“DoorBell Labs”

EC500 Final Report

Daniel Oved, Josh Manning, Bogac Sabuncu

Introduction

Security of one’s domicile is a vital aspect of everyday life. With everything becoming technological and interconnected, the security of the house is going to be inevitable. An Internet of Things home security system is a novel approach, and an untapped market for authentication and security. DoorBell Labs wants to automate and advance the act of looking through a peephole. It will recognize pictures of a face off of a whitelist, and send these photos to the user’s phone utilizing the Twilio API. From there, the user can unlock the door right from their phone.

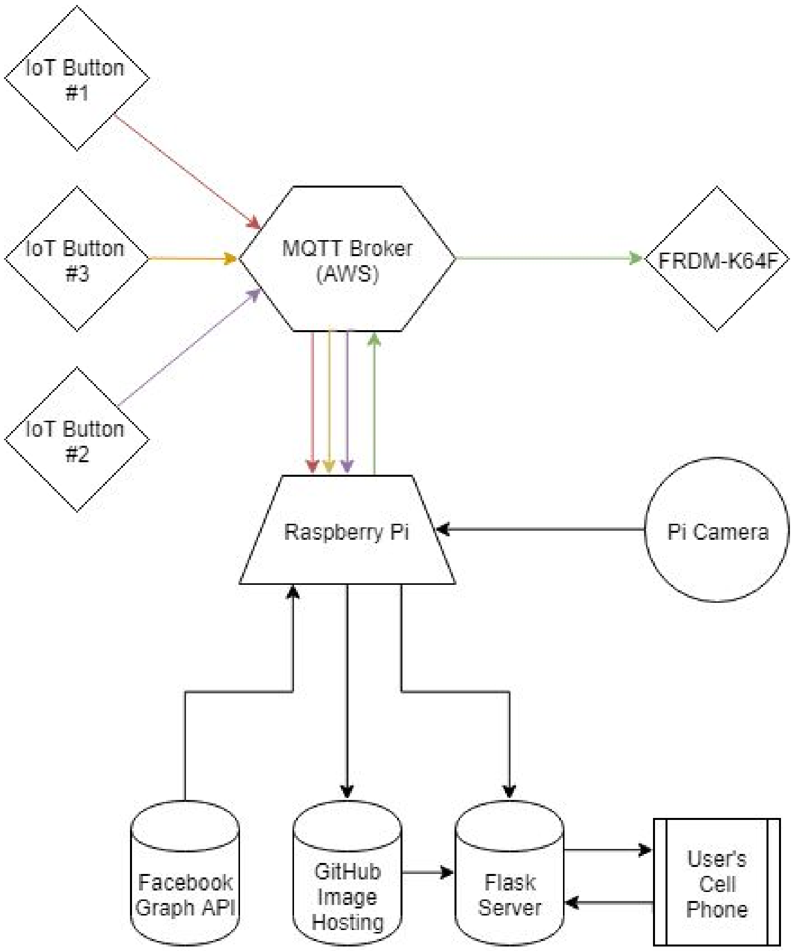

Figure 0: Connection Diagram

Technical Aspects

<u>MQTT:</u>

The driver of information between the pressing of a doorbell button (AWS IoT button) and the Raspberry Pi camera taking a picture, as well as the driver between the Pi and a FRDM board (with an LED that emulates a door being locked when off, unlocked when on) is the MQTT protocol. As we learned in class, MQTT is a lightweight protocol which is easy to use with AWS as the message broker. We had our Pi subscribe to the three unique IoT buttons' topics while it sat idle, and when one of them was pressed it would take a picture and begin the image processing described later. Later, if a user chooses to unlock the door, the Pi would publish a message to the FRDM's LED topic to turn it on, and then publish another one to turn it back off (since the doors auto-lock). The MQTT protocol was a valuable tool for our project. To enable our devices to communicate with it, we had to give the Pi a policy which allowed for posting to and subscribing to the topics it needed. We also had to give each device a private key and certificate, as well as the root CA certificate, which allowed the devices to not only verify their own identity to the AWS MQTT broker, but also verify AWS as the broker they intended to communicate with. All of this was covered in earlier labs and in class.

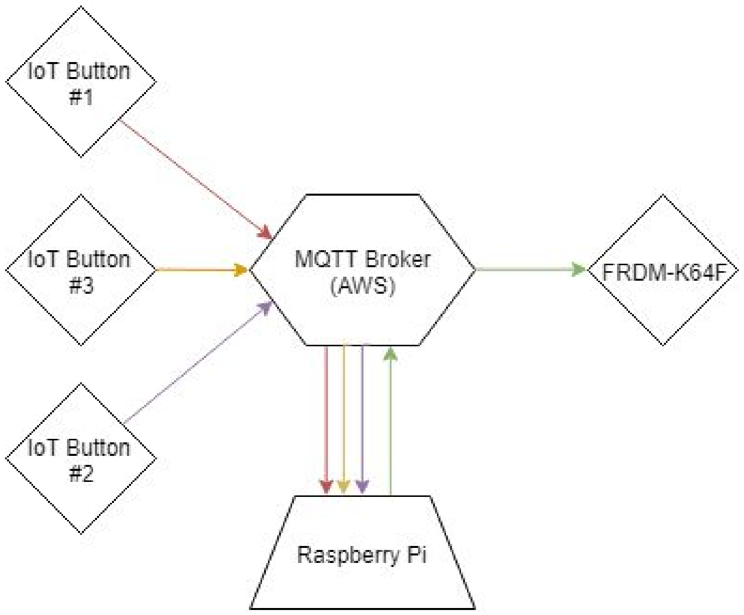

Figure 1: MQTT Topic Diagram

Twilio:

The Twilio API is a RESTful, which requires an AccountSid and AuthToken, which authenticate your account using HTTP basic authentication, which is wrapped in HTTPS for confidentiality purposes. These two values act as your username and password respectively. These values are also used as your API Keys; normally different API keys are used to differentiate between developers or subsystems, but this was unnecessary for our design.

One aspect of our design is to simply send a message to the user that is tied to the user's button. This user, obviously has a phone number tied to their name, which is the number to which that the Twilio API sends its message. We use Twilio's Programmable SMS functionality in order to deliver this message. The API in our application implements an HTTP request to the Twilio API, and the API interprets the data set to send to the appropriate phone number. We may send this in the form of either a XML or JSON , which has multiple fields, including a body, media URL, a to phone number and a from phone number, among other fields. We create a message with these fields set to the variables we want, depending on the actual message sent, depending on whether it detected a face(s) and whether it recognized that face(s). We make sure to post the photos online, specifically on Github, so that there is an actual URL associated with the image, and we can therefore send is via MMS.

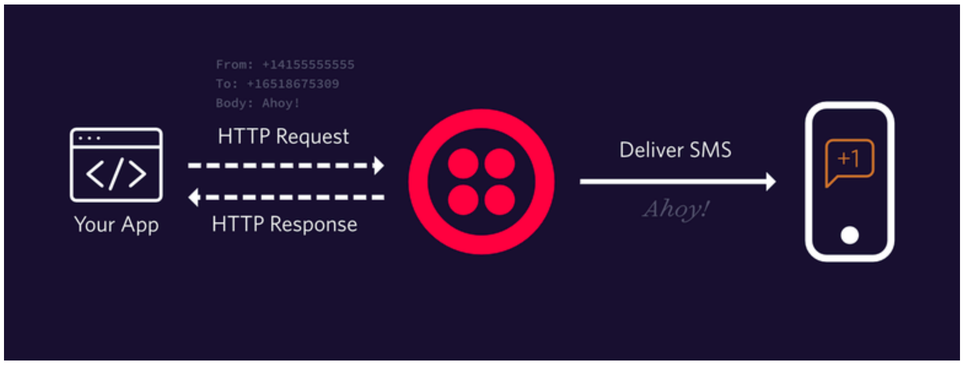

Figure 2: A simplified version of the usage of a Twilio API

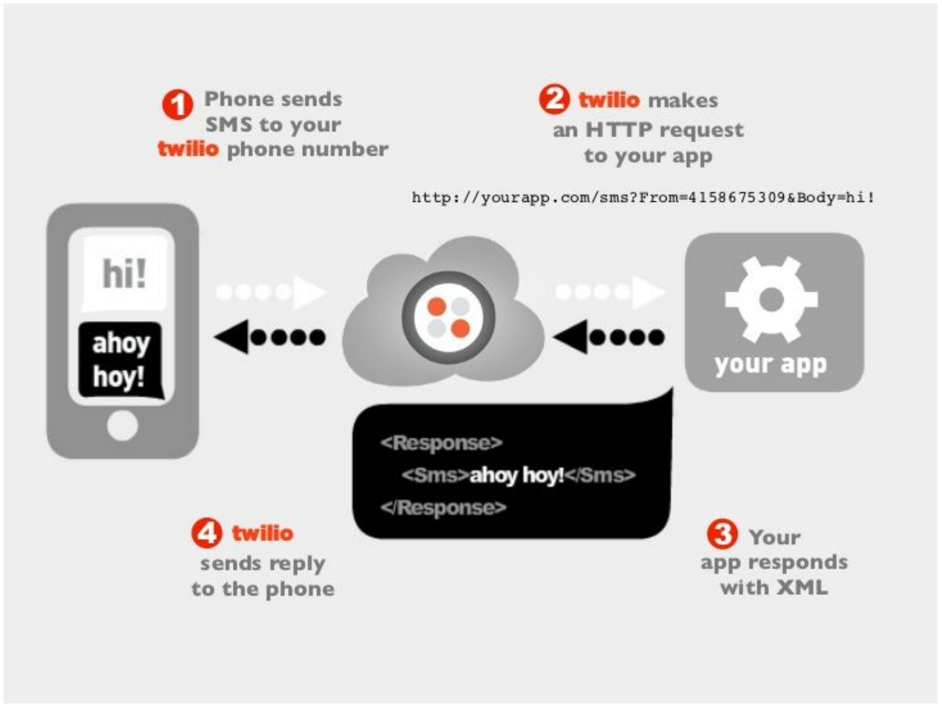
1 Phone sends SMS to your twilio phone number
2 twilio makes an HTTP request to your app
http://yourapp.com/sms?From=4158675309&Body=hi!
hi!
ahoy hoy!
your app
<Response>
<Sms>ahoy hoy!</Sms>
</Response>
4 twilio sends reply to the phone
3 Your app responds with XML

Figure 3: A more verbose diagram of the response functionality

The design must also allow for the functionality of receiving a message. This message will be the response the user inputted into their phone as a result of the message they got above. To do this, we must implement a Flask application. Flask is a micro web framework which allows for a small core and for multiple applications. It runs a server which will handle SMS responses sent to the Twilio phone number. This is tied to a decorator, which is routed to a function, and is given GET and POST methods; in this case the sms function. This sms function deals with responses with Twilio's MessagingResponse() method, which sends certain messages based on the response it has recieved.

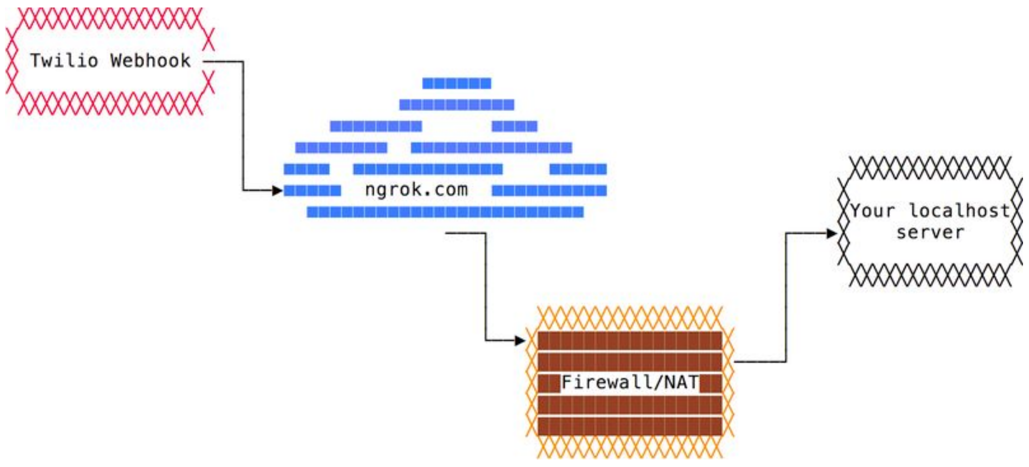

Figure 2: A diagram of the program flow of dealing with message responses

In a development environment (which is the only recommended environment for Flask applications), the application is only reachable by other programs within your computer; therefore Twilio is unable to communicate with it directly. This was done using ngrok, as it was immensely less laborious overall than deploying the application over AWS. Ngrok is a lightweight method to make the Flask application available to the internet, as it provides a unique URL on the ngrok.io domain which forward incoming requests to your local host. A custom subdomain was purchased which allowed for a consistency. On the Twilio side, all one had to do is simply tie the webhook for events of SMS/MMS to this subdomain.

A functionality of this that proved problematic is the need for the server to only run when it needed to be. This needed to be implemented such that the user may not erroneously, or (if the user is compromised) maliciously alter the state machine. In order to do this, we decided to run it as a subprocess of another Python module. This module dealt with a large state machine of idle-ness, processing images, sending and receiving Twilio messages, and opening the door. This has all been explained above. When the state is sendTwilio, a simple message is sent as described above. When the state is set to recieveTwilio, it runs the Flask server as a subprocess. This allows the user to respond to the message, and span the state machine, with different branches depending on the user's input and who is actually at the door.

When a final state has been reached, the Flask application sends a message over a socket to the original application, which is listening on a custom port. These messages are handled as such: either the user recognizes the person outside (or simply wants the door unlocked for this person), or the person is unsure. In the former case, the state is set to openDoor, the sockets are closed so they may be reused if they enter this state, and the server's process is killed so that the user may not enter more input. In the former case, the state is set to sendTwilio again, so that the user may receive another picture of the person outside; the socket and process are also killed. There is a third case, in which the user does not allow for person outside to be let in. In this case, we just simply allow the socket to timeout, kill the server and set the state to idle. We determined there is no need to implement a socket connection in this case.

<u>Facial Recognition:</u>

Facial recognition is an important part of this project, since the application needs to know who is at the door from a picture taken after the IoT button is pressed. At the beginning of the project we wanted to have a design that utilized the Facebook API to determine the identity of the faces at the door. The way this API works is that it takes an image and tries to upload it to Facebook. During this process Facebook API suggests the names of the people in the picture to tag them at the upload and the API we wanted to use returns these names and cancels the upload. Later on we decided not to use this API, since it was too simple and didn't give us any challenge.

At the end we decided use a Python API that is called Face_Recognition. This is an API that uses dbi with deep learning to accurately guess the identity of the face from a single given picture. For this API to work first we need to have pictures of people who are "known." When the application first runs it goes in to the folder named "known_people" which has the .jpeg files of known peoples faces with the example name of “Bogac Sabuncu.jpeg.” It parses through all the files getting the face encodings and the related names of the known people. This is the initialization phase of the application, after this phase the application does not need to go through the known people again since it already has the necessary data to compare new faces with the ones it already knows.

After the button is pressed the pi camera takes another picture which is processed by the API as well. The API finds the faces in the taken picture and gets the encodings of them. Then it compares these encodings with the known people encodings and if they match it finds the related name to that picture. If the face on the door is not known the application marks the person as “unknown.”

The API also finds the locations of the faces it finds at the pictures. So using the Python Image Library we take the locations of the faces given by the face_recognition API and draw boxes and the associated names around the faces. This is the final image that is uploaded to GitHub and sent to the user’s phone using Twilio. An example of the image can be seen in the figure three provided below:

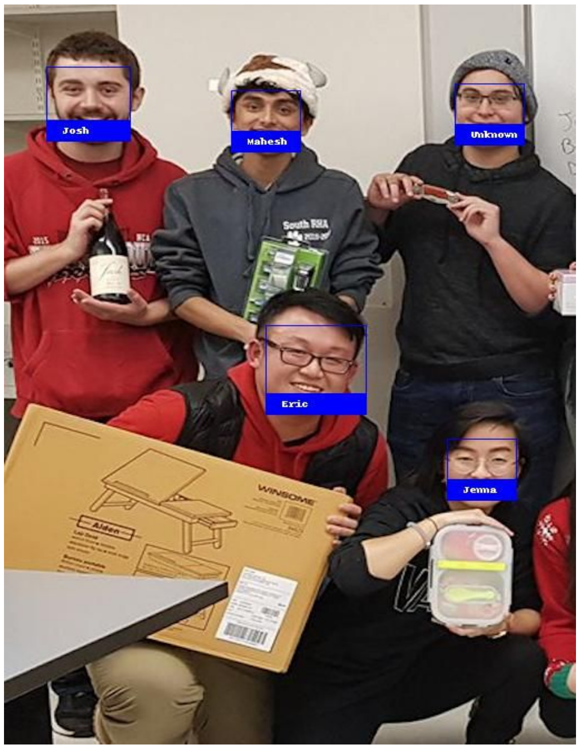

Figure 3: Image with boxes and names around the face

These steps for the facial recognition and creating the final image is all done at the Raspberry Pi after the IoT button is pressed and the picture is taken. To download the necessary dependencies to the Raspberry Pi follow the instructions at this address: https://gist.github.com/ageitgey/1ac8dbe8572f3f533df6269dab35df65

Failed Aspects

OpenCV:

After deciding not use the initial facial recognition API that used the Facebook API, we looked in to using OpenCV for our facial recognition part of our application. Later on we decided not to use OpenCV and went with the face_recognition tool mainly because of the steep learning curve of OpenCV and also the fact that the Raspberry Pi might have had difficulties on running an extensive library like OpenCv. After some research on the OpenCV library we have realized that in order for the OpenCv facial recognition to work on the Raspberry Pi we had to first train the network on our laptops then transfer the trained network to Pi due to the memory and CPU constraints of the Pi. By using the face_recognition API we have skipped this step, more efficiently automated our application without the need of a third party computer for the initialization phase.

Facebook API:

Facebook's Graph API had been deprecated in 2014, and unfortunately (for the project, but fortunately for our own privacy) no longer allows developers to access a user's friend's information when provided just the user's permission. Instead, each friend must also have provided explicit permission for their data to be accessed. Our project initially planned for us to pull our friend's profile pictures and use them for the training data of the facial recognition software. Due to these restrictions, we were only able to pull the first profile picture of friends who had also registered for Facebook's Graph API developer program. While we discovered this setback early on, we still went ahead and created a script, *fbConnector.py* which pulls images from those 10 people and saves them with their full name, just as a proof-of-concept. We considered an alternative plan of pulling images from Twitter friends, but due to time constraints this plan was not fully completed.

Videos

- Twilio: https://youtu.be/7Y_YMbGMk74
- Facebook Image Pulling: https://youtu.be/SuwQNOJufbQ
- "Door Unlocking": https://youtu.be/9LPnEjhIdXc
- Image Processing: https://youtu.be/y7aWBUux9dc