SMART VIDEO SURVEILLANCE FOR SLOW AND METERED CONNECTIONS

by

Halil Can Kaşkavalcı

Submitted to Graduate School of Natural and Applied Sciences

in Partial Fulfillment of the Requirements

for the Degree of Master of Science in

Computer Engineering

Yeditepe University

2019

SMART VIDEO SURVEILLANCE FOR SLOW AND METERED CONNECTIONS

APPROVED BY:

Prof. Dr. Sezer Gören Uğurdağ .....
(Thesis Supervisor)

Assoc. Prof. Dr. H. Fatih Uğurdağ .....

Assist. Prof. Dr. Ü. Onur Demir .....

DATE OF APPROVAL:

ACKNOWLEDGEMENTS

First and foremost I would like to thank my wife Ayşenur for everything she has done. She has been deeply supportive through this long process. She helped me as Galadriel helped Frodo through dark times. I am incredibly lucky to be with her.

I would like to express my endless gratitude to my beloved mother Ayşe. She sacrificed so much I can never pay it back. Many things would not happen if not her. Thank you for your endless love.

I wish to thank my extended family in particular Saadet Dağ, Erol Dağ, İmran Dağ, Fatoş Dağ, Cemal Sezer, Nebahat Sezer, Melehat Özer and İlhami Özer for their support throughout my entire life. I was part of this big family when I was growing up. Thank you for being so inclusive. My extended family got bigger with my wife. I cannot express my appreciation towards Mustafa Ulubaş and Pembe Ulubaş and my little sister Ceyda Ulubaş. Just keep being awesome.

Speaking of family, some friends become family. We don't have brothers but we become one with Okay Ayaz. I like to mention all my pals from Tekirdağ Anadolu Öğretmen Lisesi here, Yunus Can Kurban, Okan Kızılırmaklı, Halil Akbaş, Barış Ercan, Merve Erk Aydın and Nilay Tüfek. Finally, I would like to thank Yusuf Can Semerci for being an awesome friend.

I need to mention Dutch rail system. This work is done while commuting between Rotterdam and Amsterdam. Thank you NS for comfortable Intercity Direct trains.

Lastly and most importantly I would like to thank my thesis advisor Prof. Dr. Sezer Gören Uğurdağ. She always pushed me forward and helped me to complete this thesis.

ABSTRACT

SMART VIDEO SURVEILLANCE FOR SLOW AND METERED CONNECTIONS

Traditional surveillance is done by recording video footage to a storage device continuously. However this generates enormous amount of data and reduces the life of the storage by continuous writes. Newer devices allow user to connect to the camera on demand from the Internet. One should consider bandwidth and cloud costs. Upload speeds are usually capped by Internet Service Providers (ISP) and cloud storage is expensive. Majority of data recorded by an indoor camera does not contain any information and motionless. When there is a motion it is most likely to caused by the household members, not an intruder.

In this thesis we demonstrate a hierarchical cluster of cheap, disposable Raspberry Pi boards to monitor one or more cameras intelligently. System recognizes household members and does not save the data or trigger any action. When a face is not recognized locally, the image is uploaded to the cloud to perform further checks. If an image cannot be recognized by the cloud, it is marked as unknown. Bandwidth is only used when necessary and computation is performed by multiple devices simultaneously.

We showed that our architecture can recognize 93.56 percent of faces locally and 98.93 percent in overall pipeline. Introducing an edge server locally can reduce the bandwidth as much as 93.56 percent. We also showed Raspberry Pi run containerized applications with minor overhead and performance drawbacks. Computation time is comparable to AWS Rekognition. Introducing a layered architecture increased the recognition rate by 5.37 percent without adding significant network and cloud costs. The source code of this work is also available on Github under Mozilla Public License 2.0. Docker deployments are also open sourced. Docker images are publicly available on Docker Hub.

ÖZET

YAVAŞ VE ÖLÇÜLEN BAĞLANTILAR İÇİN AKILLI GÜVENLİK KAMERA SİSTEMİ

Geleneksel güvenlik sistemleri bir kayıt cihazına sürekli görüntü kaydı yaparlar. Lakin bu yöntem muazzam derecede veri üretmektedir. Yeni cihazlara Internet üzerinden baplanıp görüntü kaydı alınabilmektedir. Bant genişliği ve Bulut ücretlerini düşünmemiz gerekir. Karşıya yükleme hızlari genellikle Internet Servis Sağlayıcılar tarafından kısıtlanmaktadır ve Bulut'ta dosya saklamak pahalıdır. Araştırmalara göre kapalı alanlarda çekilen güvenlik görüntülerinin büyük çoğunluğu hareketsiz görüntü ya da herhangi bir bilgi içermemektedir. Hareket olduğu zaman da bu yüksek ihtimalle hırsız değil hanehalkı tarafından olmaktadır.

Bu yüksek lisans tezinde hiyerarşik kümelenmiş ucuz, emre hazır Raspberry Pi devre kartlarını kullanarak birden fazla kamerayı akıllı bir şekilde gözlemleyebilen bir sistem geliştirdik. Sistem hanehalkını tanıyor ve tanıdığı kişiler kamera karşısında olduğu zaman kayıt almıyor. Eğer kişi sistem tarafından tanınamıyorsa Bulut'a yükleniyor ve orada da tanıma algoritması çalışıyor. Eğer Bulut sistemi de kişiyi tanıyamazsa kişi tanınmadı olarak işaretleniyor. Bant genişliği sadece kişi tanınmadığı zaman kullanılıyor ve bilgisayım birden fazla devre kartı üzerinde eşzamanlı olarak yapılıyor.

Yaptığımız çalışmada yerelde yüzde $93,56$ ve Bulut ile entegre çalıştığında yüzde $98,93$ oranında yüz tanıma başarısına eriştik. Yerelde uç sunucu çalıştırarak bant genişliğini yüzde $93,56$'e kadar düşürdük. Raspberry Pi konteynıra yüklenmiş uygulamaları başarılı bir şekilde minimum ek yük ve performans farkı ile çalıştırdı. Raspberry Pi'daki bilgisayım süresi AWS Rekognition ile karşılaştırabilie seviyede olduğunu gördük. Önerdiğimiz katmanlı mimari sayesinde bant genişliğini minimum kullanıp limitli bulut masrafları ile yüz tanıma oranını yüzde $5,37$ artırdık. Projenin kaynak kodlarını açık yaparak Mozilla Public License 2.0 altında yayımladık. Docker imajlarını halka açık Docker Hub'a koyduk.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS/ABBREVIATIONS

AI	Artificial intelligence
API	Application programming interface
AWS	Amazon Web Services
BLE	Bluetooth low energy
CAD	Computer aided design
CL	Cloud Layer
CMS	Central monitoring system
CNN	Convolutional neural network
CPU	Central processing unit
DC	Data center
DML	Deep metric learning
FaaS	Function as a service
GPIO	General purpose input output
GPU	Graphical processing unit
IaaS	Infrastructure as a service
IoT	Internet of things
ISP	Internet service provider
KNN	k'th nearest neighbor
LBP	Local binary pattern
ML	Machine learning
OS	Operating system
OTG	On the go
PaaS	Platform as a service
PC	Personal computer
PL	Processing Layer
PoE	Power over ethernet
PPL	Preprocessing Layer
RAM	Random access memory

RPI	Raspberry Pi
RTCP	Real time control protocol
RTP	Real time transport protocol
RTSP	Real time streaming protocol
S3	Simple storage service
SaaS	Software as a service
SBC	Single board computer
SNS	Simple notification service
SVM	Support vector machine
UPD	Unrecognized people dataset
WAN	Wide area network
WiFi	Wireless fidelity

1. INTRODUCTION

Wealth has become more distributed among people with ascent of middle class. As people became wealthy, the urge to protect their assets has emerged. Private surveillance is one of the ways to monitor and protect people's valuables. However, private surveillance used to require expensive setups and expertise to operate.

Traditional approach of surveillance is to record video footage to a local drive or Network Attached Storage (NAS) continuously [1]. However, this generates enormous amounts of data. Seagate reports a 4TB drive can record 48 days of 30fps / 1280 × 1024 MPEG-4 encoded video footage [2]. Majority of this data contains almost no information, especially for household footage. This approach not only puts extensive load on hard drive which reduces its lifespan, but also creates a problem with its location. Any intruder can remove the hard drive from the house, leaving the system vulnerable. Placing the recording device and storage in the same place creates a single point of failure.

Internet enabled video surveillance systems are becoming more popular as prices are getting more affordable. People want to monitor their home and loved ones in an easy and secure manner. Companies see this opportunity and home automation and surveillance systems have already matured in the consumer market. Google's acquisition of Nest Labs for $3.2 billion in 2014 only proves the direction of this technology [3]. Additionally, cheap IP-cameras provide an easy solution for those who want to monitor their private property.

Instead of recording footage locally, another solution is to record video footage to cloud directly as proposed by [4]. This approach eliminates limitations of local storage systems such as single point of failure or limitations for data storage. However this approach generates high bandwidth load and potential high costs on cloud. Further, it is not possible to utilize such systems in countries where upload is capped and speed is limited. This approach can be acceptable on mission critical systems, but may hit bandwidth limitations and infeasible cloud costs for end users.

Edge computing or Fog computing [5] is coined to overcome latency and bandwidth problems with the cloud. Edge computing paradigm proposes a server at the edge of the device's network to share the load with the client and cloud. Thus, cloud is used only when necessary.

Nest Cam IQ [6] provides built-in functions which is close to an edge server. Camera comes with a 6 core Qualcomm processor to perform image recognition tasks. It can detect an unfamiliar face and create alerts. It comes with 3 hour snapshot history. If customers want more history, they need to enable cloud integration starting from 5$ a month [7]. However this method puts capturing and processing to the same place, creating a single point of failure.

In this work we aim to create a scalable system with any Web-Cam or IP-Camera powered by a Raspberry Pi (RPI) cluster as edge server. RPI is a cheap, low power, credit card size computer with WiFi module. We employed face recognition algorithm on the edge and compared the performance with state-of-art computer vision Application Programming Interface (API) "Rekognition" from AWS.

We focused on home surveillance to verify our design. However, our model is applicable to variety of applications to reduce upstream bandwidth. One famous example is speed cameras. Speed cameras for traffic enforcement are deployed with mobile connections with bandwidth limitations. It is infeasible to upload continuous video stream to server for processing. Thus information should be processed on the camera or on the edge. Speed cameras perform license plate recognition and only uploads text including license plate, violation and other meta data. If license plate cannot be recognized on the camera, server can decide to download the frame from the camera and perform more complicated algorithms to extract license plate from the frame.

1.1. CONTRIBUTIONS

We have made several contributions to the literature. Using cluster of RPIs in a hierarchical architecture with Cloud integration is novel to our knowledge. We confirmed that Raspberry

Pi has first class support from Docker and can run Dockerized applications without significant performance drawbacks, even Raspberry Pi Zero [8]. We found that our system not only reduces bandwidth need for cloud connection but also increases the overall accuracy of the recognition. Finally, we realized that false positives can be quite high with recommended settings of [9]. It should be a more strict value for systems who may use the algorithm only locally. We discuss improvements and changes we did in our system compared to other published works in the rest of the section.

Architecture proposed in [10] influenced our design the most. Their Edge Layer which consisted a RPI only performed face detection. When face is detected, it is sent to Server Layer. They used Dell T630 tower server as Server Layer. They have used Eigenfaces [11] for face recognition in their Server Layer. Their design does not include Cloud. We proposed to add one more layer, Cloud Layer. We also replaced Server Layer with Processing Layer. Our Preprocessing Layer and Processing Layer can contain multiple RPIs collaborating with each other. Lastly we used Deep Metric Learning [12] for face recognition as opposed to a shallow method. Eigenfaces can have a varying accuracy when orientation (85 percent) and size variations (64 percent) occur [13]. Those variations present themselves frequently in real life surveillance footage. On the other hand, algorithm we used has a accuracy of 99.38 percent against Labeled Faces in the Wild dataset [14] [15].

Popic compared local and cloud approach to face recognition in his master thesis [16]. We have used the same hardware (RPI) and library [9] in our local processing and same cloud provider (AWS Rekognition). His contributions are mainly focused on advantages and disadvantages of local and cloud processing. We have combined those two approaches in a single work with a hierarchical architecture. We found that face recognition accuracy rate of our design is superior than using either model alone. We also reduced the amount of face frames that are transmitted to the cloud up to 93.56 percent excluding false positive face detections.

We utilised cluster of Raspberry Pi's as edge server and included cloud API AWS Rekognition in our architecture as opposed to using same algorithm on the cloud and locally in [17]. Recognition algorithm in the edge and cloud is different so their success rate to

recognize images. We reduced bandwidth by locally processing images on the edge and improved accuracy by combining different algorithms on the edge and the cloud.

System proposed in [18] values full video footage at the expense of storing and transmitting idle video frames. On contrast, our architecture is designed to focus on face frames rather than full video footage. We send frames to the cloud only if system detects a face and it is not recognized locally by our edge servers. This reduces bandwidth usage and cloud costs significantly.

Wazwaz et al suggested a similar architecture to our proposed design [19]. However their system does not include Cloud and only performs recognition locally with LBP, a shallow method. Their contribution is mainly fine tuning parameters for face detection using Haar Cascades on RPI. Detected faces then sent to cluster of servers but it is not clear how the servers are collaborating. Hardware specifications of the servers are not specified and their average processing time is $0.43s$ with 100 faces in the training dataset. Our average processing time of $1.9s$ with four RPIs with 364 training dataset. Our design uses a load balancer with Round Robin algorithm to choose next server. We use deep method for face recognition and we perform face recognition locally and on the cloud with a success rate of 98.8 percent.

2. BACKGROUND

This work combines deep learning, embedded systems, highly available systems [20] and cloud computing. Deep learning is utilized for face recognition. Embedded systems come into play by using tiny computer boards. We utilize a number of boards to collaborate. System is designed to be fault-tolerant. Any system failure should not result in system shut down. This is achieved by high availability. Finally we utilize public cloud APIs to utilize their infinite processing power and machine learning algorithms.

In this chapter we will discuss current trends and solutions to home surveillance problems. We discuss existing solutions in Section 2.1. A brief introduction to Edge Computing will be in Section 2.3. We will then discuss current trends, abilities and limitations of Single Board Computers (SBC) in Section 2.4. Current face detection and recognition algorithms will be presented in Section 2.6. Finally we will introduce cloud computing and services used in this thesis such as computer vision, storage service, database and notification service in Section 2.7.

2.1. HOME SURVEILLANCE

Continuous recording of video requires significant amount of storage. In order to cope with local storage problems [4] proposed a cloud based solution. They suggest an architecture where all cameras are attached to a local ”Surveillance Client” and footage is uploaded continuously to the cloud. AWS S3 [21] is utilized as their storage backend. Proposed architecture includes a cloud server to perform processing on the video footage, such as vision algorithms to detect safety events and trigger alarms. In other to protect user privacy, all communication is carried out with SSL protocol.

Jayakumar et al proposed a smart surveillance system where Raspberry Pi 2 Model B is used as camera driver and pre-processing device [22]. Passive Infrared Sensor (PIR) is used to detect motion. After a motion is detected, Raspberry Pi saves camera footage locally and uploads it to cloud simultaneously. A server is used to perform face recognition tasks

because Raspberry Pi is found to have difficulties perform this task in real time. Face recognition is performed with predefined face database and if not recognized labeled as ”unknown” with a rectangular frame. Face is extracted from the footage and it’s location is send back to Raspberry Pi with serial connection. Raspberry Pi then controls the camera position with servo motors to track intruder’s face. Authors [22] did not publish performance values but claimed that system was operational in real time.

2.2. IP CAMERAS

IP cameras are cameras that are accessible over the network in which they are connected. They typically have Ethernet port or WiFi module. They are configurable using their administrator program. IP cameras are powered by their power cable or Power over Ethernet (PoE). In this setup, power is transmitted over Ethernet cable, reducing the need of wiring another cable to the camera. They are widely used in surveillance [23].

Video footage from IP cameras is usually transmitted using Real Time Transport Protocol (RTP) [24]. This protocol is combination of streaming and control of the stream. RTP uses TCP or UDP depending on the delivery of the file (unicast or multicast). It has a control protocol called Real Time Control Protocol (RTCP). RTCP has important control commands such as `DESCRIBE`, `SETUP` and `PLAY`. Payload is transferred using RTP and controlling is performed via RTCP. They are designed to be independent from underlying transport and network layers.

Vendors provide management and Central Monitoring System (CMS) software with their product, usually a windows application. This program can configure camera and setup alarms depending on the model and the features of the camera. Figure 2.1 shows network configuration of the camera via CMS software. It is also possible to watch cameras from this software as shown in Figure 2.2.

CMS software is not always maintained and up-to-date. Some software may require old browsers such as Internet Explorer 11. In order to manage the camera it is recommended to install this software in a Virtual Machine. Software may be infected with malware or viruses

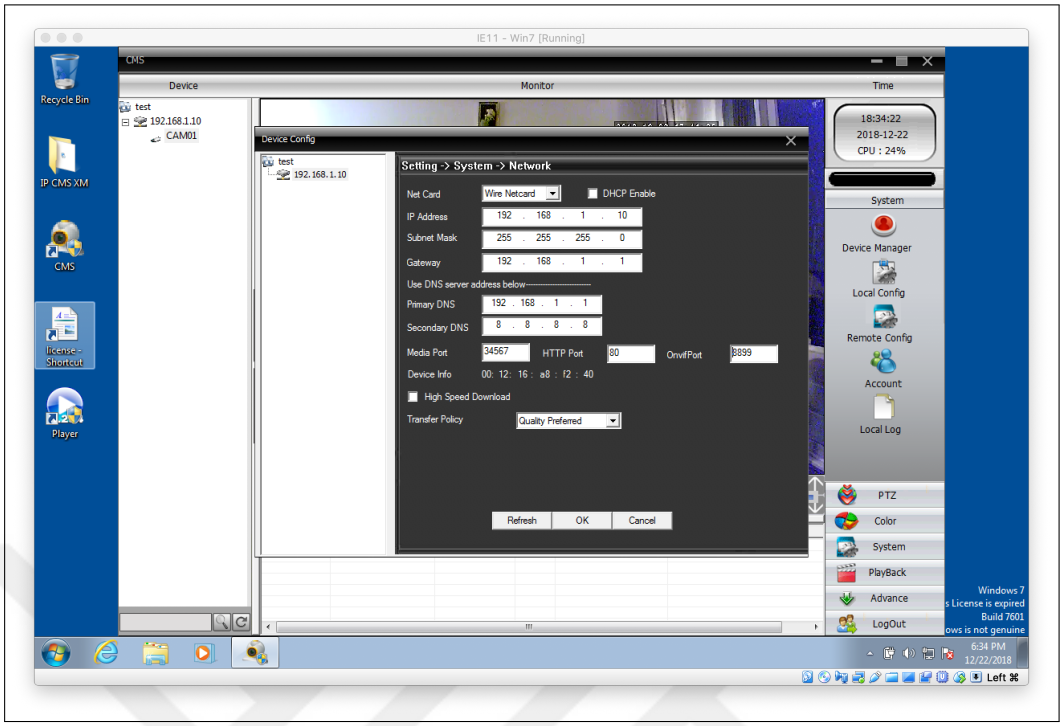

Figure 2.1. Configuration of IP camera via their CMS software

and it may not patched for latest security updates. Microsoft provides free virtual machine images for old browsers from Internet Explorer 8 to MS Edge [25].

Security is an important aspect with IP cameras. Since they are connected to Internet and they provide surveillance to valuable places it is crucial they access to the cameras are limited and authorized. Unfortunately this step is often forgotten and cameras are left with default passwords.

Insecam [26] is a project from an anonymous hacker that exposes thousands of unsecured webcam footage online. As of December 2018 website hosts 19,069 cameras around the world, majority is from the United States. At its peak, website is claimed to host 73,000 cameras [27]. This is a sharp example of security vulnerabilities in everyday devices. Manufacturers urge to change camera passwords regularly but this advice is not practical.

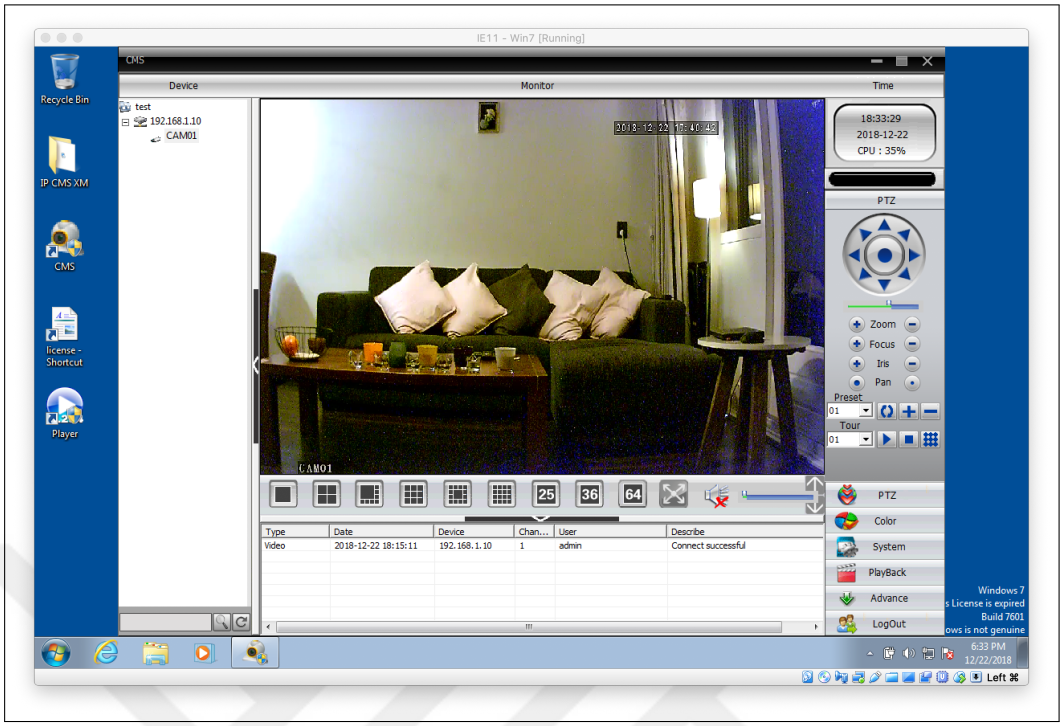

Figure 2.2. CMS software provided from IP camera vendors

2.3. EDGE COMPUTING

Edge computing is a paradigm to divide computational load onto nodes which reside at the edge of the network. It is widely used in clustered embedded and Internet of Things (IoT) devices. These devices often lack the processing power or battery requirements to perform some tasks. A server is needed to outsource the computation. However if the server resides many hops away from the device latency will suffer. Real time applications cannot tolerate such delays. Advantage of this design is having a low network latency than pure cloud implementations.

Implementation of edge computing is fairly simple for immobile devices or restricted area usage. One or cluster of servers are placed in the same network of the devices and devices offload tasks to the edge. Examples [5][28] can include a WiFi network in an organization, wireless sensor networks, connected vehicles, etc. If application targets mobile phones then edge server needs to be on the base station or Internet Service Provider (ISP) backbone. This obviously requires wider permits on operator or ISP level.

Park et al designed a hierarchical architecture with edge node for home surveillance [10]. Cameras, sensors and actuators belong to the device layer and connect to the nearest edge node. In edge layer, an edge server pre-processes the device data and feed it to the server layer when it detects an intrusion or abnormal event. Server layer controls edge nodes and can request device data at any time. Server can perform automated software updates and detailed analysis of the device data. Edge node can control devices independently thus reducing the latency on emergencies. They used Raspberry Pi 2 Model B as edge and Dell T630 tower server as server. When edge node detects an intruder, it extracts the face with Haar feature-based cascade classifiers and sends the face not the full video footage. This reduced the CPU utilization by 50 percent and network usage from 15 MB to 80 KB. Otherwise the sensor's process, image uploading into cloud, continuous distance/proximity monitoring using ultrasonic sensor and remote access of the camera live video stream monitoring and controlling of the camera and all are performed in the Raspberry Pi system itself.

Muslim et al [17] created an image recognition model with support from edge server. Image recognition procedure first detects a face and then applies Local binary pattern (LBP) algorithm to recognize and label faces. The same process is applied in both client and edge server. If edge server recognizes the faces on the image, it labels and sends the image back to the client. They utilized a smartphone as client and a powerful workstation as edge server. $13.3/1.6$ Mbps network is used between the client and edge server and server is located 7 hops from the client. Their experiments showed that processing in the edge server including the latency during the transfer is significantly faster than processing only locally. They did not address the selection of processing place, client or edge server. It is left as future work.

Alsmirat et al [18] designed a reliable IoT-based wireless video surveillance system that provides an optimal bandwidth distribution and allocation to minimize the overall surveillance video distortion. Cameras are viewed as things and connected to Mobile Edge Computing (MEC) servers. MEC servers reside at the edge of the network such as base station for mobile devices and local server for WiFi. Surveillance footage is sent to MEC server and saved locally until it is uploaded to the central cloud server. This enables continuous video delivery from the cameras even though network connection to the central

cloud is not available. They tested their system with NS-3 [29] simulator and results show cloud fully utilizes available bandwidth budget and provides high scalability.

Wazwaz et al created a face detection and recognition system which utilizes a RPI and a cluster of servers [19]. A camera is connected to RPI physically and RPI streams live video from the camera. Boosted Cascade of Simple Features Algorithm (Haar Feature-based Cascade Classifiers [30]) is used to detect faces on RPI. They run extended tests to fine tune algorithm variables to achieve maximum face detection rate. Detected faces are sent to cluster of servers for face recognition phase. They used one to three servers and compared the duration per each frame. Local Binary Pattern (LBP) algorithm is used for face recognition. Before system initialization each known face is processed and LBP values are generated. They published only duration of the recognition process but not the accuracy rate of their recognition.

2.4. SINGLE BOARD COMPUTERS

Single Board Computer (SBC) is a complete computer that comes with micro-controller unit as well as memory, GPIOs and other convenience ports on a single printed circuit board. They are used in variety of domains. Most notably in educational systems, proof-of-concept systems and as embedded computer controllers.They are increasingly getting popular because of their simplicity, cost and power requirements.

2.4.1. Raspberry Pi

Raspberry Pi is developed by Raspberry foundation in United Kingdom [31]. It is open source and designed to promote computer science education in schools and developing countries.

Raspberry Pi currently dominates the market for SBCs. As of March 2018 Raspberry Foundation reported 19 million unit sales [32]. This makes Raspberry Pi the third best-selling General Purpose Computer ever after Apple Macintosh and Microsoft Windows Personal Computers (PCs). Currently there are three models of Raspberry Pi on the market.

Device details are given in Table 2.1 All devices contains HAT-compatible 40-pin header, Micro USB power, Mini HDMI port and Micro USB on-the-go (OTG) ports.

Table 2.1. Raspberry Pi models on the market

Model Name	**Release Date**	**Component**	**Specifications**
Raspberry Pi 3 Model B+	March 2018	CPU	4x Cortex-A53 1.4 GHz
		GPU	Broadcom VideoCore IV
		WiFi	2.4GHz and 5GHz 802.11b/g/n/ac
		RAM	1 GB (Shared with GPU)
		Ethernet	Gigabit
		Bluetooth	4.2 and BLE
		Connectivity	4x USB 2.0 port
Raspberry Pi Zero Wireless	February 2017	CPU	ARM1176JZF-S 1GHz
		GPU	Broadcom VideoCore IV
		WiFi	2.4GHz 802.11b/g/n
		RAM	512 MB (Shared with GPU)
		Bluetooth	4.1 and BLE
		Connectivity	1x USB On-The-Go
Raspberry Pi Zero	May 2016	CPU	ARM1176JZF-S 1GHz
		GPU	Broadcom VideoCore IV
		RAM	512 MB (Shared with GPU)
		Connectivity	1x USB On-The-Go

In order to demonstrate power of Raspberry Pi and show it is not just a toy device Tso et al created a Data Center (DC) simulation with 56 Model B Raspberry Pi's [33]. They set up a canonical multi-root tree topology network among devices to reflect a typical DC network. Devices are divided into racks and each device in the rack is connected to same Top of Rack switch. Each device support virtualization up to 3 guest hosts by using Linux Containers. They also built a management API to head node (*pimaster*) to control custom IP policies, DHCP and DNS rules. Overall they replicated cloud simulation software Cloud DC but

in real life. This system is capable of running distributed processes on a real network and hardware platform.

2.4.2. Raspbian Operating System

Raspbian is the official operating system (OS) of Raspberry Pi. It is an open source OS based on Debian distribution of Linux. Benefit of using Raspbian is that it is highly optimized for Raspberry devices. *apt* package manager is available for Raspbian to install arm compiled packages to Raspberry Pi. Raspbian comes with a configuration tool called *rasp-config* to manage Raspberry Pi. It can set variety of settings from locale to WiFi configuration.

Raspbian operating system can be ordered pre installed on SD card when buying Raspberry Pi in a bundle. Individual users should choose a program to create bootable disks. Etcher [34] is the official recommendation to create a bootable Raspberry disk.

2.5. VIRTUALIZATION AND CONTAINERIZATION

In the early days of DCs there were only bare-metal servers, meaning there is only one tenant on the physical server. This architecture has several limitations. First, it was expensive to book the whole machine when load is low. Second, it was time consuming and partly manual process to scale up when load is high. DC should have sufficient amount of machines to meet demand of all its customers which increases the overall price. There are some advantages too. Security is utmost in this approach leaving only the network compromised since there is only one tenant utilising the server. Vulnerabilities such as Spectre and Meltdown [35] in 2018 showed that multi tenant servers can be a security issue for sensitive applications.

Virtualization solved this problem by separating customers on OS level. Hypervisor is a software, firmware or hardware that creates and runs virtual machines. In this architecture OS assumes it talks to the real hardware but in reality it talks to a virtual machine. There are two types of hypervisors, Type 1 and Type 2. Type 1 or native hypervisors runs directly on host's hardware shown on Figure 2.3(a). Type 2 or hosted hypervisors runs on top of the host OS shown on Figure 2.3(b).

Virtualization became so efficient that it dominated Infrastructure as a Service (IaaS) offerings. This way customers can share the same hardware with minimal security risk which reduces costs significantly. In case of peak load new virtual machines can be added from a blueprint to the existing cluster within minutes. This provides a easy and automated way to scale up and down to meet the demand and save money when necessary.

Disadvantage of hypervisor based virtualization is its boot up time and additional layers to computation. Creating a new VM and bootstrapping it takes some time depending on the complexity of the blueprint. This is a handicap to meet bursts of load and requires load estimation which is highly complex. Additional layers adds a computational burden and lowers the performance of the machine. In Type 2 hypervisors even though hardware is shared, kernel and OS is not. This duplication results in poorer performance.

Containerization also known as operating-system level virtualization, is a method to separate applications on operating system layer in isolated user space instances called containers. A containerized architecture is shown on Figure 2.3(c). Figure a represents Type 1 Hypervisor VM b represents Type 2 Hypervisor VM and c shows Containerization. In this approach from application's perspective they are contained in an operating system. Normally an application can access all OS resources including peripherals. However an application inside a container only sees what is available in this container. This method has almost zero cost bootstrap and execution time and lightweight containers. This is because operating system in container is usually stripped version linux, like Alpine distribution which is only 5 MB. Any dependencies are added so container size is equal to application size and its direct dependencies. Currently most widely used containerization software is Docker [36] and it is invested heavily by the industry.

2.6. FACE DETECTION AND RECOGNITION

OpenCV [37] and dlib [38] is used in this work to perform face detection and face recognition tasks. OpenCV is a C++ library aimed at real-time computer vision problems. It contains highly optimized algorithms to make computer see the image. It can utilize both CPU and GPU. Although library is written in C++ it has Python, Java and MATLAB

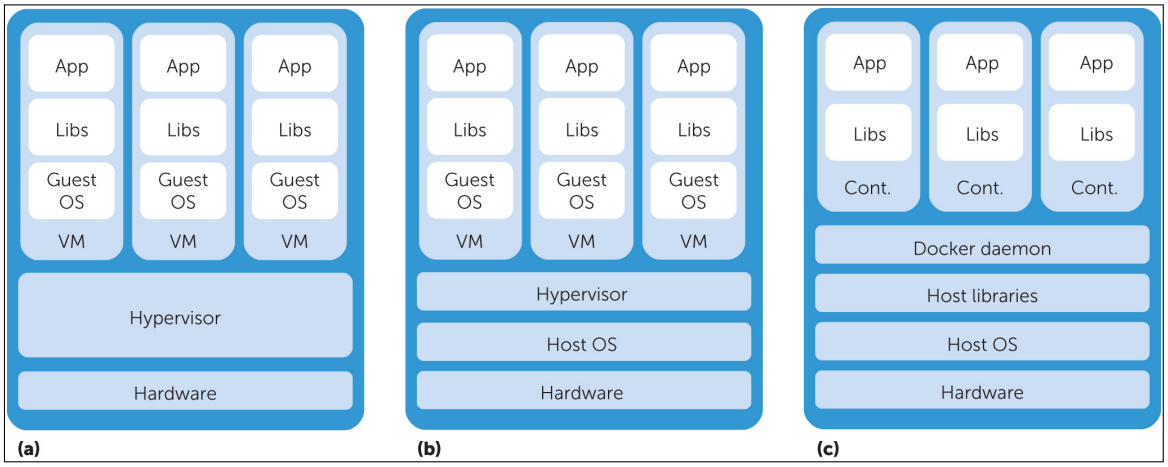

Figure 2.3. Virtualization types [39]

bindings.

Dlib-ml is a cross platform open source software library written in the C++ programming language published in 2009. It is a complete toolset with variety of components. As of 2018 dlib has libraries to handle the following: networking, threads, graphical user interfaces, data structures, linear algebra, machine learning, image processing, data mining, XML and text parsing, numerical optimization and Bayesian networks. Dlib is licensed under permissive Boost Software License which allows both open source and corporate use. Library is being used heavily both in academia and industry. Robotics, mobile devices, embedded systems, healthcare systems, artificial intelligence and machine learning systems are subset of domains where dlib is used successfully.

Dlib and OpenCV can be used together. OpenCV offers tools to read and process images. Dlib can use this information to create models that can be used to train machine learning algorithms.

Popic, in his master thesis [16], compared cloud based facial recognition system (AWS Rekognition) and facial recognition with IoT (Raspberry Pi). Advantages and disadvantages of both systems are documented. face_recognition library [9] is used on Raspberry Pi. He concluded that local processing on Raspberry Pi is immensely faster than processing on the

Cloud, due to network overhead. Accuracy rate of both systems are comparable, although AWS Rekognition is slightly more accurate. On contrast to local approach, cloud approach does not require maintenance and adding new faces to database is relatively easy. In the end, both systems are worthy of the task and should be used whenever necessary.

2.6.1. Face Detection

Haar-like feature based cascade classifiers are the industry standard for face detection. Algorithm is available in most vision libraries as well as in OpenCV. In this work we have used `haarcascade_frontalface_default` settings to detect a face. There are many others for different use cases.

Haar feature-based cascade classifier algorithm [30] is developed for rapid and accurate face detection using Haar-like features. Algorithm based on the observation that face has certain light and dark patterns. For instance eyes are darker than cheeks. Using these features rather than light intensity of a pixel creates regions where adjacent intensity values of pixels are being considered. Two or three adjacent pixel groups with certain intensity values form a Haar-like feature shown on Figure 2.4.

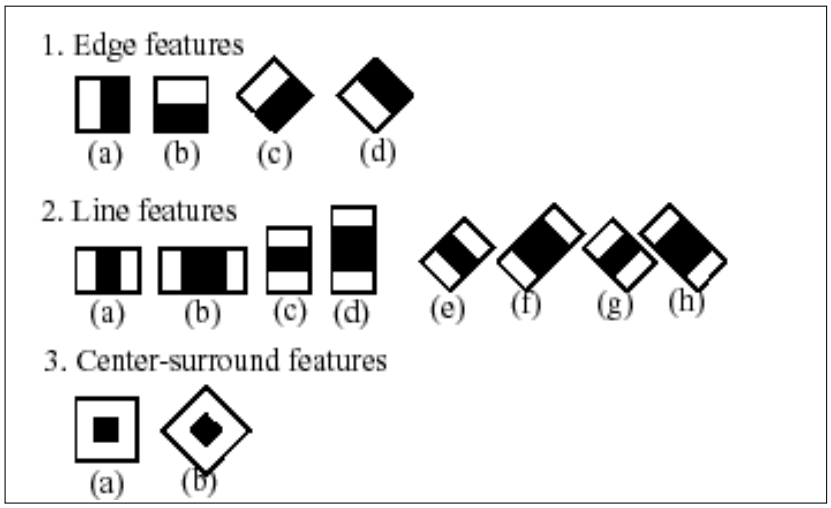

Figure 2.4. Haar-like features [40]

Haar-cascade classifier algorithm [30][40] is used for face detection which has an accuracy rate of 95 percent. This approach starts with primitive classifiers which classify the potential

face candidates into groups based on Haar features. These primitive classifiers have no computational complexity but operate on large amount of data. Using progressively more efficient classifiers based on more additional features, the algorithm eliminates some of the face candidates. The number of potential face candidates decreases at each successive stages but the complexity of the calculation increases at the same rate. As a result, the complexity of computation remains approximately same at each stage. The final stage generates the detected face with high confidence.

2.6.2. Face Recognition

Face recognition is very popular these days both in academia and industry. In essence face recognition allows computers to recognize a face in picture or video stream. This is a complex problem best suited to solve by machine learning.

This is also a controversial topic [41]. State of art face recognition on wrong hands may easily track and abuse people. It can oppressive government or marketing tool to suggest products you likely to buy. Microsoft's president recently called for action regulating facial recognition software to create a healthy market and good competition [42]. Unregulated facial recognition poses a privacy problem which is feared to lead to severe consequences under oppressive governments.

Face recognition has several domains which can have different applications.

i Face Verification: Verify if given face is same as a source face
ii Recognition: Recognize this face from a database of faces
iii Clustering: Group similar faces together

In this work we focus on face recognition. For this problem we need to build a database of known faces and use this to recognize a new image using the database.

Face recognition attracted significant attention from the past. Many methods proposed for this problem. For the sake of categorization we will define them in two areas: shallow and

deep methods. Shallow methods do not perform deep learning and deep methods uses deep learning for recognition.

Shallow methods tries to imitate what humans do to recognize a face, try to extract features and based on that recognize a face. Shallow methods extract the representation of the image. They use local image descriptors like SIFT [43], LBP and HOG [44]. After the extraction, descriptors are combined into a generic face descriptor with a pooling algorithm such as Fisher Vector. There are many methods to extract features and create descriptors which is not in the scope of this work. However shallow methods suffer when pose, light change or person wears a make-up for instance.

Most distinguished attribute of deep methods are they use Convolutional Neural Networks (CNN) for feature extraction. Weight function for the CNN is trained by supervised learning where labeled face are given to the algorithm. In the work FaceNet [45] from Google scientists trained the deep convolutional network with 100-200 million thumbnails of 8 million different people to generate a model. They used triplet loss function and generated a 128 dimensional vector for each face. Triplet loss function works as the following. 3 faces are provided to the algorithm, 2 faces of one identity and another face of a different identity. Since result of the CNN is a $128 - d$ vector space it is in fact an Euclidean space. Algorithm minimizes the Euclidean distance of the two faces of the same person and maximizes the distance from the third face. After weights are trained CNN model is used to generate a 128-D vector for unseen face. Comparing the result of the vectors it is possible to determine if the face is recognized. k-th nearest neighbor is used in Facenet to perform the final classification.

Research shows that deep methods are superior than shallow methods. Evidently, attributes humans use to recognize a face are not effective for computers to recognize a face. Thus it is important that computers generate a model that can successfully find attributes to distinguish faces in sample set. Once generated models can be used to recognize even faces which it is not trained to. Since each face is mapped to a $128 - d$ space with this model, an unseen face will also be mapped to a point. Using this information we determine faces maps to that particular point are faces of the new person we train.

We have used the algorithm described in [15] with Python wrappers [9] [46]. Library uses k-th nearest neighbor (KNN) algorithm to return closest matches. However, KNN is not suitable for creating a confidence level of the recognition. Thus library only returns binary results for found matches. Attempts to use Support Vector Machine (SVM) instead of KNN in the library have been made but not completed yet.

2.7. CLOUD COMPUTING

Cloud computing received a lot attention from the industry over the last decade. Amazon created its subsidiary Amazon Web Services and released its Infrastructure as a Service (IaaS) Elastic Compute Cloud (EC2) in 2006 [47]. In 2008 Google released their Google App Engine in beta [48]. Microsoft announced Microsoft Azure in 2008 and released it in 2010 [49]. Forbes estimated that 83 percent of all enterprise workloads will be on the cloud by 2020 [50].

Cloud is an appealing choice for both large enterprises and small scale startups. Operating a mission critical system on premise is often costly and requires dedicated specialists to monitor the health of the overall system. Outsourcing this burden leaves companies where they should focus, their product. Therefore cloud adoption is increasing. Even skeptics such as defence or government agencies is now considering and announcing their cloud adoption [51]. Pentagon's $10 billion JEDI cloud contract only shows the direction of the sector [52].

Cloud offers different products to varying requirements of customers. They are categorized under three services.

- Software as a Service (SaaS)
- Platform as a Service (PaaS)
- Infrastructure as a Service (IaaS)

SaaS provides software and its services as a product. Customers pays to access the software and do not manage any of its dependencies. Uptime guarantees of the service solely is vendors responsibility. SaaS clients are usually thin web based apps but not necessarily.

SaaS became common delivery model for many software offerings including previously one-time licensed applications like office software (Microsoft Office), CAD software (AutoCAD), Music (Spotify) even password managers (1Password). This generates continuous revenue for the companies instead of perpetual license. According to Cisco's Global Cloud Index 59 percent of all cloud workflows will be SaaS by the end of 2018. 73 percent of the organizations say nearly all their apps will be SaaS by the end of 2020 [53]. SaaS is an effective and desirable choice of software delivery because companies and people choose to pay for continuous service rather than maintain it on their own. For organizations it is cheaper to rent than buy and maintain.

PaaS provides networks, storage, servers, operating system, middleware (Java Runtime, .NET runtime, etc), database and other services to customers. Mentioned systems are managed by PaaS provider and customers can focus on Application and its Data exclusively. PaaS typically allows automatic scale-ups when load is high. This means when application is under heavy load, meaning high number of requests coming, PaaS automatically adds more resources to the application to meet the demand. Examples of this service are Google App Engine [54], Heroku [55], Amazon Elastic Beanstalk [56]. All of them handles application deployment, capacity provisioning, load balancing, auto scaling and health monitoring. Advantages of PaaS are reduced complexity, automatic scaling, load balancing and maintenance. Disadvantages of PaaS can be listed as reduced control over infrastructure and lack of operational features. However this is expected since PaaS promise to manage them automatically.

IaaS is the most basic cloud service model according to the Internet Engineering Task Force (IETF). Vendors offer IT infrastructure only, meaning virtual machines and networking as a service to subscribers. Anything above that are managed by the customers. Almost any cloud vendor offers this service. AWS Elastic Compute Cloud (EC2), Google Compute Engine and Azure Virtual Machines are examples of major IaaS offerings.

Differences between as-a-service offerings and on premise are given in Figure 2.5. On premise is defined by a private cloud where cloud owner is responsible for everything.

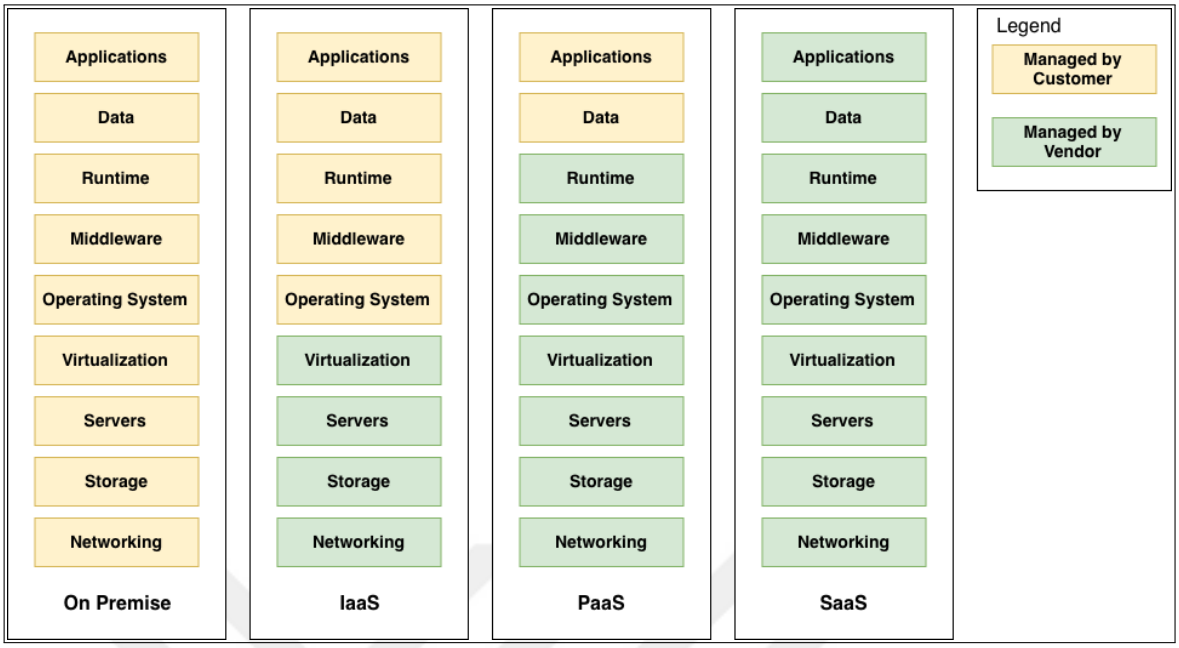

Figure 2.5. Cloud service offerings

2.7.1. Serverless Computing

Serverless computing [57], also known as Function as a Service (FaaS) is a cloud execution model where cloud provider acts as a server, dynamically allocating resources among virtual machines to handle incoming requests. Pricing is based on actual resources consumed by the application rather than a fixed pre-purchased amount of resources. Serverless is particularly good for applications where load varies unexpectedly. Since customer only pays per execution model, it does not cost anything for idle services. AWS Lambda is the first public offering of serverless computing. It is later followed by Google Cloud Functions and Azure Functions. Advantage of this approach is cost and high scalability. Disadvantages are slow bootstrap time, resource limits and limitations in debugging.

FaaS services usually calls a function in given source code when the service is triggered. It is called Handler. Handler is the execution entrypoint of the serverless function. Handler is usually provided with parameters where event details are provided. Event could be a file uploaded to a bucket or an event emitted from another service. This function usually has access to other services for logging and storing result of the execution. Finally function

needs to inform FaaS platform about the result of the execution and report any failures such as exceptions.

FaaS functions are written in the supported languages by the platform. Many mainstream languages such as Java, Python, Node.Js, Go and C# are already supported.

2.7.2. Object Storage Service

In addition to running virtual machines and offering serverless execution models, cloud vendors offer variety of services to their customers. Object Storage Services are designed to store data in a reliable way. Objects stored in such services are usually stored in multiple servers in different physical DCs. This guarantees the safety of the objects in case of a disaster.

Notable examples of this service are AWS Simple Storage Service (S3), Google Cloud Storage and Azure Blob Storage. This storages not only offers storing objects but also manage them. Buckets are resources with unique name that is available in all namespaces and regions. An application in Europe region providing objects to its bucket is also available for an application in US region. Buckets can contain both files and directories.

Companies dealing with big data favors object storages because managing of the data is outsourced. Uploading large amounts of data over the internet is another problem. Big data usually consists of petabytes (PB) of data.

Uploading 1 PB of data with a fairly fast 1 Gigabit Internet takes 2501 hours or 104 days. This is not feasible for most companies. Amazon introduced AWS Snowmobile [58] to handle the data transfer. Snowmobile is a high security 14 metre long container that is pulled by a semi-trailer truck. One snowmobile can transfer up to 100 PBs of data directly into the cloud storage. Snowmobile is equipped with security personnel, alarm systems, video surveillance and optional escort security.

2.7.3. Cloud Database

Cloud database is a database that runs on the cloud and managed by the cloud vendor. Database access is provided as a service. Database maintenance such as scalability and high availability is managed by the cloud vendor.

Both SQL and NoSQL databases are offered as database-as-a-service model by the major cloud vendors. NoSQL databases are natively more suited for cloud deployments since they are designed that way. However it is also possible to deploy a MySQL instance on the cloud.

Major SQL database offerings are Amazon Relational Database Service (RDS), Microsoft Azure SQL Database, Google Cloud SQL. NoSQL offerings are Amazon DynamoDB, Azure DocumentDB and Google Cloud Datastore.

2.7.4. Notification Service

Notification services are used to send messages to multiple recipient or services at once. A notification service can be used to send SMS, mobile push notification, notification to a certain application and notify connected services about an event. They can be used to warn masses about an emergency, marketing campaigns or verification methods. It is also used to connect different cloud services and applications.

Notable examples of Notification Services are Amazon Simple Notification Service (SNS), Google Cloud Messaging and Azure Notification Hubs. SNS can be used to connect an event in S3 service to be consumed by several services such as Lambda functions and Vision APIs at the same time.

2.7.5. Face Detection in the Cloud

Face recognition is still a difficult and complicated task for most of the organizations. Users need to create their dataset, find a good model online and train the model with their dataset. This requires specialized people and lot of time to fine tune the algorithm. From

retailers to government agencies face recognition can transform the way organizations work. Seeing this market opportunity big cloud companies released their artificial intelligence and machine learning APIs (see Table 2.2).

Table 2.2. Computer vision API offerings from major cloud providers

Cloud Provider	**Service**	**Face Recognition API**
Amazon Web Services	Amazon Rekognition	Yes
Google Cloud Platform	Google Cloud Vision	No
IBM	IBM Watson Visual Recognition	Yes
Microsoft	Microsoft Face API	Yes

Using cloud APIs to perform face recognition is relatively easy. Each provider differs from what they offer. Table 2.2 also indicates if face recognition software is offered with their Computer Vision API as of late 2018. In this work we used AWS and here we will give brief explanation on how to use AWS Rekognition.

Ease of using this technology brings ethical discussions to hand over such a powerful tool to wrong hands. United State Department of Defense's 10 billion USD cloud contract already attracted big players from the industry and outraged the employees and freedom advocators [51]. Microsoft's president Brad Smith urged policy makers to regulate facial recognition software to avoid a future like in George Orwell's classic book 1984 [59]. We are yet to see how this technology offering will transform the world.

In this work we used Amazon Web Services. This is a brief introduction how AWS Rekognition API works. AWS Rekognition accepts two pictures. One is a reference picture that one face is expected in it. Second image can contain many faces. API will return bounding boxes for each face and confidence ratio. Higher the ratio, more confident algorithm believes it belongs to the reference face. API expects the images in an S3 bucket and returns a JSON for its result.

3. THE PROPOSED DESIGN AND IMPLEMENTATION

In this chapter we will first discuss our proposed design, libraries that are used and how the deployments are made. In the next chapter we discuss our experimental setup to test our system.

We separated the setup into four layers: Device Layer, Preprocessing Layer (PPL), Processing Layer (PL) and Cloud Layer (CL). System overview is given in Figure 3.1. Device, Preprocessing and Processing Layers runs on premise. Hence, these layers are responsible for high availability and fault tolerance. Cloud Layer, as the name implies, runs on the cloud.

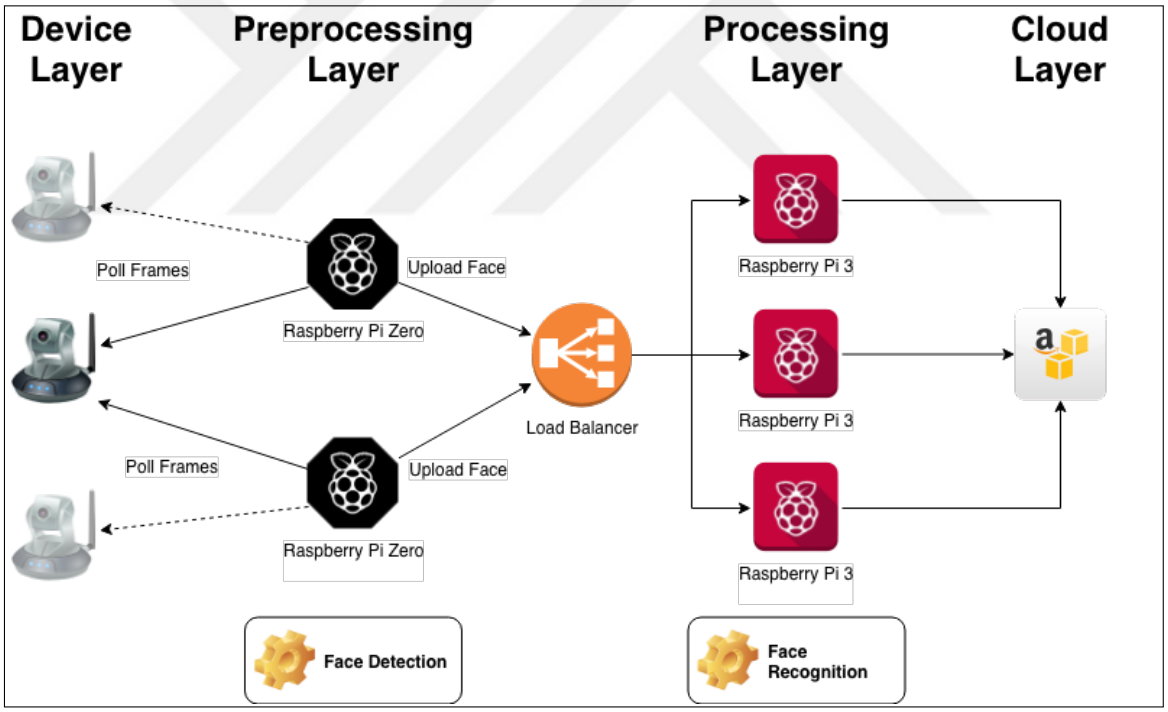

Figure 3.1. System overview

All deployments are done using Docker and all images are publicly available under *kaskavalci* repository in Docker Hub [60]. Docker is chosen for deployments because it removes the bootstrapping phase from the devices. Therefore a new device joining the system requires only Docker.

3.1. REQUIREMENTS

3.1.1. Dependencies

In this work we utilized Docker, OpenCV and Raspberry Pi. In this section we will discuss how to setup the environment and install dependencies.

We used Hypriot OS [61] initially for Docker support for Raspbian target. Later, Docker has been supported officially on Raspbian operating system as of August 2016. Installation is straightforward and same with any other OS.

```
curl -sSL https://get.docker.com | sh
```

Command `docker` should be available in the terminal after the installation is complete. Installing OpenCV is not a straightforward task. That is why we have created a docker image for this purpose. Docker image is based on work from rickryan [62]. We updated OpenCV version from 3.2 to 3.4.1 and recompiled. This is the base image we use in all other Docker images. `kaskavalci/opencv` docker image is available publicly under Docker Hub. This docker image has OpenCV v3.4.1 for Python 2.7 installed and compiled for Raspberry Pi. Installing this image can be done by

```
docker pull kaskavalci/opencv
```

This base image is used in `kaskavalci/face_recognition` image which also includes required libraries such as dlib and `face_recognition`.

We used a simple load balancer (`slb`) from Jacky Chiu [63]. This load balancer implements Round Robin algorithm and can be configured by a configuration file.

3.1.2. Generating Known Face Encodings of Dataset

We need to create a database of known faces in order to recognize an unknown face. First step is to create this database. Second step is to use this database to generate face encodings which will be used in runtime to recognize an unknown face. We used face_recognition Python module [9] for both training known faces and face recognition.

We need to have a directory structure similar to Figure 3.2. Inside dataset directory there may be multiple directories of different subjects. Each sub-directory contains photos of one person only and directory name is the label or name of the person. Photos inside the directory should only contain face of the subject alone. Since face detection algorithm sometimes gives false positives, it advised to only contain cropped faces and not complete pictures in the dataset folder.

One option to create this dataset is to use a camera to capture face of the individual and save cropped faces inside the given directory. Most obvious advantage of this approach is its simplicity and speed. An organization with many employees may spend 1 minute per employee to generate enough samples for face recognition. Biggest disadvantage is that since all samples are generated in one setting face recognition may suffer in the long term.

We used a simple Python script to crop faces from camera stream and save them under a given folder. Subjects stand in front of the camera with different poses and `face_recorder` saves each pose with one second delay. After our dataset is ready as shown in Figure 3.2, we can generate face encodings for each image. Each directory name is the label for the faces it contains.

Training part is done by getting face encodings for each face in the dataset by calling `face_recognition.face_encodings()`. Then, this information is combined by face label, which is the directory name and stored in an array. After processing each face and saving its label, this information is pickled and saved. This process only needs to be performed once. We followed the tutorial from pyimagesearch.com [64] to generate this model [65]. Later, pickled resource can be used directly from runtime. Algorithm is

```
dataset
├── aysenur
│   ├── aysenur_001.jpg
│   ├── aysenur_002.jpg
│   ├── aysenur_003.jpg
│   ├── aysenur_004.jpg
│   └── aysenur_005.jpg
└── halil
    ├── halil_001.jpg
    ├── halil_002.jpg
    ├── halil_003.jpg
    ├── halil_004.jpg
    └── halil_005.jpg
```

Figure 3.2. Directory structure for face encoding training

provided in Appendix A, Listing 1.

Pickled model is saved on the S3 bucket and downloaded each time Processing Layer is started. This approach simplifies the process of updating the model. Model can be propagated to the devices with a simple update or via background process that updates model some fixed intervals.

3.2. IMPLEMENTATION

3.2.1. Device Layer

Device Layer consist of hardware to capture video footage. It is possible to use any camera in this layer as long as it is compatible with upper layer, Preprocessing Layer. In order to achieve high availability and fault tolerance multiple cameras can be used.

IP cameras with PoE is a good choice because they can be connected to Wide Area Network (WAN) and draw their power from their Ethernet cable. Preprocessing Layer should be able to access devices in the layer. In our design all communication is done over network. Thus the device should be accessible over network or merge with Preprocessing Layer.

In our experiments we used one IP camera however it is possible to extend this indefinitely. Development is open source under Mozilla Public License 2.0 hosted on GitHub.

3.2.2. Preprocessing Layer

Preprocessing Layer is used for face detection and cropping only. This layer reduces the complete image size to faces only and feed it to Processing Layer. Performance of this layer determines the Frames Per Second (FPS) rate of the system. This is because only Preprocessing Layer interacts with Device Layer.

In our experiments we noticed camera stream is lagged when used with default OpenCV settings. In order to solve this problem we had to fine tune OpenCV settings. First, we set FPS and buffer size manually. Reducing buffer size and FPS speed improved the stream to be more real-time, avoiding lagged frames. However that was not enough. Code block to initialize a camera is given in Appendix A, Listing 2.

There are two ways in OpenCV to read frames from a real-time stream such as a webcam: `VideoCapture.read()` or `VideoCapture.retrieve()`. Former executes two functions the hood: `VideoCapture.grab()` and `VideoCapture.retrieve()`, latter retrieves the previously grabbed frame. If calling `VideoCapture.read()` from the code is lagged due to execution, then it will return a buffered frame rather than the fresh one which is lagged in time. For this reason it is advised to create a grabber thread to get latest frames at all times and retrieve frame when it will be.

In our implementation Grabber is a simple Python thread runs indefinitely and polls frames from the camera. Grab only caches the frame and retrieve reads the frame. In order to synchronize two threads on the same stream a mutex `__camLock` is used. Implementation is given in Appendix A, Listing 3.

Another problem we faced is motionless stream even though there are motion. Authors believe this is due to a bug in IP camera's firmware. In order to solve this issue we implemented a mechanism to re-establish RTSP connection when there are no motion for

predetermined amount of frames. Motion detection algorithm is presented in Appendix A, Listing 4. If motion is false for 500 frames `initcam` is called to reinitialize the connection.

Face detection is performed using Haar Cascade algorithm with default Frontal Face Alt (`haarcascade_frontalface_alt.xml`) configuration which is available under OpenCV repository [66].

In order to upload found faces to Processing Layer we have two options: write image as jpg to disk using `cv2.imwrite()` or serialize object on the fly. Former option adds extra overhead to write and read the image which is already in the memory. In order to transfer numpy array representation of face we used pickle [67] for object serialization. Filename is also provided as a header to the server. Filename is simply timestamp of the frame. Code sample is given in Appendix A, Listing 6.

3.2.3. Processing Layer

Processing Layer expects cropped faces from Preprocessing Layer. This layer has an endpoint open for incoming requests and accepts pickled payload which is encoded in JPG format. We have used `BaseHTTPServer` of Python and modified handler to include our Recognizer class.

Known face encodings are read as a pickled resource. An unknown face is assumed to be recognized if similarity of given face exceed 0.6 threshold with one of the faces in the catalog. Similarity is defined by the distance of two faces in 128 dimensional space. 0.6 is the industry standard but it is possible to fine tune it with experimentation. Code sample is given in Appendix A, Listing 7.

If a face is recognized, no action is taken by the Processing Layer. If face does not match with any encodings it is assumed to be unrecognized face. Face is uploaded to AWS S3 bucket. This way unrecognized face is saved in the cloud and it can be processed by Cloud Layer. Saving an unrecognized activity outside of the recording area is crucial since intruder can tamper with the video footage. Uploading suspicious activity to the cloud ensures that

evidence remains safe. Uploading only the suspicious activity reduces the bandwidth and cloud costs significantly.

3.2.4. Cloud Layer

We have chosen Amazon Web Services as our cloud provider. They provide free one year trial in many of their services and they have a huge community. Cloud services utilized in this work are: Simple Storage Service (S3), Simple Notification Service (SNS), Lambda (serverless function), DynamoDB (NoSQL database) and Rekognition (Computer Vision API). These services are explained in detail in Section 2.7. Architectural overview of the layer is given in Figure 3.3.

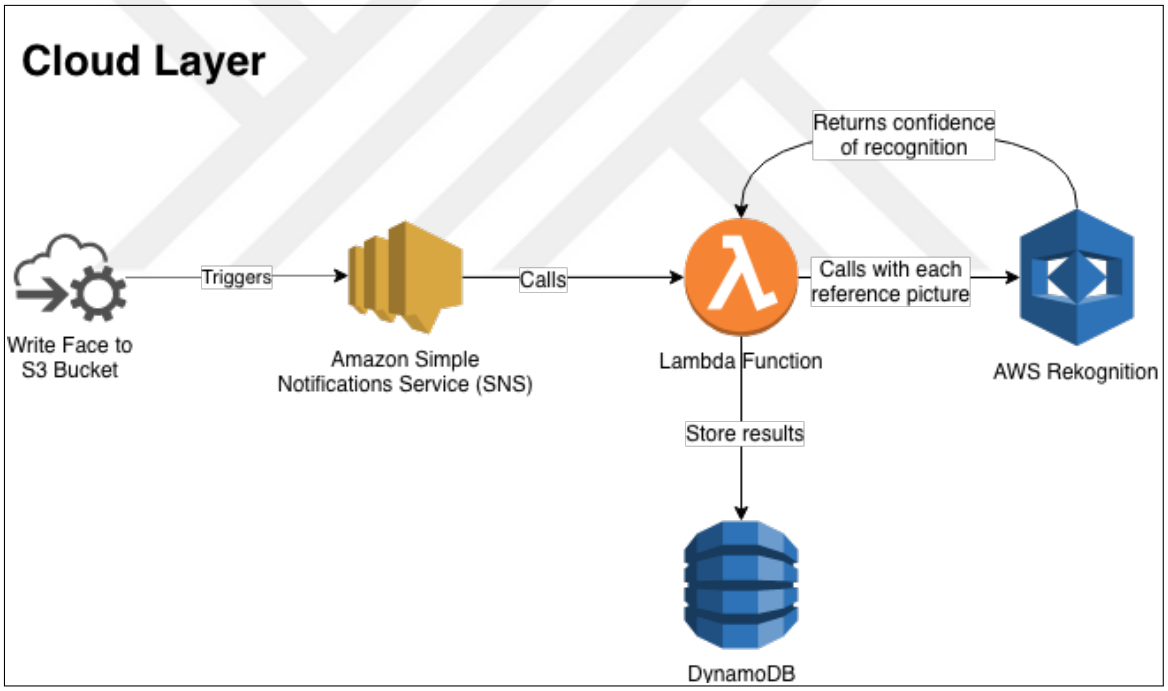

Figure 3.3. Cloud Layer

Cloud Layer starts with Processing Layer uploading an unrecognized face to a pre-defined S3 bucket. This ensures that unrecognized face will remain in the cloud even if surveillance system is damaged. Simple Notification Service listens on the changes in the S3 bucket. When a file is uploaded, notification rule is invoked.

In order to setup SNS rule to be triggered when an S3 object is created, we should create

a Topic. After Topic is created from SNS page on AWS Console, corresponding S3 bucket should be specified. This can be done by updating Topic Policy. Under Advanced View on Edit Topic Policy screen, `Condition` attribute should be updated as the following: `{"ArnLike":{"aws:SourceArn":"arn:aws:s3:::bucket-name"}}`. After policy is saved, Topic can access the bucket.

In order to define which events will trigger the Topic, an event policy should be set on S3. After choosing the bucket, on Properties screen there is Events. Under Events section we will add a notification with `All object create events` rule to an SNS Topic. Under SNS we specify the Topic we used earlier. Figure 3.4 shows the final settings on S3 Bucket Events.

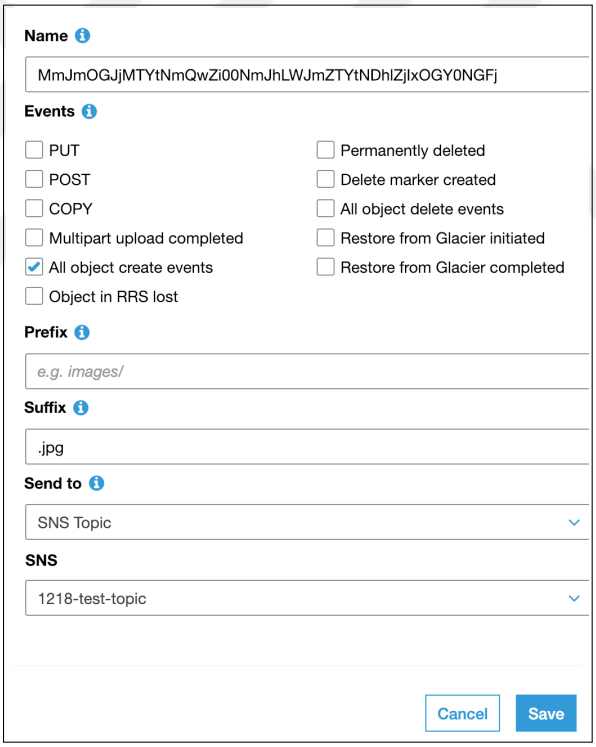

Figure 3.4. Adding SNS event to S3 bucket

Lambda function is called when SNS rule is triggered. We used Python 2.7 as Lambda language and 60 percent for similarity threshold. When function is invoked, the file that has been uploaded is provided as function parameters. Known faces in JPG form are listed in another S3 bucket. Rekognition API requires two images in S3 bucket, source and target. It returns the confidence on similarity between the face in the source image and faces in

target image. Lambda function reads through each file in the known faces bucket and calls Rekognition API for facial comparison. Code sample to call Rekognition is given in Appendix A, Listing 8.

Result of Rekognition is saved on DynamoDB database along with the filename and timestamp. It is possible to take further action from that point but authors concluded that it is beyond the scope of this work.

4. TEST RESULTS

In this chapter we discuss our experimental setup to test our pipeline. We implemented and deployed our design in Figure 3.1 in author's living room. System run for three months and captured thousands of images. We extracted faces of household from this dataset and tested face recognition success rate of our system.

In Device Layer, we used one IP camera that is shown on Figure 4.1. Since it has a server it can serve multi tenant. This means multiple agents in PPL can connect to same camera and one agent in PPL can connect to multiple cameras. Our design physically separates hardware from PPL.

Figure 4.1. IP camera

In Preprocessing Layer, we used two Raspberry Pi Zero shown on Figure 4.2. We used a ethernet port that is connected to micro USB port of RPI Zero which is then connected to a router. It is also possible to use RPI Zero W which comes with onboard 2.4G WiFi module. Since RPI Zero has very limited computer power, running a face detection platform is a heavy load for this device. We experienced a FPS rate of 0.5 to 1 while running face detection on streamed data from Device Layer.

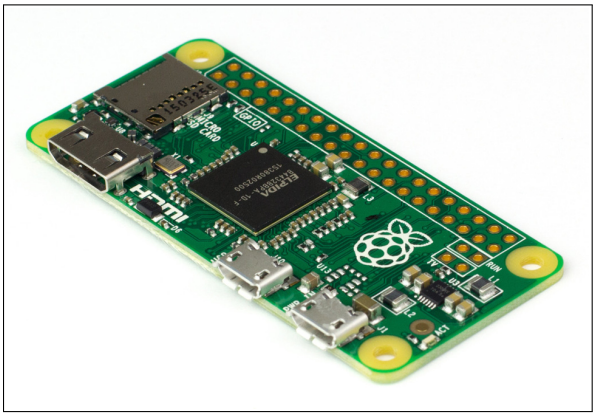

Figure 4.2. Raspberry Pi Zero

In Processing Layer, we used four Raspberry Pi Model 3B shown on Figure 4.3. One of them is assigned to perform Load Balancing along with face recognition task. Each RPI is set to listen incoming connections from port 80 and load balancer forwards requests to next available RPI. We used Python 3.7 and OpenCV 3.4.1 both in PPL and PL.

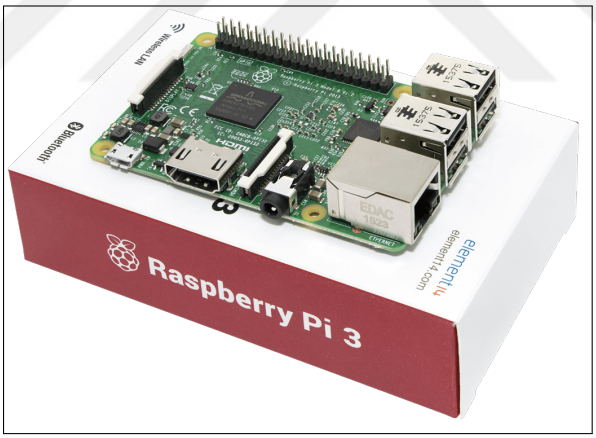

Figure 4.3. Raspberry Pi 3 Model B

In Cloud Layer, we chose Amazon Web Services. We utilized variety of services from AWS including: Simple Storage Service (S3), Simple Notification Service (SNS), Serverless functions (Lambda), Cloud Formation, NoSQL database (DynamoDB) and face recognition API (Rekognition).

A data set is created to verify and compare the results of our architecture with AWS Rekognition. We saved all the faces that Preprocessing Layer captured during our test phase.

Including false positives it resulted in 11587 images with faces. By using parameters from [19] we reduced the dataset to 4932 images. This resulted in elimination of false positives more than 50 percent.

Google Photos are used to improve the quality of our dataset. Google Photos is a photo storage service where users can upload their photos free of charge. Application offers some services such as face recognition (see Figure 4.4) which we used. After Photos service successfully tagged faces we exported people from the application.

Dataset contains 1423 face samples of household member Aysenur and 1291 of Halil, a total of 2714 images. Some face samples contain multiple faces in one frame if they are close to each other. In this case, these files are duplicated in both dataset. We identified 14 photos that are present in both individuals. Remaining 2218 photos include other people in the house (guests) and also false positives.

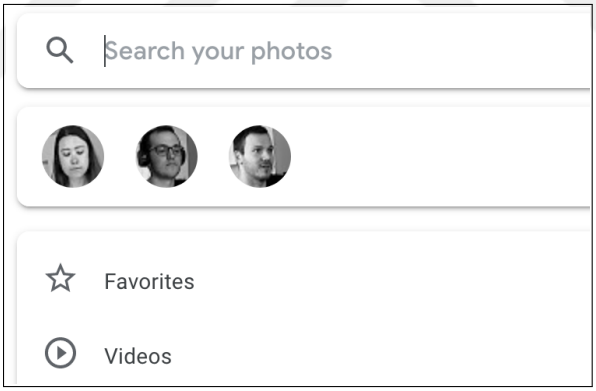

Figure 4.4. Google Photos face recognition

We also changed the way Processing Layer works to compare performance of AWS Rekognition and our face recognition. For the sake of this purpose Processing Layer uploads each frame it receives to Cloud Layer. This way we have a complete picture of two systems.

Results are populated in AWS DynamoDB. We exported this data as CSV file using DynamoDBtoCSV tool [68]. We then imported the data to Google Sheets for simple analysis and MySQL database for more complicated analysis.

Let,

$$A = \{\, f \mid f \text{ is frame recognized as Aysenur} \,\}$$

$$H = \{\, f \mid f \text{ is frame recognized as Halil} \,\}$$

$$A \cap H = \{\, f \mid f \text{ is frame recognized as Aysenur and Halil} \,\}$$

$$A \cup H = \{\, f \mid f \text{ is frame recognized as either Aysenur or Halil} \,\}$$

$A \cap H$ contains frames where Halil and Aysenur recognized together. $A \cap H$ defines all frames that has been recognized as either of them. 83 images are recognized as both people present in the frame by Processing Layer and 26 by Cloud Layer. We have 16 frames that was presented in both dataset by Google Photos. Processing Layer mistakenly recognized face as Halil and Aysenur. Algorithm is presented in Appendix A, Listing 7. It should be noted that the algorithm does not skip a face once it has been tagged as recognized. If the same face encoding is matched in multiple encodings we expect multiple successful matches. In other words, same face can be recognized as two different people.

Table 4.1 shows number of recognized images of Halil (H) and Aysenur (A) on Processing and Cloud Layers. Overall 2526 frames are recognized by Processing Layer and 2522 by Cloud Layer.

Processing Layer successfully recognized 93.56 percent of our dataset whereas Cloud Layer recognized 93.41 percent. Clearly our recognition platform performs as good as AWS Rekognition platform. In order to compare Processing and Cloud Layers we uploaded all images to the Cloud Layer. However our system design skips frame that has been recognized locally and only uploads unrecognized frames. Our system can reduce the bandwidth consumption as much as 93.56 percent. Compared to [10] we not only reduce bandwidth by only sending face frames but also reduce our Cloud bandwidth as much as 93.56 percent by processing it locally.

Table 4.1. Number of recognized images in Processing and Cloud Layers

	Processing Layer	Cloud Layer	Sample Size
$\|H\|$	1182	1151	1291
$\|A\|$	1427	1397	1423
$\|A \cap H\|$	83	26	14
$\|A \cup H\|$	2526	2522	2714

We analyzed number of frames that has been recognized successfully by Cloud Layer and frames that have been missed by both systems. Let,

$$CL = \{\, f \mid f \text{ is frame recognized by Cloud Layer} \,\}$$

$$PL = \{\, f \mid f \text{ is frame recognized by Processing Layer} \,\}$$

Let Number of frames that has been recognized by CL but not PL as $NCL = CL \setminus PL$. Figure 4.5 shows Venn Diagram representation of recognition rates while highlighting NCL. Universal Set $\mathcal{E}$ contains frames that are not recognized by neither PL nor CL. Labels inside circles are cardinality of their corresponding intersections.

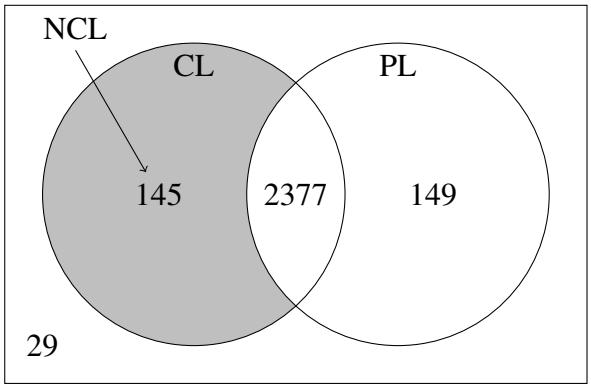

Figure 4.5. Venn diagram of recognition success rates

Our experiments show that 2526 frames are recognized locally by Processing Layer and

145 by Cloud Layer. This leaves 29 frames that could not be recognized by both systems. This is because we use different recognition algorithms in those layers. Faces that has been missed by PL can be recognized in CL. Both systems perform similarly compared to each other but they perform superior when they are combined. Overall our architecture successfully recognizes 98.93 percent of our dataset. We confirm [16] that performance of dlib is comparible to AWS. In addition, we found that using both approaches together increases overall success rate rather than using same algorithm in both layers [17].

Table 4.2. Relationship matrix of UPD

	Aysenur	Halil
Ayse	Affinity	Mother
Mustafa	Father	Affinity
Pembe	Mother	Affinity
Willem	Non-Relative	Non-Relative

False positives, number of photos wrongly recognized as a known person, are measured during our experiments. We have generated an Unrecognized People Dataset (UPD) which are not trained by the system and should be not be recognized. Table 4.2 shows a relationship matrix with household members and people in UPD. Table 4.3 gives sample size of each person in UPD and number of frames that has been marked as recognized for each household member. We found that out of 441 faces of UPD, 68 of them labeled as either Aysenur or Halil. Algorithm rarely gives false positives for opposite genders. It is noteworthy that false positives for non relatives can be quite high (Willem and Halil: 23 percent). This benchmark is achieved by using 0.6 for distance threshold which is the recommended settings. Lower is more strict, since similar faces should be grouped together in the space.

Processing time is measured and showed on Table 4.4. Processing Layer is slightly slower than Cloud Layer. On average Cloud Layer outperforms Processing Layer by 247 milliseconds. Considering for a home surveillance application where FPS rate is not required to be high this performance drawback can be compensated. We also see that σ

Table 4.3. False positives in Processing Layer. 0.6 distance threshold

Person	Sample Size	Aysenur	Halil	Aysenur (%)	Halil (%)
Ayse	70	11	1	15.71	1.43
Mustafa	185	9	6	4.86	3.24
Pembe	173	38	0	21.97	0.00
Willem	13	0	3	0.00	23.08
Total	441	58	10	13.15	2.27

between two systems are similar and does not differ much.

Table 4.4. Performance of recognition

	Success Rate (%)	Mean (seconds)	Standard Deviation (σ)
Processing Layer	93.56	1.871	0.480
Cloud Layer	93.41	1.624	0.424
Overall Pipeline	98.93	1.746	

5. CONCLUSIONS

In this thesis, a hierarchical architecture is proposed for home surveillance using Raspberry Pi as Edge Server. We used Docker in Raspberry Pi successfully and setup a cluster of Raspberry Pi devices to create a highly available system locally. Our recognition algorithm utilized 364 faces of two individuals in its face database. We have used four Raspberry Pi 3 Model B in our Processing Layer while one also acted as a load balancer.

We have connected this local cluster to Cloud Layer and utilized AWS technologies. We used Lambda for serverless execution, Rekognition for face recognition, S3 for object storage, SNS for notification system and DynamoDB as NoSQL database. Rekognition only allows for one reference picture thus we used two reference pictures of two individuals in our Cloud Layer.

Our architecture was able to recognize 93.56 percent of our dataset locally and 98.93 percent in overall pipeline. Using AWS Rekognition exclusively resulted in 93.41 percent success rate, slightly worse than dlib's performance. However in our pipeline where only not recognized photos are uploaded to the cloud, overall success rate becomes far superior. We realized that false positives can be quite high in [9] with their recommended distance threshold 0.6 (smaller is more strict). Reducing it to 0.5 eliminates all false positives in our dataset while reducing overall recognition success rate.

We experienced more than 50 percent drop in false positives in face detection by using parameters from [19]. However this also removed some valid faces from our initial database. Our conclusion is that Haar Cascade algorithm can be used for face detection but it requires fine tuned parameters for specific distances. For real time applications, Haar Cascade may result in plenty of false positives and it could omit some valid faces if they are distant or do not fit the model. We recommend to use a deep method for face detection.

On average a face is recognized in $1.871s$ on Raspberry Pi and $1.642s$ in AWS Lambda. We achieved comparable results with AWS Rekognition in our local processing with 364 faces

in face database. Fewer faces in database results in faster computation. It should be noted that duration of the AWS Lambda does not take bootstrap time of AWS Lambda function and transmission time to the Cloud. Although locally recognition on Raspberry Pi is slightly slower than AWS Lambda, difference is not crucial. Network overhead is excluded from Cloud Layer duration in our statistics. Poor Internet connection can dwarf this result and can increase Cloud Layer duration significantly. Processing Layer is a safeguard against this with a minor delay in processing time. If high FPS is required by the application more powerful Edge Servers can reduce this delay significantly.

Our implementation permits adding more edge servers easily to the existing infrastructure. Each server is stateless and containerized. State is persisted on DynamoDB database on Processing and Cloud Layers. In case of multiple cameras and high demanding applications, system can be scaled horizontally by adding more servers.

We published our source code under Mozilla Public License on Github [69]. Docker images used in this work are publicly available under Docker Hub [60]. Configuration parameters are defined as environmental variables and documented under the Github project. We believe this work can be duplicated by other researches with minimal effort.

REFERENCES

1. Lin CC. Network video recording system. Google Patents; 2003. US Patent App. 10/139,923.

2. Video Surveillance Storage: How Much Is Enough? — Seagate US; [cited 2018 21 May]. Available from: https://www.seagate.com/tech-insights/how-much-video-surveillance-storage-is-enough-master-ti/.

3. Oreskovic A. Google to acquire Nest for $3.2 billion in cash. — Reuters [cited 2018 21 May]. Available from: http://reut.rs/1hQJ0qP.

4. Rodríguez-Silva DA, Adkinson-Orellana L, Gonz'lez-Castano F, Armino-Franco I, Gonz'lez-Martinez D. Video surveillance based on cloud storage. *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*; 2012:IEEE

5. Yi S, Li C, Li Q. A survey of fog computing: concepts, applications and issues. *Proceedings of the 2015 Workshop on Mobile Big Data*; 2015:ACM.

6. Nest Cam IQ Indoor — The Sharper Home Security Camera — Nest; [cited 2018 26 May]. Available from: https://nest.com/cameras/nest-cam-iq-indoor/overview/.

7. Nest. Nest Aware | Continuous Video Recording for Nest Cams; 2019. [cited 2019 9 January]. Available from: https://nest.com/cameras/nest-aware.

8. Raspberry Pi Zero - Raspberry Pi; [cited 2018 18 July]. Available from: https://www.raspberrypi.org/products/raspberry-pi-zero/.

9. Geitgey A. ageitgey/face_recognition: The world's simplest facial recognition api for Python and the command line; [cited 2018 7 September]. Available from: https://github.com/ageitgey/face_recognition.

10. Park HD, Min OG, Lee YJ. Scalable architecture for an automated surveillance system using edge computing. *The Journal of Supercomputing*. 2017;73(3):926–939.

11. Menezes P, Barreto JC, Dias J. Face tracking based on haar-like features and eigenfaces. *IFAC Proceedings Volumes*. 2004;37(8):304–309.

12. He K, Zhang X, Ren S, Sun J. Deep residual learning for image recognition. *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*; 2016:IEEE.

13. Turk M, Pentland A. Eigenfaces for recognition. *Journal of Cognitive Neuroscience*. 1991;3(1):71–86.

14. Huang GB, Mattar M, Berg T, Learned-Miller E. Labeled faces in the wild: a database for studying face recognition in unconstrained environments. *Workshop on Faces in 'Real-Life' Images: Detection, Alignment, and Recognition*. 2008:Inria.

15. Parkhi OM, Vedaldi A, Zisserman A. Deep face recognition. *Proceedings of the British Machine Vision Conference*. 2015:BMVC

16. Popic U. Two approaches for face recognition with IOT technologies [Master's thesis]. Politecnico di Milano. Piazza Leonardo da Vinci, 32 20133 Milano, Italy; 2018.

17. Muslim N, Islam S. Face recognition in the edge cloud. *Proceedings of the International Conference on Imaging, Signal Processing and Communication*; 2017:ACM

18. Alsmirat MA, Jararweh Y, Obaidat I, Gupta BB. Internet of surveillance: A cloud supported large-scale wireless surveillance system. *The Journal of Supercomputing*. 2017;73(3):973–992.

19. Wazwaz AA, Herbawi AO, Teeti MJ, Hmeed SY. Raspberry Pi and computers-based face detection and recognition system. *2018 4th International Conference on*

Computer and Technology Applications (ICCTA). 2018:IEEE

20. Piedad F, Hawkins M. *High availability: design, techniques, and processes*. New Jersey: Prentice Hall Professional; 2001.

21. Cloud Object Storage — Store & Retrieve Data Anywhere — Amazon Simple Storage Service; [cited 2018 06 June]. Available from: https://aws.amazon.com/s3/.

22. Jayakumar AJK, Muthulakshmi S. Raspberry Pi-Based surveillance system with IoT. *Intelligent Embedded Systems.*; 2018;492:173–185.

23. Yang MJ, Tham JY, Wu D, Goh KH. Cost effective IP camera for video surveillance. *Industrial Electronics and Applications, 2009. ICIEA 2009. 4th IEEE Conference on*. 2009:IEEE

24. Schulzrinne H, Casner S, Frederick R, Jacobson V. RTP: A Transport Protocol for Real-Time Applications. RFC Editor; 2003. 3550.

25. Free Virtual Machines from IE8 to MS Edge - Microsoft Edge Development; 2018. [cited 2018 23 December]. Available from: https://developer.microsoft.com/en-us/microsoft-edge/tools/vms.

26. Insecam - World biggest online cameras directory; 2018. [cited 2018 23 December]. Available from: http://www.insecam.org.

27. Smith M. Peeping into 73,000 unsecured security cameras via default passwords. [cited 2018 23 December]. Available from: https://www.csoonline.com/article/2844283/microsoft-subnet/peeping-into-73-000-unsecured-security-cameras-thanks-to-default-passwords.html.

28. Zhang T, Chowdhery A, Bahl PV, Jamieson K, Banerjee S. The design and implementation of a wireless video surveillance system. *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*; 2015:ACM

29. ns-3 — a discrete-event network simulator for internet systems. [cited 2018 03 July]. Available from: https://www.nsnam.org/.

30. Viola P, Jones M. Rapid object detection using a boosted cascade of simple features. *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. 2001:IEEE

31. Mullins R. Department of Computer Science and Technology: Raspberry Pi; 2017. [cited 2019 3 January]. Available from: https://www.cl.cam.ac.uk/projects/raspberrypi.

32. Upton E. Raspberry Pi 3 Model B+ on sale now at $35 - Raspberry Pi; 2018. [cited 2018 27 August]. Available from: https://www.raspberrypi.org/blog/raspberry-pi-3-model-bplus-sale-now-35/.

33. Tso FP, White DR, Jouet S, Singer J, Pezaros DP. The glasgow raspberry pi cloud: a scale model for cloud computing Infrastructures. *2013 IEEE 33rd International Conference on Distributed Computing Systems Workshops*. 2013:IEEE

34. Etcher; [cited 2018 15 September]. Available from: https://etcher.io/.

35. Kocher P, Genkin D, Gruss D, Haas W, Hamburg M, Lipp M, et al. Spectre Attacks: Exploiting Speculative Execution. *CoRR*. 2018;abs/1801.01203.

36. Docker - Build, Ship, and Run Any App, Anywhere; [cited 2018 29 June]. Available from: https://www.docker.com/.

37. OpenCV library; 2018. [cited 2019 3 January]. Available from: https://opencv.org.

38. King DE. Dlib-ml: A machine learning toolkit. *Journal of Machine Learning Research*. 2009;10:1755–1758.

39. Combe T, Martin A, Pietro RD. To docker or not to docker: a security perspective. *IEEE Cloud Computing*. 2016;3:54–62.

40. Lienhart R, Maydt J. An extended set of Haar-like features for rapid object detection. *Proceedings. International Conference on Image Processing*. 2002:1;I–I.

41. Thanks to Amazon, the government will soon be able to track your face — Opinion — The Guardian; [cited 2018 25 October]. Available from: https://www.theguardian.com/commentisfree/2018/jul/06/amazon-rekognition-facial-recognition-government.

42. Smith B. Facial recognition: It's time for action - Microsoft on the Issues; 2018. [cited 2018 23 December]. Available from: https://blogs.microsoft.com/on-the-issues/2018/12/06/facial-recognition-its-time-for-action.

43. Lowe DG. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*. 2004;60(2):91–110.

44. Wang X, Han TX, Yan S. An HOG-LBP human detector with partial occlusion handling. *Computer Vision, 2009 IEEE 12th International Conference on*. 2009:IEEE

45. Schroff F, Kalenichenko D, Philbin J. Facenet: A unified embedding for face recognition and clustering. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2015:IEEE

46. dlib C++ Library: High Quality Face Recognition with Deep Metric Learning; [cited 2018 24 July]. Available from: http://blog.dlib.net/2017/02/high-quality-face-recognition-with-deep.html.

47. Announcing Amazon Elastic Compute Cloud (Amazon EC2) - beta; 2018. [cited 2018 8 December]. Available from: https://aws.amazon.com/about-aws/whats-new/2006/08/24/announcing-amazon-elastic-compute-cloud-amazon-ec2—beta.

48. Introducing Google App Engine + our new blog; 2008. [cited 2018 8 December]. Available from: http://googleappengine.blogspot.com/2008/04/introducing-google-app-engine-our-new.html.

49. Windows Azure General Availability; 2010. [cited 2018 8 December]. Available from: https://blogs.technet.microsoft.com/microsoft_blog/2010/02/01/windows-azure-general-availability.

50. Columbus L. 83 percent of Enterprise Workloads Will Be In The Cloud By 2020. Forbes Magazine; 2018. [cited 2018 8 December]. Available from: https://www.forbes.com/sites/louiscolumbus/2018/01/07/83-of-enterprise-workloads-will-be-in-the-cloud-by-2020.

51. Employees at Google, Amazon, and Microsoft protest government contracts - Vox; [cited 2018 14 November]. Available from: https://www.vox.com/technology/2018/10/18/17989482/google-amazon-employee-ethics-contracts.

52. Nix N. Why Pentagon Cloud-Computing Contract Is a Huge Deal. [cited 2018 12 October]. Available from: https://www.bloomberg.com/news/articles/2018-10-04/why-pentagon-cloud-computing-contract-is-a-huge-deal-quicktake.

53. Lambert S. 2018 SaaS Industry Market Report: Key Global Trends & Growth Forecasts - Financesonline.com; 2018. [cited 2018 08 December]. Available from: https://goo.gl/jv81JY.

54. App Engine - Build Scalable Web & Mobile Backends in Any Language | App Engine; 2018. [cited 2018 8 December]. Available from: https://cloud.google.com/appengine.

55. Cloud Application Platform | Heroku; 2018. [cited 2018 11 October]. Available from: https://www.heroku.com.

56. AWS Elastic Beanstalk — Deploy Web Applications; 2018. [cited 2018 22 October]. Available from: https://aws.amazon.com/elasticbeanstalk.

57. Baldini I, Castro P, Chang K, Cheng P, Fink S, Ishakian V, et al. Serverless computing: Current trends and open problems. *Research Advances in Cloud Computing*. 2017;1-

20.

58. Exabyte-scale Data Transfer | AWS Snowmobile; 2018. [cited 2018 26 October]. Available from: https://aws.amazon.com/snowmobile.

59. Facial recognition technology: The need for public regulation and corporate responsibility - Microsoft on the Issues; [cited 2018 15 November]. Available from: https://goo.gl/tsVkti.

60. kaskavalci - Docker Hub; 2018. [cited 2018 28 October]. Available from: https://hub.docker.com/r/kaskavalci.

61. Getting started with Docker on your Raspberry Pi · Docker Pirates ARMed with explosive stuff; 2018. [cited 2018 12 December]. Available from: https://blog.hypriot.com/getting-started-with-docker-on-your-arm-device.

62. rickryan/rpi-jessie-opencv3.2: Dockerfile to create a container for Raspberry Pi with jessie and opencv3.2; 2018. [cited 2018 12 December]. Available from: https://github.com/rickryan/rpi-jessie-opencv3.2.

63. JackyChiu/slb: Simple Load Balancer; 2018. [cited 2018 8 December]. Available from: https://github.com/JackyChiu/slb.

64. Face recognition with OpenCV, Python, and deep learning — PyImageSearch; 2018. [cited 2018 29 December]. Available from: https://www.pyimagesearch.com/2018/06/18/face-recognition-with-opencv-python-and-deep-learning.

65. GitHub - davisking/dlib-models: Trained model files for dlib example programs.; [cited 2018 09 July]. Available from: https://github.com/davisking/dlib-models.

66. Open Source Computer Vision Library; 2018. [cited 2018 2 December]. Available from: https://github.com/opencv/opencv.

67. pickle — Python object serialization — Python 2.7.15 documentation; [cited 2018 02 December]. Available from: https://docs.python.org/2/library/pickle.html.

68. edasque/DynamoDBtoCSV: Dump DynamoDB data into a CSV file; 2018. [cited 2018 30 December]. Available from: https://github.com/edasque/DynamoDBtoCSV.

69. kaskavalci/PiGuard: PiGuard makes your Raspberry Pi(s) fully functional survellience device; 2019. [cited 2019 8 January]. Available from: https://github.com/kaskavalci/PiGuard.

APPENDIX A: ALGORITHMS

Listing A. 1. Generate face encodings of the dataset.

```
# compute the facial embedding for each face in the dataset
encodings = face_recognition.face_encodings(rgb, face)
# loop over the encodings
for encoding in encodings:
    # add each encoding + name to our set of known names and
    # encodings
    knownEncodings.append(encoding)
    knownNames.append(name)
# Combine encodings and labels in one struct
data = {"encodings": knownEncodings, "names": knownNames}
# pickle and save it
f.write(pickle.dumps(data))
```

Listing A. 2. IP Camera initialization.

```
import cv2

def initcam(self):
    # Define IP camera's RTSP endpoint.
    # This can be obtained from device manual
    # Usually username: admin, password: <empty>
    url = "rtsp://192.168.1.10:554/" +
     ↪  "user=admin&password=&channel=1&stream=1.sdp"
    self.__vcap = cv2.VideoCapture(url)

    # Manually set FPS
    self.__vcap.set(cv2.CAP_PROP_FPS, 1)
    # Set a buffer size to escape from lagged stream
    self.__vcap.set(cv2.CAP_PROP_BUFFERSIZE, 3)

    # Get camera resolution
    w = self.__vcap.get(cv2.CAP_PROP_FRAME_WIDTH)
    h = self.__vcap.get(cv2.CAP_PROP_FRAME_HEIGHT)
    self.__resolution = (w, h)

    # Create grabber thread to avoid lagged stream
    self.__grabberThread =
     ↪  threading.Thread(target=self.grabber)
    self.startGrabber()
```

Listing A. 3. Grabber thread and retrieve in action.

```
__camLock = threading.Lock()

def grabber(self):
    while True:
        self.__camLock.acquire()
        self.__vcap.grab()
        self.__camLock.release()

def run(self):
    # Retrieve frames
    while True:
        self.__camLock.acquire()
        retval, frame = self.__vcap.retrieve()
        self.__camLock.release()
        # ... process the frame
```

Listing A. 4. Algorithm to detect motion.

```
# returns true if there is motion between current frame and the previous frame
def isThereMotion(self, frame):
    blur = cv2.GaussianBlur(frame, (21, 21), 0)
    if self.__previousFrame is None:
        self.__previousFrame = blur
        return True

    # compute the absolute difference between the current frame and first frame
    frameDelta = cv2.absdiff(self.__previousFrame, blur)
    thresh = cv2.threshold(frameDelta, 25, 255, cv2.THRESH_BINARY)[1]

    # dilate the thresholded image to fill in holes, then find contours
    # on thresholded image
    thresh = cv2.dilate(thresh, None, iterations=2)
    _, contours, _ = cv2.findContours(thresh, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
    self.__previousFrame = blur

    # If there are contours within the difference then there are motion
    if len(contours) > 0:
        return True

    return False
```

Listing A. 5. Face Detection Algorithm.

```
# Crops (any) faces from the frame and calls upload function for each face
def find_faces(self, frame):
    img = cv2.resize(frame, (1024, 768))
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

    faces = self.__faceCascade.detectMultiScale(
            img,
            scaleFactor=1.1,
            minNeighbors=5,
            minSize=(100, 100),
            flags=cv2.CASCADE_SCALE_IMAGE
    )

    for face in faces:
        crop = self.crop(img, face)
        fname = date.isoformat() + '.jpg'
        self.upload(img, fname)
```

Listing A. 6. Upload face using Pickle.

```
def upload(self, picture, fname):
    try:
        _, img_encoded = cv2.imencode('.jpg', picture)
        # Host is a FQDN which is hard-coded
        response = requests.put(self.__args.host,
                    data=pickle.dumps(img_encoded),
                    headers={
                        'content-type': 'image/jpeg',
                        'Filename': fname},
                    )
        if response.status_code != 204:
            logging.error('failed to send picture to server:
            {}'.format(response.content))
    except:
        print("web server is not available")
```

Listing A. 7. Performing face recognition with `face_recognition` library.

```
# Initialize dictionary with False values for each person
# Then copy the dictionary at each recognition attempt
recognized_faces = dict(self._recognized_names)
encodings = face_recognition.face_encodings(unknown_face)
for face_encoding in encodings:
    # See if the face is a match for the known face(s)
    match = face_recognition.compare_faces(
        known_faces["encodings"], face_encoding, 0.6)

    for i, m in enumerate(match):
        name = self._face_encodings["names"][i]
        if m:
            recognized_faces[name] = True
return recognized_faces
```

Listing A. 8. Performing face recognition with `AWS Rekognition`.

```
def compare_faces(bucket, key, source):
    src={'S3Object':{'Bucket':"known-faces",'Name':source}}
    dst={'S3Object':{'Bucket':bucket,'Name':key}}
    response = rekognition.compare_faces(
        SimilarityThreshold=60,
        SourceImage=src,
        TargetImage=dst)

    return response
```