GIN206

Energy efficiency for camera surveillance

Bruno David, Gabriel Molina

Mai 2024

Contents

List of Figures

Listings

1 Introduction

In recent years, the proliferation of Internet of Things (IoT) devices has markedly contributed to advancements in diverse fields including surveillance, automation, and remote sensing. These devices are pivotal in capturing, processing, and transmitting data, making them indispensable components of intelligent systems. However, the widespread adoption of such systems also escalates their cumulative energy consumption, which poses significant challenges. Particularly in home automation, it is crucial that these systems do not become a substantial burden on household energy expenses. Therefore, optimizing IoT implementations to ensure they deliver adequate quality of service without excessive power consumption becomes essential.

This challenge leads us to explore the development and performance of a cost-effective, energy-efficient surveillance solution. The goal of this system is to be installed at strategic exterior locations such as front doors or gardens to detect and respond to movement or potential security breaches. Upon detecting such events, the camera system will be programmed to capture and send images to a dashboard, thereby alerting homeowners to potential anomalies.

The proposed solution employs an ESP32 CAM[1]. This module hosts an HTTP server, which interacts with a Raspberry Pi 3[2] that serves as the processing and communication hub. This setup not only facilitates real-time surveillance but also ensures minimal energy consumption by leveraging the low-power capabilities of the ESP32. Additionally, we conducted a detailed analysis of the power consumption on this device to validate our claims of energy efficiency. Indeed, we are going to study the impact of using low-resolution images for image processing and high-resolution images for displaying images to users instead of only high-resolution images for these two operations.

Through this project, we aim to demonstrate that it is possible to deploy effective surveillance systems that are both cost-effective and energy-efficient, ensuring that homeowners can enhance their security without incurring prohibitive expenses and seeing their energy consumption rise dramatically.

[1] `https://www.handsontec.com/dataspecs/module/ESP32-CAM.pdf`
[2] `https://datasheets.raspberrypi.com/rpi3/raspberry-pi-3-b-plus-product-brief.pdf`

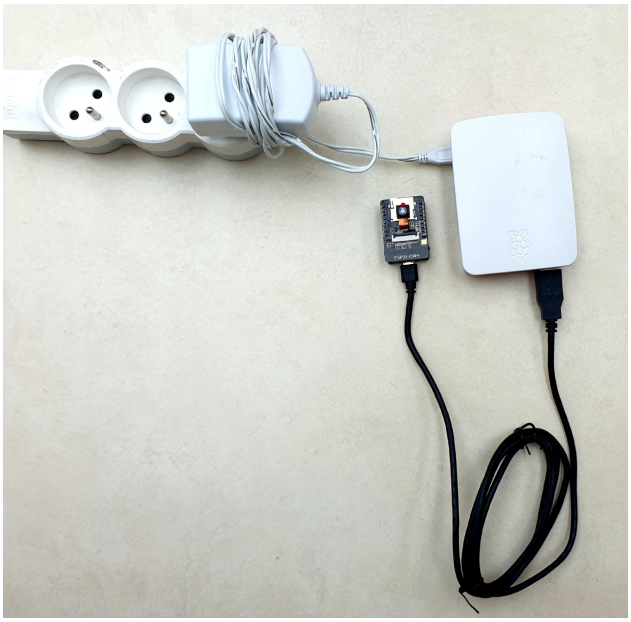

Figure 1: Set up of the camera and Raspberry PI 3

2 Camera implementation

Implementing a surveillance solution requires careful considerations of two primary constraints: energy efficiency and component limitations. In this project, we sought to optimize resource utilization within these constraints to establish a reliable monitoring system.

2.1 Preliminary tests

In the initial phase, we explored several approaches using the available devices, mindful that each component interaction could incrementally increase power consumption and reduce system efficiency.

Subsequently, we developed a motion detection algorithm based on differential analysis between successive frames. This technique operates by defining two image resolutions: a high-resolution setting at 800×600 pixels and a low-resolution setting at 320×240 pixels. While the ESP32 module offers precise results, its limited resources (520 KB SRAM + 4 MB PSRAM) required consideration of alternative solutions.

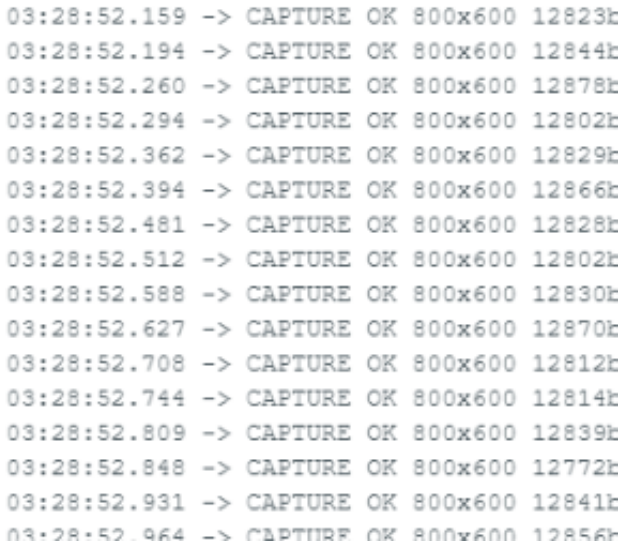

Figure 2: Motion capture process comparing consecutive high-resolution frames

To circumvent continuous motion detection — which would for instance exhaust the ESP32's resources — we proposed intermittent analysis at fixed intervals, thereby conserving computational power. However, this measure alone was insufficient.

2.2 Solution finally adopted

Therefore, we introduced further optimizations: reducing the frame rate to limit the number of frames processed per second, decreasing resolution during motion detection to minimize data processing, and targeting specific areas within frames where motion is most likely to occur.

Ultimately, integrating a Raspberry Pi 3 served to alleviate the ESP32's workload. We established a WebServer on the ESP32, providing URLs for accessing both high and low-resolution images. This setup allowed for efficient resource allocation, with error handling routines in place to manage resolution changes seamlessly.

The implementation showcases a balance between energy efficiency and performance by dynamically adjusting the camera's resolution based on the surveillance requirements. By leveraging the ESP32's low-power operation mode and the intelligent resolution management, the system promises to deliver a reliable security solution without imposing excessive energy demands.

```
#include <WebServer.h>
#include <WiFi.h>
#include <esp32cam.h>

const char* WIFI_SSID = "YOUR_SSID";
const char* WIFI_PASS = "YOUR_PASSWORD";

WebServer server(80);

static auto loRes = esp32cam::Resolution::find(320, 240);
static auto hiRes = esp32cam::Resolution::find(800, 600);
void serveJpg()
{
  auto frame = esp32cam::capture();
  if (frame == nullptr) {
    Serial.println("CAPTURE FAIL");
    server.send(503, "", "");
    return;
  }
  Serial.printf("CAPTURE OK %dx%d %db\n", frame->getWidth(), frame->getHeight
    (),
                static_cast<int>(frame->size()));

  server.setContentLength(frame->size());
  server.send(200, "image/jpeg");
  WiFiClient client = server.client();
  frame->writeTo(client);
}

void handleJpgLo()
{
  if (!esp32cam::Camera.changeResolution(loRes)) {
    Serial.println("SET-LO-RES FAIL");
  }
  serveJpg();
}

void handleJpgHi()
{
  if (!esp32cam::Camera.changeResolution(hiRes)) {
    Serial.println("SET-HI-RES FAIL");
  }
  serveJpg();
}


void  setup(){
```

```
47    Serial.begin(115200);
48    Serial.println();
49    {
50      using namespace esp32cam;
51      Config cfg;
52      cfg.setPins(pins::AiThinker);
53      cfg.setResolution(hiRes);
54      cfg.setBufferCount(2);
55      cfg.setJpeg(80);
56
57      bool ok = Camera.begin(cfg);
58      Serial.println(ok ? "CAMERA OK" : "CAMERA FAIL");
59    }
60    WiFi.persistent(false);
61    WiFi.mode(WIFI_STA);
62    WiFi.begin(WIFI_SSID, WIFI_PASS);
63    while (WiFi.status() != WL_CONNECTED) {
64      delay(500);
65    }
66    Serial.print("http://");
67    Serial.println(WiFi.localIP());
68    Serial.println("  /cam-lo.jpg");
69    Serial.println("  /cam-hi.jpg");
70
71    server.on("/cam-lo.jpg", handleJpgLo);
72    server.on("/cam-hi.jpg", handleJpgHi);
73
74    server.begin();
75  }
76
77  void loop()
78  {
79    server.handleClient();
80  }
```

Listing 1: Code for the camera

3 Raspberry-PI implementation

3.1 Raspberry connections

For our simulation, the Raspberry PI is powered from the mains. It also acts as the supply for the camera.

3.2 Code for the Raspberry

Our project implemented two solutions in order to compare them: the first solution will request high-resolution images from the HTTP server on the camera, analyze them and finally display them if needed on the Thingsboard, while the second will analyze low-resolution images and request high-resolution images only if they need to be displayed, in other words if a face or movement has been detected.

```
import cv2
from matplotlib.pyplot import plot, savefig, legend, close, ylabel, xlabel
import numpy as np
from time import sleep, time
from threading import Thread
from psutil import cpu_percent
import warnings

FACE_CASCADE = cv2.CascadeClassifier('./haarcascade_frontalface_default.xml')
URL_HIGH = 'http://192.168.43.5/cam-hi.jpg'
warnings.filterwarnings("ignore")

def getenergy() :
    cpu = []
    times = []
    t_begin = time()
    while True :
        cpu.append(cpu_percent())
        times.append(time() - t_begin)
        mean = round(sum(cpu) / len(cpu), 2)
        plot(times, cpu, label="CPU used (%)")
        plot(times, [mean for i in range(0, len(times))], label=f"CPU used in average: {mean}%", ls='--')
        xlabel("Time (s)")
        ylabel("CPU used (%)")
        legend()
        savefig('cpu_high_only.pdf')
        close()
        sleep(1)

def getsimilarity(image1, image2) :
    image1_gray = cv2.cvtColor(image1, cv2.COLOR_BGR2GRAY)
    image2_gray = cv2.cvtColor(image2, cv2.COLOR_BGR2GRAY)
    image1_gray = cv2.GaussianBlur(image1_gray, (21, 21), 0)
    image2_gray = cv2.GaussianBlur(image2_gray, (21, 21), 0)
    delta = cv2.absdiff(image1_gray, image2_gray)
    thresh = cv2.threshold(delta, 25, 255, cv2.THRESH_BINARY)[1]
    thresh = cv2.dilate(thresh, None, iterations=2)
```

```
    contours, _ = cv2.findContours(thresh.copy(), cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

    for contour in contours:
        if cv2.contourArea(contour) < 500:
            continue  # Small changes might be noise, ignore them
        return True  # Motion detected
    return False

def facedetection(image) :
    gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    faces = FACE_CASCADE.detectMultiScale(gray_image, scaleFactor=1.1, minNeighbors=5, minSize=(30,30))
    return faces

def get_high_image() :
    try :
        response = requests.get(URL_HIGH)
        if response.status_code == 200 :
            image_array = np.frombuffer(response.content, dtype=np.uint8)
            image = cv2.imdecode(image_array, cv2.IMREAD_COLOR)
            return image

    except Exception as e:
        print(f'Error while getting image : {e}')

    return False

def save_face(image, faces) :
    for (x, y, w, h) in faces :
        cv2.rectangle(image, (x, y), (x+w, y+h), (0, 255, 0), 2)
    cv2.imwrite('./face.jpg', image)

def save_movement(image) :
    cv2.imwrite('./movement.jpg', image)

if __name__ == '__main__' :
    energy_thread = Thread(target = getenergy)
    energy_thread.start()
    prev_image = False
    curr_image = False
    while True :
        t_begin = time()
        image = get_high_image()
        if type(image) != bool :
            prev_image = curr_image
            curr_image = image
            if type(curr_image) != bool :
                faces = facedetection(curr_image)
                movement_detected = False
                if type(prev_image) != bool :
                    movement_detected = getsimilarity(prev_image, curr_image)
                if movement_detected or len(faces) > 0 :
                    if movement_detected :
                        save_movement(curr_image)
                    if len(faces) > 0 :
                        save_face(curr_image, faces)
        time_to_sleep = 3 - (time() - t_begin)
        if (time_to_sleep > 0) :
            sleep(time_to_sleep)
```

Listing 2: Code for the Raspberry PI - high-resolution images

Listing 2 is the program we implemented on the Raspberry PI 3 to retrieve and analyze high-resolution images. This program will also register them if needed in order to communicate them to the Thingsboard. It also monitor CPU consumption.

Here are the additions needed to implement the second method:

```
# URL to get low resolution images

URL_LOW = 'http://192.168.43.5/cam-lo.jpg'

# Coefficients to adapt the rectangle dimensions in case of face detection
    from low resolution to high resolution image

ALPHA = 800/320
BETA = 600/240

def save_face(image, faces) :
    for (x, y, w, h) in faces :
        x = int(x*ALPHA)
        y = int(y*BETA)
        w = int(w*ALPHA)
        h = int(h*BETA)
        cv2.rectangle(image, (x, y), (x+w, y+h), (0, 255, 0), 2)
    cv2.imwrite('./face.jpg', image)

# Method to get low-resolution image from the camera

def get_low_image() :
    try :
        response = requests.get(URL_LOW)
        if response.status_code == 200 :
            image_array = np.frombuffer(response.content, dtype=np.uint8)
            image = cv2.imdecode(image_array, cv2.IMREAD_COLOR)
            return image

    except Exception as e:
        print(f'Error while getting image : {e}')
    return False

# New main code

if __name__ == '__main__' :
    energy_thread = Thread(target = getenergy)
    energy_thread.start()
    prev_image = False
    curr_image = False
    while True :
        t_begin = time()
        image = get_low_image()
        if type(image) != bool :
            prev_image = curr_image
            curr_image = image
            if type(curr_image) != bool :
                faces = facedetection(curr_image)
                movement_detected = False
                if type(prev_image) != bool :
                    movement_detected = getsimilarity(prev_image, curr_image)
                if movement_detected or len(faces) > 0 :
                    high_image = get_high_image()
```

```
53                      if len(faces) > 0 :
54                          save_face(high_image, faces)
55                      if movement_detected :
56                          save_movement(high_image)
57          time_to_sleep = 3 - (time() - t_begin)
58          if (time_to_sleep > 0) :
59              sleep(time_to_sleep)
```

Listing 3: Code for the Raspberry PI - modifications to also use low-resolution images

3.2.1 Image acquisition

To get the image from the camera in the python script, we requested the HTTP server described in section 2.

3.2.2 Image processing

To make a movement analysis on the images, we implemented a function named 'getsimilarity'. It converts the images given as arguments to grayscaled images, and then adds a Gaussian blur to reduce the image noise. Finally, a threshold[3] is used to convert the absolute difference of the two images into a binary image. To detect some movement, the function analyses the contours of the binary image.

To make a face detection, we implemented a function named 'facedetection'. It uses the 'CascadeClassifier' function[4] from cv2 to detect a face or not. This function returns a list containing the information needed to draw a rectangle around the detected face.

3.2.3 CPU consumption

To check the Raspberry's CPU consumption, we have implemented a function that uses the 'psutil.cpu_percent' function[5] to obtain this data every second and matplotlib.pyplot library[6] to construct a graph of consumption versus time.
This code is executed in a separated thread so that it runs in the same time as the main program.

3.2.4 Main functions of the code

In the listing 2, the main function first starts the 'getenergy' function in another thread. In the while loop, it asks the camera to get a high-resolution image to analyse it in order to detect face or movement, comparing it to the previous one. If face or movement is detected, it calls the appropriate function to save this high-resolution image so that it can be send to the dashboard. The while loop executes its code every three seconds.

[3] https://docs.opencv.org/4.x/d7/d4d/tutorial_py_thresholding.html
[4] https://docs.opencv.org/3.4/db/d28/tutorial_cascade_classifier.html
[5] https://psutil.readthedocs.io/#psutil.cpu_percent
[6] https://matplotlib.org/stable/tutorials/pyplot.html

In the listing 3, the main function is similar to the other. The few changes are that it firstly asks a low-resolution image to detect movement or face and then asks a high-resolution image only if one of these events is detected.

3.2.5 Connection to the Thingsboard

We implemented a HTTP server to provide the registered images to the Thingsboard, that will request it when it needs another image.

```
from flask import Flask, send_file, jsonify

app = Flask(__name__)

@app.route('/cam_move.jpg', methods=['GET'])
def get_movement():
    try:
        # Envoyer l'image
        return send_file('./movement.jpg', mimetype='image/jpeg')
    except Exception as e:
        return jsonify({"message": f"Erreur lors de la r cup ration de l'
    image : {e}"}), 500

@app.route('/cam_face.jpg', methods=['GET'])
def get_face():
    try:
        #Enoyer l'image
        return send_file('./face.jpg', mimetype='image/jpeg')
    except Exception as e:
        return jsonify({"message": f"Erreur lors de la r cup ration de l'
    image : {e}"}), 500

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

Listing 4: Code for the Raspberry PI - HTTP server

4 Thingsboard implementation

To display images on the Thingsboard, we added two html cards which retrieves the high-resolution images for movement and face detection. The requests are made every three seconds as the python code processes on images every three seconds explained in section 3.

```
<div class='card'>
    <iframe id="face" width="850" height="650" src="http
    ://192.168.43.141:5000/cam_face.jpg" frameborder="0" allowfullscreen>
    </iframe>
</div>

<script>
    function refreshFrame() {
        var frame = document.getElementById("face");
        frame.src = frame.src;
    }

    setInterval(refreshFrame, 3000);
</script>
```

Listing 5: Code for the Thingsboard - images for face detection

```
<div class='card'>
    <iframe id="movement" width="850" height="650" src="http
    ://192.168.43.141:5000/cam_move.jpg" frameborder="0" allowfullscreen>
    </iframe>
</div>

<script>
    function refreshFrame() {
        var frame = document.getElementById("movement");
        frame.src = frame.src;
    }

    setInterval(refreshFrame, 3000);
</script>
```

Listing 6: Code for the Thingsboard - images for movement detection

In our implementation, the ESP32 camera, the Raspberry-PI 3 and the computer displaying the Thingsboard were in the same subnet. Therefore, there is only 192.168.0.0/16 addresses in our codes. In practice, only the ESP32 camera and the Raspberry-PI 3 must be in the same subnet.

For instance, users can connect the ESP32 camera and the Raspberry-PI 3 to their private wifi network and make port redirection so that the Thingsboard can request the HTTP server running on the Raspberry without being at home.

5 Results

5.1 Thingsboard display

Our solution enables users to observe the movement and faces of people passing in front of the camera on their Thingsboard.

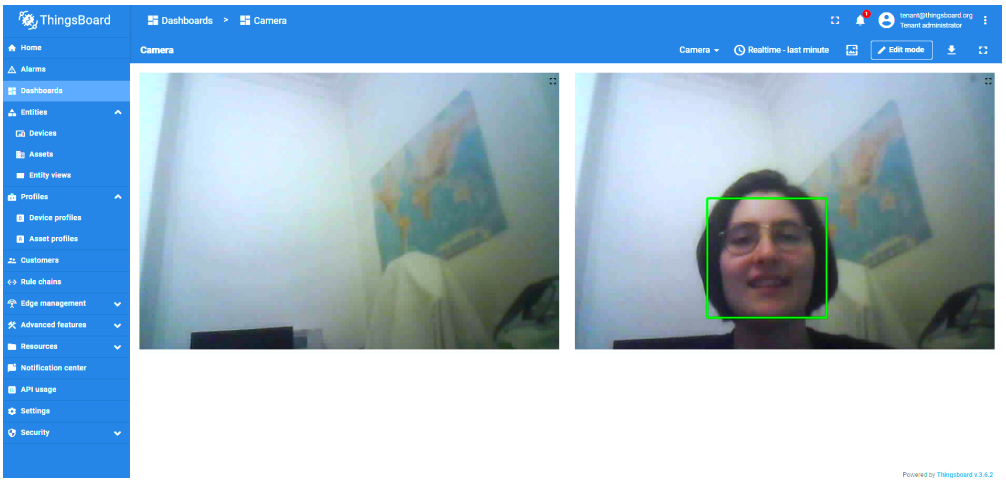

Figure 3: Capture of the Thingsboard with movement (left) and face (right) detection

5.2 Energy savings on the network

The solution using low- and high-resolution images is more energy-efficient concerning the network. We can get the size of camera HTTP server responses in the python code by adding `print(len(response.content))` after receiving the images.

```
gin206@raspberrypi:~/projet $ python3 master_low_and_high.py
Low image  - content (bytes) = 3230
High image - content (bytes) = 13011
Low image  - content (bytes) = 3034
High image - content (bytes) = 12665
Low image  - content (bytes) = 3016
High image - content (bytes) = 12315
Low image  - content (bytes) = 3008
High image - content (bytes) = 12321
Low image  - content (bytes) = 3377
High image - content (bytes) = 12523
Low image  - content (bytes) = 3047
High image - content (bytes) = 13055
Low image  - content (bytes) = 3424
High image - content (bytes) = 12588
Low image  - content (bytes) = 3491
High image - content (bytes) = 12408
Low image  - content (bytes) = 3017
High image - content (bytes) = 12124
Low image  - content (bytes) = 3002
High image - content (bytes) = 12401
Low image  - content (bytes) = 3254
High image - content (bytes) = 12659
Low image  - content (bytes) = 3059
High image - content (bytes) = 13641
Low image  - content (bytes) = 3026
High image - content (bytes) = 12308
Low image  - content (bytes) = 2997
High image - content (bytes) = 11950
```

Figure 4: Size of camera HTTP server responses when requesting low- and high-resolution images

We can see that the responses contains around 4 times less bytes for low-resolution images. The fact that only low-resolution images are required for image analysis in solution 2 is therefore an advantage when it comes to energy consumption on the network. Indeed, when no movement or face is detected, it avoids overloading the network with requests whose responses contain 4 times more bytes.

5.3 Raspberry's CPU consumption

To obtain the following results, we tested our codes placing the camera in a way to always detect movement and face. It can be considered as the worst case for the solution using low- and high-resolution images because it must retrieve these two images every three seconds. For this code the best case would be an absence of face and movement detection so there is no need to retrieve high-resolution images.

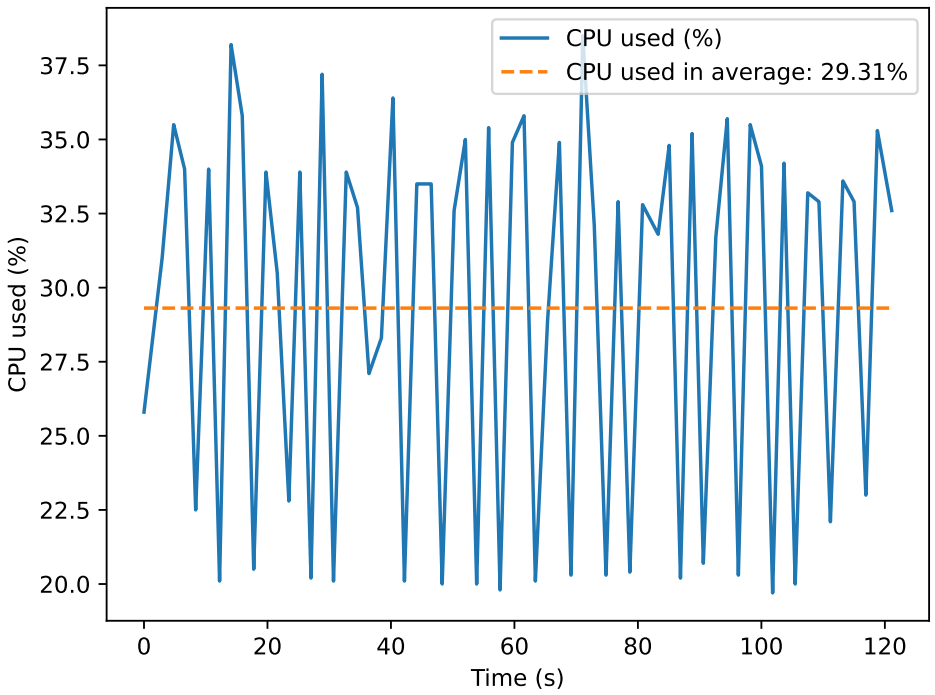

Figure 5: CPU utilization for the code using only high-resolution images - solution 1

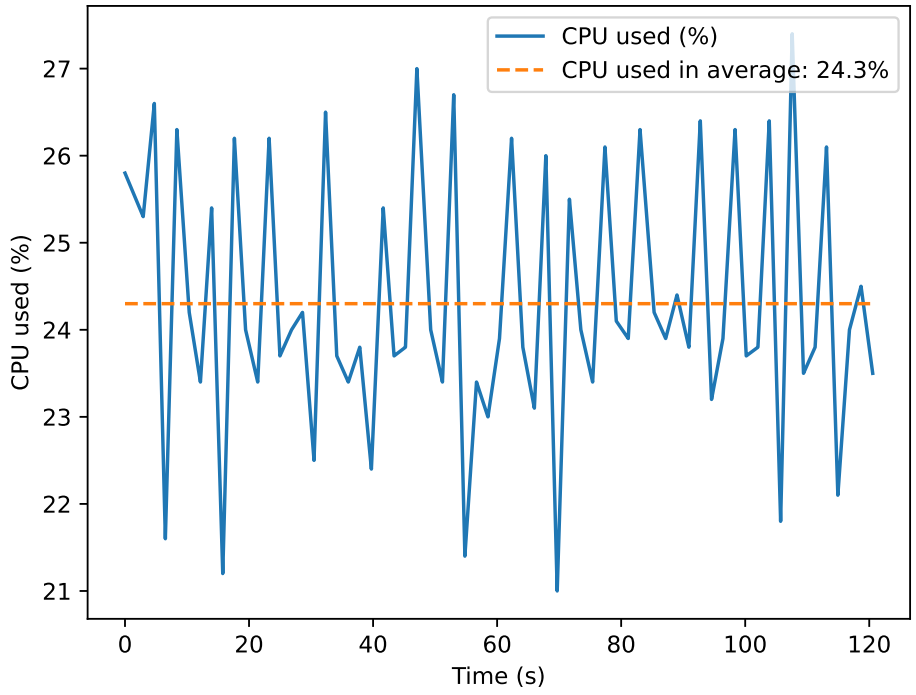

Figure 6: CPU utilization for the code using low- and high-resolution images - solution 2

On these two figures, we can see that the average of CPU utilization is lower for the second solution. The average CPU consumption is therefore 5,01 points lower for the second solution. Moreover, the main peaks are less prominent in solution 2.

All these observations can be explained by the image analysis that is carried out on low-resolution images in solution 2, containing $\frac{600*800}{240*320} = 6.25$ times fewer pixels than the high-resolution images used for image processing in solution 1. As the only changes we made between our two solutions are the processing on low- or high-resolution images, we can therefore consider that the CPU consumption is linked with the energy consumption[7]. So solution 2 is more energy efficient than solution 1 in terms of CPUs used.

[7] `https://www.pidramble.com/wiki/benchmarks/power-consumption`

6 Conclusion

In conclusion, the solution that performs image analysis on low-resolution images and only requires high-resolution images in the event of motion or face detection is more energy-efficient. In fact, when no motion or face is detected, query responses contain fewer bytes, reducing network consumption. The Raspberry PI 3 also uses less CPU for low-resolution image analysis than for high-resolution images. This saves energy too, and therefore reduces users' bills.

In terms of network savings, of course, it's important to keep in mind that motion or face detection is an occasional event, so it's rare to need to request a high-resolution image from the camera server.

7 Sources

1. ESP32-CAM Datasheet: `https://www.handsontec.com/dataspecs/module/ESP32-CAM.pdf`
2. Raspberry PI 3 Datasheet: `https://datasheets.raspberrypi.com/rpi3/raspberry-pi-3-b-plus-product-brief.pdf`
3. OpenCV Thresholding Tutorial: `https://docs.opencv.org/4.x/d7/d4d/tutorial_py_thresholding.html`
4. OpenCV Cascade Classifier Tutorial: `https://docs.opencv.org/3.4/db/d28/tutorial_cascade_classifier.html`
5. Psutil Documentation: `https://psutil.readthedocs.io/en/latest/#psutil.cpu_percent`
6. Matplotlib Tutorial: `https://matplotlib.org/stable/tutorials/pyplot.html`
7. Power Consumption Benchmarks: `https://www.pidramble.com/wiki/benchmarks/power-consumption`