IoT: Building a Raspberry Pi security system with facial recognition

David Gsponer
As IoT grows, devices become more cheaply available and facial recognition develops rapidly, the need for a system using these three components arises. Combining these three technologies will allow tackling a diversity of use cases, from replacing door locks to smart CCTV systems.

Some products which are making use of these components combined exist, but none have been implemented on a low-budget system under 100 euros. For this reason, the aim of the thesis is to work only with open-source software, no webservice, and only with the cheapest hardware available and the question whether it is possible to build such a system with the given hardware.

The following thesis will first focus on facial recognition and IoT in a theoretical framework, which is then followed by its implementation throughout the project’s five phases and a final presentation of the product.

The result of the presented project is a working system using facial recognition with OpenCV, IoT over the MQTT protocol and Clients on the Raspberry Pi as well as on an Android mobile application. The thesis builds a foundation using these three components on a system applicable in many further scenarios of facial recognition in combination with IoT on low-budget hardware.

**Keywords**

home security, facial recognition, Raspberry Pi, Internet of Things
# Table of Contents

Table of Contents ........................................................................................................... i
Terms and Abbreviations ............................................................................................... iii
1 Introduction .................................................................................................................. 1
2 Research Question ...................................................................................................... 2
   2.1 Objective of the project .......................................................................................... 2
   2.2 Scope of the project .............................................................................................. 2
   2.3 Out of scope .......................................................................................................... 2
3 Background Study ....................................................................................................... 3
   3.1 Facial recognition .................................................................................................. 3
      3.1.1 Introduction to facial recognition ................................................................. 3
      3.1.2 Techniques .................................................................................................... 3
      3.1.3 Algorithms ................................................................................................... 4
      3.1.4 Available solutions ....................................................................................... 6
      3.1.5 Algorithms and solutions comparison ....................................................... 7
   3.2 The Internet of Things ......................................................................................... 8
      3.2.1 Introduction to the Internet of Things ............................................................ 8
      3.2.2 Protocols and techniques ............................................................................. 8
      3.2.3 Available solutions ....................................................................................... 10
      3.2.4 Techniques and solutions comparison ....................................................... 11
4 Empirical Section ....................................................................................................... 12
   4.1 Project overview ................................................................................................. 12
   4.2 Project phasing .................................................................................................... 13
   4.3 Project implementation ....................................................................................... 13
      4.3.1 Phase 1: Raspberry Pi setup, network structure .......................................... 13
      4.3.2 Phase 2: Implementing facial recognition using USB camera .................... 15
      4.3.3 Phase 3: Implementing push/pull notifications over IoT hub/server ........... 22
      4.3.4 Phase 4: Implementing mobile application notifications ............................ 29
      4.3.5 Phase 5: Testing, extending functionality (false positives, UI App, Port forwarding) ................................................................. 35
   4.4 Project result ....................................................................................................... 39
5 Discussion and Conclusions ...................................................................................... 47
6 References .................................................................................................................. 49
### Terms and Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>AWS</td>
<td>Amazon Web Services</td>
</tr>
<tr>
<td>coAP</td>
<td>The Constrained Application Protocol</td>
</tr>
<tr>
<td>CCTV</td>
<td>Closed-circuit Television</td>
</tr>
<tr>
<td>FLDA</td>
<td>Fisher’s Linear Discriminant Analysis</td>
</tr>
<tr>
<td>HOG</td>
<td>Histogram of oriented gradients</td>
</tr>
<tr>
<td>IoT</td>
<td>Internet of Things</td>
</tr>
<tr>
<td>ISO</td>
<td>International Organization for Standardization</td>
</tr>
<tr>
<td>LBPH</td>
<td>Local Binary Patterns Histogram</td>
</tr>
<tr>
<td>M2M</td>
<td>Machine-to-machine</td>
</tr>
<tr>
<td>MQTT</td>
<td>Message Queuing Telemetry Transport</td>
</tr>
<tr>
<td>OpenBR</td>
<td>Open-Source Biometric Recognition</td>
</tr>
<tr>
<td>OpenCV</td>
<td>Open-Source Computer Vision Library</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>PCA</td>
<td>Principal Component Analysis</td>
</tr>
<tr>
<td>REST</td>
<td>Representational state transfer</td>
</tr>
<tr>
<td>SDK</td>
<td>Software Development Kit</td>
</tr>
<tr>
<td>XMPP</td>
<td>Extensible Messaging and Presence Protocol</td>
</tr>
</tbody>
</table>
1 Introduction

Facial recognition is being rapidly developed on mobile phones of IOS, Android and other operating systems. There are also numerous implementations working with different operating systems, but only on a single device. As IoT is also advancing and smart gadgets are getting more available to the end consumers, the need for a system which connects facial recognition with smart gadgets and low-budget hardware arises. A Business Insider’s article on facial recognition’s accuracy shows that a study by the University of Cambridge has shown that even with a face hidden by scarves or hats the algorithms are able to detect faces with accuracies of up to 69% (Condiffe 2017). The technology of facial recognition is developed to the point of precise recognition, therefore making it applicable for home security solutions.

There are numerous use cases where IoT systems with facial recognition in private or commercial use would be usable. For example, as a control mechanism to check if your child came home safely after school. Moreover, it would also be possible to implement a system which automatically checks if students came to class, or it could be used as a simple door mechanism which would replace keys. Looking at current available solutions using IoT and facial recognition supports the growing market need for such products. Whether it be AICure’s product of confirming patients’ medication intake or MasterCard’s facial recognition payment possibility, these are examples of products that are using these technologies and can save or gain earnings in hundreds of million figures (Sennaar 2018).

Sennaar’s article on available products using facial recognition and IoT shows that the market exists and is expanding, therefore making a need for a system that uses these two technologies on a foundation clear. These corporations have millions in development and research to spend. Because the ordinary person does not have that kind of money, the following thesis will focus on using the technologies on a low-budget system and if it is possible to implement such a system with only using open-source software.

As a result, the proposed thesis would provide a way for any person to implement his own low-budget security system using facial recognition and IoT. It will make the system’s boundaries clear and what can be expected from using the hardware.
2 Research Question

2.1 Objective of the project

The aim of this project is to present a basis of how to implement a system using facial recognition and IoT. It will provide a way for anyone to implement a solution with low-budget hardware.

2.2 Scope of the project

The two technologies will first be presented to make clear what exactly will be coded, how it works on a general level and what techniques are available. The different ways of how to use the technologies will then be gone through with the intention of showing possibilities of how to implement such a system. As practical solutions have been presented, the choices of available solutions will be made clear with reasoning on why a specific solution was chosen.

Each solution chosen will then be explained with it’s code throughout the development phases of the project. It will go into detail on how the code works for each of the libraries used. The finished product will finally be presented, finishing with a discussion and conclusion.

2.3 Out of scope

The following features are out of scope, as it is not feasible to include them in the project, as it would be too big to code all of them:

- Encryption between the communication mobile device to Raspberry Pi
- Security in terms of accessing the Raspberry Pi from a third-party device
- Security in terms of communication between Raspberry Pi and mobile application
- Lock mechanism to attach system to
- Detailed mathematical explanation of researched algorithms
3 Background Study

3.1 Facial recognition

3.1.1 Introduction to facial recognition

The use of computers to recognize human faces is called facial recognition. It is a biometric system which is generally used for security purposes, though it has seen potential in a wider range of applications. There are different techniques used for facial recognition, although most of them make use of nodal points for their recognition function (Techopedia 2018).

3.1.2 Techniques

Traditional

In the traditional method for facial recognition, nodal points are essential for the recognition to work. Eighty nodal points are therefore placed on a human face to determine landmarks of its shape. Through this process, it is possible to measure the width of the nose, distance of the eyes, depth of the eye socket, chin, cheekbone and the jawline. The points are then used to create a numerical code representation containing strings of numbers which can be used to recognize a face. It is also called a face print and allows detection with only 14 to 22 of the nodal points recognized (Rao & Parmar 2013, 1029.).

3-dimensional

This method of facial recognition uses 3-D models of a person's facial surface. Difficulties in this recognition lie in the comparison to 2-D video streams with the 3-D model. Therefore, when a 3-D image is taken, different points like the outside of the eye, inside of the eye or the tip of the nose will be taken. With those measurements, an algorithm which converts the 3-D image to 2-D is then applied. As the base model is 3-D, the accuracy of the converted 2-D model is more exact. The 3-D representation also allows extracting more information than taking a 2-D image (Bonsor & Johnson 2018).

Skin texture

The idea behind this technique is to gather more information about the individuality of a person's face other than from measurements. Therefore, the skin texture analysis tries to identify the uniqueness of a face on the surface (wrinkles, scars) and the texture
(birthmarks, moles). With these unique identifiers alone, it is possible to recognize a face (Pierrard & Vetter 2007).

**Thermal**

This technique of facial recognition works with thermal images of human faces. It allows detecting and distinguishing between persons through a human face’s heat. The patterns for the recognition are therefore primarily derived from superficial blood vessels under the skin. A human face is unique in its vein and tissue structure, therefore making this technique viable for facial recognition. The advantage of thermal recognition is that it is independent of lightening of pictures, as the infrared camera only captures heat of a face which is possible in any lightening condition (Bhowmik, Saha, Majumder, Saha, Sarma, Bhattacharjee, Basu & Nasipuri 2011, 113.).

### 3.1.3 Algorithms

The following algorithms presented are based on the traditional recognition technique.

**Eigenface**

Eigenface is a facial recognition algorithm developed in 1991 by Matthew Turk and Alex Pentland. The algorithm uses Eigenvalues and Eigenvectors to calculate Eigenfaces. These Eigenfaces are then compared to an input image. A weight depending on the algorithm’s calculation results is given for each value in the input image. The higher the weight, the surer the algorithm is of the recognition. The algorithm requires the use of Principal Component Analysis (PCA), which is a statistical approach of transforming a given set of possible correlated variables into a set of linearly uncorrelated variables. The Figure below shows the reconstruction of the applied Eigenface algorithm. Seen in the top left corner image is the final constructed Eigenface with its Eigenvalues and Eigenvectors. Using the top-left dataset, the algorithm can predict facial recognitions (Turk & Pentland 1991, 71.).
Fisherface

Fisherface is an enhancement of the Eigenface facial recognition algorithm. The process of the algorithm is the same, with the difference being that Fisherface uses Fisher’s Linear Discriminant Analysis (FLDA or LDA) instead of Principal Component Analysis (PCA) for its dimensional reduction. Because of that Fisherface allows classification of pictures, thus minimizing the variations within a class. Fisherman is also able to eliminate the first three principal components which are responsible for lightening changes, thus making the Fisherface method less fragile to lightening irregularities. Seen in the figure below is again the reconstruction of the algorithm, this time Fisherface. Noticeable differences to the Eigenface reconstruction is the missing illumination and differences between the pictures. This is caused through the available classification of the Fisherface algorithm and because the algorithm never had the chance to illuminate the given faces, therefore making each face important for the recognition (Belhumeur, Hespanha & Kriegman 1997, 711.).

Figure 1 Eigenface reconstruction (OpenCV 2017)

Figure 2 Fisherface reconstruction (OpenCV 2017)
Local Binary Patterns Histogram (LBPH)

Local Binary Patterns is an algorithm which does not use a holistic approach like the Eigenface and Fisherman algorithms. LBPH makes use of the pixels in a grayscaled picture and splits it in 3x3 blocks. Through comparisons from the middle pixel, the algorithm is able to detect edges depending on if the value around the centre is higher or lower to the centre value. Each 3x3 block can then be decoded in a decimal number of 8 bits starting from the north-west corner of the block. Applying this mechanism to a full picture allows creating a Histogram which is a representation of the result of applying the Local Binary Patterns algorithm on a picture. These created Histograms can then be used to apply facial or object recognition by comparing them to another grayscaled picture. Down below is an example of a Local Binary Patterns Histogram. In the upper section the input face is shown and below is the output histogram. In the output, it is visible to see the 3x3 blocks that were decoded to decimal numbers, which were then represented in black and white. For example a calculation of a 3x3 block would result in 000001110, therefore making all northern, eastern and western cells white and the southern cells black (Ojala, Pietikäinen & Harwood 1996, 51.).

![Figure 3 LBPH reconstruction (OpenCV 2017)](image)

3.1.4 Available solutions

Sennaar’s article on facial recognition applications says that the market is growing very rapidly (Sennaar 2018). Another look at available solutions shows that the big players like Amazon, Microsoft or smaller start-ups want to have a piece of it. With the aim of building a low-budget solution, only open-source libraries and APIs will be researched in a more extended manner.
**OpenCV**

OpenCV (Open-Source Computer Vision Library) is an open-source library offering computer vision and machine learning algorithms. It has been developed since 1999 and with over 47,000 community members is the biggest community-driven library for computer vision and machine learning algorithms. It is written natively in C++ and offers interfaces in Python, C++, Java and MATLAB. The library offers more than 2500 algorithms and is therefore the biggest available solution on computer vision and machine learning. Whether it be government bodies or corporations like Google or Microsoft, they all make use of this library, therefore making it the go-to library for computer vision and machine learning. OpenCV was especially designed for real-time applications, meaning face recognition using a camera to get instant feedback of a camera stream (OpenCV 2018).

**OpenFace**

OpenFace is a facial recognition library using deep convolutional neural networks for its facial recognition. It uses the neural network approach based on Google’s Facenet systems. It contains a neural network that was trained with 500 thousand pictures. Recognition can be done by providing the Python access point images and calling the face detection plus pre-processing methods on the given picture. OpenFace will then add it to the neural network thus making recognition simple to implement (Amos, Ludwiczuk & Satyanarayanan 2016, 1.).

**OpenBR**

OpenBR is an open-source framework launched in 2013. It is based on OpenCV for its algorithms and uses the Qt framework for its cross-platform compatibility. The framework is especially designed for fast algorithm prototyping and is able to provide a mature core framework. Besides the core framework, OpenBR offers plug-ins like HOG, PCA and more, which can be added if needed (Klontz, Klare, Klum, Jain & Burge 2013).

### 3.1.5 Algorithms and solutions comparison

Whether it be Eigenface, Fisherface or Local Binary Patterns Histogram, each solution to the facial recognition requirement has its own advantages and disadvantages. Therefore, the presented algorithms are nowadays used in conjunction to each other depending on a project’s needs. Although the algorithm’s recognition rate can be benchmarked, usability heavily depends on the given use cases it is applied to.
The research of available open-source solutions has shown that OpenCV is the basis of facial recognition which has been developed for a long time, whereas OpenFace and OpenBR are rather new. The later solutions do also tackle different problems of facial recognition. OpenBR for example intensively is designed for fast algorithm prototyping, whilst OpenFace is trying more of a different neural network approach. OpenFace and OpenBR are also out-of-the-box solutions, therefore offering rather less learning experience when implementing a project. OpenCV is also especially designed for real-time applications. The project aims for real-time appliance through the camera stream, therefore making this feature a core functionality that is needed for the project to work.

Delbiaggio’s research on facial recognition algorithm benchmarks shows that OpenFace and LBPH have the highest recognition accuracy on a test set of 5 different persons with 40 pictures each, with Eigenface and Fisherface following with lower accuracy (Delbiaggio 2017).

The combined research therefore determines use of the LBPH algorithm to get accurate facial recognition results as well as using OpenCV for learning experience and real-time appliance.

3.2 The Internet of Things

3.2.1 Introduction to the Internet of Things

The Internet of Things (IoT) is a technology to connect embedded systems over a network which can communicate without human intervention. It is also called machine-to-machine (M2M) communication. It allows these devices to collect data, evaluate it and generate an answer (Burgess 2018). The devices connected to the Internet of Things can be any physical object. The physical objects, which are considered under the term IoT, are devices that would generally not be expected to have an Internet connection, such as cameras, coffee-makers, fridges, etc. The data collected depends on the sensors and information the device provides (Ranger 2018).

3.2.2 Protocols and techniques

REST

REST is a well-known architectural style for defining constraints on an HTTP-based webservice. To achieve a RESTful system, six constraints must be applied, which are client-server architecture, statelessness, cacheability, layered system, code on demand and a uniformed interface. In terms of IoT the architectural style is mostly used in
combination with the coAP protocol or http-based traffic. It is therefore possible to map devices through URLs as well as defining GET, POST, PUT and DELETE requests on them (Fielding 2000, 1.).

**MQTT**

Message Queuing Telemetry Transport, MQTT, is a lightweight publish/subscribe messaging application layer protocol developed by IBM. MQTT is an ISO standard and works on top of the TCP/IP protocol stack. It allows publishing and subscribing from client as well as the server side. Because the publish/subscribe mechanism has a small overhead, it is possible for the protocol to minimize network traffic. The protocol will only send a message if a client or server publishes a message (Cohn & Coppen 2014).

**Constrained Application Protocol**

The Constrained Application Protocol (coAP) was developed to eliminate the use of hard-coded devices connected to webservices and therefore especially for machine-to-machine communication in the Internet of Things. It offers a broad spectrum of functionality like built-in discovery of services, request/response between application end points and multicast. The protocol is also designed to be easily integrated with http for Web interfaces (Shelby, Hartke & Bormann 2014, 1.).

**XMPP**

Extensible Messaging and Presence Protocol (XMPP) is an application layer protocol developed by Cisco using Extensible Markup Language, XML. Through the use of XML, it can offer near-instant messaging for structured data between two or more network entities. To work, XMPP clients are connected to an XMPP server. If a client is online, it shares the status of the client with all other clients connected to the server, thus making messaging instant between two clients that are online at the same time. For communication between an offline client connected to the same server, the message will get relayed to the offline client regardless. When the to-be-reached client is not on the same network, the XMPP server will query other servers until the to-be-reached client is found (Saint-Andre 2011, 1.).
3.2.3 Available solutions

**AWS IoT Core**

AWS IoT Core is one of Amazon’s paid services for IoT solutions. It offers a broad spectrum of functionality which consists of a Device SDK, Device gateway, Message Broker, Authentication, Registry, Device shadow and a rules engine. The listed functionality therefore allows communication between any device to Amazon’s cloud over HTTPS, WebSockets or secure MQTT. Data sent to the cloud can also be pre-processed with the rules engine allowing filtering and transformation of the sent data. Furthermore, the Registry functionality keeps track of connecting devices and the device shadow allows integration of cloud and mobile application to the AWS IoT Service (Amazon 2018a). Pricing for the service is done by what functionalities are used, meaning price per connectivity/day, per million messages sent, per rules applied in the rule engine or per device shadow and registry used. Prices are also depending on which cloud computing location will be used, for example Frankfurt is more expensive than Virginia (Amazon 2018b).

**Azure IoT hub**

Azure IoT Hub is a paid service included in the Azure IoT suite package provided by Microsoft. The services offered, similar to AWS IoT Core, range from Device SDKs, gateways to authentication and message brokers. Noticeable differences are device twins and price point. Device twins can store, query and synchronize metadata of a device in JSON format (Berdy & Betts 2018). Azure IoT Hub’s price point comes at 21.09 euros per month for the standard version S1 allowing 400,000 thousand messages of 4KB a day. In contrast to the S1 version, Azure allows pricing per device, meaning 8.433 euros per device per month with the same amount of messages/size (Microsoft Azure 2018).

**Paho MQTT**

Paho MQTT is an open-source client implementation of the MQTT protocol used in most open-source IoT scenarios developed by Eclipse. It offers client integration for embedded systems in C, Python and Java. With its broad integration possibilities, it is therefore considered one of the go-to libraries for MQTT protocol scenarios where the client needs to be integrated in running C, Python or Java code (Eclipse Wiki 2018).
**Mosquitto**

Mosquitto is a lightweight open-source MQTT broker/server for MQTT clients to connect, publish messages or subscribe to topics. It was developed through the iot.eclipse.org project by Eclipse and is suitable for all kinds of devices ranging from single boards to full servers (Mosquitto 2018).

### 3.2.4 Techniques and solutions comparison

The listed protocols and techniques all provide their own usability for given use cases. Comparing to what is used by big corporations like Amazon and Microsoft in IoT solutions, it becomes clear to see that the direction for IoT network traffic implementations is heading towards MQTT and HTTPS.

Comparing the available solutions, AWS and Azure, shows that if a project prefers using open-source software, scalability not being a dominant factor and Python integration is needed, Eclipse’s open-source MQTT implementation is to be chosen.

Based on these analyses in the research, an MQTT broker/server using the Mosquitto library as well as an Android and Raspberry Pi client using the Paho MQTT library will be used.
4 Empirical Section

The following empirical section will first present the project idea and the project phasing to achieve the final product. It will then present each individual phase with the technologies used and its code basis.

4.1 Project overview

The end product will have the following functionality:

![Diagram of General system set-up](image)

**Figure 4** General system set-up

Starting from the Raspberry Pi in the Figure above, the achieved system has a camera connected to the Pi over USB. The Raspberry Pi is responsible for detecting the homeowner's face. It will handle dataset generation, machine learning and recognition. As soon as the owner’s face gets recognized or an unknown face enters the apartment, a notification will be sent to the user’s mobile phone. For this functionality a message is sent to an IoT hub/server which will then forward it to our mobile phone. When the application receives a message from the server, it will then notify the user through notification on the phone.
4.2 Project phasing

To achieve the demonstrated system, the project will be split into 5 phases.

- Phase 1: Raspberry Pi setup, network structure
- Phase 2: Implementing facial recognition using USB camera
- Phase 3: Implementing push/pull notifications over IoT hub/server
- Phase 4: Implementing mobile application notifications
- Phase 5: Testing, extending functionality (false positives, UI App, Port forwarding)

4.3 Project implementation

4.3.1 Phase 1: Raspberry Pi setup, network structure

Raspberry Pi setup

The operating system running on Raspberry Pi is most commonly Raspbian. All the used technologies have a Raspbian-supported port, therefore the operating system chosen for the Raspberry Pi is Raspbian. As for versioning, the system will be using the newest available version, which is Raspbian Stretch. Wheezy and Jessie versions are not wise to use, as sources for used are mostly open-source, which constantly change. Therefore, having the newest available OS version running is an advantage. To not work with the default image given by Debian, a system upgrade was run on the Raspberry Pi to get all the latest updates and programmes.
Network structure

To connect to the Raspberry Pi and work on it, a network structure which has an Internet connection directly from the Raspberry Pi and a way to debug locally is essential. Consequently, a local network which allows connecting from the laptop to the Raspberry Pi whilst also having an Internet connection on both devices is needed. Furthermore, a server to send messages from the Raspberry Pi to the Android application must be set up. As the thesis aims for a low-budget home security system, buying a server from a provider is not a solution. Therefore, our Raspberry Pi will need to act as client and server at the same time. Access to the Raspberry Pi from the outside is given by port forwarding on the local network’s router. The Figure below demonstrates the project’s network set-up.

![Network set-up diagram]

Figure 5 Network set-up
4.3.2 Phase 2: Implementing facial recognition using USB camera

As seen in the project overview chapter, the first implementation to do is the facial recognition with the USB camera.

Camera

Because most facial recognition open-source libraries support USB cameras, no Raspberry Pi camera will be used. The camera should preferably be a USB camera which works out-of-the-box on any Linux distribution, as it eliminates the need to bother with drivers. The only available cameras in Helsinki are therefore from the brands Logitech and Creative. Logitech’s price point was mostly higher at around 40 to 50 euros, whereas Creative’s is at 20 to 30 euros. Therefore, a “Creative Live! Cam Sync HD Webcam VF0770” will be used for the facial recognition. The camera allows getting up to 4 frames per second and Linux support out-of-the-box.

Facial Recognition in OpenCV

The project’s facial recognition is developed in 3 parts:

- Data gathering
- Machine learning
- Facial recognition

To make the technology clear, each part will be explained and gone through by its Python code individually.

Data gathering

The data gathering phase (docs.opencv.org, 2017) (Kar, 2017) is used to create a dataset that is structured in a concurrent way. Without a well-structured dataset of high quality, the recognition will be worse, as the machine will learn on the created dataset. This step is therefore crucial for a successful operation. For these reasons, a script which automates this process was coded. It contains the following:
import cv2
import time

# gets the camera object
cam = cv2.VideoCapture(0)

# initialize face detection using default cascade
detector=cv2.CascadeClassifier('haarcascade_frontalface_default.xml')

# FaceId, used to iterate picture files later, has to be Int
faceId=raw_input('enter ID')

# counter for pictures taken
counter=0

while(True):
    # opens video stream
    vstream = cam.read()
    # input grayscale image, holds video stream frames
    gray = cv2.cvtColor(vstream, cv2.COLOR_BGR2GRAY)
    # detects faces in video stream with default cascade
    faces = detector.detectMultiScale(gray, 1.3, 5)

    for (x,y,w,h) in faces:
        # draws a rectangle
        cv2.rectangle(vstream,(x,y),(x+w,y+h),(255,0,0),2)
        # increment counter
        counter+=1
        # declares the image path
        imagePath="dataSet/User."+faceId+'.'+str(counter)+".jpg"
        # get the detected face from input grayscale image
        detectedFace=gray[y:y+h,x:x+w]
        # writes image to disk
        cv2.imwrite(imagePath,detectedFace)
        # reads image from disk
        cv2.imread(imagePath)
        # sleeps 50 milliseconds before detecting next face
        time.sleep(.5)

    if counter>50:
        break

# cleanup
print("\nsuccessful generation, exiting")
cam.release()
cv2.destroyAllWindows

Figure 6 datasetGeneratorScript.py

The script works in general as an automatic picture-taking program. It will start the camera, detect if a face was found in the camera stream and, if yes, save that picture on the disc in a concurrent way.

The script uses OpenCV's face detection algorithm using a default classifier, which is a given dataset by OpenCV with thousands of faces. It is used to detect a face from an image, in this example a frame from the video stream. Line 7.

To process an image in OpenCV, a grayscale image is needed. Therefore, each frame from the video stream (vstream) will get pre-processed to a grayscale image Lines 15, 17. Finally gets run through the detection algorithm Line 19. Parameters in detectMultiScale function are scaleFactor, 1.3, and minNeighbors, 5. The scaleFactor defines what happens if the face on the video stream is smaller than the one the algorithm must
compare it to. In a simpler way, at what point should a face be detected when the person is far away from the camera. The minNeighbors parameter specifies how many neighbors each candidate rectangle should have to retain it. In other words, it will define how strict a recognition gets processed, the higher the number, the stricter. The values 1.3 and 5 were chosen through testing the algorithm’s accuracy and recommendations to reduce false positive and true positive recognitions.

If one or multiple faces get detected by the detector, the coordinates get saved on the “faces” object. For each frame it thus contains a detected face in the form of a rectangle. X and y tells where in the picture the face got detected. W and h contains the width and height of the face from the x, y coordinates, line 19 callback. It then goes and loops the object for each face that has been detected in the frame, line 20. Finally, only the face itself is relevant to get saved on the disc. Therefore, the face gets extracted from the pre-processed grayscale frame in the beginning with the coordinates of the detection, line 28. The extracted face now gets saved on the disc, line 30.

For concurrency purposes, the script saves each picture in a similar fashion on the disc “dataset/User.Id.Counter”. This means that the given Id at the start of the programme will replace the “Id” in the path and the “counter” will show the picture’s number as 50 pictures will be taken for the dataset. Lines 10, 12, 24, 26, 30.

To show the user the taken picture, it will read the just saved picture from the disc, line 32. To have a chance to take pictures from different angles, the algorithm waits 50 milliseconds between each picture taken, line 34.

As soon as 50 pictures have been taken, the programme exits with some clean-up. Lines 36 – 41.
Machine learning

In the machine-learning phase, a training file which contains information of our face in LBPH form is created (OpenCV 2017; Kar 2017). Based on this file, the algorithm will later in the facial recognition phase decide, whether a face is in the video stream or not. The pictures must be converted to uint8 and all appended to a single array for the facial recognition to create a training file, therefore needing careful pre-processing.

```python
1 import cv2, os
2 import numpy as np
3 from PIL import Image
4
5 #initialize facial recognition using LBPH algorithm
6 recognizer = cv2.face.LBPHFaceRecognizer_create()
7 #initialize face detection using default cascade
8 detector = cv2.CascadeClassifier("haarcascade_frontalface_default.xml");
9
10 def getDataSetFromPath(path):
11     #gets the paths for all the files in the folder
12     imagePaths=[os.path.join(path,f) for f in os.listdir(path)]
13     #creates empty list for faces
14     facesToTrain=[]
15     #create empty list for Ids
16     Ids=[]
17     #now looping through all the image paths
18     for imagePath in imagePaths:
19         # only use files with a .jpg extension :
20         if os.path.split(imagePath)[-1].split(".")[-1]!="jpg":
21             continue
22
23         #loads the image and converts it to gray scale
24         imagePIL=Image.open(imagePath).convert(‘L’)
25         #convert PIL image to numpy array
26         imageNumpy=np.array(imagePIL,"uint8")
27         #gets the id of the image from the image path
28         Id=int(os.path.split(imagePath)[-1].split(".")[1])
29         #gets the face from the numpy image loaded
30         faces=detector.detectMultiScale(imageNumpy)
31         #append to face to list facesToTrain and id to list Ids
32         for (x,y,w,h) in faces:
33             facesToTrain.append(imageNumpy[y:y+h,x:x+w])
34             Ids.append(Id)
35             #return the lists
36             return facesToTrain,Ids
37
38 #load the lists from method
39 faces,Ids = getDataSetFromPath(‘dataSet’)
40 #train the algorithm
41 recognizer.train(faces, np.array(Ids))
42 #write the trained cascade file to disk
43 recognizer.write(‘trainer/trainer.yml’)
```

Figure 7 trainingScript.py
Generally, the script will first extract all the pictures taken from the file path of where they were saved before one by one. The programme then converts the pictures to uint8 and saves them in an array. As the pictures are uint8, the faces must be recognized again to get their coordinates. The programme then tries to detect a face with the old coordinates on the new uint8 picture and then appends it to the object that holds all of the faces. Finally, the training function of OpenCV gets called with the pre-processed data and the training file gets written on the disc once the algorithm is done training.

The script uses OpenCV for face detection and training, PIL to read images from the file system and numpy to create the multidimensional array needed for the training. From the OpenCV library 2 objects are created. One is the recognizer object which will be responsible for creating the trained data using LBHP recognition and the detector object which is the same as in the previous script, detecting faces. Lines 1 – 8.

The function getDataSetFromPath() will take care of the pre-processing needed before the training. It will return the Id of the picture, which is needed in case there are multiple faces to train the algorithm on, and the faces in the form of a numpy array, as the recognizer object which creates the training files needs an array in uint8. Lines 10, 12, 14, 16, 37.

The algorithm then loops all of the files in the path and first checks if the file selected is a picture with the .jpg extension, if it is, then it continues, and if not, it will be ignoring the file. Lines 18 – 22.

It will then open the picture by the given path, convert it to grayscale and save it to a variable. The programme will then continue to convert the read grayscale picture to a numpy array in uint8 format and save it to a variable using the numpy library. After that, the Id of the picture gets extracted from the path and saved on a variable. Lines 25 – 29.

To eliminate false positives, the face detection algorithm gets run over the picture taken again. This is done in case the previous algorithm picked up a face where there was none. If a face is detected again, both Id and the face are added to the list, which will be returned after all paths have been processed. Lines 31 – 37.

Finally, the function created to get the pre-processed data is called and saved on variables. It then calls the method train from the LBHP recognizer object and gives it the pre-processed data. As soon as the training finishes, the training file, which is a Local Binary Patterns Histogram, is written on the disc. Lines 40 – 44.
Facial recognition

The following script is the heart of the project, containing logic, IoT and timers (OpenCV 2017; Rovai 2018). To simplify, the code has been reduced to the facial recognition part only. Later, the script will be extended with the needed functionalities. In its current state, the following script will be able to fully recognize a face with the trained data.

```python
import cv2
import os

recognizer = cv2.face.LBPHFaceRecognizer_create()
recognizer.read('trainer/trainer.yml')
detector = cv2.CascadeClassifier("haarcascade_frontalface_default.xml")

font = cv2.FONT_HERSHEY_SIMPLEX

id = 0
names = ['None', 'David']
cam = cv2.VideoCapture(0)

while True:
    _, vstream = cam.read()
    gray = cv2.cvtColor(vstream, cv2.COLOR_BGR2GRAY)
    faces = detector.detectMultiScale(gray, scaleFactor=1.2, minNeighbors=5, minSize=(int(minW), int(minH)),)

    for (x,y,w,h) in faces:
        cv2.rectangle(vstream, (x,y), (x+w,y+h), (0,255,0), 2)
        id, confidence = recognizer.predict(gray[y:y+h,x:x+w])

        if (confidence < 70):
            id = names[id]
        else:
            id = "unknown"

        cv2.putText(vstream, str(id), (x+5,y-5), font, 1, (255,255,255), 2)
```

20
In general, the script will first detect faces from a video stream the same way as the datasetGenerator script used in the first part. When a face is detected it will then compare it to the trained dataset that was created with the previous script. The OpenCV algorithm will then give a confidence level on how exactly it recognized a face or not. Based on the confidence level, a rectangle with the name of the recognized person will be drawn on the frame.

The script first makes imports for the required libraries, this time being the OpenCV and the OS access libraries. In the initialising phase, first two different OpenCV objects are created. One being again the detector using the default cascade file to detect faces, and the other is an empty LBPH face recognition object to which the training data from before will be fed. The script then gets a font constant from OpenCV. It then initialises an Id counter and an array containing the names for the given Ids in the file path. For example, in the datasetGenerator script the Id 1 got typed. Therefore, if it’s the face of a person called David, the second position in the array needs to contain the string David. The array is needed to later show the name of the recognized person on the video stream.

The script then initialises the camera with the width and height of the window to be opened, as well as a minimal size of a face to is to be recognized on the stream. For example, how far away can a person be for a camera to still recognize a face.

As initialising is done, the script then enters an endless loop till the user gives the input of the “ESC” key to exit the programme with clean-ups.

After entering the endless loop, it will read the camera stream frame by frame and try to detect a face from it using the converted grayscale frame and the minimal face detection sizes declared in lines 24 and 25.

When a face is detected, the algorithm will loop it and draw a rectangle around it. It then gives the face to the OpenCV LBHP face recognition object containing our trained faces.
This method will return a confidence level of the highest prediction of the training data. For example, a face with an Id of 1 got recognized with a chance of 75. With this information the algorithm will then decide whether the recognition was good enough to be sure that it is really the correct person. Some testing decided that 70, gives the best results, thus when it returns lower than 70 the name of the recognized person is written next to the rectangle of the face. If the prediction is higher than or equal to 70, it will write unknown instead of the person’s name. Lastly the output is opened so that the user can see the video stream. Lines 38 – 48.

4.3.3 Phase 3: Implementing push/pull notifications over IoT hub/server

With Phase 2 completed, facial recognition is now fully working on the Raspberry Pi. Therefore, the project can move on to the next phase, which is implementing an IoT solution to push and pull notifications from a server. Referencing Figure 1 in the Product overview chapter, the following phase will focus on the server and client creation on the Raspberry Pi as well as on the Android application.

Mosquitto broker/server installation/set-up

The installation process for the Mosquitto broker/server is extremely simple on Debian Stretch versions as the Mosquitto package is included in the apt-get source list. Running apt-get install mosquitto will install the server. For the set-up the default config can be used as it is up to the clients to subscribe to or publish topics. Therefore, restarting the service with a “service mosquitto restart” command and a run of “mosquitto” will be enough to make the server working on the Raspberry Pi.

Paho MQTT client Raspberry Pi

For the Raspberry Pi client, the solution will make use of a package called Paho MQTT. It is an open-source client package created by Eclipse using the MQTT protocol. As the facial recognition script is coded in Python, it is essential for the MQTT client to be integrated in this Python script. Paho MQTT offers the needed integration to the Python script. The Paho MQTT package can be installed through a “pip install paho-mqtt” command on the Raspberry Pi.
Paho MQTT Client Raspberry Pi integration

At this point, the facial recognition script gets extended with the Paho MQTT client (Eclipse 2018). Therefore, the following code will be used. It will allow subscribing to and publishing messages on the MQTT broker/server.

```python
import cv2
import os
import paho.mqtt.client as mqtt

# ========mqtt client init ========
def connectionStatus(client, userdata, flags, rc):
    mqttClient.subscribe("FacialRecognition/reset")

# ========facial recognition init ========
def on_publish(client, userdata, result):
    print("data published \n")

# init mqtt client name
clientName = "RPI"

# init mqtt server address
serverAddress = "127.0.0.1"

# init mqtt client object
mqttClient = mqtt.Client(clientName)

# init all functions for mqtt client
mqttClient.on_connect = connectionStatus

# Connect client to server
mqttClient.connect(serverAddress)

# Assign the callback function
mqttClient.on_publish = on_publish

# initialize facial recognition using LBPH algorithm
recognizer = cv2.face.LBPHFaceRecognizer_create()

# initialize face recognition using before trained data
```
recognizer.read('trainer/trainer.yml')

# Initialize face detection using default cascade
detector= cv2.CascadeClassifier("haarcascade_frontalface_default.xml");

# gets font constant from opencv
font = cv2.FONT_HERSHEY_SIMPLEX

# initialize id counter
id = 0

# names for ids (0,1,2,...):
names = ['None', 'David']

# initializes the camera object
cam = cv2.VideoCapture(0)

# set width
cam.set(3, 640)

# set height
cam.set(4, 480)

# Define min window size to be recognized as a face
minW = 0.1*cam.get(3)
minH = 0.1*cam.get(4)

while True:
    # opens video stream
    _, vstream = cam.read()

    # input grayscale image, holds video stream frames
    gray = cv2.cvtColor(vstream, cv2.COLOR_BGR2GRAY)

    # detects faces in video stream with default cascade
    faces = detector.detectMultiScale(gray,
                                       scaleFactor = 1.2,
                                       minNeighbors = 5,
                                       minSize = (int(minW), int(minH)),
                                       )

    for(x,y,w,h) in faces:
        cv2.rectangle(vstream, (x,y), (x+w,y+h), (0,255,0), 2)

        # Check if confidence is less than 70 => "0" is perfect match
        id, confidence = recognizer.predict(gray[y:y+h,x:x+w])

        if (confidence < 70):
            #print(""Face: ", names[id])
            ret = mqttClient.publish("facialrecognition/detect",id)
            else:
            id = "unknown"
            ret= mqttClient.publish("facialrecognition/detect","false")
        cv2.putText(vstream, str(id), (x+5,y-5), font, 1,
                    (255,255,255), 2)

        cv2.imshow('camera',vstream)
        k = cv2.waitKey(10) & 0xff
        if k == 27:
            break

    # cleanup
print("\n Exiting program on user command")
cam.release()
cv2.destroyAllWindows()

Figure 9 recognitionScript.py with Paho MQTT client
In general, the script works the same as the previous with the difference that a message now gets sent to the MQTT broker/server every time a face is detected. If it’s an unknown face, the algorithm will send “false” to the server, else it will send the recognized face name.

After importing the package, Paho MQTT requires defining callback functions to publish and subscribe to messages. It also needs a callback defined for the client-server connections. Lines 3 – 9.

The script then defines the client name and the server’s address. It then goes ahead to initialise the client object and defines the callback for it. As soon as the object and callback for the client are defined, it will get connected to the server. Lastly, the callback function to publish messages is assigned to the client object. Lines 12 – 22.

The client is now well formed and connected to the server. Messages will be sent if a face is recognized. It has two options for sending messages. A face which is unknown or known is recognized. Either way, the client will send a message to the same topic called “facialrecognition/detect” with the payload being “false” for unknown recognitions and “Name” for known recognitions. The Id variable contains the recognized face’s name. Lines 65 and 68.

**Paho MQTT client Android application**

To get messages to the Android phone, an Android application which will connect to the MQTT broker/server is created. As the same client library, Paho MQTT from Eclipse, is coded in Java, it naturally works for Android Studio through compiling the library as Android Studio uses Java as its main programming language.

**Android application constraints**

As Android is a rapidly developing IDE, constraints for the mobile app first must be defined. To get the most out of the current available offerings from Android, the Oreo SDK version 8.1 will be used. Therefore, API level 27 must be chosen for the app and all the used libraries. The application will be running on a OnePlus 5T mobile phone which is also up to date, running the latest Android 8.0.0 version.
Paho MQTT client Android application integration

Therefore, the following MQTT client class is created (Thumar 2017). It acts the same way as the Paho MQTT client on the Raspberry Pi with the difference being that the callbacks are overwritten by our own need instead of being defined completely, thus making the code on Android Studio easier.

```java
1 package MQTT;
2 import android.content.Context;
4 import org.eclipse.paho.android.service.MqttAndroidClient;
6 import org.eclipse.paho.client.mqttv3.DisconnectedBufferOptions;
7 import org.eclipse.paho.client.mqttv3.IMqttActionListener;
8 import org.eclipse.paho.client.mqttv3.IMqttDeliveryToken;
9 import org.eclipse.paho.client.mqttv3.IMqttToken;
10 import org.eclipse.paho.client.mqttv3.MqttCallbackExtended;
11 import org.eclipse.paho.client.mqttv3.MqttConnectOptions;
12 import org.eclipse.paho.client.mqttv3.MqttException;
14
15 public class MqttClient {
16   public MqttAndroidClient mqttAndroidClient;
18   final String serverIP = "tcp://"+public ip*@1883;"
20   final String clientId = "AndroidClient@sperson";
21   final String subscriptionTopic = "facialrecognition/detect";
23
25   public MqttClient(Context context){
26       mqttAndroidClient = new MqttAndroidClient(context, serverIP, clientId);
27       mqttAndroidClient.setCallback(new MqttCallbackExtended()) {
28           @Override
29           public void connectComplete(boolean b, String s) {
30               
31           }
32           @Override
33           public void connectionLost(Throwable throweable) {
34               
35           }
36           @Override
37           public void messageArrived(String topic,
38              MqttMessage mqttMessage) throws Exception {
39               
40           }
41           @Override
42           public void deliveryComplete(IMqttDeliveryToken iMqttDeliveryToken) {
43               
44           }
45           
46       });
47       connect();
48   }
50   public void setCallback(MqttCallbackExtended callback) {
51       mqttAndroidClient.setCallback(callback);
52   }
```
```java
private void connect(){
    MqttConnectOptions mqttConnectOptions = new MqttConnectOptions();
    mqttConnectOptions.setAutoReconnect(true);
    mqttConnectOptions.setCleanSession(false);
    try {
        mqttAndroidClient.connect(mqttConnectOptions,
            null, new IMqttActionListener() {
                @Override
                public void onSuccess(IMqttToken asyncActionToken) {
                    DisconnectedBufferOptions disconnectedBufferOptions =
                        new DisconnectedBufferOptions();
                    disconnectedBufferOptions.setBufferEnabled(true);
                    disconnectedBufferOptions.setBufferSize(100);
                    disconnectedBufferOptions.setPersistBuffer(false);
                    disconnectedBufferOptions.setDeleteOldestMessages(false);
                    mqttAndroidClient.setBufferOpts(disconnectedBufferOptions);
                    subscribeToTopic();
                }
                @Override
                public void onFailure(IMqttToken asyncActionToken,
                                       Throwable exception) {
                    System.err.println("Error connecting to server!");
                    exception.printStackTrace();
                }
            });
        } catch (MqttException ex){
            ex.printStackTrace();
        }
    }
    private void subscribeToTopic() {
        try {
            mqttAndroidClient.subscribe(subscriptionTopic,
                0, null, new IMqttActionListener() {
                    @Override
                    public void onSuccess(IMqttToken asyncActionToken) {
                    }
                    @Override
                    public void onFailure(IMqttToken asyncActionToken,
                                           Throwable exception) {
                    }
            });
        } catch (MqttException ex) {
            System.err.println("Error subscribing!");
            ex.printStackTrace();
        }
    }

Figure 10 Paho MQTT client Android
```
The code in general works as an adaption for the given client by Paho MQTT. Therefore, methods must be overwritten and their callbacks defined. Other than that, this class works on its own when initiated anywhere else in the app.

First on, the programme imports all of the used classes in the client library given by Eclipse, which were compiled through the gradle file. Next on the server IP, the name it will connect as and the topic to subscribe to, will be defined. On class initialisation, the constructor will be called, which will initiate the client-server connection, set the callbacks, define the topics to subscribe to and keep it alive.

The messaging and establishment of the client-server connection is coded in the default way of how to code it according to the Paho MQTT documentation.

**Android app activity**

For each screen in an Android application, a logic class written in Java and an XML file which renders the UI is needed. For demonstration purposes and simplification, the UI will not be shown, but only the logic of the Paho MQTT client connection in the Java logic class.

```java
package fi.helia.haaga.thesisprojectotfacialrecognitionspoer;

import android.app.NotificationChannel;
import android.app.NotificationManager;
import android.content.Context;
import android.support.v4.app.NotificationCompat;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;

import org.eclipse.paho.client.mqttv3.IMqttDeliveryToken;
import org.eclipse.paho.client.mqttv3.MqttCallbackExtended;

import MQTT.MqttClient;

public class MainActivity extends AppCompatActivity {
    private MqttClient mqttClient;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        startMqtt();
    }
}
```
The above code works in a way that the client-server connection gets automatically initialised by creating a client object from the code before. Because what should happen as soon as a message arrives must be shown in the app UI or as notification, this functionality of the Paho MQTT client must be coded in the activity of the app.

The onCreate function at the beginning of the code triggers the startMqtt() method, which will then connect to the server and wait for messages. What should happen when a message arrives is defined in the override of the callback in the method “messagesArrived”. It will give back the message and it then is saved on a variable, whilst being casted to a string format.

### 4.3.4 Phase 4: Implementing mobile application notifications

With communication between Raspberry Pi and Android application over a server running on the Raspberry Pi now fully working, Phase 3 has ended. Therefore, the next development phase can begin, which will focus on sending notifications on the mobile phone through the Android application. It will focus on the Android library used. It will also extend the facial recognition script with functionality to not permanently send messages for each frame, thus not spamming the Android app anymore.
Notifications Android app

To push notification on a phone, the Android library support-compa will be used and compiled in version 27.1.0 (Android Developers 2018). The following code shows the activity holding the Paho MQTT client extended with the notification functionality.

```java
package fi.helia.haaga.thesisprojectiotfacialrecognitionsponer;

import android.app.NotificationChannel;
import android.app.NotificationManager;
import android.content.Context;
import android.support.v4.app.NotificationCompat;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;

import org.eclipse.paho.client.mqttv3.IMqttDeliveryToken;
import org.eclipse.paho.client.mqttv3.MqttCallbackExtended;
import MQTT.MqttClient;

public class MainActivity extends AppCompatActivity {
    private MqttClient mqttClient;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        startMqtt();
    }

    private void startMqtt() {
        mqttClient = new MqttClient(getApplicationContext());
        mqttClient.setCallback(new MqttCallbackExtended() {
            @Override
            public void connectComplete(boolean b, String s) {

            }

            @Override
            public void connectionLost(Throwable throwable) {

            }
        };
    }
```
The notifications in the support-compat library work in a way that a channel for the notification first needs to be created. After channel creation, the notification can be defined with a builder class and the necessary information it needs to hold. Because the channel has been established, firing the notification will directly tunnel through the created channel and therefore notify the phone with the defined message.
For the above code the changes mean that the information a notification should hold is defined by the message that is sent to the application (intruder or owner). Therefore, the method is called in the messageArrived callback which holds this necessary information. If the server sends “false”, then an intruder has been detected, therefore notifying the phone with an all-caps message and a warning icon. Else, it will send the detected face name, and the application will notify the phone with a welcome home message according to the name sent.

The shownotification method makes the notifications possible by using the support-compat library given builder and manager classes. It will first establish a new manager object, which will get the phone’s notification service. After that, variables for notifications, channel and importance of notification must be defined. As the channel solution is a new way of sending notification, the programme will first check if the phone’s Android version is 8.0.0 or higher. If it is, a channel needs to be created, if not, it can be ignored. Finally, the necessary notification is built and sent to the phone.
Notifications facial recognition script

To make the functionality of notification correctly available, changes in the script are needed. These include not spamming a recognition message for each frame where a face is recognized. The following code change is a workaround to wait for a specific number of frames before sending the next message. It is coded this way because a time.wait call from a library would pause the video stream and continue where it left off, therefore sending the next frame which got cached but not the real-time frame.

```python
1 import cv2
2 import os
3 import paho.mqtt.client as mqtt
4
5 #---mqtt client init----
6 def connectionStatus(client, userdata, flags, rc):
7     mqttClient.subscribe("facialrecognition/reset")
8
9 def on_publish(client, userdata, result):
10     print("data published \n")
11
12 #Init mqtt client name
13 clientId = "RPI"
14 #Init mqtt server address
15 serverAddress = "127.0.0.1"
16 #Init mqtt client object
17 mqttClient = mqtt.Client(clientId)
18 #Init all functions for mqtt client
19 mqttClient.on_connect = connectionStatus
20 mqttClient.on_message = messageDecoder
21 #Connect client to server
22 mqttClient.connect(serverAddress)
23 #Assign the callback function
24 mqttClient.on_publish = on_publish
25 #counts the frames when a message has been send
26 framecounter = 0
27 #switches to true when a message has been send
28 messageSend = False
29
30 #---facial recognition init------
31 #initialize facial recognition using LBPH algorithm
32 recognizer = cv2.face.LBPHFaceRecognizer_create()
33 #initialize face recognition using before trained data
34 recognizer.read("trainer/trainer.yml")
35 #initialize face detection using default cascade
36 detector = cv2.CascadeClassifier("haarcascade_frontalface_default.xml");
37 font = cv2.FONT_HERSHEY_SIMPLEX
38 #initialize id counter
39 id = 0
40 # names for ids (0,1,2,...):
41 names = ["None", "David"]
42 #Initializes the camera object
43 cam = cv2.VideoCapture(0)
44 # set width
45 cam.set(3, 640)
46 # set height
47 cam.set(4, 480)
48 # Define min window size to be recognized as a face
49 minW = 0.1*cam.get(3)
50 minH = 0.1*cam.get(4)
```
First the needed variables are defined. Lines 26 and 28.

Then the algorithm will first check if there has been no message sent yet before trying to recognize faces from the video stream. If that is the case, the algorithm continues the same way as previously with the only difference being the messageSend variable being given the value true as soon as a message is sent to the IoT server/hub. Lines 57, 72 and 77.

**Figure 13** recognitionScript.py with Paho MQTT client and notifications

The code changed with the functionality of now waiting around 40 seconds before sending another notification to the phone over MQTT.
If the messageSend variable is set to true, meaning a message was sent, then the algorithm will wait 1000 frames, which is equal to around 40 seconds, before setting the variable back to false and continuing the recognition. Lines 80 – 86.

4.3.5 Phase 5: Testing, extending functionality (false positives, UI App, Port forwarding)

False positives

The biggest problem with facial recognition algorithms are false positives. False positives is a problematic when faces are not being recognized when they should be, hence the algorithm throws false when it should throw true. As the project is building a home security solution, these false positives need to be eliminated as much as possible. From experience during the project, it was noticed that most false positives come from the first couple of frames where a face is detected. The reason for this is because in most scenarios a face is detected very early on in the facial recognition process, meaning the head of the person might be slightly tilted to the left, right, up or down in the camera stream. As the person would usually open the door in the home security solution where the camera is set up at the entrance, it would throw the same kind of false positives which have to be fixed. Therefore, the following changes have been made to the script:

```python
1 import cv2
2 import numpy as np
3 import paho.mqtt.client as mqtt
4 #---mqtt client init-----
5
6 def connectionStatus(client, userdata, flags, rc):
7     mqttClient.subscribe("facialrecognition/reset")
8
9 def on_publish(client,userdata,result):
10     print("data published \n")
11 #init mqtt client name
12 clientId = "RPI"
13 #init mqtt server address
14 serverAddress = "127.0.0.1"
15 #init mqtt client object
16 mqttClient = mqtt.Client(clientId)
17 #init all functions for mqtt client
18 mqttClient.on_connect = connectionStatus
19 mqttClient.on_message = messageDecoder
20 #Connect client to server
21 mqttClient.connect(serverAddress)
22 #Assign the callback function
23 mqttClient.on_publish = on_publish
24```
#### facial recognition init

```python
# initialize facial recognition using LBPH algorithm
recognizer = cv2.face.LBPHFaceRecognizer_create()
# initialize face recognition using before trained data
recognizer.read('trainer/trainer.yml')
# initialize face detection using default cascade
detector = cv2.CascadeClassifier('haarcascade_frontalface_default.xml');
font = cv2.FONT_HERSHEY_SIMPLEX
# initialize id counter
id = 0
framecounter = 0
messagesend = False
# Ignore the first 3 frames of face detection
facedetectcounter = 0
frameswaited = False
# names for ids (0,1,2,...):
names = ['None', 'David']
# initializes the camera object
cam = cv2.VideoCapture(0)
# set width
cam.set(3, 640)
# set height
cam.set(4, 480)
# Define min window size to be recognized as a face
minW = 0.1*cam.get(3)
minH = 0.1*cam.get(4)
while True:
    # opens video stream
    _, vstream = cam.read()
    # input grayscale image, holds video stream frames
    gray = cv2.cvtColor(vstream, cv2.COLOR_BGR2GRAY)
    if not messagesend:
        # detects faces in video stream with default cascade
        faces = detector.detectMultiScale(gray,
                                           scaleFactor = 1.2,
                                           minNeighbors = 5,
                                           minSize = (int(minW), int(minH)),
                                           )
        for (x1,y1,w1,h1) in faces:
            cv2.rectangle(vstream, (x1,y1), (x1+w1,y1+h1), (0,255,0), 2)
            id, confidence = recognizer.predict(gray[y1:y1+h1, x1:x1+w1])
            if frameswaited:
                # Check if confidence is
                # less than 70 => "0" is perfect match
                if (confidence < 70):
                    id = names[id]
                    msg = mqttClient.publish
                    ("facialrecognition/detect", id)
                else:
                    id = "unknown"
            messagesend = True
```
The recognition script was changed slightly, so that it will only do the face recognition when three consecutive faces have been detected on the video stream.

First a Boolean and a counter-variable are declared, which will be needed to control the frames and when the recognition should be executed. Lines 38 – 40.

After a face is detected, the algorithm will now check if the framesWaited variable is set to true, meaning 3 face detected frames have been waited or not. If it is, the algorithm continues as usual. If not, the algorithm checks if the waited face detected frames are 3. If that is the case, the framesWaited Boolean can get set to true, else the counter gets increased by one. Lines 68 and 84 – 90.
User interface application

The user interface is kept simple. For the system to work, the app must be launched and run in the background. The main functionality consists in the notifications, which a user can also receive outside of the app. Therefore, only text that explains the system’s functionality is added inside the home screen of the application.

```
1 <string name="app_welcome">Welcome</string>
2 <string name="app_info">
3     To use the app, please make sure it is running actively or in the background.
4     As soon as your face or an intruder will get detected, you will receive a notification.
5 </string>
```

Figure 15 Android app strings

Above are the 2 strings which are added to the project’s strings.xml file.

```
1 <xml version="1.0" encoding="utf-8"?>
2     <android.support.constraint.ConstraintLayout
3         xmlns:android="http://schemas.android.com/apk/res/android"
4         xmlns:app="http://schemas.android.com/apk/res-auto"
5         xmlns:tools="http://schemas.android.com/tools"
6         android:layout_width="match_parent"
7         android:layout_height="match_parent"
8         tools:context="fi.helia.haaga.thesisproject1otfacialrecognitionwsponer.MainActivity">
9       <TextView
10          android:id="@+id/welcomeText"
11          android:layout_width="144dp"
12          android:layout_height="31dp"
13          android:layout_marginBottom="8dp"
14          android:layout_marginEnd="8dp"
15          android:layout_marginStart="8dp"
16          android:layout_marginTop="8dp"
17          android:text="@string/app_welcome"
18          android:textAppearance="@style/TextAppearance.AppCompat.Large"
19          app:layout_constraintBottom_toTopOf="@+id/infoText"
20          app:layout_constraintEnd_toEndOf="parent"
21          app:layout_constraintStart_toStartOf="parent"
22          app:layout_constraintTop_toTopOf="parent"
23          app:layout_constraintVertical_bias="1.0" />
24       </TextView>
25
26       <TextView
27          android:id="@+id/infoText"
28          android:layout_width="292dp"
29          android:layout_height="117dp"
30          android:text="@string/app_info"
31          android:.textAlignment="center"
32          app:layout_constraintBottom_toBottomOf="parent"
33          app:layout_constraintEnd_toEndOf="parent"
34          app:layout_constraintStart_toStartOf="parent"
35          app:layout_constraintTop_toTopOf="parent" />
36
37
38     </android.support.constraint.ConstraintLayout>
```

Figure 16 Android app UI
The xml file above is the main activity’s user interface file. It consists of a constraint layout to position elements and the two text views which hold the strings declared in the strings.xml file.

**Port forwarding**

The Raspberry Pi needs to be accessible from outside the home network, as the server needs to send messages to the phone even when it is not connected to the home network. To make the accessibility, the port used for the MQTT server/broker needs to be port forwarded on the home router. It is done through creating a virtual server on a router’s Web interface.

![Port forwarding](image)

**Figure 17** Port forwarding 1883

The Raspberry Pi in the home network has a static IP address of 192.168.43.30, therefore with the above configuration all packets with port number 1883 and IP 192.168.43.30 will be forwarded to the Raspberry Pi.

**4.4 Project result**

This chapter will show all the scripts in execution. It will start off with the facial recognition configuration and continue to the home set-up for the camera. Furthermore, a scenario of me as a homeowner who is coming home and a scenario where an intruder enters the apartment will be shown.
Data gathering

As pictures of a homeowner are needed, the script to gather faces is executed. On execution it will prompt an Id which is used to index the person of whom the pictures are taken. If it were the second person who will be added to the system, Id number 2 needs to be given. The results shown will start from a not configured set-up, therefore 1 will be entered.

When prompting Id 1, the camera will start. The camera now needs to be pointed to the owner’s face. The script will only take a picture when a face is recognized. To improve the quality of the later machine learning and recognition, tilting the head slightly in different directions is recommended, so that pictures of the face can be taken from different angles.

As soon as 50 pictures have been taken, the script exits, and the camera will be stopped. Looking in the dataSet folder on the Raspberry Pi, it is now possible to spot the 50 pictures that have been taken from the executed script.

Figure 18 datasetGeneratorScript.py execution

Figure 19 DataSet folder after execution
Machine learning

This step is simple in the set-up. The machine learning script has to be executed and the programme does the rest. Checking the trainer folder will show that it now contains a histogram trained with the pictures that were taken before.

```
pi@raspberrypi:~/thesisProject/facialRecognition $ python trainingScript.py
```

**Figure 21** trainingScript.py execution

![Figure 22 Trainer folder after execution](image)

**Figure 22** Trainer folder after execution
As the next script that will be used is the facial recognition script, which is connected to the MQTT broker, the broker needs to be running and listening for connections. Executing a mosquitto command on the Raspberry Pi will start the broker.

```
oci@raspberrypi:/thesisProject/facialRecognition $ mosquitto
1524762889: mosquitto version 1.4.15 (build date Wed, 28 Feb 2018 11:29:47 +0000) starting
1524762889: Using default config.
1524762889: Opening ipv4 listen socket on port 1883.
1524762889: Opening ipv4 listen socket on port 1883.
```

Figure 24 Mosquito broker start

Camera set-up

To get a realistic approach, the camera will be set-up in a place where it will constantly have the door in its frame. Taking this approach, anyone who enters the room will be forced to be in the video stream’s frame, therefore making a recognition in any given case.
Android application

To receive the facial recognition notifications on the phone, the Android app must be used actively or be running in the background. Starting the application will automatically connect it to the MQTT broker.
Facial recognition

Finally, it is now time to execute the facial recognition script. Checking the terminal shows that after execution both Android application and the Paho MQTT client on the Raspberry Pi are connected to the mosquitto MQTT broker.

```
pi@raspberrypi:~/thesisProject/facialRecognition $ python3.5 recognitionScript.py
```

**Figure 27** recognitionScript.py execution

```
pi@raspberrypi:~ $ mosquitto
1524674079: mosquitto version 1.4.15 (build date Wed, 28 Feb 2018 11:29:47 +0000) starting
1524674079: Using default config.
1524674079: Opening ipv4 listen socket on port 1883.
1524674079: Opening ipv6 listen socket on port 1883.
1524674907: New connection from 127.0.0.1 on port 1883.
1524674907: New client connected from 127.0.0.1 as RPI (ci, k60).
1524674922: New connection from . . . on port 1883.
1524674922: New client connected from . . . as AndroidClientSpsoner (c0, k60).
```

**Figure 28** MQTT broker connections

The system is now running. Following, two test cases are presented. Each will show the recognition on the video stream and notification on the mobile phone.
Case One: Homeowner enters

As the homeowner enters the room, the video capture shows that he was recognized, and the script will therefore send a message containing the recognized face to the Android application. The app will respond to a message containing a name with a greeting notification to the phone including the homeowner’s name.

Figure 29 Homeowner recognition

Figure 30 Homeowner in app and general notification
**Case Two: Intruder enters**

In this scenario, a person who is not registered in the system enters the room. The system will thereby determine the recognized face as “unknown” and send a message containing “unknown” to the Android application. The app will respond to a message containing “unknown” with a warning message saying that an intruder has been detected at home.

![Figure 31 Unknown recognition](image)

**Figure 31 Unknown recognition**

![Figure 32 Unknown in app and general notification](image)

**Figure 32 Unknown in app and general notification**
5 Discussion and Conclusions

Goal of the thesis was building a low-budget home security system using facial recognition. Facial recognition and an IoT solution were implemented by using only open-source software, therefore making the costs of the system only hardware related. With the USB cameras ranging from 20 to 40 euros and a Raspberry Pi 3 model around 30-40 euros, the final product was built on hardware which costs less than 80 euros therefore completing the thesis goal.

As mentioned in the introduction, there are many use cases in the growing market of IoT and facial recognition where both technologies could be used in conjunction. The presented use case is only one of these. The thesis’s aim was therefore also to provide a low-budget solution for the need to prototype such systems. As the project’s code is explained and working, the thesis is also building a foundation of a to-be-easily-extendable system using facial recognition and the Internet of Things.

The MQTT protocol was used through the Paho MQTT library, therefore allowing IoT communication. Currently, the Android application only receives messages and the Paho MQTT client only sends messages. Nonetheless, functionality of sending from the Android application as well as receiving from the Paho MQTT client is coded. Future improvements to the system could consist of a picture view of the detected image sent to the Android app or even starting the facial recognition script by clicking a button on the application. The code to send and receive from both clients is already integrated and will therefore make future improvements to the system simple.

Concerning false positives or positive false predictions on the facial recognition algorithm, the used script can still be improved. The workaround of waiting the 3 first frames is a solution to eliminate the most common false positives or positive false errors, but it does not eliminate wrong predictions to 90% plus. Improvement could be done through testing facial recognition on the Raspberry Pi using the upcoming neural networks and deep learning algorithms. These algorithms show promising results, but are at the moment not feasible for Raspberry Pi implementations because of the missing computational power of the device. Further improvements in the algorithms through OpenCV or other libraries that provide neural networks and deep learning for facial recognition could allow for future integration in the system, therefore making its facial recognition part even more accurate.

Summarizing, a home security system using facial recognition and the Internet of Things was built with using only low-budget hardware and open-source software. IoT communication in both directions, Android to Raspberry Pi and vice versa, is coded and
can be used in further development. False positives have been reduced to a minimum with the recognition showing reliable results. Further improvements on the recognition part will depend on the improvements of the algorithms through neural networks and deep learning.
References


OpenCV 2017. Face Recognition with OpenCV. URL: https://docs.opencv.org/3.3.1/da/d60/tutorial_face_main.html. Accessed: 1 March 2018


