IoT System
Design and implementation of a Framework to load balance data analysis on the device or at the edge

Barbanti Francesco - matr. 161051

28 gennaio 2022

1 Introduction

The goal of this project is to implement a load balancer that, starting from three different simple algorithms, decides how much computation to perform on the device and how much computation distribute to the edge devices. In addition, a dashboard was developed to monitor the system status, the number of connected devices, how the computational load is distributed and much more.

2 Tools

This section describes the technologies used in order to achieve the objectives just presented. First let's break down the dashboard logic from the load balancer logic:

Dashboard tools

The implementation of the dashboard is based on Node-RED [1], a flow-based development tool that provides a browser-based editor that makes it easy to wire together flows using the wide range of nodes in the palette that can be deployed to its runtime.
The choice fell on this tool as it is commonly used in the IoT world and thanks to its features, it is possible to run at the edge of the network on low-cost hardware such as the Raspberry Pi as well as in the cloud. In addition, Node-RED allows the programmer to write Javascript code, in order to customize the behavior of the dashboard.
The modules installed from the Node-red palette are:

- node-red-dashboard: a module that provides a series of widgets useful for interacting with the user
- node-red-node-sqlite: a module that provides an instance of a sqlite database
- node-red-node-ui-table: a module that allows you to transform Javascript objects into HTML tables

Load balancer tools

The load balancer code was written on Visual Studio Code [2]. In particular, it was installed a plugin named PlatformIO [3], an alternative of the most famous Arduino IDE. I chose PlatformIO because I feel better with the VSCode development environment instead of the Arduino one.
The code was developed for ESP32 boards and the following libraries have been included:

- ArduinoJson [4]: a simple and efficient JSON library for embedded C++.
- WiFi [5]: a library that allows ESP board to connect to Wifi stations.

- Preferences [6]: a useful library to save data permanently on ESP32 flash memory.
- AsyncMqttClient [7]: an Arduino for ESP8266 and ESP32 asynchronous MQTT client implementation.
- AsyncTCP [8]: a fully asynchronous TCP library, aimed at enabling trouble-free, multi-connection network environment for Espressif's ESP32 MCUs.
- ESPAsyncWebServer [9]: library that provides an easy way to build an asynchronous web server.
- AsyncElegantOTA [10]: library that allows OTA updates to ESP32 boards

As explained in section 6, the messaging protocol used is MQTT. In particular, the broker used is the well-known Mosquitto [11].

3 Project idea

The project is made up of various devices, which take on different roles. In fact, as shown in figure 1, there is a device that takes the role of leader, one or more edge devices and the laptop on which the mqtt broker and the dashboard run (but it could be replaced by a raspberry pi, for example).
The idea is that the user, through the dashboard, specifies the algorithm that the system must execute, as well as the parameters of this algorithm and the execution frequency, expressed in number of iterations per second. After that, the dashboard communicates the information requested by the user via the MQTT protocol with the leader. At this point the leader, through the logic described in section 5, understands if he is able to perform all the computation locally or if it is necessary to offload it to the edge devices (if they are active). In the first case, every second the leader carries out the iterations specified by the user and, once finished, returns the result to the dashboard using an appropriate topic. Instead, in the second case the leader offload the iterations that he cannot bear to the devices, keeping the computation locally as much as possible. On the edge device side, the computation is distributed on the basis of computational capabilities: for example, if the leader fails to perform 100 iterations per second and two ESP32s and a Raspberry Pi are available, to the first will be required to perform 20 iterations each, while to the second the remaining 60.
The dashboard shows different performance evaluation metrics, the result of the algorithms, the number of online devices etc. It also provides a page where you can dynamically configure the parameters for the wifi connection of each device (see section 4).
The computational problems that every device must be able to solve are:

- Calculate the first n prime numbers
- Count the words contained in a sentence
- Multiply two vectors of equal size

The project is also expected to implement OTA updates: for this reason each device (both edge and leader) initializes a web server, reachable at the page "ip_address/update", in which it is possible to upload a .bin file.

4 Dynamic Wifi credentials

One of the fundamental requirements of the project was to make it as dynamic as possible. For this reason, hard-coding network credentials is not a good idea. To solve this problem the flow chart shown in figure 2 has been implemented.
At the beginning the device tries to read the network credentials inside the EEPROM memory and, if this operation is successful, it tries to connect to the WiFi router for 60 seconds: if it succeeds then it continues its flow, otherwise it acts as an access point . Then

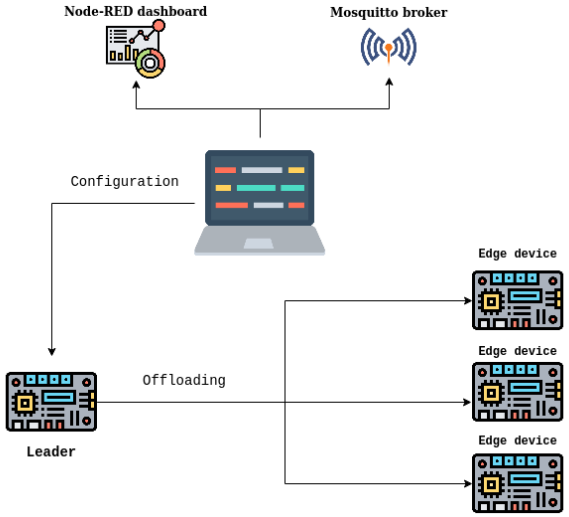

Figura 1: System diagram

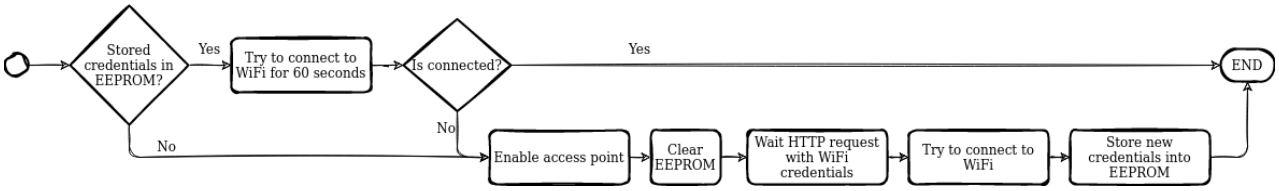

Figura 2: Flow chart of dynamic WiFi credentials

the user, through the appropriate page on the dashboard, connects to it and sends an HTTP request containing the new WiFi credentials. At this point the board connects to the router and saves the new credentials in permanent memory.
Furthermore, within the http request it is required also to enter the IP address of the MQtt broker; in fact, changing the WiFi network could obviously also change the broker's IP address assignment. This information is also stored in the board's permanent memory, and is removed only if the WiFi network credentials are changed.

5 Computational capabilities

This section describes the mechanism for discovering the computational capabilities of the various devices. In particular, each device, after connecting to the WiFi network and to the MQTT broker, executes an algorithm capable of detecting the computational capabilities of the latter. The idea is to run each computational problem 100 times in the worst conditions, calculate the execution time at each iteration, make an average and find the number of iterations of the same algorithm that the device is able to support in 1 second. This returns the number of iterations per second that the device is able to perform for each different computational problem.
The worst case conditions for a problem are defined on the basis of the largest input they can support. More specifically:

- Prime number: calculates maximum the first 400 prime numbers
- Word count: the size of the text must be a maximum of 500 characters
- Vector multiplication: the maximum vector size is 30 elements

Let's take a practical example: an ESP32, to return its computational capabilities in performing the calculation of the first n prime numbers, must call the function shown in listing

```
int getPrimeNumCapability(){
  unsigned long start = 0;
  unsigned long endd =0;
  unsigned long delta = 0;
  float num_sec=0;
  int ris[400];
  for(int i=0; i<100; i++){
    start = micros();
    getPrimeNumbers(2, 400, ris);
    endd = micros();
    delta = endd-start;
    num_sec = num_sec + delta;
  }
  num_sec = 1/((num_sec/100)/1000000);
  return (int(num_sec-50));
}
```

Listing 1: getPrimeNumCapability() function

	Prime number	**Word count**	**Vector Mult.**
Num sec.	2438	23276	19099

Tabella 1

1. Note how 50 iterations per second have been removed in the capability calculation: this is done for greater system stability. Similarly, this process can also be done for the other two computational problems and the final result is shown in table 1.

6 Communication protocol

As already mentioned, the messaging protocol used is MQTT. The choice fell on this protocol as the project lends itself very well to a publisher-subscriber paradigm. In fact, since the leader device must communicate with multiple edge devices at the same time the same message (see section 8), MQTT is a better choice than COAP. Also, since there is not supposed to be a packet loss problem in the environment, keeping a TCP connection open for each client shouldn't be a problem. The disadvantage that MQTT has in this scenario is that all clients must know the message formats up-front to allow communication.

7 Status of devices

This section explains how the leader is able to understand which edge devices are online and which are offline (i.e devices running out of battery). For this purpose are used the retain and lastWill type messages offered by the MQTT protocol.
Each time an edge device connects to the MQTT broker, it sends a retained message on the client topic "clients/id/status" with payload 1, where the id is the MAC address of the same. It also sets a retained lastWill message on the "clients/id/status" topic with payload 0. In this way, if the device does not communicate with the mqtt broker for more than 15 seconds, then it will be considered offline and the lastwill message will be published. At this point, to monitor the status of the edge devices, the leader can subscribe to the topic "clients/+/status".
This methodology is also used for the device leader, who publishes messages on the topic "leader/lead/status". This is useful for the dashboard to show the user whether the leader is online or not.
Similarly, after publishing the status message, the edge devices publish a message containing their computational capabilities on the topic "clients/id/capability" (see listing 2), as well

```
{
    "prime_num": 3239,
    "word_count": 24423,
    "vect_mult": 12423
}
```

Listing 2: Example of message published on the topic "clients/id/capability"

```
{
    "task": "vect_mult",
    "data": "{12 3 43 65}, {22 412 87 9}",
    "num_sec": 250
}
```

Listing 3: Example of message published on the topic "dashboard/task"

as a message on the topic "clients/id/ip" containing the ip address of the device (useful for showing on the dashboard at which address the Web server can be reached for OTA updates).

8 How the project works

When the user configures a computational problem from the dashboard and requests its execution, the process linked to Node-RED publishes a message on the topic "dashboard/-task". An example of this message is shown in listing 3.
Then the leader, who has subscribed to the topic, decodes the received message, understands which computational problem is required, checks its capabilities and decides whether to offload part of the computation to the edge nodes. In fact, if it is unable to fully bear the computational load, it checks which edge nodes are online and partitions the load based on their computational capabilities. So, in this case, the leader publishes a message on the topic "leader/task/algorithm", where algorithm can assume 3 different values (since there are three computational problems):

- leader/task/prime_num
- leader/task/word_count
- leader/task/vect_mult

Listing 4 shows an example of a message posted on the topic "leader/task/vect_mult".
Subsequently, once each device has completed its iterations, it publishes the result of the last iteration on the topic "task/result". In this way, each device that is running a computational problem publishes only one message per second, avoiding sending the single result of each iteration (in fact they are all the same).
It is a performance metric used to see how the system scales as the number of iterations per second it has to perform increases. Figure 3 shows the flow chart of the leader.

NB: obviously the leader maintains a data structure in which to store the status and computational capabilities of each edge device; in particular, during the implementation phase, it was decided that the maximum number of edge devices that the leader can manage is 20 (but it is possible to change this limit by updating a constant via OTA update).

9 Dashboard

The dashboard, implemented through Node-RED, subscribes to different topics of the MQTT broker, in order to monitor the status of mosquitto. It also shows the status, computational

```
{
    "dev": [
        {
            "id": "2c549188c9e3",
            "num_sec": 123
        },
        {
            "id": "v4g2hg9ipo2w",
            "num_sec": 421
        },
        {
            "id": "4rag2lm8bc6r",
            "num_sec": 67
        },
    ],
    "A": "1 2 3 4 5",
    "B": "2 3 4 3 1"
}
```

Listing 4: Example of message published on the topic "leader/task/vect_mult"

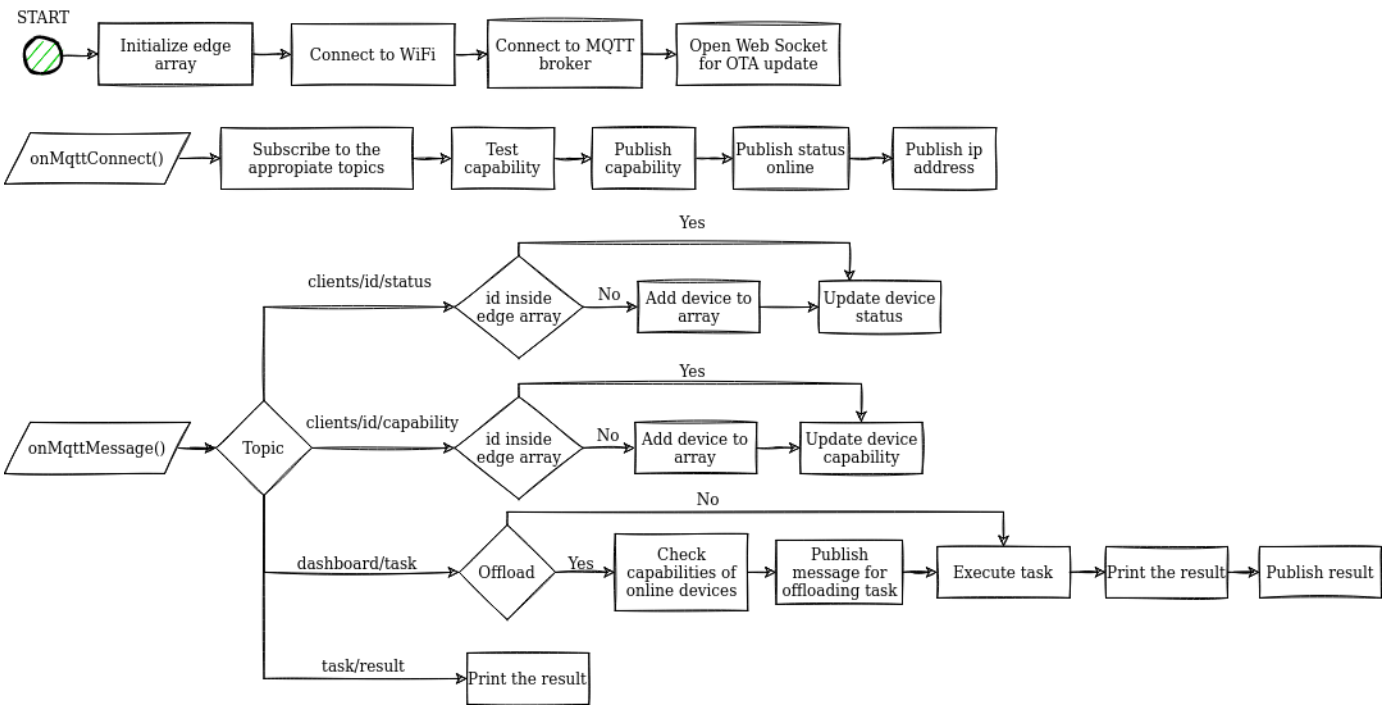

Figura 3: Flow chart of the device leader

capabilities and ip address of each device. The performance page shows several graphs, such as the workload the leader device is subjected to, the number of iterations that the edge nodes must perform and much more. Through a specific Node-Red module it was possible to run a small instance of SQLite, in order to save the information of all the devices connected to the MQTT broker.
The dashboard is useful for monitoring latency as the number of edge devices connected to the Mqtt broker increases. To do this, a python script has been written that runs 10 virtualized edge devices (computational problems are run on the laptop) with random capabilities. Each of these virtualized devices behaves exactly like a physical edge device on the network. To monitor the latency the following approach was followed: a mqtt client was written in python which, every two seconds, publishes a message on the topic "clients/latency/measure". Then the dashboard, which is subscribed to the previous topic, replies the message on the topic "dashboard/latency/measure"; at this point the python client, which is subscribed to this topic, is able to measure the time of sending and receiving the new message, and the latency is represented by half of this difference. This approach was followed as I didn't want to delegate the task of measuring latency to physical devices: either they run computational problems or they measure latency, they can't do both. If the client that measures the latency is running on the same machine that is also running the mqtt broker, then the latency seen is local, not affected by the data transmission time in the network. However it can be useful all the same, as it is a measure of how overloaded the mqtt broker is and how it is capable of forwarding messages.
Furthermore, from the dashboard it is possible to set a maximum latency. This means that if the measured latency exceeds a certain threshold value, then the system is too loaded and all running activities are stopped.

10 Final remarks

Even if the WLAN technology used in the project is WiFI, the most suitable in this application context would be ZigBee. In fact, the project was designed to support a large number of edge devices; thanks to the mesh network, each device becomes a node and a repeater, thus increasing the total signal range and optimizing the path of data exchanged between devices and hubs. While WiFi is able to support high data rates and long transmission distances, on the other hand it has a very high power consumption, unlike ZigBee. Since, the maximum size of the MQTT packet that the leader sends every second is about 2000 bytes (in case there are 20 edge devices connected and the input size of the computational problem is 500 bytes) and the data rate that Zigbee can reach is 250kbps, then the message can be delivered in about 10ms.
Table 2 shows all the Mqtt topics used in the project.

Riferimenti bibliografici

[1] URL: `https://nodered.org/`.

[2] URL: `https://code.visualstudio.com/`.

[3] URL: `https://platformio.org/`.

[4] URL: `https://arduinojson.org/`.

[5] URL: `https://github.com/espressif/arduino-esp32/tree/master/libraries/WiFi`.

[6] URL: `https://github.com/espressif/arduino-esp32/tree/master/libraries/Preferences`.

[7] URL: `https://github.com/marvinroger/async-mqtt-client`.

[8] URL: `https://github.com/me-no-dev/AsyncTCP`.

[9] URL: `https://github.com/me-no-dev/ESPAsyncWebServer`.

Topic	Publisher	Subscriber	Message Type	QoS
dashboard/task	dashboard	leader		1
leader/task/prime_num	leader	device dashboard		1
leader/task/word_count	leader	device dashboard		1
leader/task/vect_mult	leader	device dashboard		1
task/result	device leader	leader dashboard		1
dashboard/latency/measure	dashboard	python client		0
clients/latency/measure	python client	dashboard		0
clients/latency/value	python client	dashboard		0
clients/+/status	device	leader dashboard	retained, lastWill	1
clients/+/capability	device	leader dashboard	retained	1
leader/lead/status	leader	dashboard	retained, lastWill	1
leader/lead/capability	leader	dashboard	retained	1
leader/lead/ip	leader	dashboard	retained	1
clients/+/ip	device	dashboard	retained	1
$SYS/broker/bytes/received	broker	dashboard		
$SYS/broker/bytes/sent	broker	dashboard		
$SYS/broker/store/messages/bytes	broker	dashboard		

Tabella 2: Mqtt topic table

[10] URL: https://github.com/ayushsharma82/AsyncElegantOTA.

[11] URL: https://mosquitto.org/.