ENG5220

Real-Time Embedded Programming

Project Report

Team 16: Fun Door Security System

Bin Liu	2522070L
Shuaiqi Liu	2629952L
Genyuan Su	2534443S
Yuhan Lin	2614570L

1. Introduction

People often have the need of installing an effective security system to protect their private spaces and to grant access to themselves and their family or friends. Following this clue, our team thought about developing a raspberry-pi-based real-time security system and using face recognition to control the opening of the door lock. Face recognition is a quick and accurate way to distinguish between authorized hosts and unwelcomed people. This report will first demonstrate few points should be considered before we start our program, including problems may encountered, how our system is supposed to work and what hardware we should prepare.

2. Problems May Encountered

1. How to ensure the performance of camera in abnormal circumstance (e.g., too light or too dark, camera angle);
2. How to upload and save face data to database easily and quickly. The form of face data saved;
3. What methods could be applied to matching the scanned face and face data.

3. How This System Works

When the whole system is activated, all the events start, which include face recognition, motor, LED strip, sound detection, and LED bulb. The camera for face recognition is always activated while the LED bulb is on only when a sound is detected (somebody approaching the door). The purpose of this function is to make sure the system can work properly in a dark environment. If the face captured by the camera can match any faces in the pre-created facial recognition library, a motor connected to the door lock will unlock the door and the LED strip installed inside the house will turn green. Otherwise, the motor will not rotate and the door will stay locked. If someone comes and tries to unlock the door but fails, the LED strip will turn red, which alarms the host in the house that an unauthorised visitor is outside. The flow chart of the system is shown in Figure 1.

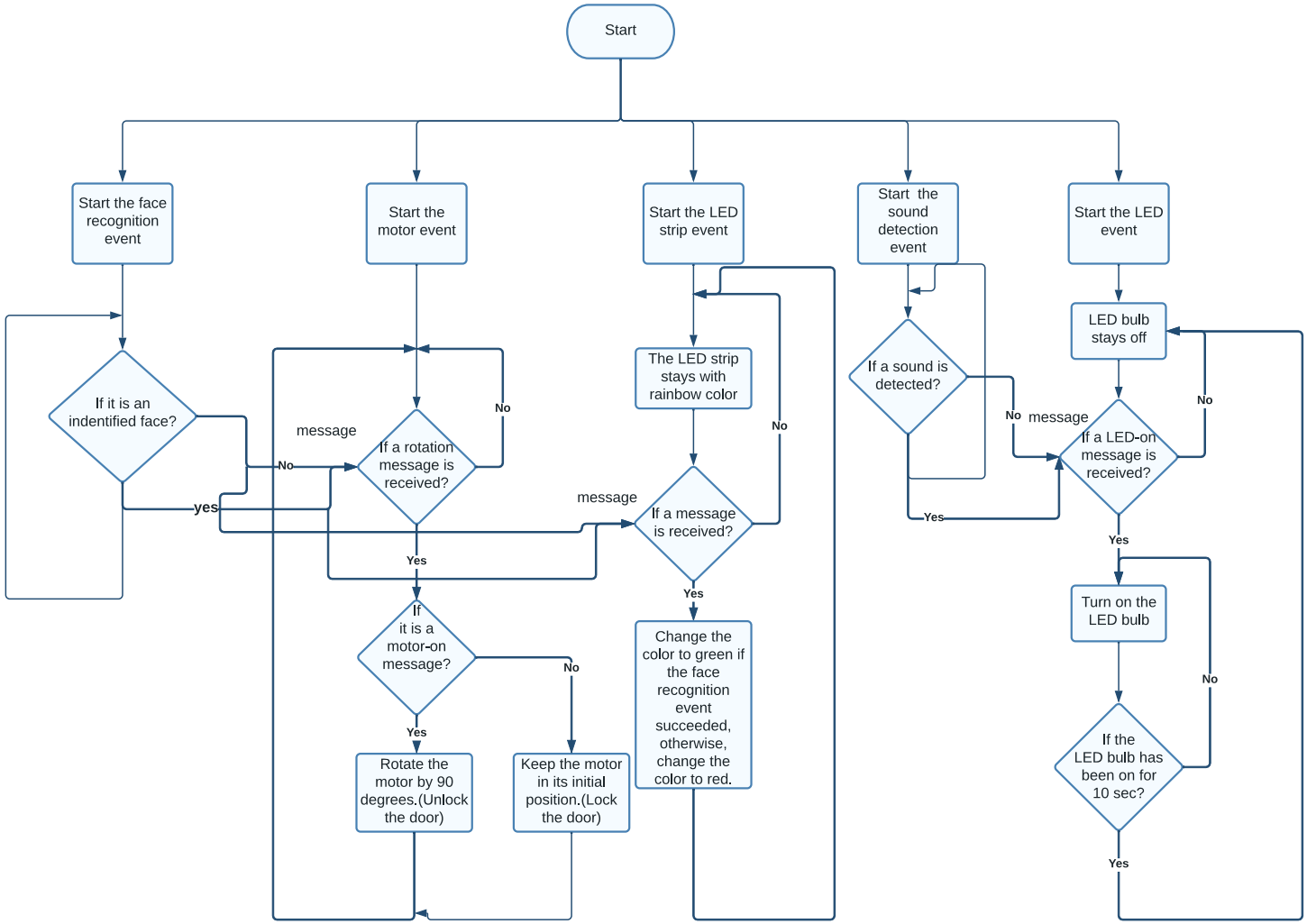

Figure 1. The flow chart of the system

4. Hardware

- Raspberry Pi 3B
- Sound Sensor
- 8MP Auto Focus Camera Module
- Digital Programmable LED Strip
- SG90 Micro Motor
- Breadboard, Dupont Lines, etc.

5. Methodology

5.1 Event-Driven Programming

Event-driven programming is a paradigm that involves building applications by sending and receiving events. When the program triggers an event, it can respond by running any callback functions registered in the event manager. At the same time, it can also pass relevant data to the responding function. In this mode, if the program does not subscribe to any functions when the event is triggered, it will not throw an error because the event is sent to the event manager.

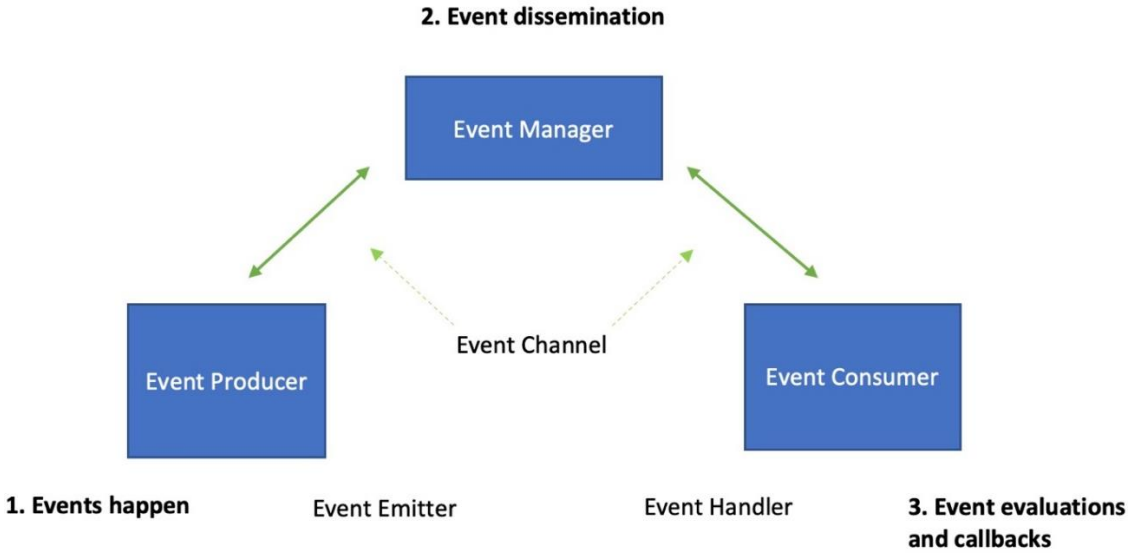

Figure 2. Event-driven programming architecture

As shown in Figure 2, when a registered event happens, it will be transmitted to the event manager. Then the event manager will disseminate this message to inform a consumer or multiple consumers that have

registered for this event. Consumers would assess if it was needed to respond to this event.

Emitters have the function to detect all the events, any happened events would be noticed and gathered by emitters. Then emitters will transmit events to the event manager. So far, an Event emitter's job is finished for one circulation. An event emitter does not have the information of the consumers of any events. Its job is to convey the information of all those happened events to the event manager.

Event handlers are responsible to respond events, which is applying some specific actions that were registered by consumers. From a programming perspective, it refers to functions. Event handles are required to respond to events as soon as possible to ensure real-time performance.
Event channels are data pipes that connect event emitters and the event manager as well as event consumers and the event manager. They are responsible to transmit the data of events from event emitters to the event manager and then eventually to event consumers. In addition, event channels have the data of the correct distribution of events, which ensures all the consumers respond to the events they have registered.

5.1.1 Observer pattern

The observer pattern is a software design pattern. In this pattern, a target object manages all observer objects that depend on it and actively notifies the observers when its state changes. This is usually achieved by calling the methods provided by each observer. This pattern is usually used to implement an event processing system, which makes one method to implement event-driven programming.

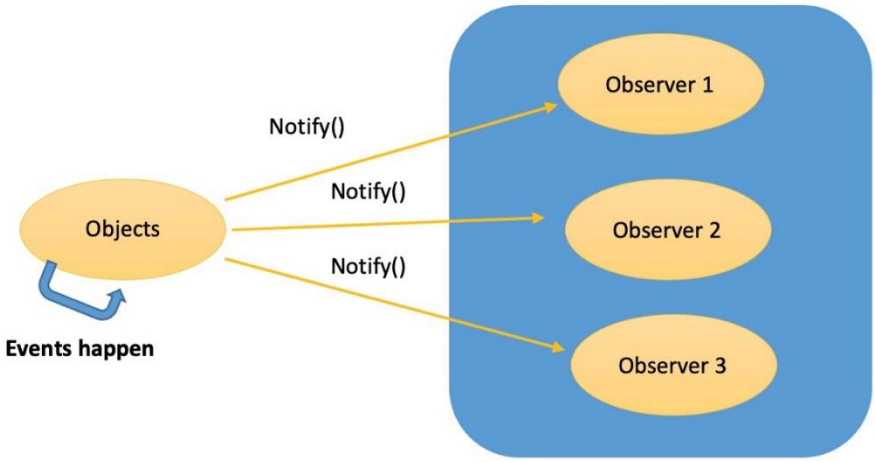

Figure 3. Observer Pattern

As we can see from Figure 3, when an event happens, the object would notify observers that have registered on the object for this event. Then observers would apply actions to respond to the event. This is a one-to-many pattern. You may be wondering why this pattern does not correspond to the architecture in the previous section. It is not true. The Object has combined event producers and the event manager. It can detect events when they happen, also, it maintains a queue of all these events that will be transmitted to related observers.

5.1.2 Publish-Subscribe pattern

The Publish-Subscribe model has two participants: publisher and subscriber. The publisher publishes a message to a channel, and the subscriber binds to the channel. When a message is published to the channel, it will receive a notification. The most important point is that publishers and subscribers are completely decoupled and do not know the existence of each other. It can be seen from the definition that the two sides are completely decoupled in the Publish-Subscribe mode, while in the observer mode, the target object manages the observer, and the two sides are coupled, which is the main difference. In addition, there is an intermediate layer channel in the Publish-Subscribe mode.

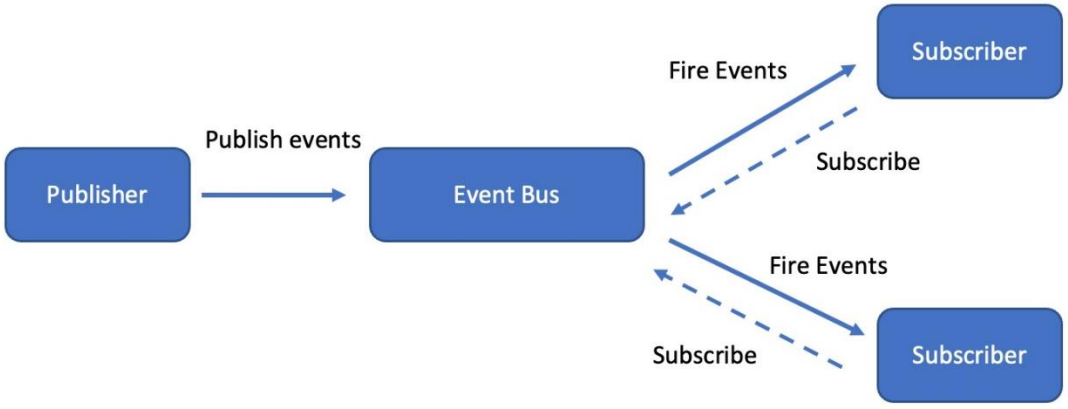

Figure 4. ***Publish-Subscribe Pattern***

As we can see from Figure 4, when an event happens, the publisher would publish this event on the event bus. Then messages would be sent to the subscribers that have subscribed to this message. After that, subscribers would respond to this message.

5.2 Implementation

In this project, we applied the Publish-Subscribe pattern. As you can see from the directory tree, we built a class named "dispatchEventService.h" which acts as the event bus as mentioned in the previous section.

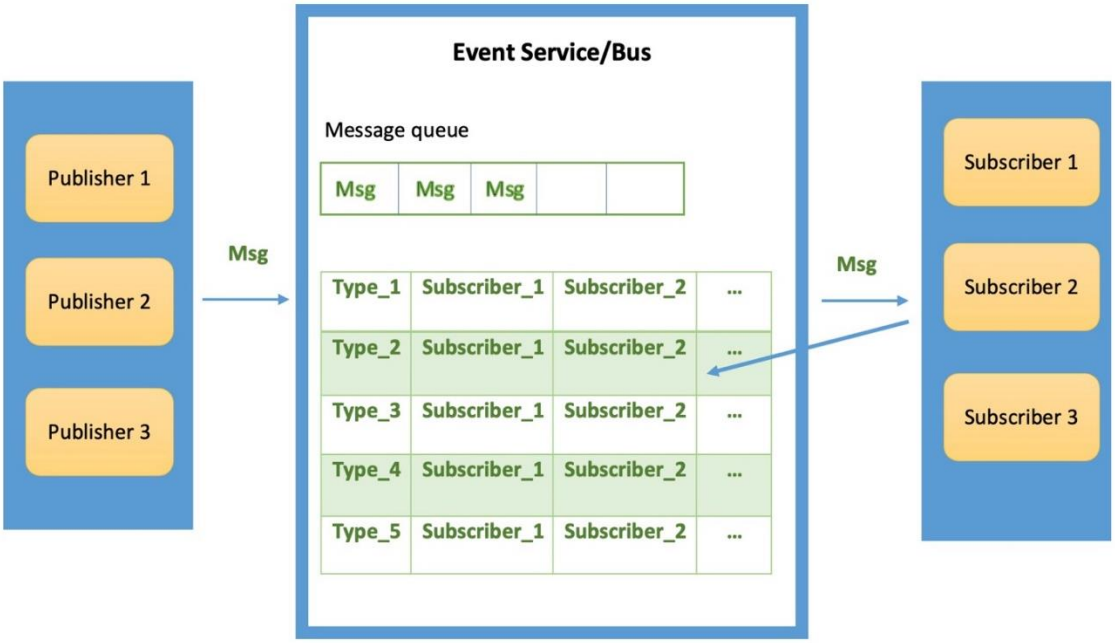

Figure 5. Mechanism

The mechanism of the event-driven architecture we applied is shown above. The event service maintains a message queue and a table. The message queue stores messages from publishers. Every message has a type name which we use to distinguish different types of events. When an event happens, a message will be added to the queue. The table stores the information of subscribers that have subscribed to a certain type of message or event. In the meanwhile, the event service would handle all the messages in the queue in a loop. Also, the process is in a thread.

From the programming perspective, we have a class named "EventCallback" which is an abstract class, it has only one function which is the "callback()" function. Every event handler inherits from this class and actualizes this function.

5.3 Computer Vision

The computer vision (CV) part mainly concentrates on the face detection and face recognition, which is the core module of the whole system. The former determines whether a face is present and returns its pixel position, while the latter validates the input face against known information to determine whether there is a matched face in the database. Detection is an a priori condition for recognition. Therefore, OpenCV built-in function are used to detect the face first, and then performing face recognition. The face detection code is shown below.

```
CascadeClassifier classifier;
classifier.load("/home/pi/raspi_project_16/cascades/haarcasca
de_frontalface_alt.xml");
classifier.detectMultiScale(frame, faces, 1.2, 5);
```

There are three methods given by OpenCV regards face recognition, which are Eigen Faces, Fisher Faces and Local Binary Patterns Histogram (LBPH), respectively. The project needs of CV are:

1. Superior accuracy;
2. High robustness and versatility;
3. Low hardware performance requirements;

4. Easy to update models;
5. Low cost.

Thus, the LBPH is used here. The advantage of this algorithm is that it is not affected by lighting, scaling, rotation and panning, and can also run on the not very high-performance platforms smoothly, making it feasible on Raspberry Pi.

The LBPH algorithm is to compare the grey value of the pixels (x_p, y_p) on a circle of radius R with that of the current pixel (x_c, y_c) , and if the surrounding pixel has a greater grey value, the position of the pixel is marked as 1, otherwise it is 0. By arranging the value of each point, the current pixel is given a binary LBP value. The relationship between (x_p, y_p) and (x_c, y_c) is:

$$\begin{cases} x_p = x_c + R\cos\left(\dfrac{2\pi i}{P}\right) \\ y_p = y_c - R\sin\left(\dfrac{2\pi i}{P}\right) \end{cases}, \ i \in P$$

where P is the number of sampling points. However, as the number of sampling points grows, the number of LBP patterns increases exponentially, to 2^P in total, which is not conducive to expressing graphic information. The LBP patterns, therefore, are integrated through a certain rule called Equivalent Model, reducing the dimensionality of LBP patterns, so that the information of the image can be best represented with a reduced amount of data. The content about means of reducing pattern is not represented here.

With the method mentioned above, each pixel is given an LBP value based on surround information. The next step is LBP feature matching and the process is as follows:

1. Dividing images into non-overlapping areas;
2. Construct a grey scale histogram within each area;
3. Stitching the grey-scale histogram features of the whole image in a certain order to construct the overall features;
4. The similarity between the face to be recognized and the features

from the database is calculated, and the highest similarity which is greater than the threshold is considered as the same person.

To train the model, the folders are named after the registrants' names and the photos corresponding to the registrants are put under the folders. During training, the folder name is used as the label and the vector obtained from the training is stored in .xml extension files, which is the model for face recognition. When validation, the confidence level returned by the function is compared to a set threshold to determine if it is a known individual.

6. Conclusion

Even though we have this project worked properly, there are still many improvements that can be done to have a better performance. The face recognition event does not work 100 percent precisely, sometimes, it could not recognize the same face it has identified before. Efforts on this are needed to improve the practicability. In addition, the function of our system is not hundred percent compatible with a real-life scenario, and it could be more fun. We were planning to build a dancing light that can dance with music in the environment. It requires extra sensors and efforts, but we believe we can work this out if we have enough time. Also, we need more explorations on event-driven programming approach. This was like an unknown continent to most of us, which makes it hard for us to embark on coding before studying the concepts of it. Even though we finally got the system worked fine, we believe more efforts need to be invested on this to benefit our future career.

With the continuous hardworking of all of our team members, we managed to get this project done. We learnt a lot from our lecturers, Dr. Bernd Porr and Dr. Nicholas Bailey. Thanks to their guides and all the materials they offered, we got a better knowing of real-time embedding programming; we gained knowledge of many sensors and how to apply them in coding. We now have a clear understanding of what skills we need to enhance for the career we are pursuing.