Intelligent platform for access tracking in research laboratories

Author: Smetanca Ioan
Coordinator: Conf.dr.ing. Octavian Cornea
Faculty of Electrical and Energetical Engineering, Dept. of Electrical Engineering, UPT

Abstract— **In this paper, we aim to develop a real-time system design to monitor laboratory access on a Raspberry Pi 3B+ using a webcam. The purpose of this work is to monitor the presence of students in real time so that everything is automated and there is no longer a need to write down student names on paper.**

To implement this system, OpenCV was used for image processing, the Gmail API for sending alerts when unknown persons are detected, and Node-Red Dashboard with Flask for real-time monitoring of student entry and exit times. Port forwarding and Ngrok enabled external access during testing.

The functionality of this system is divided into several steps. It collects images with the purpose of identifying faces. These images are then used in order to map the encodings to each unique individual.

INTRODUCTION

Access monitoring in dedicated research laboratories is sometimes necessary to restrict presence to individuals who are authorized to work in those spaces. The presence of unauthorized people can lead to damage of sensitive experimental setups, compromise of experiments, and other issues. This article presents such a platform for monitoring access to a research laboratory, which uses facial recognition technology to identify the individuals entering the lab. This technology is primarily based on the detailed analysis of facial features to identify or verify a person's identity.This method combines multiple techniques and machine learning algorithms with computer vision, simultaneously transforming a facial image into a series of unique numerical indicators, thus allowing precise facial comparisons depending on the system's performance and the number of training models used.

This technology operates in several stages. First, face detection must be as clear as possible (the system identifies the presence of a face in an image or video stream), followed by the extraction of that face's features (using specialized algorithms capable of quickly recognizing contours and key features), and then the data are compared with the training data.If a face is detected in that video stream or image, it is then processed to extract a set of essential characteristics — from the distance between the eyes and the shape of the nose to the fine textures of the skin.

The latest neural networks (such as FaceNet or ArcFace) convert this information into a fixed-size numerical vector [1]. Once the data is converted into vectors, these vectors are compared with those in an existing database using mathematical methods such as Euclidean distance or cosine similarity. This is also known, if the differences are below a certain threshold, it can be concluded that the identified faces belong to the same person. We can use this technology in payment systems, and banks can utilize it to verify identities during financial transactions, thereby reducing the risk of fraud.

II.REAL-TIME FACIAL RECOGNITION ON RASPBERRY PI WITH OPENCV AND PYTHON

The Raspberry Pi is a low-cost SBC (Single Board Computer), which means a fully functional computer integrated on a single circuit board, available at an affordable price. Various electronic components (sensors, buttons, relays) can of course be connected to it, allowing us to create a wide range of electronic projects. It was developed by the Raspberry Pi Foundation, a NGO based in the United Kingdom [2].

For this system, we will use a Raspberry Pi 3B+, along with a 64 GB Class 10 Kingston microSD card. It will be necessary to install the Raspberry Pi OS on this microSD card (a USB stick can also be used if the microSD card is not available).

After installing the Raspberry Pi OS, we can proceed with installing the necessary dependencies for facial recognition, such as OpenCV.

III INSTALLING THE NECESSARY DEPENDENCIES FOR FACIAL RECOGNITION

In this step, we will install the required dependencies: *OpenCV*, *face-recognition* and *imutils*. We will then make some modifications to the *swapfile* to increase the swap space so that our system does not crash during the installation of these libraries.

For daily use, a swap memory of 100MB is usually sufficient. The swap space is used when the physical memory (RAM) is full. If the system needs more resources and the RAM is full, inactive memory pages are moved to the swap memory, which helps prevent the system from crashing. It is important to note that swap memory is slower than RAM [3].

However, for building OpenCV, we will need at least 5.8GB of memory. The term "OpenCV" stands for "Open Source Computer Vision." Its architecture consists of software, a database, and pre-programmed plugins that support the integration of computer vision applications. We will also use

the *HAAR CASCADE* algorithm, which is an efficient method for object detection. It is a ML-based approach which trains a cascade of classifiers on a large number of positive and negative images, where a positive image has at least a face detected in it and a negative image has no faces detected. It is designed to detect objects from different angles and perspectives. Lastly, we will install *imutils*, a Python package that provides simple functions to ease basic image processing operations in OpenCV. We will move on to changing the swap memory. We open a terminal and type *sudo nano /sbin/dphys-swapfile*, and at change the value of *CONF_MAXSWAP* to 5120. This allocates 5GB of microSD's disk space to be designated as swap memory. If we add the 1GB of RAM memory and the swap memory allocated we get approximately 6GB. The second command we type in the terminal is sudo nano /etc/dphys-swapfile, and for CONF_SWAPSIZE, we chose the value 5120. We can check in the terminal using the command free -m to see how much physical and swap memory we have.

Finally, we will restart the system using the command sudo reboot and run the following commands from the table below:

Table1 – Necessary dependencies

1	sudo apt install cmake build-essential pkg-config git
2	sudo apt install libjpeg-dev libtiff-dev libjasper-dev libpng-dev libwebp-dev libopenexr-dev
3	sudo apt install libavcodec-dev libavformat-dev libswscale-dev libv4l-dev libxvidcore-dev libx264-dev libdc1394-22-dev libgstreamer-plugins-base1.0-dev libgstreamer1.0-dev
4	sudo apt install libgtk-3-dev libqtgui4 libqtwebkit4 libqt4-test python3-pyqt5
5	sudo apt install libatlas-base-dev liblapacke-dev gfortran
6	sudo apt install libhdf5-dev libhdf5-103
7	sudo apt install python3-dev python3-pip python3-numpy
8	git clone https://github.com/opencv/opencv.git
9	git clone https://github.com/opencv/opencv_contrib.git
10	mkdir ~/opencv/build
11	cd ~/opencv/build
12	cmake -D CMAKE_BUILD_TYPE=RELEASE \
13	-D CMAKE_INSTALL_PREFIX=/usr/local \
14	-D OPENCV_EXTRA_MODULES_PATH=~/opencv_contrib/module s \
15	-D ENABLE_NEON=ON \
16	-D ENABLE_VFPV3=ON \
17	-D BUILD_TESTS=OFF \
18	-D INSTALL_PYTHON_EXAMPLES=OFF \
19	-D INSTALL_PYTHON_EXAMPLES=OFF \
20	-D CMAKE_SHARED_LINKER_FLAGS=-latomic \
21	-D BUILD_EXAMPLES=OFF ..
22	make -j$(nproc)
23	sudo make install
24	sudo ldconfig
25	pip install face-recognition
25	pip install imutils

IV TRAINING THE FACIAL RECOGNITION MODEL

We will run this command to obtain all the necessary files/folders from Ivan13s repository:

git clone https://github.com/Ivan13s/facial_recognition

Then, we will go to the dataset folder and create a new folder. The name of this folder must be our first or last name. In the headshots.py file, we will open it with the *Geany* text editor and, at line 3, we will change that name with our own name [4].

To run this file, we will need to click on the icon that looks like a paper plane.

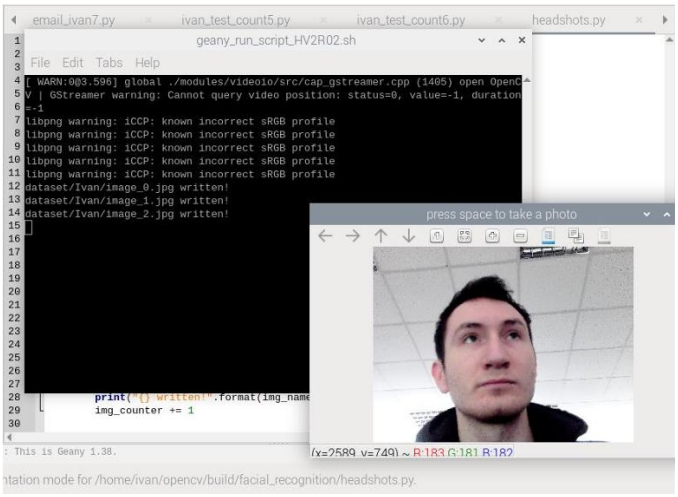

Fig1. Capturing Facial Images for Recognition: Data Collection in Progress

It is necessary to press the space bar to take these pictures. All these photos will be saved in the dataset/Ivan folder because we have previously the name 'Ivan' in headshots.py on line 3, and the folder we created (the new folder) was also named 'Ivan'. If we want to take pictures for other people, we will repeat the same process for each. It's important to mention that it is necessary to have as many photos from as many different angles as possible to increase the chances that the photos we will train on will have a much higher accuracy. Now we have collected these photos and successfully stored them in the dataset folder for each person.

To train the respective models, we will go to the terminal and run the command *cd opencv/build/facial_recognition*. Once we are in the facial_recognition folder, we will run the command python train_model.py. From this moment, a window will open, and we will receive a message indicating that the model training process has started. If we encounter an error stating that a specific module is missing, we will need to create a virtual environment and install all the dependencies mentioned in the previous step. In the terminal, we type *python -m venv venv*, and then we will follow the steps provided for installing the dependent libraries. It is ideal to create this virtual environment from the beginning and install all the modules in it to avoid affecting the system when we install the modules. To activate the virtual environment on *Raspberry Pi(* Debian/Linux), we will type *source venv/bin/activate*, and to exit that environment, we simply type *deactivate*. It's worth mentioning that the

train_model.py file will analyze the photos in the dataset folder where we created a new folder and collected the photos. Once the training is complete, this file (*train_model.py*) will generate another file called *encodings.pickle*, which contains the face identification criteria. We use a detection method called *HOG (Histogram of Oriented Gradients)* along with the *Haar Cascade* algorithm. *HOG* is a feature descriptor commonly used in computer vision and image processing for object detection. To verify if the model has been trained correctly, we run the following command: *python facial_req.py*. If it doesn't work, try first entering the virtual environment and running it from there, or make sure you are in the folder where the *facial_req.py* file is located.

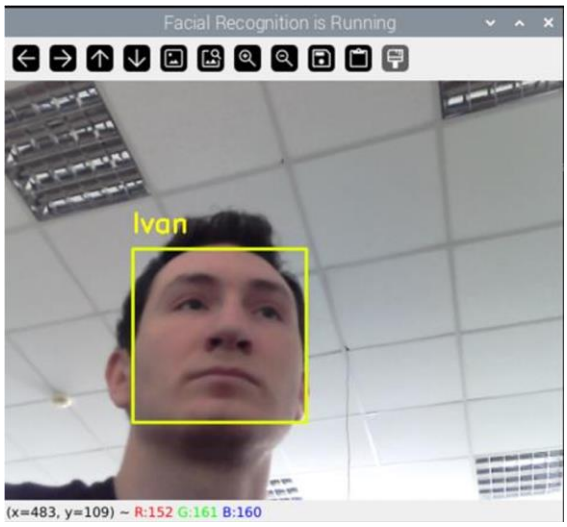

Fig2. Real-time facial recognition testing

Now that we see this is working, we will try to send an email in case we are detected as an unknown person. To do this, we will use the *GMAIL API*.

V. USING THE GMAIL API TO SEND EMAILS

To be able to use the Gmail API, we will need the credentials from the account we want to send the email from. To do this, open a browser and go to *console.cloud.google.com*. If you're not logged in, you'll need to log in first. Once logged in, click on "*Select Project*" and create a new project (*New Project*). We created a new project named *LABD003*.

The second step is to create an OAuth consent screen. Click the "Navigation Menu" (the three horizontal lines in the top-left corner), then go to *APIs & Services → OAuth consent screen*. For User Type, select External, then click Create. After that, name your application *LABD003 APP*. In the Developer Contact Information section, you can enter your own email address. Once you've filled out the required fields, proceed to the Testing section and click *Publish App*. Next, go back to *APIs & Services*, but this time navigate to *Credentials → CREATE CREDENTIALS* to generate your *credentials*. These credentials are necessary to authenticate and allow your Python script to send emails. Without them, sending emails through the *Gmail API* from a Python script will not be possible. For the *Application Type*, we will choose *Desktop App*, and the name we selected was *D003LAB1*. Finally, we click *Create* and we will receive a *JSON file* with our credentials, which we will download (Download JSON). The *.json file* we downloaded will be renamed to *D003LAB1.json*.

After renaming it, the next steps are: we go back to *APIs & Services*, then to *Library*, and in the search bar we look for *GMAIL API*, which we enable, and we do the same *for Google Drive API*. In the next step, we will need to obtain a *token.json* using the credentials we just downloaded. But before doing this, we will install the necessary libraries: *pip install --upgrade google-api-python-client google-auth-httplib2 google-auth-oauthlib* [4].

Now that we have installed all the necessary libraries, we will run the command *python for_token.py* (this file alongside all the source code are provided on *GitHub*) and we will obtain the *token.json* file. If we are not able to obtain this *token.json* on the *Raspberry Pi*, it will be necessary to run the script on another device and then move *token.json* to the *Raspberry Pi*. This *token.json* should be placed in the project root [5].

VI.NODE-RED DASHBOARD

Node-RED is a flow-based programming tool that was initially developed by *IBM* and is now open-source. It is generally used for connecting hardware devices and is very suitable for IoT (Internet of Things) applications. In this step, we will use *Node-RED* to create the interface for the scripts we are running in the background. Specifically, we will add buttons, create folders, and upload photos, so that we no longer need to use the terminal for these actions, and the interface can be available to users if needed (or if someone wants this feature) [6].

We will begin by installing *Node-RED* on the *Raspberry Pi* using the following command in the terminal:

*bash <(curl -sL https://raw.githubusercontent.com/node-red/linux-installers/master/deb/update-nodejs-and-nodered).*In the terminal, we will also type *ifconfig* to check the IP address. Once we see the IP address (next to inet, this will show the IP assigned to the *Raspberry Pi*), we will be able to connect to it via *SSH* using the command *ssh ivan@192.168.1.120* in our case. This can be done from a terminal or from a Windows machine using *PuTTY*. However, before we can connect via *SSH*, it is necessary to make sure that *SSH* communication is enabled on the *Raspberry Pi*. To do this, we go to the settings in the top left corner (there's a raspberry icon there), then to *Raspberry Pi Configuration → Interfaces*, and there we will find *SSH*, which we must activate if it is not already enabled.

It is not mandatory to connect via *SSH* at this point, but in order to have remote control and to be able to troubleshoot in case something goes wrong, it is a necessary step. To improve security and protect the network, we can enable the firewall using the command *sudo ufw enable*, and in order to disable it we use *sudo ufw disable*. If the firewall is enabled, we can allow communication on specific ports or for specific protocols, for example: *sudo ufw allow 22/tcp. T*his means communication is allowed on port 22, which is used for *SSH*, even when the

firewall is active.

Now that we've briefly discussed SSH and the firewall, we can proceed to start the *Node-RED* server using the command node-red-start. To stop it, we use node-red-stop, and to check the log, we type node-red-log in the terminal.Once the *Node-RED* server is running, it will display the URL we need to access. In our case: *127.0.0.1:1880* or you can directly type *192.168.1.120:1880*. To access the graphical user interface (UI), go to *192.168.1.120:1880/ui*. To secure the editor, visit *https://nodered.org/docs/user-guide/runtime/securing-node-red* and follow the steps provided.

Just like we needed to install a list of certain libraries and dependencies on the *Raspberry Pi*, the same is true for *Node-RED*. In the *Node-RED editor* (or development environment), go to the top *right menu* (three lines) → *Manage Palette*, then go to the *Install tab* and search for *node-red-dashboard*. Several options will appear, and you can install the one you want by clicking the Install button next to it. Here is a table with the options we to install for this project. Some are optional or were only used for testing.

Table2 - List of modules and contributions in Node-Red

1	@background404/node-red-contrib-python-venv
2	node-red
3	node-red-contrib-buffer-parser
4	node-red-contrib-chunks-to-lines
5	node-red-contrib-file-upload
6	node-red-contrib-filesystem
7	node-red-contrib-fs-ops
8	node-red-contrib-function
9	node-red-contrib-pythonshell
10	node-red-contrib-queue-gate
11	node-red-contrib-require
12	node-red-contrib-ui-upload
13	node-red-dashboard
14	node-red-iot-mqtt-api
15	node-red-node-base64
16	node-red-node-pi-gpio
17	node-red-node-ping
18	node-red-node-random
19	node-red-node-serialport
20	node-red-node-smooth
21	node-red-node-ui-list
22	ui-upload-mf

After installing the necessary components in the development environment (editor), we can begin creating the button for starting the script (*ivanc12.py*, the one with *Flask* or *ivan_c7bun.py*). At this point, we will use the script with Flask. We will start by taking a button from the Dashboard section, which can be found almost at the bottom left, where all the necessary UI nodes are located. We will drag a button node onto the flow, and this button will be connected to a *pythonshell in* node. This node is found under the input section of the network. A debug node will be attached to the `pythonshell in` in order to see inspect the output. After linking the 3 nodes (*button → pythonshell in → debug*), we will configure them. For the button, we will change the label to *START FACIAL RECOGNITION*, for the *pythonshell in* node, under py file, we will put the corresponding path of the script, which in our case is:

/home/ivan/opencv/build/facial_recognition_var2/ivanc12.py

and under *Virtual Environments Path*, we will specify /home/ivan/venv.For the debug node, we will check all three boxes: debug window, system console, node status, and for output, we will set it to *msg.payload*.

Next, we will try to connect 3 buttons to a switch, the switch to a function, and the function to a *TCP request node*. So, we will have 3 buttons (we take them from where we got the first one) and rename them as follows: *START SERVER*, START *TRAIN MODEL*, STOP. The first button will send a payload with the string "b", the second one with "t", and the third one with "q". All three buttons will be connected to the switch, where we will change the Property to *msg.payload*. Here we will also add 3 rules by clicking (add an item), which will be of type contains and will contain the 3 strings *(b, t, q)* in the order mentioned earlier. The function code can be found on GitHub or downloaded from there and added to the corresponding function in the *Node-Red* editor. For the *TCP request* node, we will set the server IP (*192.168.1.120 in our case*) and port *5005*. For return, we will use it as a string, and for *Close*, we will choose when character received is *q*.

To start the model training script, we will add a new button called Start app train model, linked to a pythonshell in node connected to a debug node. We will configure the pythonshell in node by adding the path to the training model script as well as the virtual environment. In pythonshell in, under Py file, we will set the path:

/home/ivan/opencv/build/facial_recognition_var2/train_model1.py

and for Virtual Environment Path, we will set /home/ivan/venv.Next, we will add in Node-RED the ability to create a folder, so that if anyone wants, they can create a folder with their desired name directly from the Dashboard. We will use a text input button for the user to input their name, a second fs mkdir node to create the folder, and finally, the last Path node. We will start by configuring the first text input button. In Mode, select text input, set the Delay to 0. Under Topic, write msg.payload, and for Label, set it as CREATE FOLDER and check all other boxes.

For the second node with *fs mkdir*, we will select it to be regular, not temporary, set *Foldername* to *msg.payload*, and set *Result* to also as *msg.payload* and check the recursive option. For *Path*, in the *Value Format* section, we will put {{msg.payload}}. After we have created the folder, it is necessary to choose the folder where we want to upload the photos (this applies, for example, when we want to upload photos we received via email with *unknown* individuals or when a person is not present at that moment to take pictures but has other photos they can send us so we can upload and later train them). That being said, we will choose a *text input button*; for *Mode* we select *text input*, *Delay* will be set to 0, and *Topic* should be *msg.payload.* This button will be linked to a function that contains a small piece of code, namely:

context.global.set('Variabila_Input', msg.payload); return msg; With this code we define a global input variable received from the *text input button* in practice, the user writes the folder they want to access so they can upload photos into it. We move on to the next button, which we will use to upload the photos; this button is called *upload-mf* (multi upload), which is connected to a *join1* node, the *join1 node* being connected to a *base64 node*, and the *base64 node* being connected to a function that converts *base64* data into a *Buffer* and saves the image; each saved image will be unique because a timestamp is used.

This function will be linked to an *fs access node*, which will have the following configurations: for *Path*, *msg.payload*, and *for Filename*, *msg.filename*. For the join and *base64* buttons we will have the following configurations: for *join1*, we choose manual mode, for Combine each we select *msg.payload,* and for to create, we choose *Buffer*. For the *base64 node* we set *ACTION to Convert Buffer<->Base64*, and for *Property* we write *msg.payload*. In the next part, for displaying the files (photos) and folders, we will need two buttons: one for displaying folders and one for displaying files, both buttons are taken from the Dashboard section and are of type button. We will also add two *fs list* nodes, which we find on the left in the *Storage* section, and we will also need a list node found in the *Dashboard* section and two functions, one named function 48 and the other storage folder, both functions are available on GitHub. For configuring the two buttons (display folders/files), we will set the path that goes to the dataset directory, and for Topic, we leave it as *msg.topic*. For the fs list node connected to the Display Folder button, we will set *Path* to *msg.payload*, *Pattern* to *, *Filter to folders only (checked)*, and *Result* to *msg.payload*. We do the same for the second fs list node, only that for *Filter we check files only*.

These two fs list nodes will be connected to a list node that we configure as follows: for *List Type* we choose *Single-line*, for *Action* we choose *switch to send changed item*, and we check allow *HTML* in *displayed text*; this node will be connected to *function 48* and this function to another function, *storage folder*. To *delete folders/files*, we will use a button from the Dashboard section and connect it to a *function 49*, which will then be connected to an *exec* node.Will rename that button to *Delete Folder* and send a payload with the string *DELETE* because this payload will be sent to the corresponding function where a condition is set upon pressing the button. For Topic, we'll use msg.topic. *Function 49* will be available on *GitHub*, and now we'll configure the exec node. For Append, we'll check it and write *msg.payload*, and for Output, we'll leave it as "*when the command is complete*" (exec mode).

We'll have another button taken from the Dashboard section and rename it to *Entries/Exits*. When pressed, this button will display in real time who enters and exits the lab. For this, we'll use that button along with two functions and a template: *Function 52*, *Function 53*, and a template, which you can download from *GitHub* and link to this flow. Next, we'll add another button for local photo collection specifically, a text input button renamed to Student Name. This button will be linked to *Function 58*, which connects to a *PythonShell in* node containing the script to open a GUI window. When pressing *Space*, it will create the corresponding folder, take the photos, and save them inside, making it very easy to collect and store these images. Afterward, we only need to train the model.

Now, let's configure the first node: for Topic, we'll use *msg.payload*, *Delay* set to 0, and *Mode* set to *text input*. *Function 58* is available on GitHub.

In the PythonShell we have the Py file where we'll put this path:

/home/ivan/opencv/build/facial recognition var2/headshots node.py /home/ivan/opencv/build/facial_recognition_var2/headshots_node.py, and for *Virtual Environment Path* we'll put */home/ivan/venv* with both boxes checked. To integrate that Flask we'll need another template where we'll specify the source at the address where Flask is running at that moment, in our code it will be:

```
<img src="http://192.168.1.120:5000/video_feed" width="100%">
```

This way we know that on the specific address and route on which *Flask* runs and integrate it in the *Dashboard*. If all steps have been followed correctly up to this point, we should have a *Dashboard* similar to Fig3.

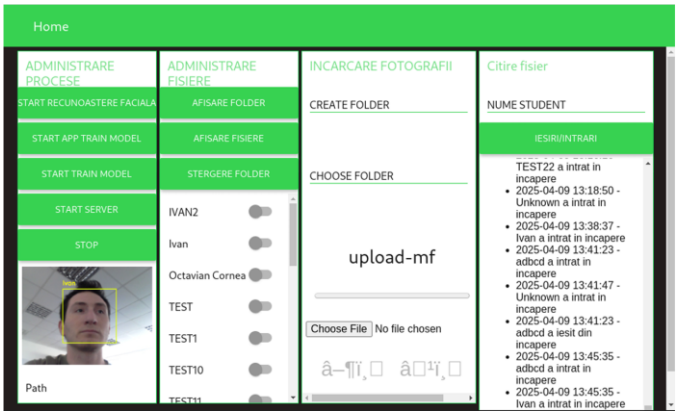

Fig3. Dashboard View

Sure, we can also use the other option without Flask.

However, that video stream won't be integrated into the dashboard and will only be available locally through the GUI. If we're on the same network and try to start the facial recognition feature from a phone, the GUI window will open on the respective monitor, but it won't be visible on the phone.

Using *Flask* makes this possible we can view the video stream integrated into the Dashboard directly from the phone. One downside of using *Flask* is that it consumes more resources compared to displaying a local GUI, since *Flask* sends data over *HTTP*. Additionally, the configuration and code required to use Flask are slightly more complex.

VII. REMOTE ACCESS AND ADMINISTRATION OF A NODE-RED DASHBOARD SERVER

In today's rapidly evolving technological landscape, where connectivity has become a defining element of modern society, the need to control systems remotely is essential in most cases. Whether we're talking about industrial, educational, or home applications, efficient real-time information management plays

a crucial role in improving performance and optimizing processes.In this context, I set out to explore the potential of Node-RED, a platform developed by IBM that facilitates process automation through a visual, flow-based approach. We successfully implemented the visual interface as shown in Fig3, and now we want to connect remotely to this Node-RED server.

For this to be achievable so anyone can access our Dashboard (UI) with proper username and password authentication, we need to configure port forwarding. When we want to make this server accessible over the internet, we must understand how local networks and routers work. Devices on a local network (computers, Raspberry Pi, laptops) are not directly visible from the outside because the router hides them behind a public IP address using NAT (Network Address Translation).

This is where port forwarding comes in - a method where we can explicitly tell the router what to do. For example: "Everything coming to port X from the internet should be forwarded to device Y on the local network at port Z." Essentially, we're opening a port between the external network and a specific service on our internal network.

We attempted to implement this but encountered difficulties because we're behind a CGNAT (Carrier-Grade Network Address Translation). This means we need to configure the higher-level router responsible for internet traffic routing. Configuring port forwarding on our local router isn't sufficient, as the data passes through multiple routers before reaching the internet. Only the top-level router, which handles the final external connection, can properly expose our service.[7].

Without access to this top-level router, we would likely need to contact our ISP to configure it for us and obtain a public IP address. Only then could we properly access the server remotely through that public address. We can access our server using a public address and the corresponding port. In our case, this didn't work because we're behind *CGNAT* and don't have access to that higher-level router. Instead, we used *Ngrok* to gain external network access and verify this functionality.

Ngrok is a tool that creates secure tunnels between a local server and the internet, providing a way to make a local server publicly visible online with a public link. Using *Ngrok* made router configuration unnecessary. Instead, we needed to go to ngrok.com, register for an account to obtain a token, and download *Ngrok* for our operating system where we found the necessary instructions. Then we ran the command *ngrok* config add-authtoken (inserting our token) followed by *ngrok http 1889* (we changed *Node-RED*'s server port to *1889*) [8]. Using this command generates a link that we can access from our phone using mobile data. In our case, this link was *https://813b-193-226-9-80.ngrok-free.app* (we're using a free application where each *ngrok http 1889* command generates a different link) and this is just for the development environment. For the interface, we would use */ui* after .app.

If we notice the video feed doesn't display anything on our mobile device, then we need to also start *ngrok http 5000*, but we'll need to modify the template with the new link obtained from *ngrok http 5000*. This project will remain local in our case, but if someone wants to publicly expose that *Node-RED server*, it's recommended to use port forwarding instead of *Ngrok*, as many organizations may block access to public websites or IP addresses obtained through *Ngrok*.

VII CONCLUSIONS

The purpose of this paper was to explore and integrate modern technologies in order to implement a feature rich real time laboratory access monitoring system.The work is structured into several sections, each addressing specific aspects of the development process, from image processing to communication management and interaction with the end user.

The integration of the various technologies I used (*OpenCV, Node-Red, Flask, Gmail API*) led to the successful development of a flexible and robust system for presence monitoring in an academic environment.

Following this result, it was found that this method is much more efficient than the classical method

In addition to enhancing security, this solution can be extended and improved, as well as adapted to other applications in the field of automation and image processing.

This work can serve as an example of applicability in real-world contexts, with significant benefits both in academia and in industry.

That being said, the integration of these technologies that I have used helped me gain a much better understanding of how they work and how important it is for most technologies to remain interconnected.

References

[1] F. Schroff, D. Kalenichenko, and J. Philbin, “FaceNet: A Unified Embedding for Face Recognition and Clustering,” Mar. 2015, doi: 10.1109/CVPR.2015.7298682.

[2] J. W. Jolles, “Broad-scale applications of the Raspberry Pi: A review and guide for biologists,” Sep. 01, 2021, *British Ecological Society*. doi: 10.1111/2041-210X.13652.

[3] “Swap-Memory.” Accessed: Mar. 03, 2025. [Online]. Available: https://phoenixnap.com/kb/swap-memory

[4] “GMAIL API.” Accessed: Feb. 09, 2025. [Online]. Available: https://www.youtube.com/watch?v=PKLG5pfs4nY

[5] “Facial_Recognition GitHub.” Accessed: Feb. 02, 2025. [Online]. Available: https://github.com/Ivan13s/facial_recognition

[6] “Node-Red.” Accessed: Jan. 12, 2025. [Online]. Available: https://en.wikipedia.org/wiki/Node-RED

[7] D. F. Hritcan and D. Balan, “Exposing IoT Platforms Securely and Anonymously behind CGNAT,” in *Proceedings - RoEduNet IEEE International Conference*, IEEE Computer Society, 2024. doi: 10.1109/RoEduNet64292.2024.10722287.

[8] “ngrok.” Accessed: May 10, 2025. [Online]. Available: https://ngrok.com/blog-post/what-is-port-forwarding