Licence plate recognition and actuation system

Project report of the IoT course– Academic Year 2023-2024

Matteo Amorth 238379 University of Trento

Abstract— **Licence plates serve as unique identification codes that are required for almost all the vehicles worldwide. Just as ID cards allow the identification of people and provide accesses to specific services, such as financial transactions, healthcare prescriptions and voting, also licence plates can be used to unlock many benefits. This paper is going to present how licence plate recognition can be exploited to control and manage different types of gates, barriers and similar systems with the contribution of distributed IoT devices, machine learning models and fog-cloud network management.**

Keywords - plate recognition, IoT system, machine learning

I. INTRODUCTION

Since the creation of the first mainframe and the initial connections between computers, computer scientists have understood that connecting devices could unleash an immense amount of data and computation power. Historically, powerful data centres shared resources via high-speed connections, often relying on specialized and expensive hardware like optical fiber connections. Clients and terminals would establish connections with these data centers to request services and execute operations. However, with the growing demand for real-time data processing and energy efficiency, a new network paradigm has emerged in recent years: IoT (Internet of Things) networks. A generic definition for IoT is the following one: "A global network infrastructure, linking physical and virtual objects through the exploitation of data capture and communication capabilities. This infrastructure includes existing and evolving Internet and network developments. It will offer specific object identification, sensor and connection capability as the basis for the development of independent cooperative services and applications. These will be characterized by a high degree of autonomous data capture, event transfer, network". Technically speaking, this architecture focuses on three key aspects: distribution tasks across many connected and distributed devices, processing data closer to the source, and reducing energy consumption. To achieve these goals, IoT networks are organized into three main device categories: edge devices, fog devices and cloud. Edge devices are typically small embedded systems equipped with various types of sensors to collect data from the environment. These devices either forward raw data to the network or perform minimal processing before sending it too. Edge devices firmware is optimized for low power consumption to enable continuous operations on batteries. If the system is not real-time, it is possible to save more energy continuously collecting data and enabling communications a few times per day. Fog devices act as intermediaries between edge devices and the cloud. Their role is to collect data from edge devices and perform not too complex operations such as data filtering, message management, hosting dashboard, and bridging different protocols like gateways. These devices are crucial because they enable real-time responses by processing data quickly and close to the source. Indeed, by processing data locally, fog devices help reduce the amount of information that needs to travel to the cloud, improving bandwidth constraints. Additionally, fog devices require less energy than cloud servers, reserving cloud resources for more complex tasks. Even more, in large scale IoT deployments, fog devices allow for the distribution of workloads, preventing bottlenecks at the cloud level and enabling more scalable solutions. Finally, cloud servers provide the heavy computational power needed for tasks that cannot be completed by fog devices, such as running complex neural networks models or sophisticated algorithms. The cloud is essential when the local resources are insufficient to handle demanding operations, or the time required to complete them on small devices would be too long. In addition, cloud servers support many different programming languages and software that rarely are supported by fog devices. To support the growing complexity and requirements of IoT networks, lightweight communication protocols have been developed, focusing on energy efficiency, low bandwidth consumption, and reliability over constrained environments. Four popular protocol examples are MQTT, Zigbee, BLE, and LoRa. IoT networks pushed the development of a new type of networks called LPWAN (Low Power Wide Area Networks), that are a long-range and low-power communication for IoT devices. LoRa WAN and NB-IoT are examples of this type of network.

II. RELATED WORKS

A. Commercial solutions

Many commercial products, developed for gate control are available off the shelf. These solutions typically offer complete systems with complex and dedicated hardware. Most of these products, designed for private users, include smartphones applications to control the systems. When a user approaches the gate, he provides an input to the system. The systems often include additional tiny dashboards and management interfaces. The communication between these devices usually relies on local networks and employs proprietary and closed protocols to exchange information. However, products based on plate recognition are less common and are typically designed for large public parking areas or similar environments. These solutions tend to be even more close and less customizable than private solutions. Additionally, they often come with a lot of expensive features that may not be necessary or affordable for individual users or smaller setups

B. *Hobbyist solutions*

The Roboflow website presents a project like the one discussed in this report, which utilizes both Roboflow and Node-RED. Roboflow is a platform that allows users to train neural networks and either download or deploy models. Node-RED, on the other hand, is a platform for visual programming that allows users to create workflows, known as "flows", which integrate various tools, code, and APIs. This visual approach simplifies the process, allowing applications and workflows that would normally require significant coding to be developed with much less effort and time. This platform works seamlessly on PCs and microchip-based boards like the Raspberry Pi. However, microcontroller-based boards are not yet supported. Additionally, high-level programming interfaces, like this platform, usually lacks on the performance and optimization aspects, which are critical for energy-harvesting devices.

III. System brief description

A. *Aim of the system*

The goal of this system is to develop an integrated solution where an IoT system allows registered vehicles to open various types of gates, barriers, and automated doors without the need of any type of remote control or user interaction.

The main objective of the system is to develop a distributed environment (IoT), where many different types of simple devices work together to achieve results and perform operations, rather than building a single complex device that performs every operation stand alone.

B. *Logic flow*

The logic flow of the system can be summarized in a few consecutive steps. First, images of the environment are captured by a camera positioned with a clear line of sight to the gate entrance. These images are then transmitted to the server over a wireless connection. Once received, the server decodes the images and performs object detection using a neural network. If a licence plate is detected, another neural network recognizes the characters within the plate. Finally, the results are verified and compared with existing records, and appropriate actions are executed based on the outcome. The logic cycle of the system is showed in [1].

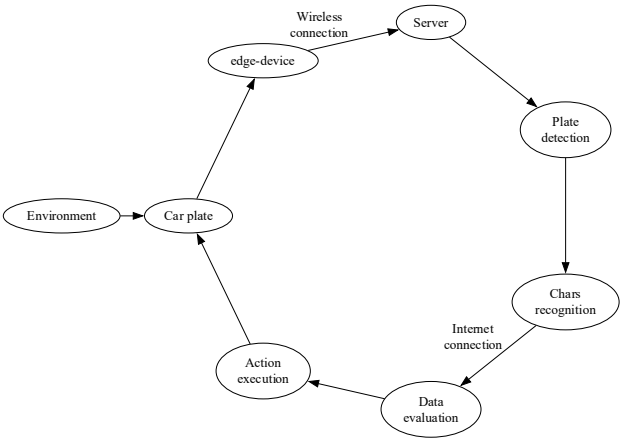

Fig. 1 – High level flow of the system.

IV. Hardware

The system consists of a full-stack IoT infrastructure, that starts from embedded devices to distributed internet servers, where every member performs specific tasks. The architecture has three pillars:

- Edge device: Collects images from the environment and forwards the data over the network.
- Local server: Manages data retrieved from the network and handles communications.
- Remote resources: Hosts neural network models necessary for data processing

A. *Edge device*

As the name suggests, an edge device connects different worlds. In this setup, the edge device's role is to acquire information from the environment using a camera and forward it to the system for processing. This device requires two main features: an integrated camera (or at least an interface that can be connected to a camera) and a network connection. A *OpenMV Cam H7 Plus* is a suitable option for this role [2]. This device is a small, low-power, microcontroller board that allows to implement computer vision applications in the real-world. It can be programmed with high-level Python scripts, thanks to its MicroPython implementation (that includes a small subset of Python libraries). This board has an integrated camera mounted in the upper part and a powerful ARM Cortex M7 processor capable to perform machine learning operations directly on the board. However, to run neural network models on such boards, an optimization and adaption process is required that can be complex. Additionally, this device should have low energy consumptions, allowing to operate only with batteries supply or alternative solutions.

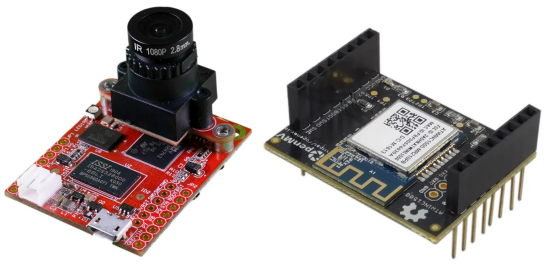

Fig. 2 – The OpenMV Cam and WiFi shield boards

The network connection can be established with the external boards called "shields", that extend the OpenMV board's capabilities through the connection with the available pins. As the name suggests, the WiFi Shield [2] enables wireless connection. The connection between the main board and the shield is performed through an SPI connection (Serial Peripheral Interface) leaving three remaining GPIO pins (P4, P5 and P9), available for other SPI connections or I/O operations [3]. In the setup test, two pins have been used: one to read the current status of the gate and another to provide an actuator signal as an output pin. In other words, the camera board acts as the actuator of the system, becoming the first and the last element of the system's chain.

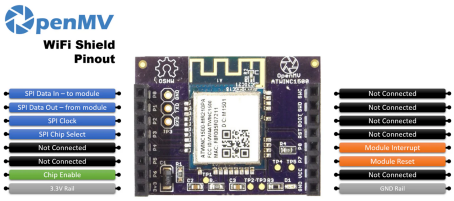

Fig. 3 – The WiFi shield pinout

B. Local server

The local server provides various services that can be requested to complete operations. In this system, a RaspberryPi is well-suited for this role: it is powerful enough to perform complex operation while remaining compact and energy efficient. It supports the installation of a complete operating system that unlocks the possibility to run many different services such as network connections, neural networks models, algorithms, webpages and complex programs. For this setup, initially a PC has been used to develop and monitor the system, and once it has been verified, all the system was transferred on a RaspberryPi [4].

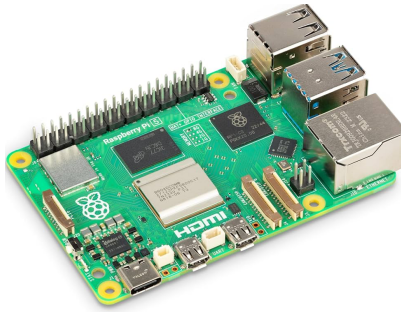

Fig. 4 – A RaspberryPi board

C. Remote resources

Remote resources refer to all the services provided by external servers that enable the execution of complex operations, like training and hosting neural networks models. Although it is possible to host these models on a local server, it is often much faster to interact with models in a powerful remote datacentre rather than running them locally. In addition, the system goal is functioning as a distributed architecture involving edge, fog and cloud devices. In this setup, the Roboflow web environment was exploited to train, test and validate the plates detection model. For the characters recognition, a pre-trained keras model was implemented without any training hardware.

V. Software

A. Core software

All the programs executed by the system follow a series of consecutive steps, starting from data acquisition to the execution of operations. To ensure that everything functions correctly, a core program running on the local server coordinates all the activities within the system. It is responsible of many different tasks:

- Message acquisition
- Network management
- Neural networks models interaction
- Data manipulation
- Database interrogation
- Message delivery
- Errors handling
- Logging activity

A state machine structure was chosen for this program, thanks to the ability of task management as separated processes, enhancing the readability and development of the code. In [5] is showed the state machine states.

```
def run(self):
    state_functions = {
        'idle': self.idle,
        'plate_detection': self.plate_detection,
        'chars_detection': self.chars_detection,
        'string_detection': self.string_detection,
        'img_crop': self.img_crop,
        'exit': self.exit
    }

    while True:
        state_function = state_functions.get(self.state)
        if state_function:
            state_function()
        else:
            log.error(f"[CORE] Unknown state: {self.state}")
            self.send_msg(self.mqtt_dbg, '[MQTT]     Unknown state - Closing program')
            break
```

Fig. 5 – The state function machine

B. Data acquisition

The OpenMV Cam board comes equipped with all the necessary libraries to capture and process images using its embedded camera. With just a single line of code, images can be acquired and various modifications can be applied. These operations are performed entirely on the board, enhancing the real-time image acquisition process. As a starting point, many examples available on the OpenMV IDE offer guidance during the development phase.

C. Wireless connection

Wireless connection is used to connect the edge-device with the local server on the same network via WLAN protocol (Wireless Local Area Network). To establish the connection, it is sufficient to include the network library provided by the board manufacturer and configure a few parameters as outlined in [6].

```
def wireless_setup(self, SSID, KEY):
    print("[SETUP] WIFI")
    self.wlan = network.WLAN(network.STA_IF)
    self.wlan.active(True)
    self.wlan.connect(SSID, KEY)

    while not self.wlan.isconnected():
        dprint('Connecting to "{:s}"...'.format(SSID))
        time.sleep_ms(1000)

    dprint("[Setup] WiFi Connected ", self.wlan.ifconfig())
```

Fig. 6 – The connection setup function

D. MQTT transmission

Once the devices are connected in the same network, a messaging protocol is needed to enable the communication between them. The MQTT (Message Queuing Telemetry Transport) protocol allows the exchange of information via a publisher-subscriber messaging model. Thanks to its lightweight structure, it is suitable for low-power and low-

performance devices such as those in this IoT system. Sensors boards can publish data in special boxes called "topics", and clients subscribed to those topics can access the data. In this setup, three topics were used:

- test/cam_target: where the edge device publishes images and the local server, as a subscriber, reads the data
- test/plate_check: where the local server publishes results and devices, as subscribers, read and perform operations.
- test/cam_debug: where the local server publishes status and errors information.

These topics are manged by an MQTT server, known as broker, which handles the infrastructure and hosts the topics. Many developers created MQTT brokers with various features. In this setup, an Eclipse Mosquito broker was selected because for its flexible message payloads and customizable configuration settings. Additionally, it can be hosted on a RaspberryPi, allowing it to be installed on the local server.

E. Image transmission

Images must be sent via the MQTT connection to the local server. However, images typically exceed the maximum payload size allowed by brokers. For this reason, a compression and segmentation process was implemented to reduce the payload size. This process involves several steps:

1. Get the image size
2. Divide image into small packets
3. Encapsulate the packet with header information
4. Convert the data into a string.
5. Send the packet
6. Repeat previous steps until the entire image is transmitted.
7. Send a transmission completion message.

F. Message stucture

To make the messages understandable from the whole system devices, a common syntax standard was chosen. In this system, JSON (JavaScript Object Notation) objects are used as the primary format for packaging and transmitting data between devices. JSON is lightweight, easy to parse, and widely supported, making it ideal for IoT systems with low-power devices and limited resources. The role of JSON objects is to encapsulate image data along with essential metadata, ensuring that each packet of information is understandable by the receiving system. Each JSON object contains multiple fields to describe the data being sent. A typical JSON object in this system is outlined in [7]

```
message = {
    "device_id": self.device_id,
    "mode": 3,
    "payload": fragment
}
```

Fig. 7 – The JSON object for messages

where device_id is the name of the edge-device, mode identifies the type of message sent and payload the actual data to be shared. When the local server sends messages to the edgedevice, the device_id field is the target's name. This allows to filter messages when more devices are involved.

G. Message reconstruction and verification

When a new message arrives on the local server subscribed topic, several operations are triggered. First, a message handler function interrupts the current activity running in the main script and begins reading the incoming data. If the message is correctly encoded as a JSON object (organized with device ID, operational mode and payload fields), a verification process checks if the message was sent by a registered device. Then, a buffer stores the chunks of the image received by the messages until the conclusion message is received. At this point, the entire image is reassembled in a variable, ready for further processing.

H. Plate detection

Plate detection is accomplished using Roboflow tools, which enable the training of neural networks models by providing vehicle images labelled with bounding boxes around the plates. The powerful YOLO v8 (You Only Look Once) algorithm allows real-time detection of objects within the images provided. More than 1700 images were used to train the model, while test and validation phases both used around 400 images. Once the model is trained, it can be deployed to the cloud using the *Hosted image inference* service provided by Roboflow's *inference_sdk*. This function takes an image and the trained model as input, returning results that indicate the objects detected, their size, locations and the probability of matching the desired object. If multiple plates are detected, the local server script selects the one with the highest probability, while the other results are discarded. To determine whether a vehicle is arriving or leaving, a simple strategy is employed: if two consecutive frames contain a licence plate, and the latter shows a larger plate size, it indicates the vehicle is approaching; otherwise, the vehicle is leaving.

I. Characters recognition

Once the licence plate has been detected, the local server script crops the area around the plate and forwards the result to the OCR model (Optical Character Recognition). The OCR model identifies any text present in the provided image and attempts to convert it into letters and numbers. This model has been pre-trained with Keras, an open-source deep learning API built on TensorFlow libraries. To use this model, Keras and TensorFlow libraries must be installed as python packages. While the plate detection model operates in a remote server, the OCR model runs locally because it is lightweight enough to be executed quickly, even on low-specs hardware (like the local RaspberryPi server). The model receives an image as an input and returns any characters found in the image, along with their coordinates and confidence probability. Although it is possible to use the Keras model directly for character detection in the environment, the division of tasks into two models provides important benefits:

- Precision: models trained for specific tasks (e.g. detecting plates) tend to deliver more accurate results.

- Real-time performance: The OCR model runs very quickly when it has a clear focus on where text is likely to be located
- Security: The plate detection model is trained specifically to recognize licence plates, making it harder to bypass the system with counterfeit or homemade solutions.

J. *Database*

The data obtained from previous steps must be compared against records stored in a database or other storage systems. For this setup, a InfluxDB database was chosen. InfluxDB is optimized for time-series data, making it ideal for IoT systems, and it can be easily integrated into various scripts and software. Additionally, it is lightweight enough to run on low-spec devices like the local RaspberryPi server.

The database structure consists of a table for each plate, all tagged with *plate*, and grouped inside a bucket (an InfluxDB database) called *license_plate_data*. To integrate this database with the core program, the software must be installed on the server, and a database token must be provided to the script. Once connected, data can be read and written using various modes developed to interact with the database. In this system, plates are stored with an associated status value. This allows to enable and disable accesses, keeping track of the previous authorizations. Fig. [8] shows how data can be queried through the graphical interface.

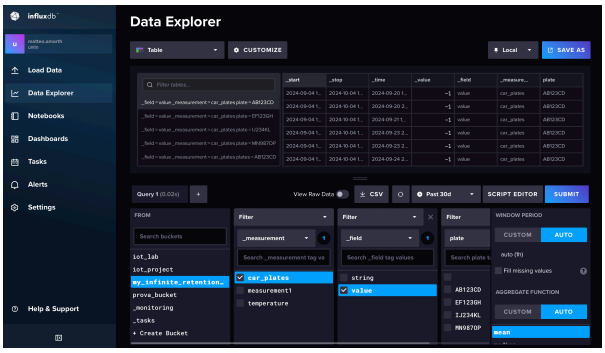

Fig. 8 – The InfluxDB interface

K. *Dashboard*

Even though there is an MQTT topic dedicated to debug and information, an interface has been implemented into the system to provide visual feedback for users. The software used to show data is Grafana, a useful tool that provides an impressive dashboard's library fully customizable and integrable with InfluxDB. This configuration allows to perform modification without the involvement of the core program. In this setup, the dashboard has been organized to show four different panels:

- General statistics: report how many accesses have been performed by vehicles.
- Daily accesses: report the history of the current day with allowed and rejected vehicles.
- Last rejected vehicle: shows the last vehicle that tried to get over the barrier.
- Last allowed access: shows the last vehicle access with time log.

Fig. 9 – The dashboard of the system

L. *Debug and logging activity*

While the core program is running, a debug topic is active, where useful information about the server's errors and status are published. This allows remote monitoring of the system's operation and helps in identifying any potential issues. Additionally, a duplicate of the status is printed in the console. Logging levels (INFO, WARNING, ERROR, DEBUG) can be configured to filter which messages are displayed. Furthermore, a dedicated log file is extended at each session, storing all logging messages for future analysis.

M. *Configuration*

The whole system can be customized with two text files called *config.ini* and *server_config.ini*, used to configure the edge-device and the local server scripts, respectively. The customizable values are:

- Device IDs
- MQTT settings
- MQTT topics
- WIFI parameters
- InfluxDB parameters
- Roboflow parameters
- GPIOs

VI. System Setup

To set up the system, the following steps were carried out on the IoT system devices.

A. *Edge-device*

- Configure properly the config file accordingly to network and MQTT parameters.
- Connect the input GPIO to a push button and the output pin to a led with protection resistor.
- Load the script developed for the board
- Run the code.

B. *Local server*

- Download and install all the python dependencies.
- Download and install a MQTT broker.
- Download and install InfluxDB. Create a bucket where the data will be stored.
- Download and install Grafana. Link the InfluxDB database. Import the dashboard template.
- Run the core program.

C. *Remote resources*

- Create a Roboflow account and upload training images.
- Train a plate detection model and export the keys to the core program.

D. *Other operations*

- Add records to allowed plates tables.
- Subscribe an external MQTT client to topics used in the system.

VII. RESULTS

The system provides a lightweight solution to manage vehicle accesses with an IoT network. Messages are exchanged across many different devices and neural network models are involved to complete tasks. The local server can complete all the tasks even with limited resources. The OpenMV cam board delivered images streaming without issues on capturing and sending data. Finally, the embedded dashboard provided insights about the system routine.

VIII. FUTURE DEVELOPMENTS

The system is designed to be expandable in various ways. The core program already supports multiple configurations. For instance, cropped images of the license plate can be sent to the server for character recognition, eliminating the dependency on Roboflow's environment. Additionally, strings can be sent directly to the local server, where they are evaluated against database records. For both cases, new neural network models suitable for edge-devices must be trained. A simple hardware expansion could be connecting relays on the output pin allowing the usage of high voltage devices. A more advanced hardware modification could involve replacing the WLAN communication system with LoRa. This could be achieved by connecting the edge device to a LoRa board via the SPI interface.

REFERENCES

[1] Keith D. Foote, "A brief History of the Internet of Things".

[2] EU Framework 7 Project, "Casagras and the Inclusive Model for the Internet of Things".

[3] IoT Analytics Research 2018, "Total Number of active device connections worldwide".

[4] A. Triantafyllou, "Network Protocols, Schemes, and Mechanisms for Internet of Things (IoT): Features, Open Challenges, and Trends", Wireless Communications and Mobile Computing Volume 2018.

[5] Commercial systems example [online]: https://www.1control.eu/en/index.php.

[6] About Node-Red [online]. Available: https://nodered.org

[7] Hobbyist solution [online]. Available: https://blog.roboflow.com/use-node-red-with-roboflow/

[8] About Roboflow inference [online]. Available: https://inference.roboflow.com

[9] About OpenMV cam [online]. Available: https://openmv.io

[10] About RaspberryPi [online]. Available: https://www.raspberrypi.com

[11] About MQTT. Available: https://mqtt.org.

[12] Ecma international, "The JSON Data Interchange Syntax", ECMA-404, 2nd edition, December 2017

[13] About InfluxDB [online]. Available: https://www.influxdata.com/index/

[14] About Grafana [online]. Available: https://grafana.com