l'immagine in un array
    monodimensionale
            images.append(img)
            labels.append(label)
```

```
21      return images, labels
22
23  # Percorsi delle immagini
24  fire_images_path = 'machinelearning/dataset/fire'
25  non_fire_images_path = 'machinelearning/dataset/non_fire'
26
27  # Caricamento delle immagini
28  fire_images, fire_labels = load_images_from_folder(fire_images_path, 1)
29  non_fire_images, non_fire_labels = load_images_from_folder(
        non_fire_images_path, 0)
30
31  # Combinazione dei dati
32  X = np.array(fire_images + non_fire_images)
33  y = np.array(fire_labels + non_fire_labels)
34
35  # Divisione del dataset in training e test set
36  X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
        =0.2, random_state=42)
37
38  # Normalizzazione dei dati
39  scaler = StandardScaler()
40  X_train_scaled = scaler.fit_transform(X_train)
41  X_test_scaled = scaler.transform(X_test)
42
43  # Creazione e addestramento del modello SVM
44  svm_model = SVC(kernel='linear')
45  svm_model.fit(X_train_scaled, y_train)
46
47  # Salvataggio del modello addestrato e dei dati di addestramento
48  model_file = 'machinelearning/model/svm_model.joblib'
49  dump(svm_model, model_file)
50
51  X_train_file = 'machinelearning/model/X_train.npy'
52  np.save(X_train_file, X_train)
53
54  Scaler_file = 'machinelearning/model/scaler.joblib'
55  dump(scaler, Scaler_file)
56
57  print(f"Modello SVM, X_train e scaler salvati in {model_file}, {
        X_train_file} e {Scaler_file} rispettivamente.")
```

5.2 Pi Camera Visual demonstration

Figure 5.1: Raspberry Pi Camera fire detection

Chapter 6

User Interface

A graphical user interface was made to show the real-time behavior of the system on the screen. The GUI is served through Apache on a Docker container. The graphical user interface was realized by use of:

- **HTML**: for the structure of the page
- **CSS**: for the graphics on the page
- **JavaScript**: front-end language
- **PHP**: back-end language
- **Ajax**: for real-time map
- **OpenStreetMap**: to use the map and the markers

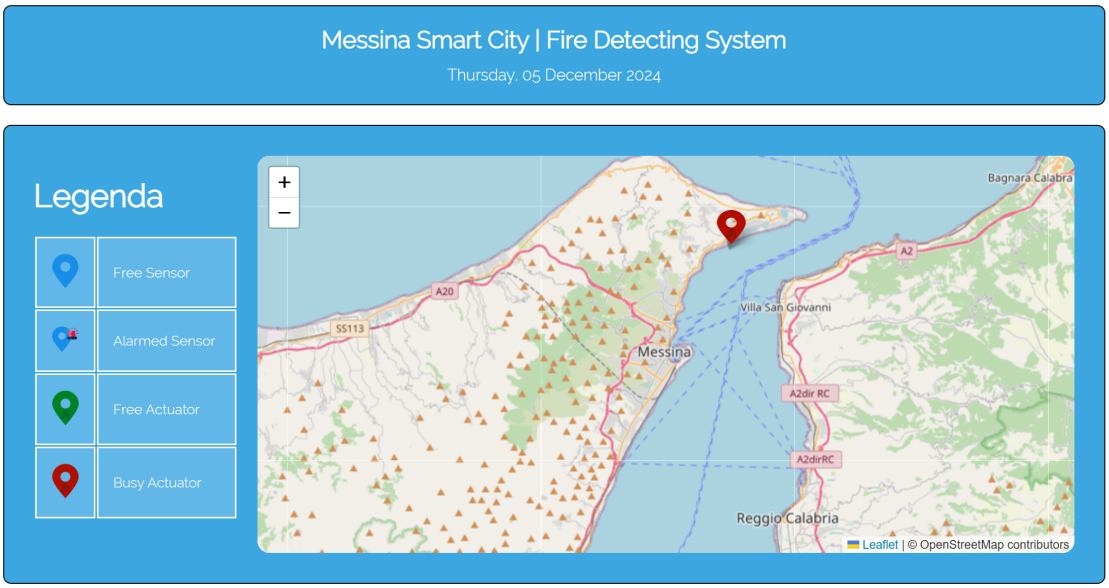


Figure 6.1: User Interface

The interface is divided into two parts:

- **Map of devices**: in which we can see in real time the devices connected to the system: sensors and actuators that may be in idle or busy state. If an alarm was triggered less than three minutes ago, a blue sensor icon with a siren logo is shown. If the actuator is clear it will be green, otherwise red.
- **History Logs**: A console where the latest system logs are shown: any event that is triggered will be shown in the bottom bar. If the event features an actuator action, for example in the case of a fire, the message will be highlighted in red to make it stand out to the eye.

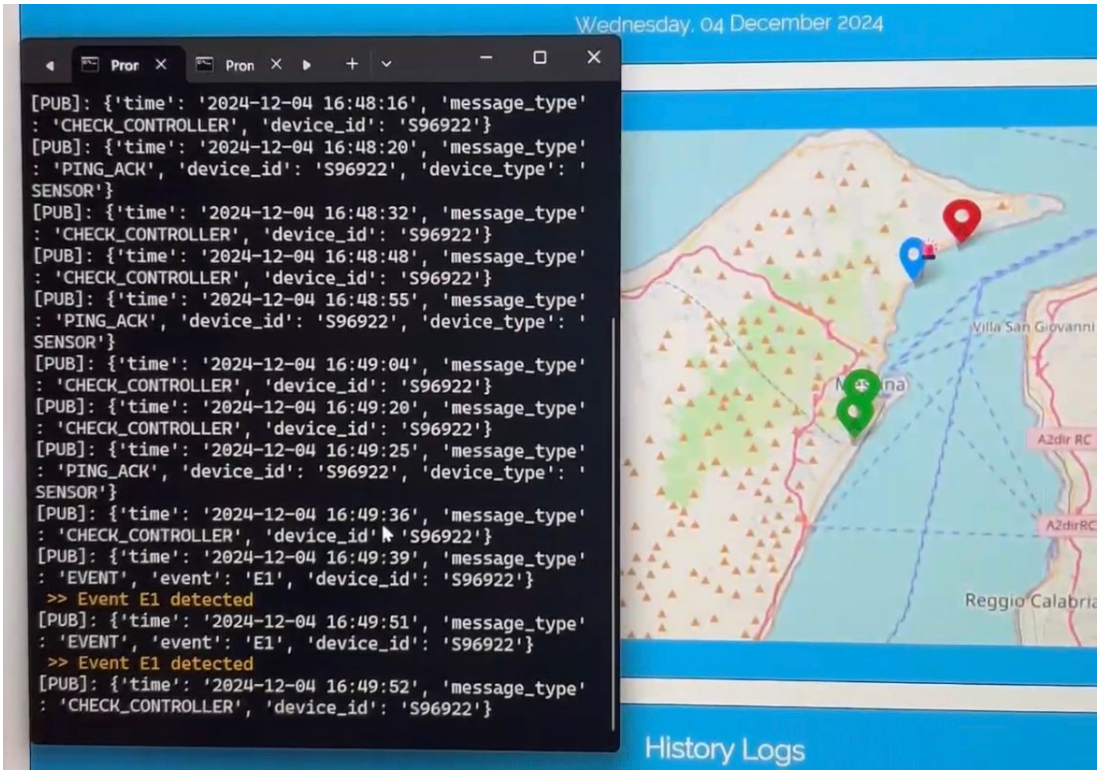


Figure 6.2: Map of devices during a event triggered

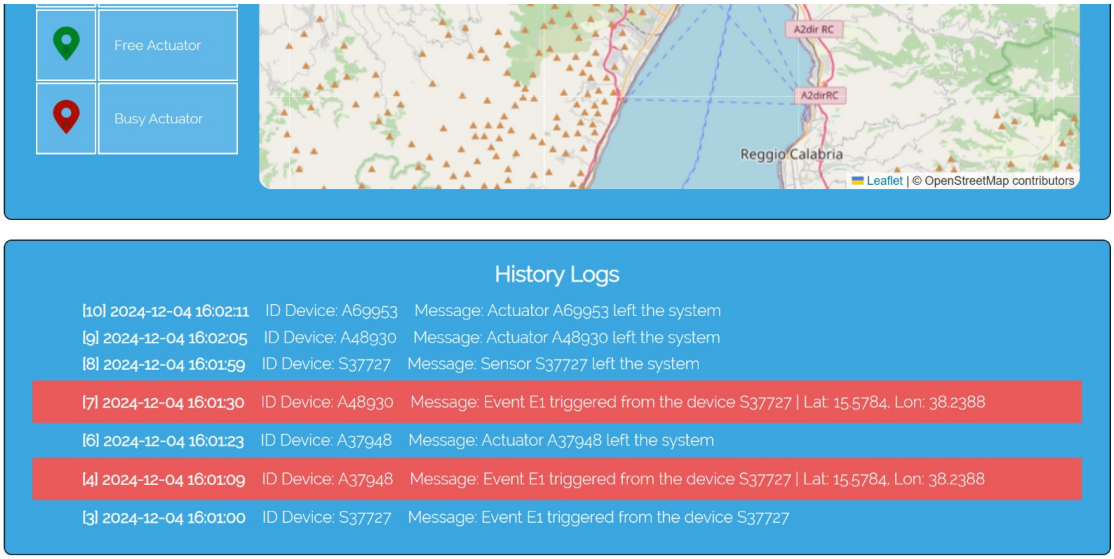


Figure 6.3: History Logs

By means of a timed Ajax call, it is therefore possible to update both the map of devices and history logs. The Ajax call calls a PHP file that opens the connection to the database on which the information with respect to devices and logs is stored. A simple retrieval of the information that already exists on the database is then performed.

The GUI therefore is a way to visualize concretely what is going on behind the scenes within the system.

Chapter 7

API test and documentation

The creation and management of APIs requires a well-structured process to ensure the proper functioning and reliability of interactions between applications. This process consists of several phases, including design, development, testing and documentation.

7.1 Postman

After designing and developing the APIs, it is important to subject them to a testing process to identify possible errors, bugs or security vulnerabilities. API testing can be divided into several categories, including unit testing, integration testing, and functional testing.

Unit testing is based on verifying individual API functionality, integration testing examines interactions between different system components, and functional testing focusses on validating API responses against requests made.

An application used for API testing is Postman.

Postman provides a complete API development environment that allows developers to send requests to APIs, visualise responses, and test complex situations. It supports different types of requests such as POST, GET, PUT, DELETE and offers advanced features such as environment variable management, authentication and authorisation.

Finally, it allows you to automate API testing and generate detailed reports.

Postman request example

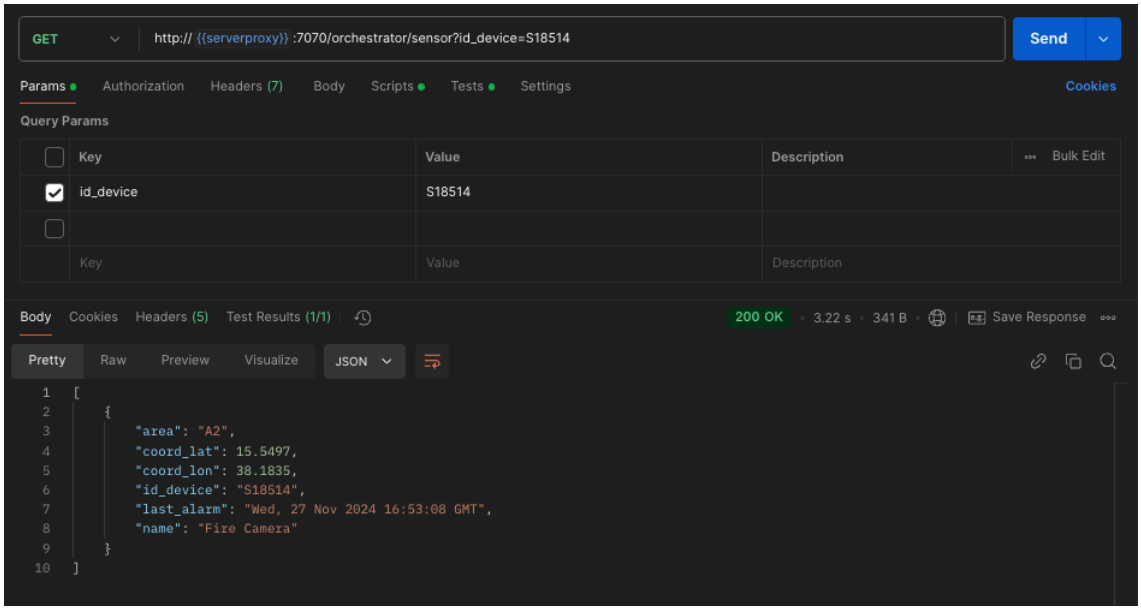


Figure 7.1: Postman Request

7.2 Swagger

Unce the testing process is completed, we move on to the phase related to the documentation of the APIs, a fundamental process to facilitate the use and resolution of any problems by external developers.
The documentation provides detailed information on API functionality, required and returned parameters, available endpoints, and any limitations.

In this regard, a tool used for API documentation is Swagger.

Swagger is a tool that allows you to automatically define, view and generate API documentation in a standardised format.
To create documentation with Swagger, you usually start by defining the API using a file in JSON or YAML format. This file contains the structure of the API, describing:

- **Endpoint disponibili** (es. /sensor, /actuator).
- **Metodi HTTP utilizzati** (GET, POST, PUT, DELETE).
- **Parametri richiesti o opzionali**.
- **Risposte attese**, con codici HTTP e eventuali payload.
- **Informazioni generali**, come titolo, versione e descrizione dell'API.

Once you have written the file, you can view and verify it using Swagger Editor, a web application accessible online.

7.2.1 Example of API documentation

File YAML

```yaml
openapi: 3.0.0
info:
  title: Documentazione API
  version: 1.0.0
paths:
  /controller/sensor:
    post:
      summary: Aggiungi un nuovo sensore
      description: Aggiunge un nuovo sensore con i dettagli forniti nel corpo della richiesta.
      requestBody:
        required: true
        content:
          application/json:
            schema:
              type: object
              properties:
                id_device:
                  type: string
                  description: ID del sensore
                  example: S4
                name:
                  type: string
                  description: Nome del sensore
                  example: Fire Camera
                area:
                  type: string
                  description: Area del sensore
                  example: A2
                coord_lat:
                  type: number
                  format: float
                  description: Coordinata X del sensore
                  example: 20.10
                coord_lon:
                  type: number
                  format: float
                  description: Coordinata Y del sensore
                  example: 10.00
      responses:
        '201':
          description: Sensore aggiunto con successo
          content:
            application/json:
              schema:
                type: object
                properties:
                  message:
                    type: string
```

```
                    example: Sensor added successfully
      '400':
        description: Parametri mancanti o non validi
        content:
          application/json:
            schema:
              type: object
              properties:
                message:
                  type: string
                  example: Missing one or more parameters
      '500':
        description: Errore interno del server, impossibile
  aggiungere il sensore
        content:
          application/json:
            schema:
              type: object
              properties:
                message:
                  type: string
                  example: Failed to add sensor
```

Swagger UI

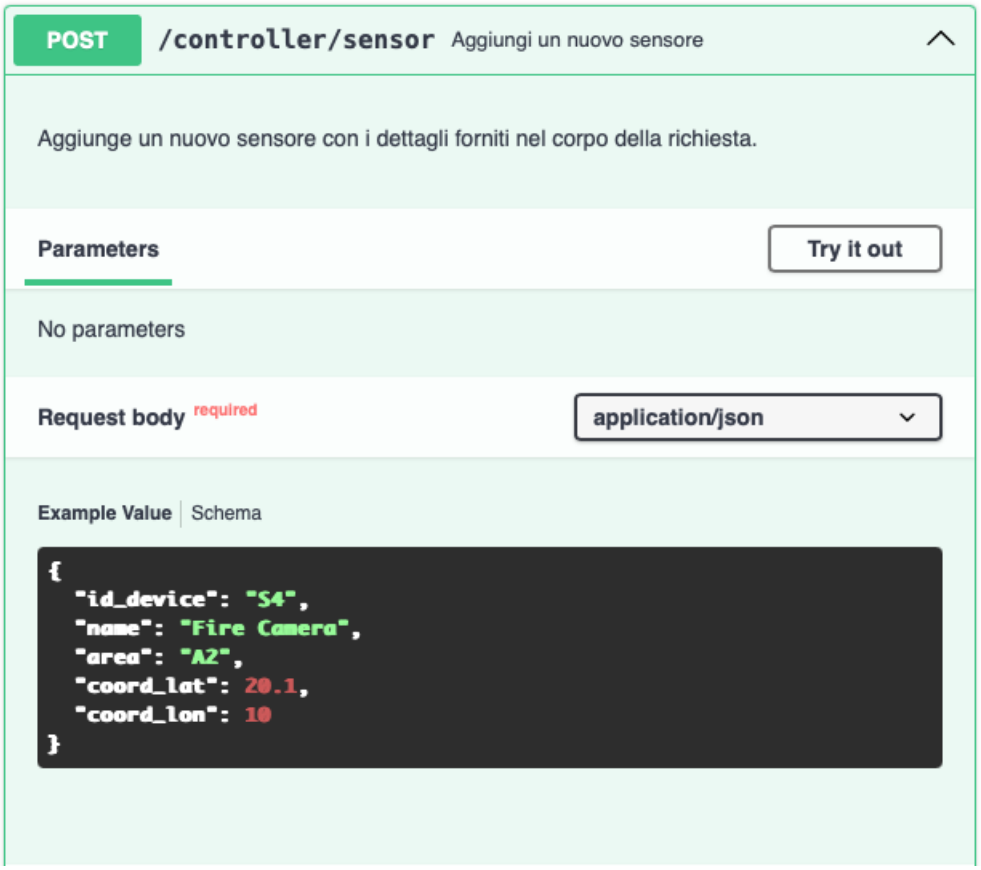


Figure 7.2


Responses
Code
Description
Links
201
Sensore aggiunto con successo
No links
Media type
application/json
Controls Accept header.
Example Value
Schema
{
"message": "Sensor added successfully"
}
400
Parametri mancanti o non validi
No links
Media type
application/json
Example Value
Schema
{
"message": "Missing one or more parameters"
}
500
Errore interno del server, impossibile aggiungere il sensore
No links
Media type
application/json
Example Value
Schema
{
"message": "Failed to add sensor"
}

Figure 7.3

Conclusions

The project presented is part of the context of smart cities, with the aim of simulating a distributed system for the detection and management of fires in real time. Through the use of a Raspberry Pi, we have implemented a solution capable of detecting a fire and activating the actuators necessary for the intervention, thus ensuring a quick and automated response to emergency situations.

The management of the entire system is based on efficient communication through the MQTT protocol, hosted on AWS. This choice made it possible to implement a scalable, reliable system that conforms to the needs of a distributed architecture. We also took advantage of the use of Docker containers to separate and manage the different components of the system: the controller, the orchestrator, the database and Nginx for the reverse proxy.

The project made it possible to explore and apply fundamental principles of distributed systems, such as modularity, interoperability and scalability. The combined use of Docker, MQTT and Nginx containers has demonstrated the possibility of building a robust and adaptable architecture, able to easily integrate into broader contexts of smart cities.

Project files references

- Google Drive Folder
- Enter to phpMyadmin
- Enter to user-interface