NOVA
NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

FILIPE LOURENÇO LOPES PREGAL
Bachelor in Computer Science

FACE-BASED PHOTO INDEXING IN EDGE COMPUTING ENVIRONMENTS

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
April, 2024

NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

FACE-BASED PHOTO INDEXING IN EDGE COMPUTING ENVIRONMENTS

FILIPE LOURENÇO LOPES PREGAL

Bachelor in Computer Science

Adviser: Hervé Miguel Cordeiro Paulino
Associate Professor, NOVA University Lisbon

Examination Committee:

Chair: Nuno Manuel Robalo Correia
Full Professor, NOVA University Lisbon

Rapporteur: António Gelásio Frazão Isidro Teófilo
Assistant Professor, Instituto Superior de Engenharia de Lisboa

Adviser: Hervé Miguel Cordeiro Paulino
Associate Professor, NOVA University Lisbon

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
April, 2024

Face-based Photo Indexing in Edge Computing Environments

Copyright © Filipe Lourenço Lopes Pregal, NOVA School of Science and Technology, NOVA University Lisbon.
The NOVA School of Science and Technology and the NOVA University Lisbon have the right, perpetual and without geographical boundaries, to file and publish this dissertation through printed copies reproduced on paper or on digital form, or by any other means known or that may be invented, and to disseminate through scientific repositories and admit its copying and distribution for non-commercial, educational or research purposes, as long as credit is given to the author and editor.

To my family and friends.

Acknowledgements

I would like to thank my professor and advisor Hervé Paulino for guiding me and believing in me through the elaboration of this dissertation. I would also like to thank NOVA School of Science and Technology, the Department of Computer Science and all the professors that taught me during the years I spent in this institution.

I would like to thank my family and friends for supporting me not only during the elaboration of this thesis, but during the whole time I have spent in the university. I would like to give a special thanks to the friends I have met in this journey: Daniel Batista, Vasco Carvalho, Pedro Franco, Nádia Mendes, Leandro Teixeira and Miguel Barreto, for helping me and standing by me in my learning process.

"Never forget what you are, for surely the world will not. Make it your strength. Then it can never be your weakness." (George R. R. Martin)

Abstract

Over recent years, smart mobile devices have grown in popularity. With such popularity growth, data traffic has also increased, which gave rise to new problems such as higher latency in data requests or less data storage capability. A paradigm that promises to nullify many of the issues with this growth is Edge Computing. A computing paradigm composed by the user devices, edge servers and the cloud. Edge Servers, located at the edge of the internet, close to the user's devices, help processing and disseminating information and data.

During this dissertation, we propose to enhance Chives, a machine learning API to identify faces in photos, in this environment, creating a cluster indexing system, in order to enable the development of a photo sharing application with lower latency in picture search through facial recognition. Such index is created using Conflict-free Replicated Data Types. On a mobile phone, the user will be able to search for photos with faces, using a photo of a similar face as an input.

Our solution proved to retrieve the correct results, while being almost as fast as the human eye perception's capability, being a good addition to the EdgeGarden environment.

Keywords: Conflict-Free Replicated Data Type, Edge Computing, Indexing at the Edge, Machine Learning, Publish/Subscribe

Resumo

Ao longo dos últimos anos, os dispositivos móveis inteligentes têm crescido em popularidade. Com esta evolução de popularidade, o tráfego de dados também aumentou, o que provocou um despertar de novos problemas como elevadas latências em pedidos de dados ou menor capacidade de armazenamento de dados. Um paradigma que promete anular muitos dos problemas associados a este crescimento é a Computação na Edge. Um paradigma computacional composto por dispositivos dos utilizadores, servidores na Edge e a Cloud. Os servidores na Edge, localizados nas periferias da Internet, perto dos dispositivos dos utilizadores, vêm ajudar a processar e distribuir informação e dados.

Durante esta dissertação, propomos melhorar o Chives, uma API de aprendizagem automática para identificar caras em fotos, neste ambiente, criando um sistema de indexação de clusters, para permitir o desenvolvimento de uma aplicação de partilha de fotos com baixa latência na procura de imagens por reconhecimento facial. Este índice é criado utilizando Conflict-free Replicated Data Types. No telemóvel, o utilizador poderá procurar fotos com caras, utilizando uma foto semelhante como input.

A nossa solução provou obter resultados corretos, enquanto se mantém quase tão rápida como a capacidade de perceção do olho humano, sendo uma boa adição ao ambiente do EdgeGarden.

Palavras-chave: Conflict-Free Replicated Data Type, Computação na Edge, Indexação na Edge, Aprendizagem Automática, Publish/Subscribe

Contents

List of Figures xi

List of Tables xii

Acronyms xiii

1 Introduction 1

1.1 Context and Motivation . . . 1
1.2 Edge Computing . . . 2
1.3 The Problem . . . 3
1.4 Proposal . . . 3
1.5 Contributions . . . 4
1.6 Document Overview . . . 4

2 Background and Related Work 6

2.1 The EdgeGarden Ecosystem . . . 6
2.1.1 General Architecture . . . 6
2.1.2 Caching Service . . . 7
2.1.3 APIs . . . 8
2.1.4 Chives . . . 12
2.2 Mutable Data in EdgeGarden . . . 13
2.2.1 Weak Consistency . . . 14
2.2.2 Eventual Consistency . . . 14
2.2.3 Conflict-free Replicated Data Types . . . 15
2.2.4 Publish/Subscribe CRDTs . . . 17
2.3 Indexing at the Edge . . . 17
2.3.1 The Approaches . . . 19
2.3.2 Discussion . . . 22

3 Developed Solution 24

3.1 Solution Overview 24
3.2 Edge Services 28
3.2.1 Publish-Subscribe 29
3.2.2 Index Creation and Publication 33
3.2.3 The Face Indexing Service 37
3.3 Mobile Client Services 38
3.3.1 Receiving and Using the Index 39
3.3.2 The Face Searching Service 41
3.3.3 Chives' API 41
3.4 Implementation Details 43
3.4.1 Distinguishing Between Mobile and Edge 43
3.4.2 Restructuring the Chives Module 43
3.4.3 Configuring Gradle 45
3.4.4 Copying Files Task 45
3.5 Final Remarks 46

4 Experimental Evaluation 47
4.1 Goals 47
4.2 Experimental Environment 48
4.2.1 Simulated Environment 48
4.2.2 Real Environment 48
4.3 Evaluation 49
4.3.1 Solution Accuracy Study 49
4.3.2 Solution Efficiency 53
4.3.3 Summary 55

5 Conclusion 56
5.1 Conclusions 56
5.2 Future Work 56
5.2.1 Integration With Oregano 57
5.2.2 Merging and Diverging Clusters 57

Bibliography 58

List of Figures

1.1 Annual Phone Users Growth 1
1.2 General Edge Computing Architecture 2

2.1 EdgeGarden Architecture 7

3.1 Architecture of the System in a Single Region 24
3.2 Creating and Sharing an Index of Faces 25
3.3 Searching for Photos Given a Photo of a Face 25
3.4 Frameworks Used in the Edge Server 27
3.5 Retrieving Data by ID on the Publish Subscribe Service 30
3.6 Functioning of Chives on the Edge 34
3.7 Clusters Under Our Context's Concept 34
3.8 Establishing a Connection Between Published Photos and Clusters 35
3.9 Difference Between Regular Sets and OR-Sets 36
3.10 The Face Indexing Service 37
3.11 APIs Used in the Mobile Phones 38
3.12 Downloading the Face Index on the Mobile Phone 40
3.13 The Face Searching Process 42
3.14 Modules Organization 44

4.1 Distance Between the Projection of the Test Photo's Face and the Closest Projection in the Index 50
4.2 Maximum Number of Projections of the Same Cluster Within the 7 Neighbours of the Input Photo's Projection 51
4.3 Maximum Number of Projections of the Same Cluster Within the 13 Neighbours of the Input Photo's Projection 52
4.4 Testing Time and Battery Consumption 54

List of Tables

2.1 System Architecture 6
2.2 Indexing at the Edge in the State of the Art 18

3.1 Possible Sets 35

4.1 Device Used for the Experimental Tests on a Simulated Environment . . . 48
4.2 Devices Used for the Experimental Ttests on a Real Environment 48

Acronyms

Δ-CRDT Delta-Based CRDT 16, 17

AI Artificial Intelligence 4
API Application Programming Interface 3, 4, 6, 7, 8, 9, 10, 26, 27, 28, 29, 32, 37, 38, 39, 41, 46, 56

CB-DHT Cluster-Based Distributed Hash Table 8, 9
COIN COordinate-based INdexing 18, 19, 21, 22, 23
CooLSM Cooperative Log-Structured Merge tree 18, 19, 20
CRDT Conflict-Free Replicated Data Type vii, viii, 4, 6, 13, 14, 15, 16, 17, 19, 30, 31, 36, 37

DC Data Center 2, 3, 18, 21, 23
DHT Distributed Hash Table 19

EC Eventual Consistency 14, 15, 17

FcE Face Extraction 26, 39, 57
FR Facial Recognition 27
FtE Feature Extraction 26, 34, 39, 43, 57

G-Set Grow-Only Set 36

IoT Internet of Things 1, 2
IR Image Recognition 6, 11, 18, 22

KNN K-Nearest Neighbors 40, 48, 49, 50, 52

MDD Mobile Dynamic Dataset 11

O-CRDT Operation-Based CRDT 16, 17, 35
OR-Set Observed-Remove Set 16, 36, 39
OS Operative System 43, 48

P/S Publish/Subscribe vii, viii, 3, 8, 17, 26, 27, 28, 29, 31, 32, 35, 36, 37, 46
PO-CRDT Pure-Operation-Based CRDT 16, 17
PS-CRDT Publish-Subscribe CRDT 4, 16, 17, 29

S-CRDT State-Based CRDT 16, 17, 35
SC Strong Consistency 13, 14, 16
SEC Strong Eventual Consistency 15
SubC Subscription with Computation 10, 11, 57

UUID Universally Unique Identifier 29

WC Weak Consistency 14, 16

1

Introduction

1.1 Context and Motivation

The *boom* of mobile smartphones over the last few years is a global phenomenon and mobile applications come with them. The preference of smartphones over feature phones is evident, as shown in Figure 1.1. While the number smartphone users grow each year, the opposite happens for the traditional feature phones. By 2026, the number of smartphones users is expected to grow up to 7455 millions [8]. It has been a while since the smartphones started to diverge from simple calling and texting devices and to allow further utilities such as taking photos and sharing content. Internet-driven communication between smartphones is another key feature that differentiates them from regular mobile phones. With smartphones, smart-televisions, computers and a set of all other application and communication-driven devices come different computing paradigms, like the Internet of Things (IoT).

The IoT is a concept that represents the connection of physical everyday devices, (i.e.

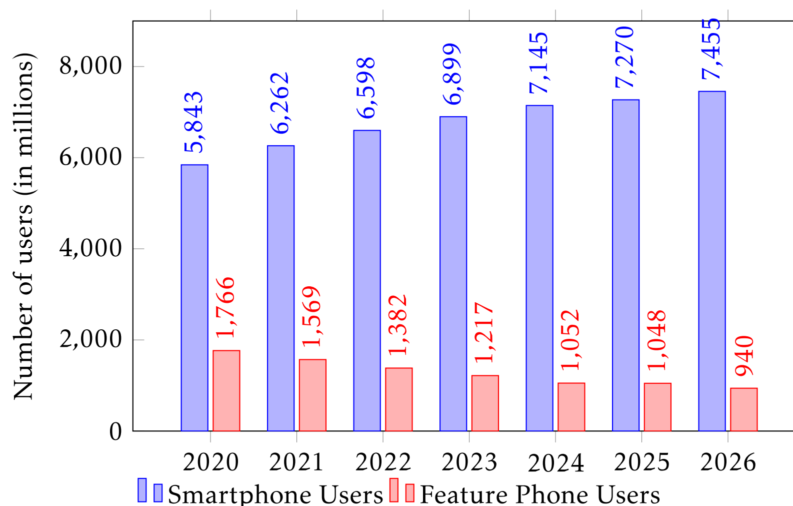

Figure 1.1: Annual Phone Users Growth (Including Expected Growth After 2021). Adapted from [8].

televisions, light bulbs, speakers, kitchen machines,...) to a local network or the internet, to enable them to trade data, process information together and remote control [26]. An example of this paradigm is presented by Baseca et al., who created an IoT environment with cameras and irrigation controllers in order to enable farmers to monitor and control when to irrigate their crops [5].

As the amount of smartphone users increases, so does the amount of data produced. In 2020, the world's total amount of data traffic was 47 exabytes per month. It is expected to grow to 221 exabytes per month, in 2026 [8]. Given the huge amount of data, sometimes with local interest, some applications generate on the edge of the internet, it does not make sense to send all these data to the cloud to allow sharing it between a set of geographically close devices. For example, in a photo-sharing application where a group of people are at a wedding, a sports event or a concert and wish to share photos between themselves. In cases like this one, it would make sense to have an environment which allowed their devices to share their photos to the destination device directly or through a closer intermediate.

1.2 Edge Computing

The Edge Computing paradigm comes as a solution to such requirements, bringing the information processing away from the cloud (and also from the devices) to edge servers, also referred to by edge nodes, which are the intermediate agents, near the devices. A general architecture of Edge Computing is composed by these three levels, as shown in Figure 1.2. Such environment allows higher performance, resulting in a better user experience [11]. The most well known example is the IoT.

As mentioned before, using the original concept of a single Data Center (DC) is not always beneficial. Suppose a voice recognition system where users record their voice to perform activities processed in the cloud. In this system, sending voice recording through the internet to a single DC would imply a bigger data flow, with higher latency,

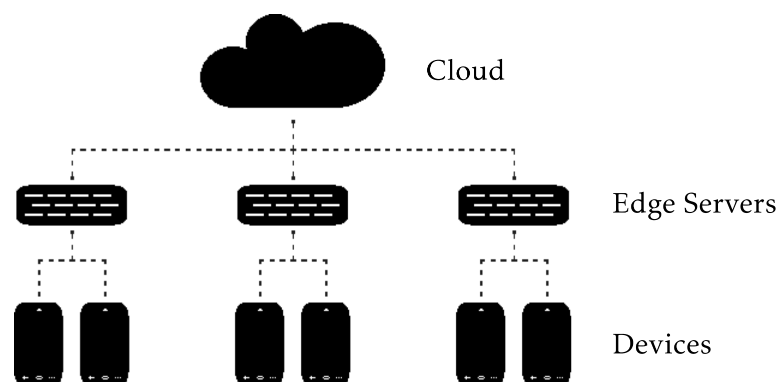

Figure 1.2: General Edge Computing Architecture

less information storage capability and a larger bandwidth than receiving only what was said in a text format. That is when edge computing takes place. In this environment, the user would record his voice in his device, the latter would send the recorded audio to the nearest edge server, which would run software to convert his voice into text. The text would then be sent to the cloud for further operations. The cloud DC would only receive the information in a text format, which would make the system more efficient than having the DC receive the audio directly and processing a huge quantity of audio files, if the application is scaled to a big quantity of users.

1.3 The Problem

EdgeGarden is an ecosystem of solutions to share data, including images, locally, on the edge, recurring to *tags* that identify and describe such data. It allows this by providing a set of usable Application Programming Interfaces (APIs) that support the programmer of the system with its intended purpose, without the need to opt for centralized services.

In the set of solutions of the ecosystem, come Thyme [24, 23], that enables data distribution and sharing between devices without the need of external servers, recurring to a Publish/Subscribe (P/S) abstraction, Basil [14], that adds Thyme a key-value storage abstraction and the option to persist data on the cloud, and Oregano [20], which enables computation on the data source device. It also allows the programmer to set up edge servers to connect multiple networks of devices and transfer some of the heavy weight, of storage and the operations, to the edge.

With such solutions, we can address the wedding example, stated in Section 1.1. Enabling the wedding attenders to use EdgeGarden, they can share their pictures of the event, setting up their own chosen *tags* and allowing other people to search for specific pictures and download those which they desire. All of this, without unwillingly sending the pictures to a public or private cloud.

It is important to notice that a detailed search, such as "*pictures with my face*", providing a specific *tag*, would imply a content-based search, which would lead to the need of applying computation to all the pictures with such *tag*, in order to identify the faces on these pictures. If the request "*pictures with the tag* bride *and my face*"is made multiple times on different devices, the same computation will be triggered, returning the same expected output. It does not seem very effective to do these same operations multiple times.

To address this problem, it is proposed to avoid repeated computation, by equipping this Ecosystem with an indexing system.

1.4 Proposal

This dissertation is following a set of other dissertations [16] in whose objective is to build an index of faces in a context of which it is not known previously whose faces are going to

appear in the photos. This previous work, named Chives, is an API that uses the Artificial Intelligence (AI) concept of *Clustering* to create groups (clusters) of images in which the same face appears. Whenever a cluster is big enough, it becomes a candidate to integrate the index.

The goal of this paper is to implement an edge service that:

1. Runs the state-of-the-art clustering algorithm on the most popular photos shared in the system;

2. Creates a device-shareable index with the mutable data on EdgeGarden, using Conflict-Free Replicated Data Types (CRDTs);

3. Allows content-based searches on the devices to, given a specific photo of a face, verify if the face is indexed and return the metadata with the necessary information to find the pictures which contain such face.

A search on the index is required to achieve item 3, in order to verify if the given face belongs anywhere near any cluster.

1.5 Contributions

With this dissertation, we expect to contribute the following way:

1. Developing system that indexes clusters of faces, allowing a more efficient search on the index and sends requests to working devices with photos for face recognition.

2. Evaluating the efficiency of such system, showing the advantages or disadvantages of using an index for content-based searches.

1.6 Document Overview

The contents and organization of this document are further described.

On chapter 2, it is presented the background of this paper, more specifically, the Edge-Garden Ecosystem, on section 2.1, with its architecture, GardenBed's Caching Service, APIs (Thyme, Basil and Oregano) and the state of the art Chives, the API that we will enhance that retrieves a stream of clustered faces and, finally, the background on Mutable Data, on section 2.2, explaining weak and eventual consistencies, CRDTs and their types, including Publish-Subscribe CRDTs (PS-CRDTs), the ones to be used, the background on Indexing at the Edge, on section 2.3, with some state of the art approaches and some comparison between them and what it is intended to be done in this paper.

On chapter 3, it is presented the solution for this dissertation, starting with an overview, on section 3.1, the solution for the edge, on section 3.2, and the one for the mobile phones, on section 3.3, followed by some implementation details, on section 3.4, and some final remarks, on section 3.5.

On chapter 4, we present the experimental evaluation performed: we start by presenting its goals, on section 4.1, and environments, on section 4.2. We, then, finish the chapter by evaluating the solution, on section 4.3. We finish this dissertation with the conclusions of the work performed, on chapter 5.

2

Background and Related Work

As previously stated in section 1.3, the EdgeGarden ecosystem is a system that searches to provide users a way to share pictures to this system between phones without having necessarily to share them to a cloud or to the edge. One can just publish their photo and it will be automatically available to any user in the garden. It also provides a way to search pictures in the system using Image Recognition (IR) in order to find pictures that contain the face of a specific person. This essay proposes the use of Conflict-Free Replicated Data Types (CRDTs) to dynamically share and update pictures between the users of the system. In order to explain this, one must understand some notions on the state-of-the-art of this system.

2.1 The EdgeGarden Ecosystem

The EdgeGarden ecosystem is a set of solutions for data dissemination, storage and computation. In this section, we will present a general overview of the ecosystem's architecture and of the Application Programming Interfaces (APIs) that are relevant for this dissertation.

2.1.1 General Architecture

The ecosystem is a 3-level architecture within the Edge Computing paradigm composed by Mobile Devices, Servers on the Edge and a Cloud, as it can be seen in Figure 2.1. The devices are connected between them, through a wireless technology, such as Wi-Fi or 5G, and to the nearest Edge Node to provide and receive data items, based on tags labeling the theme of their content.

Table 2.1: System Architecture

Level	Execution System	Communication	APIs		Data		
			Data Storage and Sharing	Computation	Method	Duration	Relevance
Devices	EdgeGarden-Mobile	Devices & Edge Nodes	Thyme [24] & Basil [14]	Oregano [20]	Stored	Ephemeral	Local
Edge Nodes	GardenBed [23]	Devices, Edge Nodes & Cloud	GardenBed's caching system	—	Cached	Ephemeral	Local & Global
Cloud	GardenBed Cloud	Edge Nodes	Any database	—	Stored	Persistent	Global

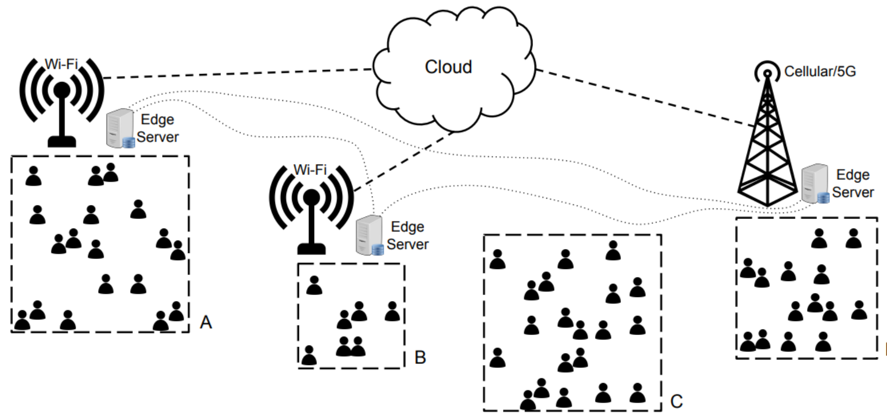

Figure 2.1: EdgeGarden Architecture. Taken from [22].

As presented in Table 2.1, the devices execute the *EdgeGarden Mobile* software stack, which manages communication, topology, data location, data replication and data subscriptions, among others. Applications interact with EdgeGarden via 3 APIs. **Thyme** allows data replication between devices and the edge servers. **Basil**, built atop of Thyme, to enable data persistence. Lastly, **Oregano**, which also extends Thyme, to enable computation requests in the devices.

Edge servers run GardenBed [23], a service-based software that provides services for its clients. To ensemble GardenBed and its clients we give the name region. Currently, GardenBed offers three services: Caching to save data items, grant data dissemination inter-regions and help the devices with the data distribution in the intra-region; Cloud connection to allow cloud storage to persist data, allowing devices to disconnect from the system while keeping such data in the system, and help with replication across the globe in specific situations where it makes sense, such as data items which users or the system consider relevant to access any time and anywhere; Namespace manager that manages the namespaces currently available on the region (more details on subsection 2.1.3).

Meanwhile, the Cloud will be responsible for storing data in a global environment [14]. In this context, devices only communicate with other devices in the same region and with the closest edge server, while edge servers can communicate with devices, between themselves and with the cloud. Furthermore, the cloud will only communicate with the edge servers.

2.1.2 Caching Service

As stated in subsection 2.1.1, GardenBed offers a caching service that is periodically requesting the most popular data on its region. This service enables the system to provide data that might no longer be available on the mobile devices of its region, ensuring the persistence of relevant data, while also enabling, through a connection to other edge servers, the sharing of such popular data to users in different regions than that on which the data was published on. GardenBed's cache resorts on Adaptive Multipart Caching, a

namespace sensitive caching system. Instead of single cache, it works on multiple caches, each one for each namespace. This has came as a solution to some cases when, during a short period of time, the interest for a specific theme or, in this case, namespace is higher than normal, resulting in a clearance of what is considered important cached data on regular times. By supporting these separate caches and establishing a bottom and top limit to the size of each namespace's cache, the EdgeGarden environment becomes more adaptive to exceptional situations, keeping always cached the most popular data in each namespace. [23] The Caching Service brings this aspect to the edge server's abstraction.

2.1.3 APIs

To further detail how the EdgeGarden ecosystem works, in this section we delve into the APIs offered to the mobile applications.

2.1.3.1 Thyme

Thyme [24, 25] is a time-aware data sharing framework to store data within the edge computing paradigm. It works with storage combined with a topic-based Publish/Subscribe (P/S) interface and makes use of a Cluster-Based Distributed Hash Table (CB-DHT), as detailed in [22].

Time-awareness As traditional storage systems function under an expected request, users have to do an explicit request to be aware of whether or not it is available for consume. On the counterpart, P/S systems typically work by disseminating data as it is published for online users, not persisting it, leading to the negation of access on the offline users. Thyme, combining both systems, solves both problems, allowing users to be notified as soon as they are online, allowing them to request the data item as long as the source device or any replica's holder are still in the service. They can even search for data that has been published prior to their entrance on the system.

Topic-based P/S Thyme exposes a topic-based P/S API [10], with which subscribers express their interest on a certain topic, referred to by *tag*, by subscribing to it. Meanwhile, publishers disseminate their content (under a certain set of *tags*) which will notify their subscribers. The subscribers can, then, choose, whether or not to download the given data item.

The operations Thyme's framework offers operations such as insert data (or **publish** in the P/S paradigm) which adds a new data item with respective *tags* and description, delete data (or **unpublish**) which deletes the item and makes it unavailable for further subscribers, query data (or **subscribe**), which returns all the data items' metadata with the *tag* requested and activates your subscription, making you receive notifications on

new publications with the queried *tag*. Furthermore, it is possible to **unsubscribe** a previously subscribed *tag*.

Cluster-Based Distributed Hash Table The data items published in Thyme are kept in the source nodes. so, these, upon a publish operation, become a source for the published item in the system. Moreover, for load-balancing and efficiency purposes, whenever a device downloads a data item, it automatically becomes a new source (of such item) for any future download operations. The data is hence distributed and replicated among multiple devices, and a mechanism to trace its whereabouts is needed.

For that purpose, Thyme makes use of a CB-DHT. The mobile devices are clustered into groups and each group will be responsible for two distributed tasks:

1. Store information of **where** is the data located in the CB-DHT, acting as a distributed broker.

2. Actively replicate the **data** along the system's devices;

On item 1, the tags of newly published objects are hashed. Each cluster of the CB-DHT is responsible for dealing with a set of hashes. Whenever an item, a cluster is responsible for, is published, this cluster is responsible for notifying all the subscribers of such hashed tag. Then, the notified devices are the ones responsible for retrieving the data if desired. Furthermore, each cluster of the CB-DHT is also responsible for managing the metadata of the publications and the subscriptions. For the publications, they need to store the information on who published them and whose devices hold their replicas. Meanwhile, for the subscriptions, they need to store the information on who are the subscribers of each *tag*.

Regarding item 2, there are cases when it is desired to keep a minimum number of replicas. In this case, it is possible to set up active replication in the configurations of the API. Whenever this happens, a minimum number of replicas is defined, and those will be stored in the devices belonging to the same cluster as the publisher.

At the edge Running Thyme on the devices enables them to communicate with themselves on an intra-region environment. It is not always possible to communicate with devices on other regions by simply running Thyme on them. To ensure inter-region communication it is recurred to the edge computing paradigm, supplied by running GardenBed [23] on the edge servers. The servers will, then retrieve the devices' published data and cache it, helping in the data dissemination within the region and sharing it with other edge servers, making it available for devices in other regions. The programmer can also decide, through filters, what data gets to be shared between regions. With the inclusion of GardenBed in the distributed system, data persistence is, then, a feature.

2.1.3.2 Basil

The Basil [14] framework is built on top of Thyme, which means that it inherits all Thyme's previous features, including time-awareness and communication with GardenBed, adding a key-value store abstraction to the system. It also enables communication with the cloud, recurring to the use of database, it allows the use of cloud persistence to the most popular data items. The API is ahead explained.

The operations To insert new data, the **put** operation is available which takes a set of keys, a data item, a description, and a handler as arguments. Such method will call Thyme's publish method.

The querying is made through the **get** method which will take a query item and a handler. The query has a set of keys and, possibly, some additional filters, which describe the data item one wants to retrieve.

Basil introduces a new method to enable the publisher of an object the addition and removal of keys from such published objects, **link** and **unlink**, respectively. These operations are disseminated throughout the network to grant that they are effectively updated in all peers of the system. The link operation also notifies subscribers of the added key of such object.

It's possible to call the method **remove** to remove a set of objects published under the same key, or, simply, a pair key/object. Only the publisher of such objects can remove the contents from the system. It's possible through a Boolean value, to define if the objects should really be deleted from the system, or only the link between the key and the object. The latter will remove such key (*tag* on Thyme's abstraction) from the set of keys of the data item.

Meanwhile, the methods **subscribe** and **unsubscribe** work similarly to what would happen in the Thyme environment, given a specified key and a set of filters.

Scopes of interest With the introduction of the cloud in the general environment, the most popular **local** objects can now become available globally, integrating in a scope of **global** interest.

When an object is published under the **global** relevance flag, this one will be sent to the device linked edge server which will be responsible to communicate with the cloud to store the data item in the database for global access. If an object is published under such circumstances and one wishes to retrieve it, it is possible to do so specifying this desire. Having the object stored in the cloud, grants its persistence, since it is no longer dependent on the devices which can disconnect any time from the system.

2.1.3.3 Oregano

Oregano [20] is a framework that serves as an extension of the Thyme one, complementing it with Subscription with Computation (SubC), which is, basically, a request which

involves computation on the source of data (the mobile device).

Subscription with Computation When a SubC is issued in Oregano, this will trigger two subscriptions in Thyme.

1. One to subscribe the item with a specific tag, providing a specific computation input;
2. The other to subscribe to the computation's result.

The subscription performed on item 1 will ensure the devices holding such tag will do the requested computation in the Oregano environment. Then, when the computation is finished, the source devices will *publish* the result with a specific tag only subscribed by the original requester, which, as it has already subscribed to it, will receive the notification informing the computation was finished. The following behavior proceeds as it does in Thyme.

To grant this does not trigger the same computation on multiple devices, when the result expected is one, Oregano has developed the concept of a Mobile Dynamic Dataset (MDD). It is simply a data set replicated through multiple devices. In Oregano, the MDDs hold all the pictures with a specific tag. So, back to the previous example, when the request for the tag 'Player' is performed, the MDD holding the tag 'Player' will be notified that a computation to recognize a specific face was requested. A device which holds a replica from this MDD will be assigned to perform IR with Ronaldo's picture which is in the publication's description.

In this paper, the goal is to allow an IR search on pictures. For example, taking in consideration the football stadium example presented in [24, 23], imagine one is sitting on the backbenches of a stadium watching a match between Portugal and Argentina. Those people do not have a great view on the players. But, using Thyme and, more specifically, Oregano, we could search for the face of a specific player, let us say, Cristiano Ronaldo. Those who are sitting next to the pitch can easily take a close-up picture of Ronaldo and share it through Oregano. Then, someone watching the match from far behind can make a search with a picture of Cristiano and the tag "*Player*". Oregano will enforce the source devices (the ones which hold the pictures with the tag "*Player*") to perform computation to run an IR operation which will tell if such player in the picture is Ronaldo or not. If the result is positive, the requester will be informed and can choose whether or not to download the picture.

Although Oregano has developed a concept that brings the computation away from the Edge Servers, it has some inconveniences. As some tags may be more popular than others, computation will be more concentrated in the devices of those who published the data. In the previous example, the football fans who are sitting closer to Ronaldo are more likely to have more pictures of him. Supposing Ronaldo is the most popular football player in the match, this cluster of devices will be excessively loaded with IR requests. To

address this problem, a face cluster indexing mechanism would prove beneficial. This will be approached further ahead.

2.1.4 Chives

In the context of this dissertation, we aim to achieve an application that allows users to search photos based on tags. Sometimes, users may want to search in the dataset for photos that contain a specific face giving a photo of such face as an input. Having the necessary tools to implement the communication and storage, now it is also necessary to have a way to retrieve faces from photos. We will work on a previously developed but unfinished framework called Chives. The state-of-the-art Chives was built with abstraction in order to be integrated with Thyme, Oregano and GardenBed [16]. Chives already clusters faces on images received through a stream and returns those clusters who have met a certain condition. By default, such condition is *clusters with more than 10 projections* but the programmer can define one that serves his purpose.

Furthermore, Chives works in three stages:

Face detection The face detection algorithm, in Chives, receives an image as input and detects every faces in it. This is done using OpenCV's model *HaarCascade FrontalFace Alternative Tree CUDA*. After detecting the faces of an image, they are extracted into a new set of shorter images. This is useful for the next step, the feature extraction.

Feature extraction To perform the feature extraction, Chives, makes use of the *Principal Component Analysis* model and two *Linear Discriminant analysis* models which receive *eigenvalues* and *eigenvectors* generated by previously trained face recognition algorithms: EigenFaces and FisherFaces. Such models generate a set of coordinates representing the features of the previously extracted faces. These will be used further for euclidean evaluation and comparison.

Clustering Making use of the coordinates generated, each feature can be represented as a point in a plane. The difference between two points can be measured through euclidean distance. The similar the features between two faces, the similar the face is considered to be. Faces which features' differences are small enough are considered the same face and are clustered together as such.

To perform such operations, Chives provides the following tools:

1. **OpenCV Face Extractor** is used to detect faces in a photo and extract them;

2. **ArcFace Feature Extractor** allows the faces' features extraction, which are represented by double-point vectors.

3. **OpenCV Feature Extractor**, another option for the feature extraction. As ArcFace's it also outputs double-pointed vectors.

4. **DBScan Clusterer**, as the name points out, clusters the previous extracted features. Each cluster represents a person's face.

5. **MOA Clusterer** is another possibility to face clustering.

6. The **Framework** itself defining what is a cluster, centroids, points... And providing interfaces: Face Extractor, Feature Extractor, Clusterer...

Goal The state-of-the-art method to search on images by face is done through Oregano's Subscription with Computation. The user that wants to search for photos with a certain face on them would trigger Subscription with Computation on all the most popular photos and respective holding phones. However, as we pointed out in Oregano, a major problem would be a lot of intensive and costly computation on the devices holding the most popular pictures. If there was an index of clusters as keys to the most popular pictures, we could avoid a lot of computation on the holding devices, solving the problem we have.

One of the goals in this dissertation is to have these indexes stored as CRDTs in order to enable them to be accessed throughout the devices and edge nodes and asynchronously mutated. To allow this goal to be fulfilled, it is needed to understand how the distributed systems work and how this access and manipulation is possible.

2.2 Mutable Data in EdgeGarden

A traditional non-distributed system (i.e. a computer, a mobile phone) works by storing data, in its disk, with unique locations. Access to such data is, then, done through simply accessing such locations. This makes it easy for such system and its programs to change and manipulate data. It is possible to run basic applications such as a scientific calculator or a camera with its photo gallery.

Although, nowadays, most applications require interaction between multiple devices, the, so called, distributed systems, in order to enable more complex activities (i.e. chatting, multiplayer gaming, photo-sharing...). Data in such systems cannot be accessed through a single location, since there are multiple devices trying to do so. Some kind of data replication is required in order for it to be accessed by these devices.

There are multiple ways to program this. Ideally one would want a program to look at the system as if it was an offline system with unique locations for its variables. Having to update each variable only once with the variable being geographically replicated in real-time is impossible. So, when creating a distributed system, the programmer has to sacrifice either the response time or the consistency of its variables. It is possible, then, to classify two different major types of systems, those with a Strong Consistency (SC) but

slower in latency, and those with a weaker consistency but faster in latency, the Weak Consistency (WC) systems [6, 18].

Strong Consistency In SC, a system will have to synchronously update each one of its replicas with its most recent update. This means that every variable in each replica will have the updated value, meaning that accessing the variable will always return the real value of the variable independently of what replica is the user connected to. The problem is that updating the variable in every replica each time you want to update a variable can overload the connection with requests, making the transition between operations slower than its consistency counterpart.

2.2.1 Weak Consistency

In WC systems, a variable might not have the most updated value when accessing it. This happens because the variable is replicated to all replicas of the system asynchronously, allowing the user to keep performing operations on the system even if there are some variables that are still outdated. This improves the latency of the system. While sacrificing consistency, the user can access the system's variables in a faster way, since he does not have to wait for it to update all its variables.

2.2.2 Eventual Consistency

There are multiple degrees of WC, of which causal and eventual are the most wellknown. Given that the EdgeGarden makes use of CRDTs, which guarantee eventual consistency, this dissertation will focus on this model. Eventual Consistency (EC), as the name implies, grants that, eventually, consistency will be achieved. This is achieved defining Eventual Visibility, which states that whenever a completed operation e is executed, it will be eventually visible in all replicas [6]. Also, whenever an operation is performed after the operation e, it will ensure that this operation will, in almost all replicas of the system, be executed after the operation e has become visible.

Although the idea of EC may seem simple, its implementation requires definition. EC as it is, does not grant a set of properties that seems required from the user's point of view. To enforce the right behavior of a system in such point of view, the user must witness Causal Consistency, which is a combination of causal arbitration and causal visibility and is implied by this set of properties, which is as follows:

Read My Writes When a message is posted, it must be visible.

Monotonic Reads If something earlier have been already read, performing a post should display more messages not fewer.

No Circular Causality One should not be able to see something that has not happened yet.

Causal Arbitration If an event of a session has happened before the event of another session it may imply a causality, so it should be in the correct order (e.g time).

Causal Visibility If an event of a session has happened after the event of another session, and it is observable, the earlier event must be observable too.

There is also another property enforceable to grant an even stronger combination of properties:

Consistent Prefix If the operations of a different session are observable, all the previous operations, in the correct order (e.g. time), must be observable too.

In this dissertation, thought, the focus is centered on replication of clusters using CRDTs, which are not visualized by the user of the application. Therefore, such focus should be on Strong Eventual Consistency (SEC), the type of consistency required by the data types. To achieve EC these three principles must be fulfilled:

Eventual delivery All correct replicas will eventually receive the same update.

Convergence All correct replicas that have delivered the same updates will eventually reach the same state

Termination All method executions terminate

This causes problems in some systems, since when a replica receives an update and executes it immediately it may clash with another unreceived update. To solve this issue, SEC was defined as EC, but instead of **Convergence**, it rather has **Strong Convergence**, defined as follows:

Strong Convergence All correct replicas that have delivered the same updates *have* the same state

To apply all criteria stated, Shapiro et al. have defined CRDT [21].

2.2.3 Conflict-free Replicated Data Types

A CRDT, is a data type that can be updated asynchronously, therefore, replicas do not require coordination updating values, and, later, corrected between replicas, granting that two replicas receiving the same set of updates would reach the same state [18]. This happens following these properties:

1. Every replica is updated, disregarding other replicas;

2. When two replicas converge, they use a set of mathematical formulas to ensure they hold the same final updated result.

This is used in WC systems as an alternative to SC systems [18]. SC systems wait for all replicas to hold the same value in their changed variables until they can proceed with the next operation. These systems often have higher latency and, as an alternative, lower latency systems using WC can make use of CRDTs when possible [6, 18].

The nature of CRDTs allow the systems to keep running, even if they are holding an outdated value, without risking to enter in inconsistency and making it impossible to fix the value.

A CRDT can be of two main types: *Operation-Based CRDT (O-CRDT)* and *State-Based CRDT (S-CRDT)*. Some other types of CRDTs have been defined and developed over the years, such as: *Pure-Operation-Based CRDT (PO-CRDT)* and *Delta-Based CRDT (Δ-CRDT)*. Also some other methods of implementing the CRDT technology have also become part of the state of the art, like the *Publish-Subscribe CRDT (PS-CRDT)*. All of these types will be addressed and discussed so a definitive final option can be achieved. This essay will discuss all of these alternatives.

O-CRDT O-CRDTs [21]. have two phases of operation: *prepare* and *effect*. Whenever an operation is executed locally, it stores the operation and when the replication happens, this edge node will send its list of performed operations and receive the list of performed operations from other replicas, order them, following the logical clock, and execute the operations, resulting in a final mutual value for all replicas of the system. An example of an O-CRDT is the Observed-Remove Set (OR-Set) [18] . The OR-Set works by supporting *add* and *remove* operations that, internally, depend on their causal history. The key characteristic of this O-CRDT is attributing each *added* element of the set an unique *tag*, one that is invisible outside of the interface. When a replica receives a *remove* message on an *added* element which *tag* is not known by such replica, it proceeds to ignore any possible *add* operations for such *tag*.

S-CRDT S-CRDTs [21] work in a different way, they execute the operation locally and them propagate their version of the state to the other replicas. All the replicas will, then, determine a common value using a *merge* function. An example of a common S-CRDT is a boolean value that registers whether a certain event has occurred at least once or not. This variable would be *false* if the event never happened or *true* if the event has happened at least once. When the replication is done in WC, it's easy to understand which will be the value of this boolean variable. Suppose two replicas as an example. If in both replicas the value is *false* (the event never happened in either replica) or *true* (the event happened in both replicas or has been already replicated), there will not be a problem. In the case of an inconsistency, the real value of this variable is *true*. This is due to the nature of this variable. If the value is *true* in at least one replica, it means that the event has happened at least once, since there's a replica that holds the value *true* in its variable. So a replication is needed, the *true* value should be replicated to the replica that still holds the value *false*.

PO-CRDT O-CRDTs function relying on the *prepare* and *effect* operations. However, during the *prepare* phase, parts of the state of a CRDT are included in the message for dissemination. PO-CRDTs [3] are a type of O-CRDTs that aim to make O-CRDTs solely operation-based, straying away from S-CRDTs by only transmitting the operation and potential arguments during replication. They resort to what is defined as a partially ordered set of operations (PO-Log) [2] and are able to implement O-CRDTs in a way that reduces delay in sending messages [18] while making the message sizes longer [18] but still relatively short [2].

Δ-CRDT Furthermore, Δ-CRDTs [1, 12, 18] are a type of S-CRDT which do not propagate the entire state but rather propagate only the changes (deltas) to the state since last synchronization. This reduces the amount of information to be send in a S-CRDT, making it a more efficient and less costly method.

2.2.4 Publish/Subscribe CRDTs

CRDTs are an approach to deal with objects' synchronization on EC systems. They do, however, lack efficiency in the context of mobile environments. This is due to the fact that when a CRDT needs to update or send its updated version, most solutions require each node to have information about all other nodes of the system, which can present some issues in efficiency and become very costly in bandwidth. Barreto et al. have, in response to this problem, came up with a PS-CRDT [4] alternative to CRDT. This solution promises to upgrade the performance of CRDTs in some specific environments such as the mobile one, the one this essay has interest on. Instead of sending and requesting updates to all edge nodes, PS-CRDTs' approach is based on the P/S abstraction. A PS-CRDT replica publishes its updates and the interested replicas are notified that there are updates to be retrieved. Upon receiving the notification, replicas have to contact the publisher in order to retrieve such updates. These notifications are performed on the broker of the system, which can be centralized or distributed. The PS-CRDT model implies that the interest of replicas in some updates is defined as topic-based, as it was explained in section 2.1.3.1. It has been able to implement S-CRDTs, O-CRDTs and Δ-CRDTs in the P/S paradigm [4].

2.3 Indexing at the Edge

Using an indexing mechanism at the edge to store the index of data items might sound like an obvious task, but requires further explanation. It can be made through a lot of diverse approaches with its own properties, advantages and disadvantages. Some of the state of the art indexing mechanisms will be discussed and compared to what this essay aims to achieve.

It is now possible to select a set of properties to classify these multiple approaches, which will be useful to analyze them. On Table 2.2, the already discussed works are

Table 2.2: Indexing at the Edge in the State of the Art

	Location	Goal	Data	Where	Access
COIN [27]	Along the Edge Servers	Reduce the number of hops between switches when retrieving a data item	Unstructured Data	Edge Servers	Through coordinates representing the location of each one of the data items
CooLSM [13]	Cloud	Separate write and compaction operations	Any data	Edge Servers (the Ingestors) and Cloud (the Compactors)	Through the Reader (closer to data sinks)
Cachier [7]	Cloud	Cache IR models	Parts of IR Models	Edge Servers	Simply, through the closest Edge Node
HDS [9]	Cloud and along the Edge Servers	Decompose the Edge Computing in three hierarchic levels	Unstructured Data	Both in the Edge Servers and in the Region Data Centers (DCs)	Using *Cuckoo Summary* for intra-region indexed items and *COordinate-based INdexing (COIN)* for inter-region indexed items
Helios [17]	Cloud	Divide the indexing scheme in two partitions, making use of both hashing (or other methods) and tree structured schemes	Pointers to Cloud Location	Edge Servers	Through the Edge Servers
Our proposal	Devices	Optimize queries over images stored in the devices	Faces in images	Edge (Clustering over a stream of images)	Local access: PS-CRDTs

summarized according to the following properties:

Location The location of where the data is. It can be on the Cloud, in the Edge Servers or on the Devices that use the system.

Goal The goal of indexing in each project. It describes the method or goal this approach uses in indexing.

Data The type of data that is being indexed. It classifies the type or content of the data. It can vary from type to content from approach to approach, depending on what the main goal of the proposal is.

Where The location where is the index computed. Each approach uses different ways on what computes the index. The majority of approaches compute it on the Edge, but some do rather compute it on the Cloud.

How How is the index computed. Explaining or labeling the process of computing the index.

Access How to access these indexes. Most approaches access the index simply by issuing a request to the nearest Edge Server, so, in such cases, this tries to explain what happens when this access is made.

2.3.1 The Approaches

COIN COordinate-based INdexing (COIN) [27] is an unstructured data indexing alternative to previous indexing mechanisms in the Edge Computing paradigm, made as an extension of the Distributed Hash Table (DHT). It strays away from this dissertation's proposal by having the data spread along the Edge Servers instead of having it stored on the users' devices and, unlike EdgeGarden that uses Conflict-Free Replicated Data Types (CRDTs), it requires a request to the nearest Edge Server whenever an access to the index is desired.

In COIN, each switch on the edge network is identified by a coordinate (x, y) (with $x, y \in [0; 1]$) and all the data items are spread along the edge network, accessible through a coordinate. To access, efficiently, a data item, one would only have to know the coordinate of the switch which its storing Edge Node is connected to. COIN achieves this by making use of the last 8 bytes of a synthesis hash function of the data identifier. From these bytes it's possible to generate a pair of two decimal numbers (of 4 bytes each), representing the coordinate. Then, if the coordinates where the data item is stored are the same coordinates as the switch to which the node is connected to, the Edge Node simply returns the data item, or, if not it must check its neighbors' coordinates. If any of the neighbors have the coordinates in question, the Edge Node requests the data item and returns it to the user. If the data item is not in the Edge Node or in the Edge Nodes of its neighbors, an Euclidean distance function is called to determine which of the Edge Node's neighbors should be closer to the Edge Node that owns the data item and send it a *virtual link*, starting a *greedy forwarding* process. After choosing the forwarding switch, this one will also check its neighbors and proceed to send the request or the *virtual link* to the desired switch, similarly to what was done before.

To generate the 2D virtual plane representing the switches' coordinates, a control plane is needed. The control plane consists of one or more components and is responsible for managing the Edge Network. The index is computed by the Edge Nodes themselves, as stated before, which know their own coordinates and those of their neighbors, and are responsible for receiving and dealing with the requests for the **write** and **read** operations.

Following this algorithm, Xie et al. claim to use 59% shorter path length and 30% less forwarding table entries in their mechanism, being a more efficient alternative to traditional DHT.

CooLSM Cooperative Log-Structured Merge tree (CooLSM) [13] is an Edge Computing approach to indexing. Unlike EdgeGarden, it decomposes a Log-Structured Merge Tree or LSM Tree [15], distributing its components along the edge system. Like COIN, it also requires an access to the nearest Edge Server to retrieve or insert indexed data items, differently from the simple device stored items access. To understand the work-flow of the CooLSM it's necessary, first, to understand the concept of a LSM Tree.

A LSM Tree is divided by levels named L_0, L_1, L_2, ... L_k. L_0 resides in memory, while

the remaining levels reside on the disk. Each level is composed of multiple tables, also called *sstables*. L_0 also contains an additional table called *memtable*.

Whenever a **write** operation is performed, the LSM Tree will acess the *memtable* and perform such operation. This *memtable* will then be appended in the L_0 set of *sstables*. If the L_0 reaches full capacity (the constant defining maximum number of *sstables* in a level). This will trigger the compaction operation.

The compaction operation consists in merging all of the *sstables* of a level, and appending the merged table, a new *sstable*, to the set of *sstables* of the next level. Merging must prioritize the last written value (the value that appears on the last appended *sstable*), in the case where different *sstables* have the same key. In the case of L_0, its *sstables* will be merged and appended as a new *sstable* to the set of tables of L_1. In the case of L_1 reaching the full capacity of *sstables*, this will trigger a compaction operation between L_1 and L_2, and so on.

To **read** values from this LSM Tree, the *memtable* will, firstly, be consulted and its value returned. In the case where the value is non-existent, the *sstables* of L_0 will be iterated until finding and returning the requested key. If this level does not hold the value for such key, the next level will be consulted, and so on, until reaching the the final level, L_k. If this key was not found, then *null* is returned, informing the user the value for such key is nonexistent.

In CooLSM, this concept is decomposed in multiple machines, to perform a more efficient division of tasks and adapt it to the edge computing paradigm. There are three main components: the Ingestor(s), the Compactor(s), and the Reader (which also works as a backup).

The levels of the LSM tree are now divided between the Ingestor(s) and the Compactor(s). The Ingestors hold identical L_0s and L_1s, while the Compactors hold the remaining levels (L_2 and L_3). When the user stores new data, this operation is performed on the Ingestor(s) which will add the new key-value pair to its *memtable*, and proceed as the LSM trees do. The compaction operations work similarly to the ones in the LSM trees, except when the L_1, in a Ingestor, is about to exceed its threshold, its *sstables* are compacted and inserted in the L_2 of the Compactor(s). The Reader is responsible for making regular replicas of all the levels in the Ingestor(s) and Compactor(s). It is on the Reader that the user should perform the **read** operations, avoiding to overload both the the remaining components, which are busy with the **writing** and **compacting** operations. As mentioned before, the Reader also behaves as a backup, in the event of an anomaly causing the remaining levels to lose their data.

The Ingestor(s) should be placed at the edge, to allow fast interactions with the user, while the Compactor(s) work on the cloud, keeping the heavy compacting operations away from the user. The Reader(s) should be placed close to data sinks and data consumers to enable high reading availability.

HDS Another indexing mechanism, HDS [9], is able to achieve fast data localization by putting in hierarchy the Edge Computing paradigm and combining the use of either a protocol called Cuckoo Summary and COIN. This strays away from EdgeGarden by not only storing the data along the Edge Servers and some Data Centers (DCs), but by making the index accessible through the Edge Servers. Basically, instead of simply having a DC and a set of Edge Servers, HDS has three different hierarchic levels: the **Cloud DC**, where all the data is stored, a set of different **Regional DCs**, containing both a *Cuckoo Summary* (an intra-region index) and a partial global index, and the traditional **Edge Servers**, containing an index of the cached data items.

This mechanism indexes all the data items from the main DC along the different Edge Servers. Each Edge Server is attributed to Region DC. When a user performs a request to its closest Edge Server, it will immediately return the cached data. If it does not have the requested data, the request will be forwarded to the corresponding Region DC, which will check its global index. If the global index does not have the cached data, the Region DC checks the Cuckoo Summary, a summary of all the cached data in the region, built with a Cuckoo hash table. The hash table is composed by pairs of key and identifier of the data holding edge server. This summary is achieved by having all the Edge Servers from each region to send its index to the Region DC. If any of the Edge Servers from the region has the data, then it is retrieved. Otherwise an inter-region request is needed. This one uses the already listed mechanism, COIN. Except, instead of implementing COIN on the Edge Servers, the mechanism, in HDS, is implemented on the Region DCs. Each Region DC is attributed a coordinate, paralleled to what happens in the original COIN algorithm. The rest of the algorithm, occurs exactly like COIN.

Using this approach, HDS is able stray away even further the index from the cloud than traditional indexing mechanisms. Making sure the data index is either in the requested edge server or one hop away from it (either on the same region or not) it is able to fastly find the requested data.

Helios Helios [17] is the mechanism used by Microsoft to perform data operations (e.g. aggregations), before inserting it in the Azure Data Lake (the cloud system they use to store data). This approach strays far away from this essay's by storing the data in the cloud, and the index being accessed through the Edge Servers, similarly to what previous approaches did. The indexing algorithm works the following way. The index is divided into two partitions. On the first partition, it is used a hashing function on the key of the object being indexed. The objects are then divided by hashed keys. If the object has no key, a round-robin scheme is used. The new approach Potharaju et al. introduces contains, also, other methods to divide the indexed data. On the second partition, for each reliability zone, the data is indexed on a tree data structure designed for the system. This data structure is inspired by several state-of-the-art ones, such as: the B-Tree, the already explained LSM-Tree and Merkle Tree. This index contains the key of the object and a pointer for the location where the object is stored in the cloud.

Cachier Cachier [7] is an approach to Image Recognition (IR) applications that aims to minimize IR latency in such systems. By combining edge computing, caching and IR trained models, Cachier allows mobile users a faster recognition retrieval on trained models by caching these models in edge servers. It is not an indexing mechanism, since it does not use an index to store its data, and rather uses a Cache to do so. But, it was decided to include this, since it also deals with IR and Edge Computing, as in this essay's approach. The concept is simple. If one is in a cinema, the probability of wanting to recognize a poster of a movie is high in that place or, if one is in a museum, the probability of wanting to recognize piece of art from that museum is also high, hence the use of a cache on the edge node of such places is, theoretically, efficient. To understand how Cachier has came up with its settings for the most efficient approach to its theory, one must understand how the cache mechanism works.

A cache is useful because it takes the most popular requests to a cloud closer to the user (e.g. in its device, on an edge node). It is as effective in reducing latency in cached requests as smaller it is, since it does not have to iterate the full set of data. A cache that would have all the possible retrievable items in it, would be as slow as the cloud performing the retrieving operation. In a parallel idea, a cache with a single item in it would not be as effective, since one would only be able to use it on this single item. So, a cache is expected to take only the most popular retrieved items by such user. When a request is done, and the item requested belongs in the cache, the operation is labeled as a *cache hit*, when the opposite happens, it is called a *cache miss*.

Cachier uses this concept, along with the previously stated theory of regional popularity, to implement its methodology. The cache size is as big as the number of relevant objects from that model and its value is calculated through a set of mathematical formulas that leads it to the one which would reduce latency the most, but also keeping a good amount of retrievable items. IR models are cached on the edge nodes based on retrieving popularity. A *cache hit* probability is, so, the probability of an item being requested in such region, making this whole system very geographically friendly, and enabling costly recognition operations stray away from the cloud. The downside of this system occurs when a *cache miss* does. Since not all objects of the model are cached in each edge node, the cached model would not be able to differentiate whenever an object is unidentified in the actual model, or simply does not belong to the set of cached objects. When this happens, the edge node forwards the retrieval operation to the cloud, identifying the desired non-cached object or not, if the object is unidentified within the model.

2.3.2 Discussion

On the location where data is stored, it seems to diverge given on the approach each author took. Indexing mechanisms like COIN and Cachier store their data on the edge nodes, while some other works have rather stored it on their cloud systems, like HDS. A common goal seems to be set on the state of the art works: retrieve data efficiently, but

some sub-goals or constraints are set for each of these approaches. HDS aims to access geographically distributed data with their index, having it distributed along the different regions' DCs, recurring to Cuckoo Summary, a protocol to achieve fast data localization in intra-region.

Xie et al. have proposed COIN, where there's an edge node responsible for computing the index, the Indexing Edge Server. So when accessing the index, the Edge Servers resend the requests to this Indexing Edge Server or request its Distributed Indexing Table, which is distributed in multiple Edge Servers and contains mapped data for their locations on the Edge Network [27]. Cachier is an alternative that caches Image Models for recognition from Image Recognition Systems. It stores the cache on the edge and, recurring to a Distribution Estimator module, tracks the number of times an object is requested, estimating the request distribution and the probability of a cache hit, relying on Probability Theory [7].

3

Developed Solution

3.1 Solution Overview

For our dissertation we have developed a mechanism in an Edge Computing environment that allows the users of such environment to search for photos in their region of mobile devices giving only a photo of a face as an example. The purpose of this development is allowing such users, providing a photo of their face, or the face of another person, to receive the most popular photos published on the system containing the given face. As shown in Figure 3.1, a single region of our ecosystem is composed by an Edge Server and a group of mobile devices in its surroundings. The general idea is narrowed down in a few steps. In Figure 3.2, the first steps of the interaction are shown: the edge server retrieves the most popular photos to the mobile phones, processes and creates an index of faces, and shares the index with the smartphones. In Figure 3.3, the user provides a photo of a face to the mobile phone, the latter will search on the face index for the existence of such face, and if it does exist, replies with the photos of such person.

We have, so, leveraged Chives, a framework built on EdgeGarden, with the ability to

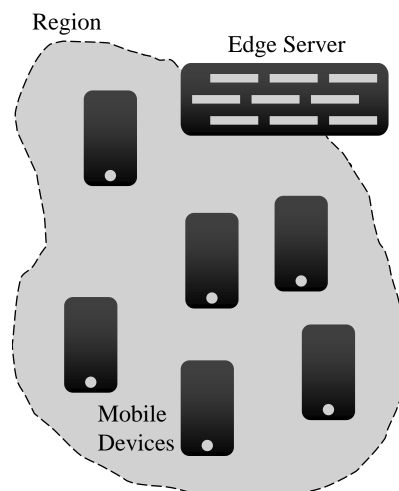

Figure 3.1: Architecture of the System in a Single Region

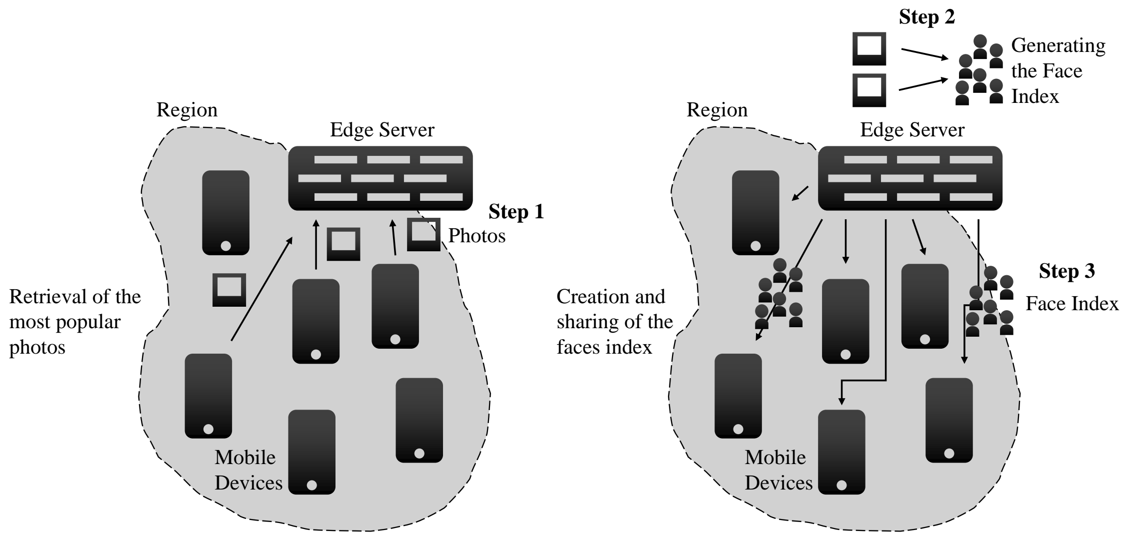

Figure 3.2: Creating and Sharing an Index of Faces

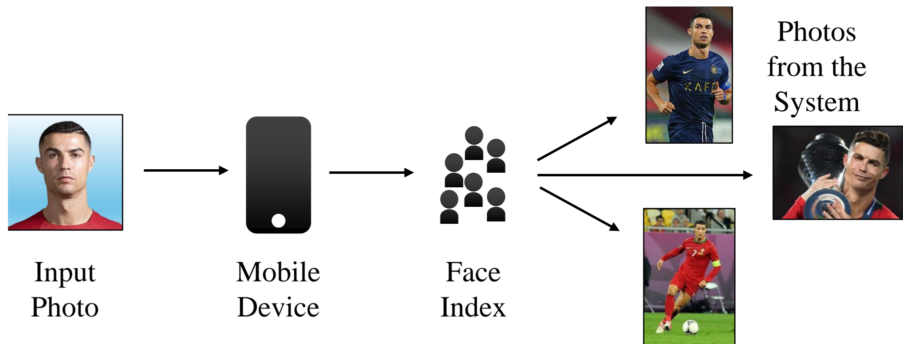

Figure 3.3: Searching for Photos Given a Photo of a Face

index data based on their content and search for such data with content examples. The EdgeGarden environment was already equipped with tools that allows searching for data based on topics. Chives intends to add to such environment the ability to search for data based on its content. Taking photos as an example, Chives would enable the user to search for photos, not only by their topic, but by the content of the photo. If a photo depicts Cristiano Ronaldo next to his birthday cake, we can describe such photo by saying *This is a photo of Cristiano Ronaldo, next to his birthday cake*. The main contents would be *Cristiano Ronaldo* and *birthday cake*. Chives allows the user to provide an example photo as an input. If one wanted to search photos with birthday cakes, providing a photo of a birthday cake as input, Chives would retrieve this photo. Another query could be with a photo of Cristiano Ronaldo, such would output this photo as well. Its implementation is run both at the edge and in a mobile phone.

Chives runs as two components with different purposes:

Edge Server On the edge, Chives is responsible to create a content-based index on a set of data;

Mobile Device On the mobile endpoint, Chives-Mobile is responsible to search for data in this index which content is similar to a provided input.

On the Edge, Chives is built on top of GardenBed, presented in subsection 2.1.1, which provides a set of services to the EdgeGarden ecosystem. The key services in the context of this dissertation are the Caching service, responsible to cache the most popular data in the region, described in subsection 2.1.2, the Publish/Subscribe (P/S) service, as the name states enables publication and subscription on behalf of the edge server, and the Face Indexing service, which deals with the creation and sharing of the index of faces, as shown in Figure 3.4.

On the counterpart, on the Mobile version, Figure 3.11, Chives is built on top of Basil, presented in subsubsection 2.1.3.2.

Although Chives is able to deal any kind of content, in the context of this dissertation, the focused content to deal with is human faces. It is important to notice that, we did not start the development of Chives from scratch, some parts of Chives were already implemented, those of which will be explained below.

What existed

At the beginning of our development, the Edge Server could request the most popular items and store them, through the Caching service. This would be useful in cases where a mobile device requested an item that was no longer available on other devices. If such item was popular enough, it would be cached at the edge for providance. GardenBed could not, however, publish or subscribe to data. On Figure 3.4, we can see the architecture of the edge server: gray parts symbolize the already implemented segments of the system while golden blocks are meant to represent new services, built during the development of this dissertation.

On the mobile side, users could post and retrieve data between themselves, and request missing data to the edge. Users could make use of many Application Programming Interfaces (APIs), introduced in subsection 2.1.3 and represented in Figure 3.11, to share data and apply computation on that data.

So far, Chives provided tools to deal with Face Extraction (FcE), the capability of, given a photo with multiple faces, extract those faces, each as singular ones; Feature Extraction (FtE), the ability to describe these faces features as double-point vectors, and Clustering, a method of grouping together close enough points. One could provide a stream of photos, and Chives would extract all the faces in those photos and create clusters of faces. Such clusters would be, no more less than, sets of the projections of faces, grouped by person. This state-of-the-art Chives served as starting point to our implementation.

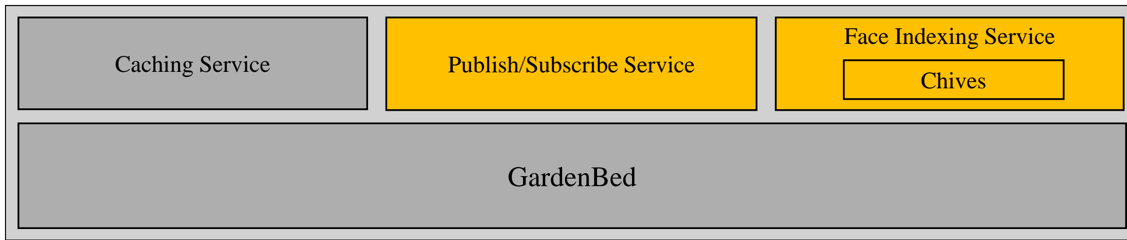

Figure 3.4: Frameworks Used in the Edge Server

What was the focus of this dissertation

The main purpose of our solution is that mobile phone users can search for published pictures that contain a specific face, making use of the P/S paradigm available in the context of EdgeGarden. For this have continued previous work done on Chives. Although such framework was well equipped with Facial Recognition (FR) tools, it still lacked the ability to index clusters, share them to the mobile phones, and, on the smartphones, receive the index and perform searching operations on it. We implemented two components of Chives, one on the Edge and another at the mobile side.

The component of Chives that runs at the edge creates an index of faces, passively iterates over a set of pictures, extracts their faces and creates clusters, adding each cluster to such index. This component was integrated in a new service created with the purpose to share this index with the mobile devices, which was named Face Indexing Service. However, the sharing of data, in our case, the index and its clusters, was not yet available as a service of GardenBed. For this purpose, we extended GardenBed with a new service, the Publish/Subscribe Service, which allowed the edge server to make their own publications and subscriptions of data they produced or required.

On the counterpart, the component running in the mobile devices will, firstly, subscribe to and download the cluster index. It will, then, reconstruct the index by subscribing to all the indexed clusters. Clusters information, such as the clusters' tags or the clusters' projections will also be requested, for this we made use of the already available Basil API (subsubsection 2.1.3.2). Finally, Chives-Mobile allows the user to provide a picture of the face of itself or another person, which will be used to search on the index for photos that contain such face, using some state-of-the-art parts of Chives (subsection 2.1.4), combined with newly implemented ones. The final result is a framework that allows the leveraging of the system by allowing users to search for photos with similar photos as input.

Below, we explain how we implemented the edge server's component of our solution.

3.2 Edge Services

The edge server is a very important part of the EdgeGarden environment. It works as a passive actor, providing services to the environment, that performs tasks requiring high performance capability, caching, or communication with other regions. As the users publish data items on EdgeGarden, GardenBed will download the most popular data. Such data will be sent to Chives' pipeline, which will allow the indexation of its content. This intends to facilitate user searches on data, without them really knowing what topics identify such data, or, in our context, to facilitate user searches on photos, in which an individual is present, using a photo of the face of this same person.

As explained in subsection 2.1.4, the state-of-the-art Chives provided an API to, given a set of pictures, detect and extract faces in such pictures, extract the features of these faces and cluster them. By concept, these clusters are groups of similar features. If two sets of features are similar enough, such would, theoretically, mean that those features were extracted from different photos of the same person. We can, then, say that each cluster is a computational representation of the face of a specific person.

In its final stage, Chives will provide a way to let the users of the system know if a certain person's face exist in any photo published by the users of the environment, and if the answer is positive, it will provide such photos to the user. In our solution, we leveraged the state-of-the-art Chives, on the edge, to be able to run through a stream of images, which are the most popular published photos on the system, and generate an index of faces detected in those images. Such index is shared, in real-time, with the mobile devices, with all of its clusters, our digital representation of human faces. As more images are read by Chives, more clusters are created and shared, or even updated, as clusters grow in size, with different new projections of the same face. Each cluster, when created is appended with a distinguished default tag, which will be used to identify the cluster in the system. To perform all of these tasks we have developed an Indexer inside of Chives, and a service that would be responsible to run these computations, while receiving input from GardenBed, we called it the Face Indexing Service. As we built our solution's edge component on top of GardenBed, we needed this framework to be able to publish data onto the devices, a functionality which had not yet been implemented. Only then, could, the Face Indexing Service, share the indexed clusters with the users on the mobile phones. For this purpose we have implemented a Publish/Subscribe service for GardenBed, which will allow the edge server to also be a creator and publisher of data for the environment. These services are depicted in Figure 3.4.

Further in this section, we will start to explain how the P/S paradigm works in our system in subsection 3.2.1, followed by the creation and publication of the index, subsection 3.2.2, with the integration in the Face Indexing Service in subsection 3.2.3.

3.2.1 Publish-Subscribe

As the state-of-the-art EdgeGarden environment worked already through the usage of P/S technologies, introduced as a communication and data transmission methodology between the mobile devices by Thyme, as referred in subsubsection 2.1.3.1, it seemed logic that the same abstraction were to be used by the edge itself as well, as a solution to the lack of a framework or service that allowed such server to share content generated by itself. The whole mechanism of data sharing between mobile nodes relied already in the usage of Publish-Subscribe CRDTs (PS-CRDTs) or, at least, normal data types shared through the P/S logic. Instead of requesting to each and every node for data on a specific topic, P/S allows one node of a distributed system to manifest its interest on a specific topic and to be notified by the holders of content on this topic. This methodology brings time and resource efficiency to data-sharing when compared to traditional data-sharing methods, as noted in subsection 2.2.4. Requiring the ability to participate in these interactions, on the edge node's behalf, we have developed a GardenBed service called **PublishSubscribeService**, which allows the edge node to publish data to the mobile nodes, and subscribe to data from such. To explain how this was achieved, we need to introduce some concepts and characteristics of the data that is transmitted through the regions.

Data Characteristics

The data shared in the EdgeGarden system can be of any type. There are no boundaries on size or type, apart from the boundaries of the owner's device itself. In order to identify such data, the EdgeGarden's framework's APIs rely on an unique object identifier, a Universally Unique Identifier (UUID), and a set of *tags*. The identifier is useful to address a specific object which identity is already known by the system, whereas the tags are mainly descriptive and distinguish data between topics but, in some cases, might identify uniquely an object as well. Two different objects will never have the same identifier, but may have the same set of tags, if none of those tags is purposely added to uniquely identify such object and by chance the circumstances allow it.

Publication

Data objects are sent through the nodes of the system inside of *publications*. These publications, introduced by Thyme, subsubsection 2.1.3.1, contain, not only the data item itself, but also the information it needs to travel along the environment. The *tags* and description of the data item, the information of whether or not it requires active replication and the scope of interest are a few of the properties of the *publication*. When a publication reaches a destination, this information is then saved in metadata appended to the data object.

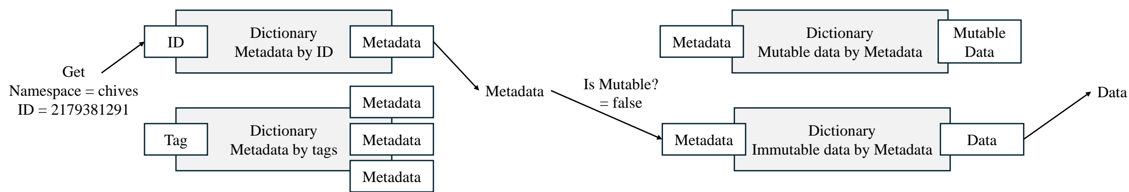

Figure 3.5: Retrieving Data by ID on the Publish Subscribe Service

Metadata

The metadata's purpose is carrying the identifying properties of the object, such as the tags and the identifier, but also describe the purpose of the sent message. They are responsible to tell the receiving subject which scope of interest (inter-region or intra-region) was this data sent for, a brief description of the contents, the address of the owner, the namespace, the location of the data in the edge computing environment, a flag indicating whether or not such data is mutable (a CRDT) or not (which purpose is important for our context and will be explained further ahead), and others.

Storage

Having these in mind, it was needed a way to store data, created or downloaded, in memory so it could be available to share, when requested. For this purpose, it is important to note that data can be either mutable or immutable. We developed a solution to Data Storage, relying on a multi-dictionary logic, using hash tables, that allowed a fast retrieval on request. Such are:

Metadata by ID Allows accessing the metadata of a specific item through its identifier;

Metadata by tags Allows accessing the metadata of one or more items through a tag;

Data by metadata Allows accessing a data item through its metadata;

Mutable Data by metadata Allows accessing a mutable data item through its metadata.

By setting up these dictionaries, we could retrieve any data, being it mutable or immutable, receiving either its identifier or one of its tags. Relevant to note that, these dictionaries were each included inside of each own other dictionary, labeling them by namespace, since such is an important aspect of the environment. An example of this storage system functioning, depicted in Figure 3.5, is when we need an immutable data item, giving its ID as input. First we choose between the Dictionary organized by tags or identifiers. since we have an ID as input, we would choose the second. We would provide the namespace and, if such item existed in the this namespace, the output would be a Metadata object. Checking the flag *isMutable*, we would proceed to next dictionary, in this example, if the tag was false, it meant that the data item was immutable, so we

would follow to the immutable data dictionary, providing the Metadata object as input, we would finally get the data item requested.

Mutable Data Items

As previously stated, EdgeGarden allows sharing of data, being it mutable (CRDTs) or immutable (traditional data items). The system was already equipped with the capability of, given a Mutable data item update, automatically trigger a notification on the environment. However, a Mutable Data Item must be, on implementation, explicitly declared as so. The Mutable Item base approach already provides a call to a method named *save*. Such method is available to, on a correct implementation of a mutable item, be triggered at the end of every update call's code. The behaviour of such functionality relies on the existence of P/S functions on the system it currently held. The approach used both in Thyme (on the mobile side) and in our implementation, automatically provides the Mutable Item with the publication of their updates. In other words, whenever a mutable data item's contents are updated on a top abstraction, an explicit call to publishing such update is not required, apart from the one to the *save* method on the implementation of such item. This differs from the normal (immutable) data items. Remembering that EdgeGarden was built with abstraction in mind, it provides the programmer with the basic interfaces and classes to be used upon each and every solution. However, some useful state-of-the-art solutions to data items were built in a previous moment than that of our environment, and might not support these functions, or even, could have not been built purposely to be used in our environment. That does not mean these are proven worthless to one's solution and must be addressed as well. Normal data items can be shared and used in any implementation, but must suffer an explicit call to an update publication, as well as an explicit subscription to its updates.

Client and Broker

The mechanisms behind the Publish Subscribe Service work under an interconnection and interaction between its client and its broker. The main difference between both, is that the broker is responsible to execute the actions on the trigger of a certain event, whereas the client's purpose is to make available a set of operations that can be called and trigger such events. Let us say, for example, if a publication is on the run to be shared, the broker would be responsible to deal with the event's consequences and executions, like, store the data in memory, read the metadata to send the publication to its set location and notify interested nodes, whereas the client would be the top entity addressed to manage the interactions between the broker of this event and its perpetrator, it is the client that notifies the server of such event as it also notifies the actor of its hypothetical success or results.

3.2.1.1 Publish Subscribe Service's API and Execution

The `PublishSubscribeService` is a GardenBed service that allows its or any other framework to perform operations on the P/S paradigm of the system. Provides methods that enables one to publish, subscribe, unpublish or unsubscribe to data based on a *tag* and a *namespace*. The API is further explained.

Publish a data item

Two operations are available to manage the publishing of data in the system, the first one publishes a data item, the latter removes a publication:

```
publish(namespace ID, publication, handler)
```

Disseminates a publication to the mobile nodes within the specified namespace. The handler returns either the ID assigned to the published object or an error message if the operation fails.

```
unPublish(namespace ID, object ID, handler)
```

Removes a previously published object given its ID from the specified namespace. The handler returns either the ID of the removed object or an error message if the operation fails.

Subscribe to a data item

Two operations are available to deal with data subscriptions. One to perform a subscription, the other to unsubscribe.

```
subscribe(subscription, handler)
```

Initiates a subscription to receive notifications based on the provided criteria encapsulated within the subscription parameter. This includes details such as the tags of the object in question. Upon successful matching, the handler returns the tags of the matched object. If the operation fails, an error message is returned. In this case, the subscription itself already contains the information about the namespace, relieving the method from requiring it.

```
unSubscribe(tags)
```

Initiates the unsubscription process for notifications based on the specified tags. This operation cancels the subscription.

Linking tags to data items

Linking tags to previously published data is also available through these two operations:

```
link(namespace ID, object ID, tags)
```

Associates the provided tags with the specified object within the given namespace. Subsequent subscriptions to these tags will include retrieval of the associated object.

```
unlink(namespace ID, object ID, tags)
```

Disassociates the specified tags from the object identified within the designated namespace. Following this operation, subscriptions to these tags will no longer retrieve the associated object.

Updates on data items

It is also possible to update a data item, updates are published and subscribe through these operations:

```
publishUpdate(namespace ID, object ID, object, update description, handler)
```

Publishes an update to the specified object within the given namespace. This operation fails if the object with such ID is a mutable data item, since mutable data items update themselves in this environment. The handler returns either the object ID or an error message if the operation fails.

```
subscribeUpdate(namespace ID, object ID, update handler, operation handler)
```

Initiates a subscription to receive updates for the specified object within the given namespace. The update handler is triggered on every update, while the operation handler is triggered upon the operation is either successful or failed.

With all the tools we needed to post and retrieve data from the EdgeGarden-Mobile devices, it is now possible to share clusters between them.

3.2.2 Index Creation and Publication

The face index is created as the images are fed to Chives and clusters are ready to be published as they are created. In Figure 3.6, we can see, in a golden colour, the components that were created or modified during the elaboration of our work. In subsection 2.1.4, we explained what the state-of-the-art Chives was prepared to do. On such framework,

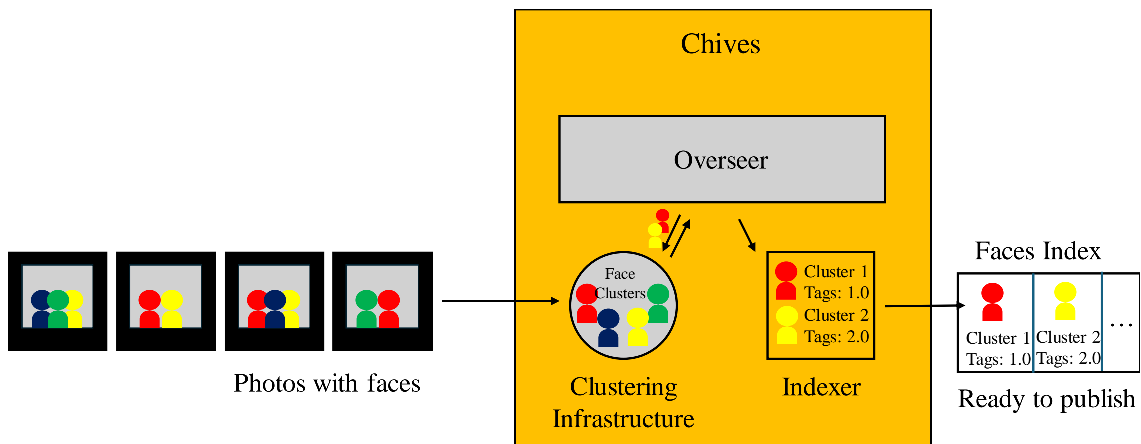

Figure 3.6: Functioning of Chives on the Edge

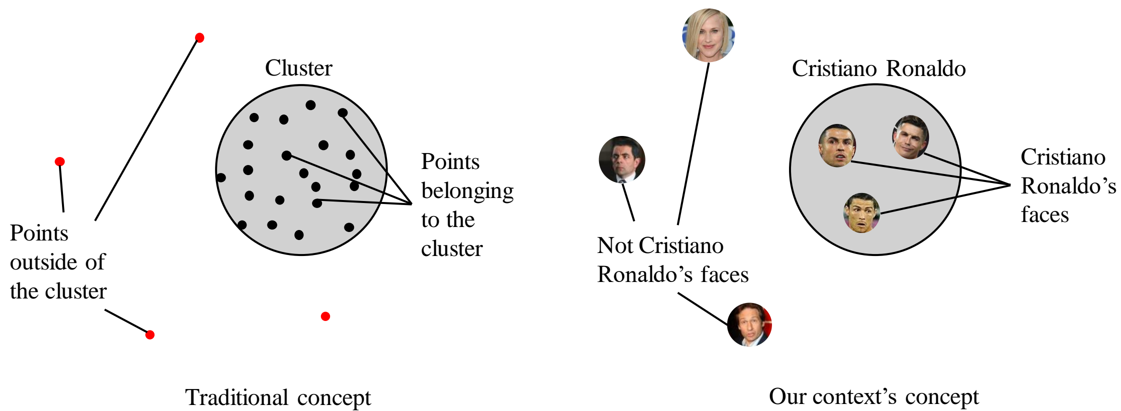

Figure 3.7: Clusters Under Our Context's Concept

we added an Indexer that will be responsible to generate and prepare the index of faces, from the face clusters it receives from the Clustering Infrastructure of Chives. This index is composed by clusters which are appended with identifying tags. Further, we explain this mechanism.

The Clusters

The traditional digital concept of cluster is a group of close-by points. In our context, such points are double-pointed, the projections, obtained through the FtE. In such concept, projections represent different faces from the same person extracted from different photos and each cluster represents a person. In Figure 3.7, we explain this with an example: this cluster contains the projections of the faces extracted from Cristiano Ronaldo's photos, any face that is not his, does not get included in his cluster. The first version of Chives dealt with normal clusters: groups of close-by projections, with an identifier. We leveraged this definition of a cluster to include also a set of tags, as we needed them to

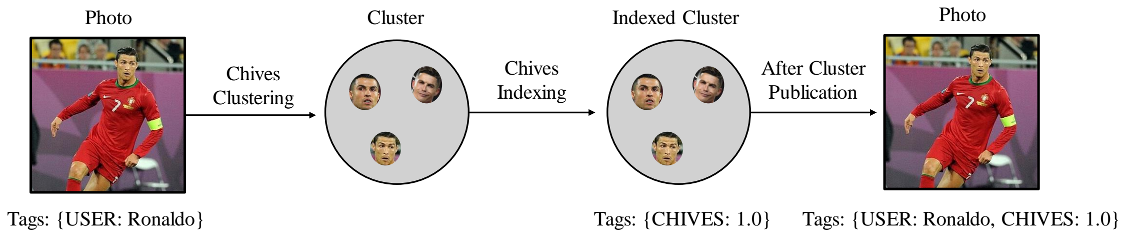

Figure 3.8: Establishing a Connection Between Published Photos and Clusters

be identified within our P/S context. We refer to them as Indexed Clusters. This way, such identification would not be lost throughout the whole edge computing system, as one node of the system could always perceive, through such tags, which clusters the operations are to be dealt on.

The Tags

Tags are a feature already present in the state-of-the-art EdgeGarden environment, as explained in subsection 2.1.1. To allow a better distinction between types of *tags*. It was created a new characteristic every *tag* will contain: the *TagType*. Tags can now be of type *USER*, the already existent and so far only type of *tag*, and of type *CHIVES*. The latter will make sure *tag*s created with the purpose of identifying and organizing clusters within the index will not be confused or mismatched with user created *tag*s. Tags of type *CHIVES* will be appended not only to the cluster it represents, but to the photos from which the projections of such cluster were extracted. This will allow the edge server and all of the mobile nodes to easily retrieve the photos in which a face is represented when searching by such face, as shown in Figure 3.8.

The Index

The key element in the whole solution is our cluster index. It is the index that contain all of the clusters, organized by *tag*s, and it is on the index that the search operations for photos by face will be executed on. The index in our solution required to fulfill two main characteristics:

1. The index required to be a set (non-ordered) with ability to add and remove elements with ease;

Table 3.1: Possible Sets

Set	Type	Add Elements	Remove Elements	Data Sent on Update
State G-Set	S-CRDT	Yes	No	The New State of the Set
Operation G-Set	O-CRDT	Yes	No	The Add or Remove Operation
State OR-Set	S-CRDT	Yes	Yes	The New State of the Set
Operation OR-Set	O-CRDT	Yes	Yes	The Add or Remove Operation

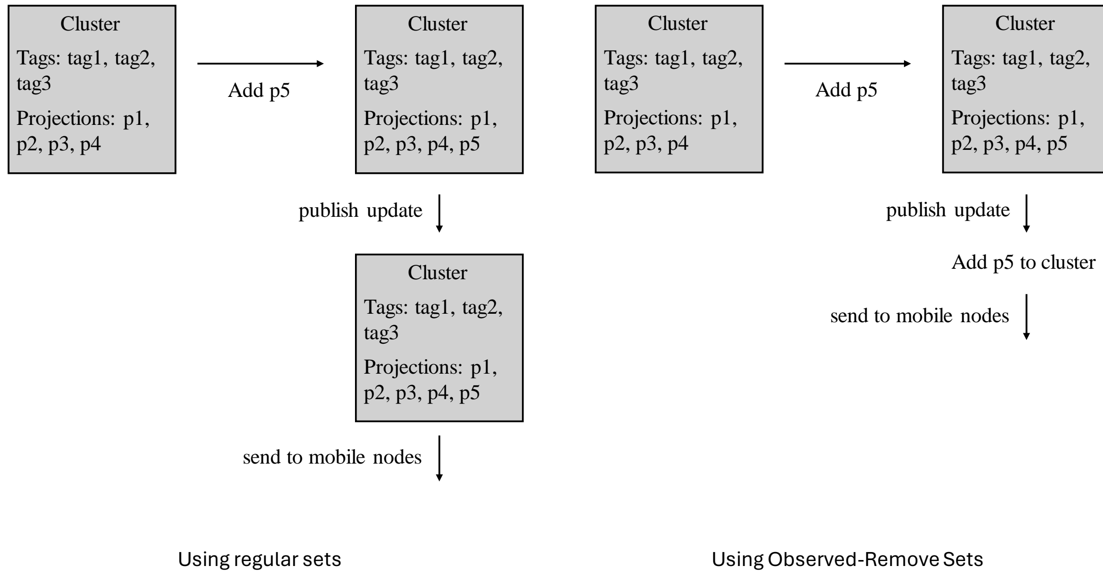

Figure 3.9: Difference Between Regular Sets and OR-Sets

2. The index required to be a shared element with integration in our P/S infrastructure.

In Table 3.1, a few CRDT sets are described given their characteristics. Using a Grow-Only Set (G-Set), we would lack the ability to remove elements, which leaves us with the Observed-Remove Sets (OR-Sets). For this purpose we used an Operation OR-Set, explained in section 2.2.3, for its ability to add previously removed elements and its independence from system-dependent timestamps [18]. The index is, so, shared through a *publication* on the edge's behalf, and read on the mobile phones, through a *subscription*, fulfilling, thus, our requirements. Taking this type of CRDT as an index, we wanted that our clusters could update each of their properties independently without having to share all of its composition every time a single detail was altered. For this purpose, we took this choice as an example and modified the our indexed clusters accordingly, turning both the *tag*s and the *projection*s of each cluster an OR-Set of their own. We could, then, add or remove either a *tag* or a *projection* without having to update the whole cluster, as represented in Figure 3.9.

The Indexer

The main actor in this component of Chives for the indexation part is the Indexer. Such Indexer's purpose is to receive a cluster out of the Clustering Infrastructure and prepare it to be ready for publication. Such actor takes the information received upon clustering and turns it into shareable information. It generates an Indexed Cluster by storing its tags and projections in the OR-Sets, referred in the last paragraph, and adds such Indexed Cluster

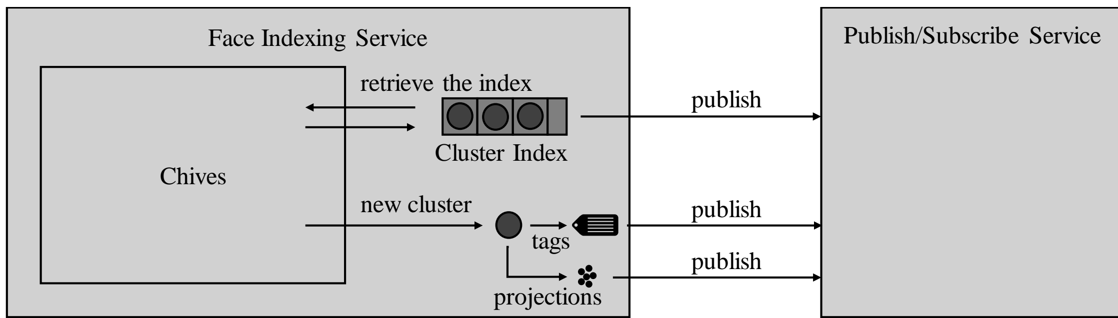

Figure 3.10: The Face Indexing Service

to the index. Chives, on the Edge, is an API built to be integrated in a GardenBed service, so we required such service. Although the indexer prepared clusters to be ready for publication, the publication itself is performed recurring to a service, explained further ahead.

3.2.3 The Face Indexing Service

The **FaceIndexingService** is the service we have developed to run *Chives*, retrieve the index from it and publish such index and its clusters. In other words, it spreads the face clusters along the mobile phones. Making use of the *P/S Service*, mentioned in subsubsection 3.2.1.1, this service does the connection between Chives (edge module) and the P/S system of the Edge Server. It fully operates within its own namespace, the *Chives* namespace and uses its own tag for the index, of type *CHIVES*, set to be subscribed by the mobile side. In Figure 3.10, we demonstrate its behaviour. The service starts by retrieving the index from Chives, and publishes it. Subsequent updates on the index are triggered automatically, since it is a Mutable Data Item, explained in section 3.2.1. When Chives notifies the service of a new indexed cluster, the service is responsible to publish both its *tags* and its *projections*, which are also mutable sets, as we pointed out in section 3.2.2. By relying on this type of CRDTs, we grant, so, that the contents of the index will be, eventually, consistent (subsection 2.2.2).

Sharing the index among other regions

During this dissertation, the main focus was to implement the faces' index in a single region, however, in future iterations of this work one might want it to spread the index among other regions. As previously stated, the properties of EdgeGarden's publications allow the programmer to set the index's scope of interest to global, which will allow future workers to configure the index in order to enable inter-region sharing.

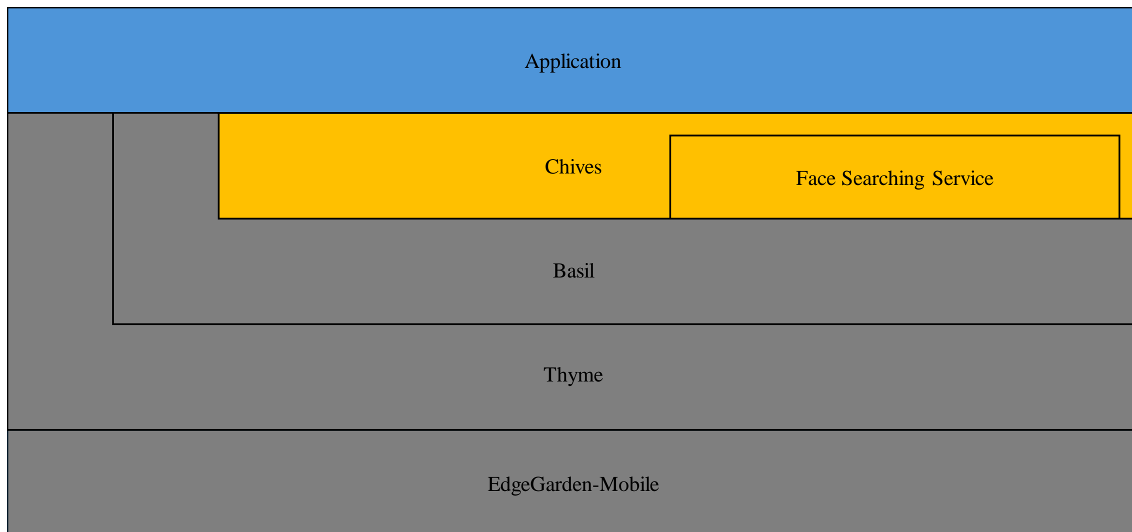

Figure 3.11: APIs Used in the Mobile Phones

Appending tags to the published photos

In order to be identifiable by face, the photos circulating on the environment, must be linked to the *tags* generated by this service. Although this operation was not yet implemented, we will explain its workflow. When retrieving the faces' features from a photo on the system, the Face Indexing Service will temporarily store the identifier of the publication of such photo along with its features until such features are detected on a cluster. Then, we use this cluster's *tag* and link it to the publication.

3.3 Mobile Client Services

The mobile phone is the most important part of the EdgeGarden environment. After all, the whole purpose of this environment is that the users are able to communicate and exchange data almost independently from external factors, depending only on an edge server to establish its region of mobile phones and provide indexation services, explained in section 3.2. It is on the mobile phone that the users will interact with whole environment. As the users publish photos on the mobile phone, such photos will be cached on the edge server, which will use them to create an index of faces. The mobile phone will, then, allow the users to provide a photo of a face, which will be used to, find previously shared photos containing this very same face.

We have, so, developed an API called Chives, on the mobile side, which will be responsible to search for photos based on the faces they contain. Chives is built on top of Basil, explained in subsubsection 2.1.3.2, and runs a service called the Face Searching Service. The latter relies on Basil to retrieve the index and its clusters from the edge server and provides to Chives a way to search on this index using a photo of a face as input. In

Figure 3.11, we can see the full stack of APIs running on the mobile phone, the golden coloured are the ones we implemented for our solution, the shaded ones are previously available APIs, already explained in the previous chapter, with the blue one being any application the software developer might want to implement using Chives. Next, we explain how the faces index is received and handled on a mobile phone.

3.3.1 Receiving and Using the Index

In order to reach the final goal of searching for pictures using a face as an input, we must, firstly, receive the faces index. Upon index reception, the user is, then, able to search on it using a face.

Retrieving the index

The index retrieval is done is coordination with the edge server. We have explained, in subsection 3.2.3, that the edge server shares the faces index through a publication of such index under a predefined *tag*. Using the default tag for the cluster index, the same used by the edge on publication, the mobile phone can subscribe and download the face index and then proceed to retrieve the clusters as well. In Figure 3.12, we demonstrate this workflow through a flowchart diagram: first, a subscription to the index is done. Upon notification that the index has been published by the edge server, we proceed to its download, followed by the subscription to all the clusters it contains. Whenever we receive a notification on a cluster, we download it and subscribe both its tags and projections. In a similar manner, a notification on both the tags and projections will be received, and we proceed to their download. Once both of these sets are downloaded, we can reconstruct the cluster using them, setting, then, the end of the process. When an update on the index is received, it means a new cluster has been introduced, and, so, we are required to download it and add it our local index. The part of the process where we subscribe and download a cluster, and subsequent actions, is repeated whenever we receive a notification on an update of the index.

As we have stated in subsection 2.1.4, the state-of-the-art Chives already provided us with the tools to perform FcE and FtE. In section 3.2, we explained our principle of feature closeness: 2 vectors of features that are similar enough are considered as being extracted from faces of different photos of the same person. We will rely on this principle to perform the index search.

Using the index

The index is an OR-Set of indexed clusters, as explained in subsection 3.2.2. Indexed clusters are defined, in such section, as clusters with appended *tags*. These *tags* are also appended to the photos from which the projections of these clusters came from. Searching for pictures of a specific person cannot be done without finding the *tags* of the clusters in

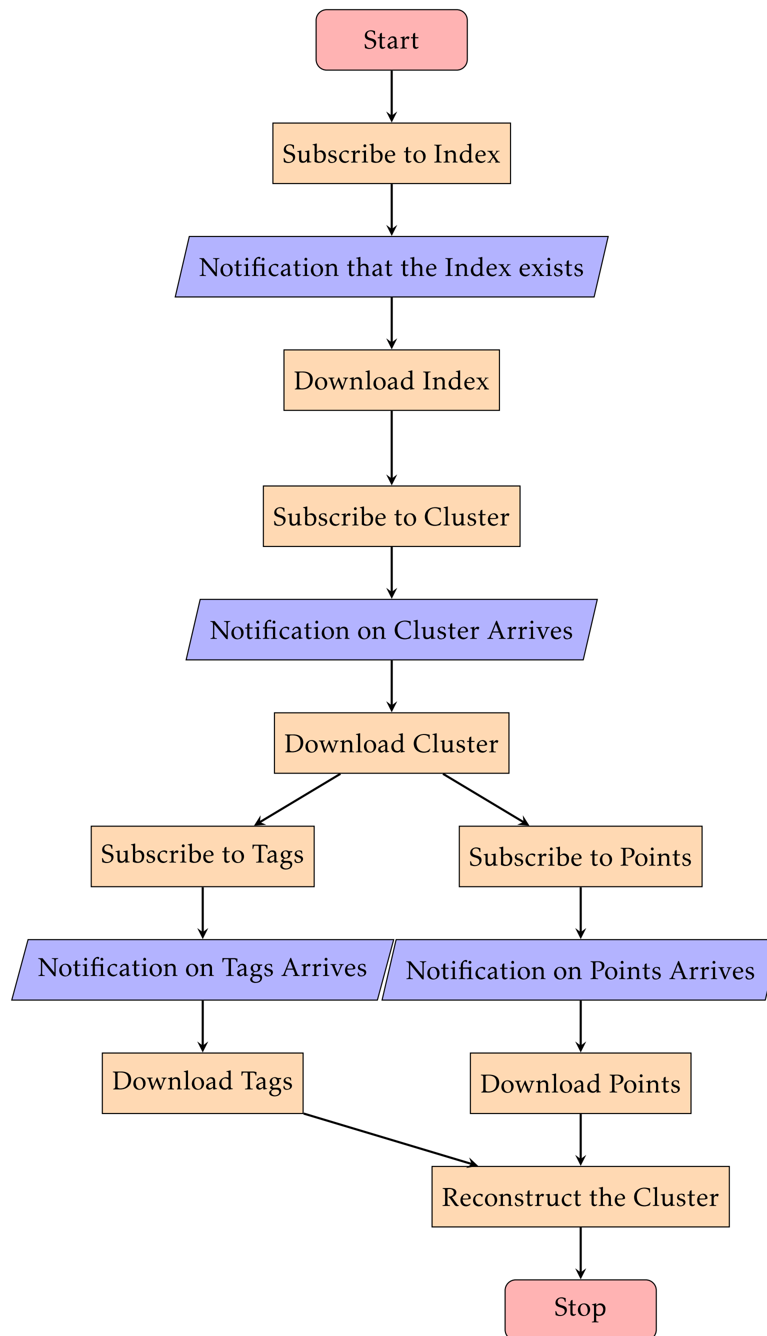

Figure 3.12: Downloading the Face Index on the Mobile Phone

which these people's faces are indexed. Looking at Figure 3.7, we can trace an equivalence between a point belonging to a certain cluster, and a face belonging to a certain person. So, if we extract the features of the face of the input photo and those features happen to belong to any cluster in the index, we can use the *tag* of such cluster to finally find all the pictures in which such person's face is included. This is the exact principle we used in the development of our solution: we extract the faces of the input photo, in each face we extract their features into points, using the K-Nearest Neighbors (KNN) algorithm, we

try to find the closest cluster to each point, and return the indexed clusters. Having these clusters, we have their *tags*, and with the *tags* we can search for the desired photos.

Searching for the photos

To finish our procedures, we must, still, find the pictures to deliver to the user. Although this part was not yet implemented, we will provide its workflow. After retrieving the *tags* of the indexed clusters, we will use Basil to find the desired photos. Making a subscription with the *tag* retrieved from the last step, on Basil, all the mobile phones in the region that contain such *tag* on, at least, one of their photos will notify the user's mobile phone. Each one of these notifications contain a thumbnail of the photo in question. The user will then decide which photos he wants to download.

It is easier, now, to understand how the Face Searching Service operates in the mobile phone.

3.3.2 The Face Searching Service

Integrated in the Chives API, the Face Searching Service is responsible to retrieve and use the index in order to find the appropriate *tags* of a certain cluster. Observing Figure 3.13, a sequence diagram displays the whole process of finding pictures with similar faces, having four actors in mind: the user of the system, Chives (on the mobile phone), Basil (on the region of mobile phones) and GardenBed (on the edge server). First, the cluster index is published by the edge server, followed by a request on the mobile phone for such index, which is retrieved and stored locally, as explained in subsection 3.3.1. When a user proceeds to submit a photo to Chives for photo searching, Chives will use this photo to extract its faces and features, followed by a search for *tags* on the index, as explained in section 3.3.1. Having the *tags* of the cluster, the Face Searching Service will request Basil for any publications with these *tags*, resulting on a request on other mobile phones of the region and, even, cached publications on the edge server, native to other regions of the environment. Such pictures will be returned to Chives, which displays them to the user on the mobile application.

The Face Searching Service is responsible for doing this whole process and making available to Chives for user usage.

3.3.3 Chives' API

In this section we present the API that Chives makes available to the user.

Finding the tags of a cluster

```
Set<Tag> findTags(OpenCVImage face)
```

This operation is responsible for retrieving the set of *tags* of an indexed cluster of projections of a face, given a similar face as input.

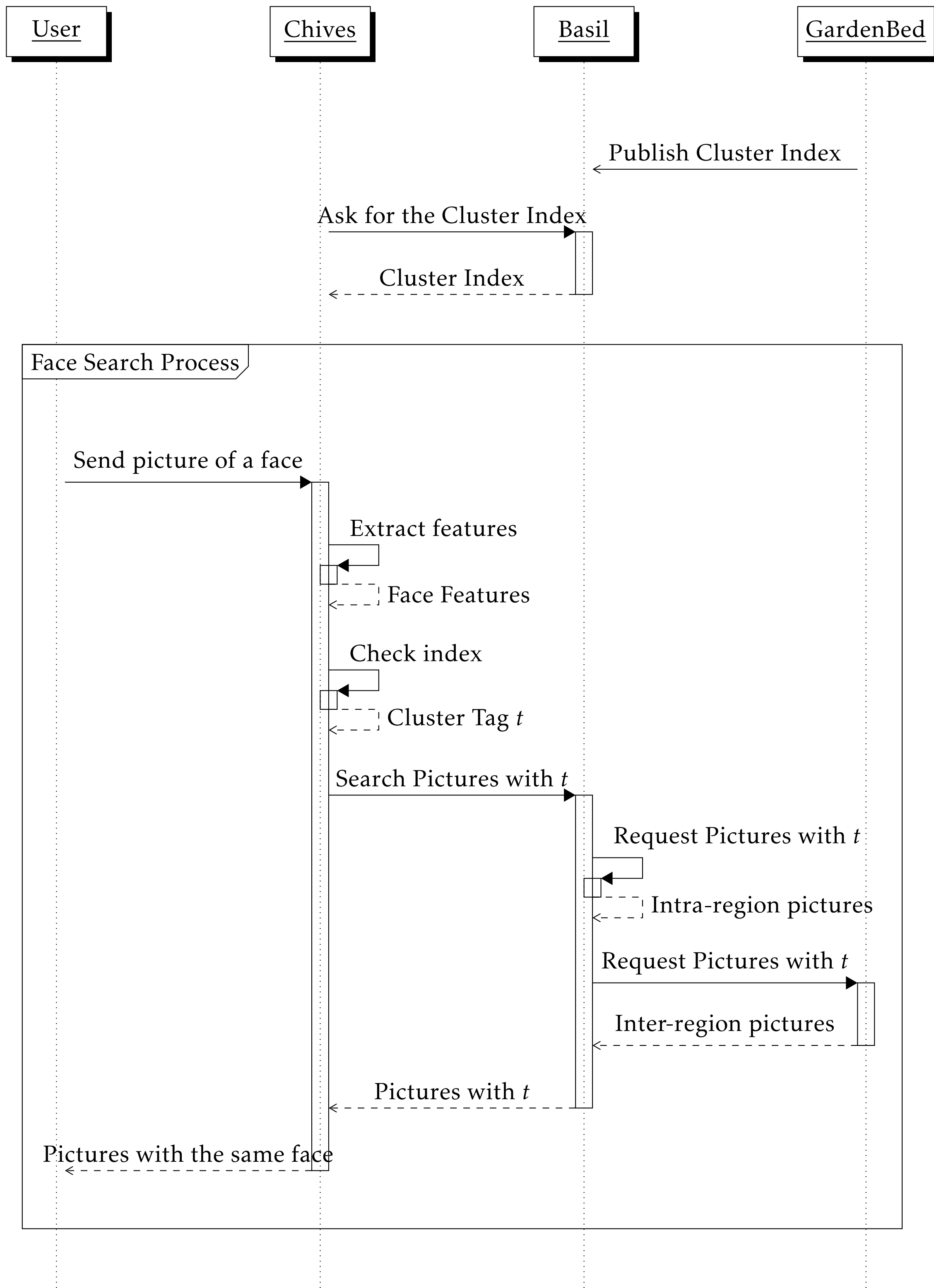

Figure 3.13: The Face Searching Process

Finding the photos given a face

```
Set<Image> getImagesWithFace(Image face)
```

This operation is responsible for retrieving all the photos that contain a specific face in them. As stated previously, this is yet to implement and will be left for future work.

3.4 Implementation Details

3.4.1 Distinguishing Between Mobile and Edge

During the development of this project, some state-of-the-art implementations working well on a pure Java environment would not run on an Android environment due to incompability of some of the libraries or classes used. Taking that into account, some modifications had to be done to some of the modules of the state-of-the-art. This was the case of the `ArcFaceFeatureExtractor` which run a class `BufferedImage` of the Java Abstract Window Toolkit (java.awt), not available in Android Java. Another issue with the implementation of `ArcFaceFeatureExtractor` was that it ran a process in Python to perform the FtE. Most Android phones do not support running Python code on their Operative System (OS). To solve this issue, we have separated the code of `ArcFaceFeatureExtractor` into two versions. One to run in Java, for the emulation and edge server environments, and another in Android, for the mobile phones. The single module *arcface-feature-extractor* was divided, so in three submodules:

arcface-feature-extractor:core in which we have integrated thr code to both versions into an `AbstractArcFaceFeatureExtractor`, common class;

arcface-feature-extractor:java in which we have integrated the parts of the code that would solely run in normal Java environments, into a class `ArcFaceFeatureExtractor` extending the abstract one. This part of the code runs on both the edge server and on emulated mobile phones;

arcface-feature-extractor:android in which we have implemented our own version of the `ArcFaceFeatureExtractor` using the class `Bitmap` from the Android Graphics (android.graphics) library as an alternative to the non-compatible one. And have integrated pre-compiled Python code to run the part that would previously run on a different process with the OS build tools, using a library called Chaquopy (com.chaquo.python).

3.4.2 Restructuring the Chives Module

A lot of Java classes in the state-of-the-art Chives had the sole purpose of serving the Edge Server. We have decided that such classes were not necessary in the Mobile Phones implementation. In a similar manner, some developed classes in our mobile solution were

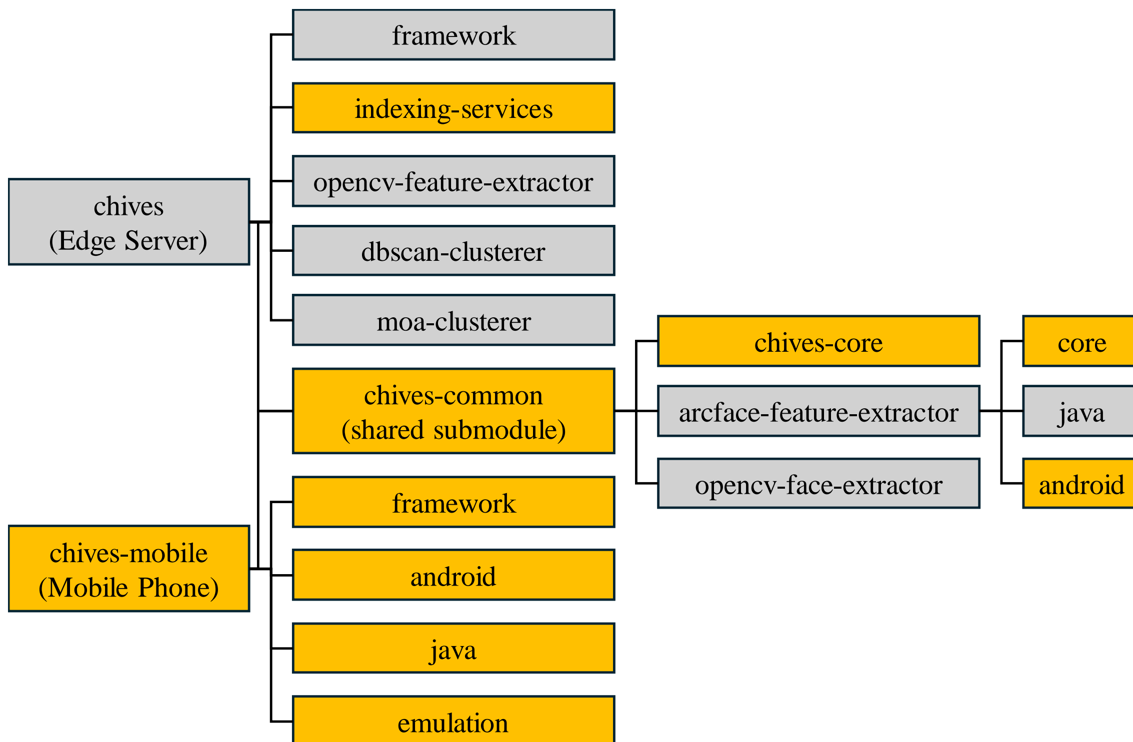

Figure 3.14: Modules Organization

irrelevant to the Edge Server. With that in mind, we have decided to restructure the whole Chives module into two modules: Chives (for the edge server version) and Chives-Mobile (for the mobile version). The common parts of the code are inside of a module called Chives-Common. In Figure 3.14, we represent an hierarchy of modules and submodules of both Chives and Chives-Mobile. In a gray colour we have the modules which suffered little to no change, while in golden we have the newly implemented modules in our solution. We describe the latter modules as follows:

indexing-services module where the FaceIndexingService is located: serves the purpose of indexing on the edge server's behalf;

chives-common shared module between chives and chives-mobile where the submodules with shared code are located;

chives-core submodule where core definitions are made. Classes for abstractions like Cluster, Projection, ClusterIndex are stored here;

framework Chives (mobile version) and the FaceSearchingService are stored here;

android Classes for chives-mobile which contain android-only code are located here;

java Classes for chives-mobile which contain java-only code are located here;

emulation submodule that deals with emulating a mobile phone.

3.4.3 Configuring Gradle

At the start of the developing work, we had the gray libraries in Figure 3.14 completely functional on a Java environment. However some of the libraries used were not prepared to run on Android, given the state-of-the-art Gradle configuration. Such was the case of the opencv-face-extractor module that used the JavaCPP, JavaCV and OpenCV libraries. Differently to the arcface-feature-extractor module, these libraries had an alternative to run on an Android environment, so the whole module didn't require restructuring. We made use of a Gradle property called *emulation*, available on the edgegarden-mobile's build configuration, and a variable created in our development called *chives-mobile* to set the usage of either the Java configurations or the Android ones. In the Gradle build file of the module chives, we set *chives-mobile* as *false*, while in the chives-mobile module, such variable was *true*.

In the case of the opencv-face-extractor module, if the running configurations were set to either the edge server or an emulated mobile phone, it would use the Java libraries. However, if the running configurations were set to real environment mobile phone, it would use the Android libraries. Using this same logic, we have set arcface-feature-extractor to use either its java submodule or its android submodule. The same was applied on chives-mobile, which would use its java submodule or its android submodule.

3.4.4 Copying Files Task

Gradle build in the whole EdgeGarden project proceed to do a copy task during compilation time. This copy task copies some source files, such as Python scripts, face detection models, property files, among others. However, as such script satisfies the need for these files in a context that the compiled code and the compilation task are executed in the same machine, such does not happen when one wants to run the compiled project in another machine. This is due to the face that the copy task script uses a folder in the *User Home* directory to copy these files into, though such folder is available in the machine if the code is run on the machine it was built on, in other devices it will not find such folder. Android allows the programmer to use these Gradle tasks to copy files to the *assets* folder, a folder that is integrated inside the compiled project. This allows such files to be accessible in the code of the project on the mobile phone, and solves half of the problem. However, the functions Android makes available to access the contents of this *assets* folder do not allow direct access to the files through a file path. When creating the OpenCV tools and the ArcFace tools in chives-mobile, they require models included in this *assets* folder. Accessing files in the *assets* folder is done through an Input Stream, Android makes available. However these libraries are only prepared to receive files as constructor parameters through their String path or File object. Searching for a solution to this issue, it was decided that, when running the app for the first time, the app would copy all the files from the *assets* folder to the app's *storage* folder, which is available directly through their content's path, enabling, so, these libraries access to the models they

required.

3.5 Final Remarks

During this chapter, it was possible to understand the whole developed solution, with all its actors and components. We have developed and explained a solution that leverages the EdgeGarden environment with an alternative to traditional topic-based P/S solutions, a content-based search.

We have described the whole process of face indexing, with subsequent modifications to the components of the existing edge server, as the process of face searching, with the creation of an API called Chives that enables the user to do this search.

In the next chapter, we will show how we have tested and evaluated the efficiency of this solution.

4

Experimental Evaluation

In this chapter, it will be presented the steps it took to evaluate our implementation. We will present our experimental goal in section 4.1, with the questions proposed to evaluate our developed solution. We will proceed to describe our experimental environments, the setup for a simulated environment in subsection 4.2.1 and a real environment with an Android app developed for testing purposes in subsection 4.2.2. Finally, we will evaluate the solution, describing the steps, experimental results and conclusions in section 4.3.

4.1 Goals

The goal of this dissertation was to propose a solution that would allow a user to perform a face-based search on photos within the EdgeGarden environment and evaluate its performance. For the latter, we have created two environments that will help us evaluate the developed solution: a simulated and a real environment. These environments will run our solution and help us answer two questions we propose to analyze our solution in terms of accuracy and efficiency in the face search results. These characteristics are described below.

Accuracy *How accurate is our solution at presenting results?* We will test our solution in a simulated environment to analyze and take conclusions on the best settings we can use in our solution for providing better results at the face search.

Efficiency *How fast is our solution?* and *How much energy is consumed during the execution of our solution?* For these question, we will perform our tests on a real environment in order to have accurate and real results. We will present absolute values on the time and battery usage it takes to retrieve results.

Device	HP ENVY 15 Notebook PC
CPU	Intel Core i7-5500U
RAM	16 GB
Storage	464 GB
Battery	1815 mAh
Wi-Fi	802.11 a/b/g/n/ac
OS	Windows 10 Home

Table 4.1: Device Used for the Experimental Tests on a Simulated Environment

Device	Lenovo Legion Y520	Samsung Galaxy S21 5G
CPU	Intel Core i5-7300HQ	Exynos 2100 (5 nm) 1x2.9 GHz Cortex-X1 & 3x2.80 GHz Cortex-A78 & 4x2.2 GHz Cortex-A55
RAM	16 GB	8 GB
Storage	237 GB + 931 GB	128 GB
Battery	4000 mAh	4000 mAh
Wi-Fi	802.11a/b/g/n/ac	802.11 a/b/g/n/ac/ax 2.4G+5GHz
OS	Windows 11 Pro	Android 13 + One UI 5.0

Table 4.2: Devices Used for the Experimental Ttests on a Real Environment

4.2 Experimental Environment

4.2.1 Simulated Environment

For the simulated environment, we have implemented a single machine environment that runs the *FaceIndexingService*, presented in Figure 3.10, to compute the cluster index, and perform KNN searches on the projections of such index.

On this environment, some statistics on characteristics will be retrieved that will help us analyze the performance of the face search. This procedure has the purpose of helping the user of the framework set the preferences that will provide better search results when setting its application. Constraints such as the amount of photos on the index might have some influence on the searching accuracy.

In Table 4.1, we present the device used for simulation purposes.

4.2.2 Real Environment

To measure the efficiency of the solution, we will use metrics extracted from running the solution on a real environment. We will run Chives on the edge on a computer, simulating an Edge Server, and Chives-Mobile on a smartphone. This will help us create a scenario of a real interaction between the edge and the mobile phone. The computing times and battery usage will be retrieved in their absolute values in order to analyze if the solution

is efficiently practical for modern day devices. The devices used during the tests on a real environment are described in Table 4.2.

Further below, we will proceed to present our evaluation.

4.3 Evaluation

4.3.1 Solution Accuracy Study

When a user provides a photo of a face in order to search on the index for similar faces, a search on the cluster index, using a KNN algorithm is performed. We questioned, during the elaboration of this dissertation, how accurate would the results retrieved, using our solution, be.

If we used a simple KNN with no constraints, we would have a result for every provided picture, as long as there were clusters in the index. There would not be any problem, if the users only provided pictures of people they know that exist in the published photos. However, we do not take for granted that the users know all the contents that exist in the system, as it would not be humanly possible in cases where multiple people are publishing photos. We must provide feedback to the user upon searching, whether or not its provided face is indexed by Chives, in order to avoid the maximum of number false positives on our solution results.

When presenting results, we must ask ourselves two questions:

1. Does the person the user wants to find exist on our system?

2. If the person exists, is our solution providing correct results?

To answer these questions we performed the indexation of faces on some photos and, then, performed searches on the index with completely different photos. For this purpose we have organized our sets the following way:

Photos to perform indexation To perform the indexation of faces on photos, we used a set of photos with the faces of ten different people, including: 168 photos of Colin Firth, 149 photos of Patricia Arquette and 43 photos of Rowan Atkinson. Note: no photos of Cristiano Ronaldo were included in this set.

Photos to perform search To perform the search for clusters on the index, we used four sets of ten photos each, each set of different people. The people we searched for were: Colin Firth, Patricia Arquette, Rowan Atkinson and Cristiano Ronaldo.

Running the first test with no constraints, in other words, allowing a result even if the person was not indexed in the first place, allowed us to take really sharp conclusions: the search algorithm was very precise in finding the right cluster as for the thirthy photos of Firth, Arquette and Atkinson, it retrieved the right clusters for every photo used as input. However, in Ronaldo's case, as he was not indexed, the algorithm returned clusters

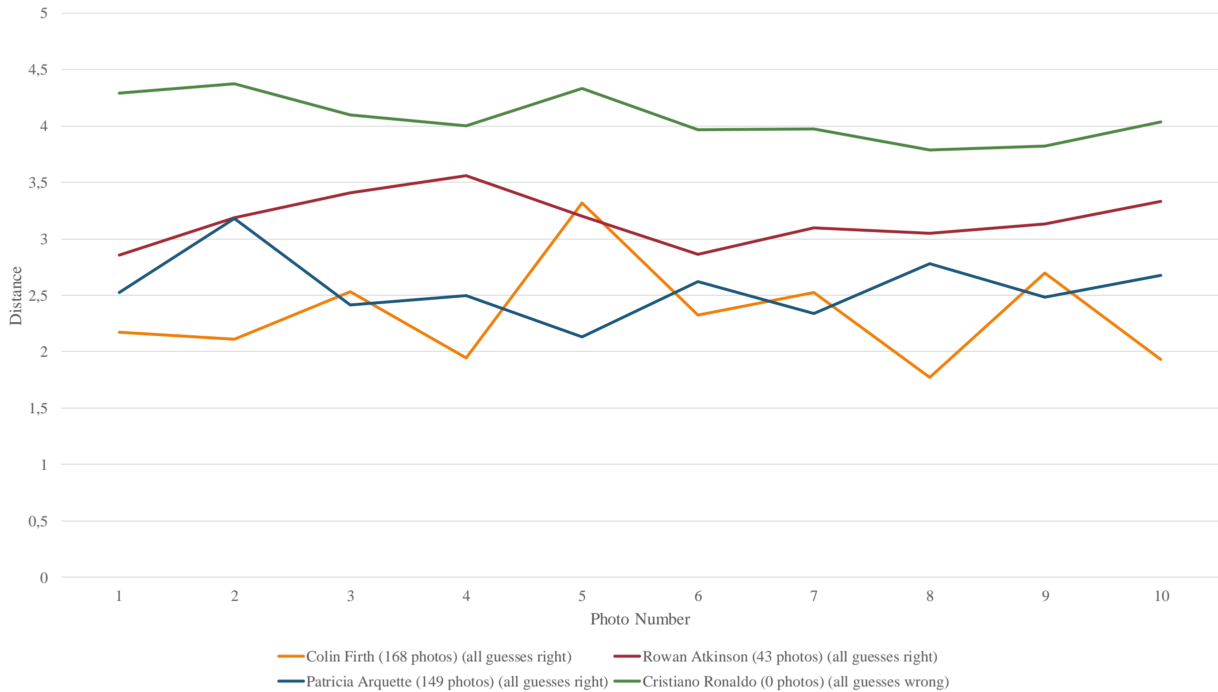

Figure 4.1: Distance Between the Projection of the Test Photo's Face and the Closest Projection in the Index

of different people. Some input photos even outputted different people's clusters when compared to other input photos. We could, however, answer the second question: *If the person exists (in the cluster index), our solution provides the correct results.*

We still required to answer our first question. When searching for projections on clusters of the index, we should be able to recognize some patterns that would allow us to tell if the result the KNN algorithm was giving us was the same person that was being searched for.

In a few words, the KNN algorithm picks the k closest projections to our input projection and outputs the cluster that has the most projections in this set of neighbours. To decide if the test person was actually the one being outputted, and take conclusions on whether or not it was indexed in the first place, we looked at two major characteristics:

1. The distance, in an abstract measurement, between the test projection and the closest indexed projection;

2. The amount of neighbour projections that belong to the same cluster.

Analyzing the Projection's distance The first metric to analyze is the distance between the input photo's face projection and the closest projection in the index. Using the set of test photos, we retrieved this distance for every input photo. The results can be seen in Figure 4.1. The horizontal axis represents the number of the photo used as user input

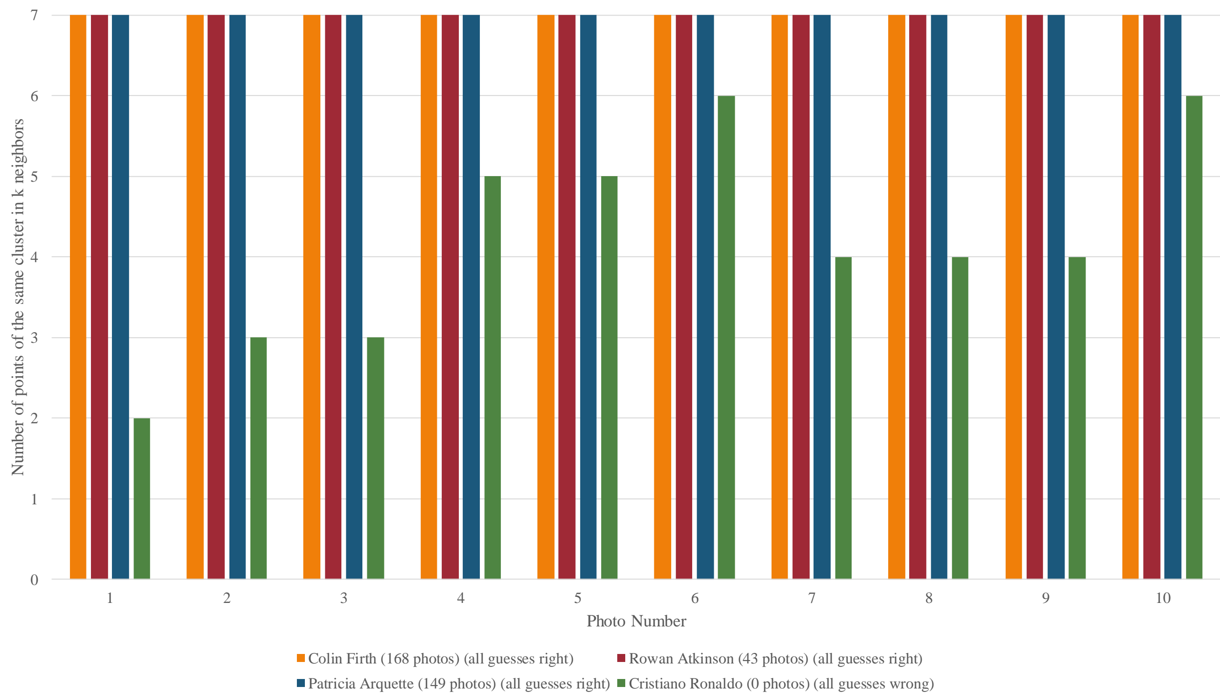

Figure 4.2: Maximum Number of Projections of the Same Cluster Within the 7 Neighbours of the Input Photo's Projection

while the vertical axis represents the retrieved distance. A first conclusion we can take is the notable difference in the results of a person that is not indexed (Cristiano Ronaldo, in our test) when compared to indexed people. The longest retrieved distance on an indexed person's test was Rowan Atkinson's fourth photo with the value of *3,558558839*. The shortest retrieved distance on a non-indexed person's test was Cristiano Ronaldo's eighth photo with the value of *3,78568798*, which is still an higher value than that of an indexed person. We can easily point out a distinguishing between people that exist in the system and people that do not exist in the system. With this amount of indexed photos, a good value to set as constraint for this distance would be somewhere in between, like the mean between these two described values: *3,6721234095*. However, this did not seem like a good metric to constraint our results as the difference between the closest two results that should output a cluster or not is too small to avoid wrong results. Another negative aspect that can be noted is accuracy of this metric when the clusters are yet too small. As we can see by comparing Rowan Atkinson's results, results of a person that has a significant difference in cluster size due to less indexed photos, to Firth's or Arquette's results, a person that has less faces indexed has projections with longer distances when compared to people with bigger clusters.

Analyzing the K neighbours Another metric used to analyze our results was the number of neighbouring projections that belong to the same cluster. The more precise search

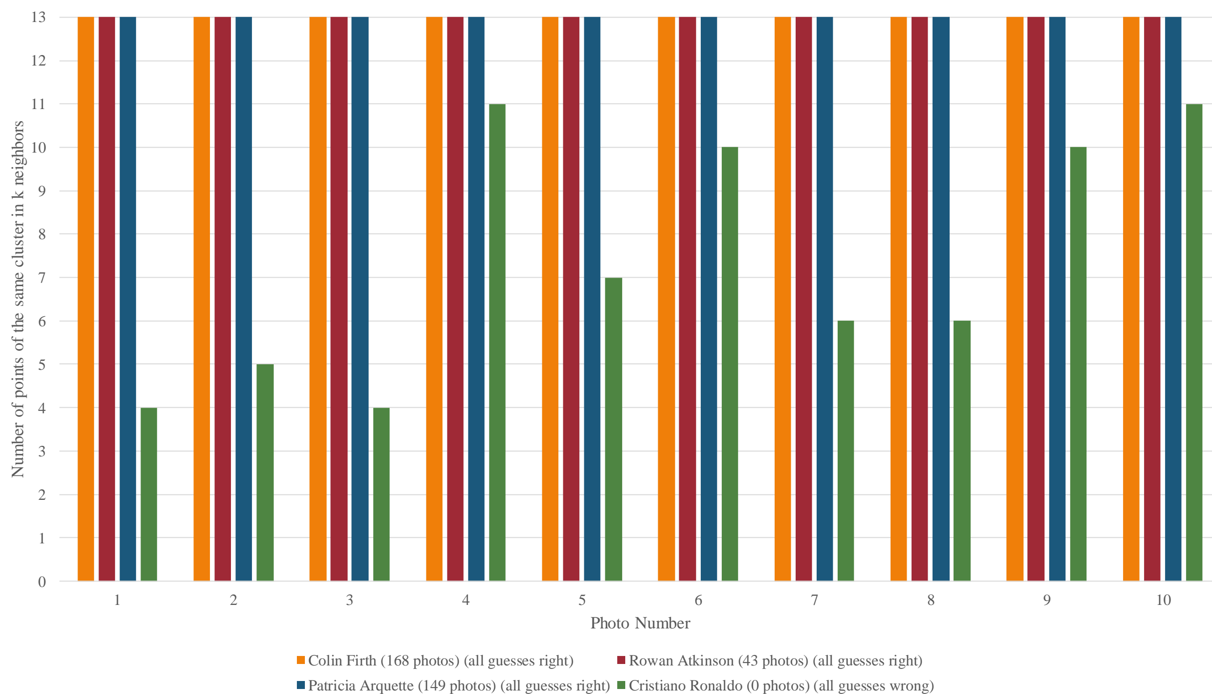

Figure 4.3: Maximum Number of Projections of the Same Cluster Within the 13 Neighbours of the Input Photo's Projection

would output all K neighbours as belonging to the same cluster, whereas a more imprecise result would be that which would result in less than K neighbours as belonging to the same cluster. To analyze this metric, we have done tests with two different values of *k*. The results will be presented in bar graphs where the horizontal axis represents the number of the test photo and the vertical axis represent the number of neighbours of the test projection that belong to the KNN's chosen cluster. We performed the first test with *k=7* and its results can be seen in Figure 4.2. As we can see, all the indexed people got accurate results with all of them having seven neighbours of the same cluster. However, when testing the person that was not indexed, the results are less precise, having the maximum number of neighbours belonging to the same cluster been six, in Cristiano Ronaldo's sixth and tenth photo. In the second test, we used *k=13* with its results presented in Figure 4.3. The same conclusions can be taken for this example. Indexed people got all thirteen neighbours of the same cluster, the outputted cluster. Non-indexed people got fewer neighbours. The best result Cristiano Ronaldo's set of test photos achieved in this example were Cristiano's fourth and tenth photos, both with eleven neighbours of the same cluster. This seemed to us a more accurate metric to use to detect non-indexed people, since we can detect that a person belongs to the index when it has the maximum number of neighbours possible, belonging to the same cluster. However, choosing the value of *k*, when performing the search does not have a visible best choice. Choosing smaller values like *k=7* might result in a less solid result, while choosing bigger values

like *k=13* require a higher number of indexed faces.

Performing these tests allowed us to take answer our first question. Choosing a preset value for either the maximum distance between the input photo's face projection and its closest projection or the minimum amount of neighbours belonging to the same cluster allow us to differentiate between an input face that does not belong to the index and one that does.

Receiving correct results is the top priority in our solution. Nevertheless, performing our search with efficiency is an aspect to deal with. Such characteristic will be analyzed further below.

4.3.2 Solution Efficiency

In terms of efficiency, we questioned ourselves how efficient would our solution be. We considered two major aspects when analyzing this characteristic: To have an efficient retrieval of results, this operation must be both *fast*, in the time it takes searching on the index, and be *energy-efficient*, in other words, consume the less energy possible.

The human eye can process ten to twelve frames per second, with any value above that being interpreted as motion [19]. This means a frame is, on average, processed every 91 ms. We considered that our application would be fast if we could retrieve our results no more than five times slower than this value. This means that, if we can achieve results on the index faster than half a second, we will label our solution as *fast*. The index search operation is an operation to be run once every time a user wants to find photos. With this purpose in mind, is expectable that energy consumption would not hit high values. However we must have some target value for this measurement. So, we have decided that, even if a user makes constant searches, with no intervals in between, during an hour, the algorithm should consume no longer than half the capacity of the battery on the user phone. Values below this threshold would make our solution *energy-efficient*.

In order to measure our characteristics in efficiency. We prepared an Android app for the case. Such app retrieves the index from the edge server and performs a thousand consecutive index searches. It would register the time and battery level difference in between these operations and proceed to present some statistics.

In Figure 4.4, we can see the application's results. One thousand operations took the phone 195537 ms or 195.5 seconds to execute and consumed 2.0% of its battery. The average operation took, so, 195 ms to execute and consumed 0.002% of battery. As the average operation takes 195 ms, no more than five times the frame processing time of a normal human, we concluded that our solution was *fast*. One minute of constant executions consumed 0.61% of battery, while one hour of these executions consumed 36.82% of battery. As even if a user performed one hour of consecutive operations the battery level would not decrease more than 50%, we considered our solution to be *energy-efficient*.

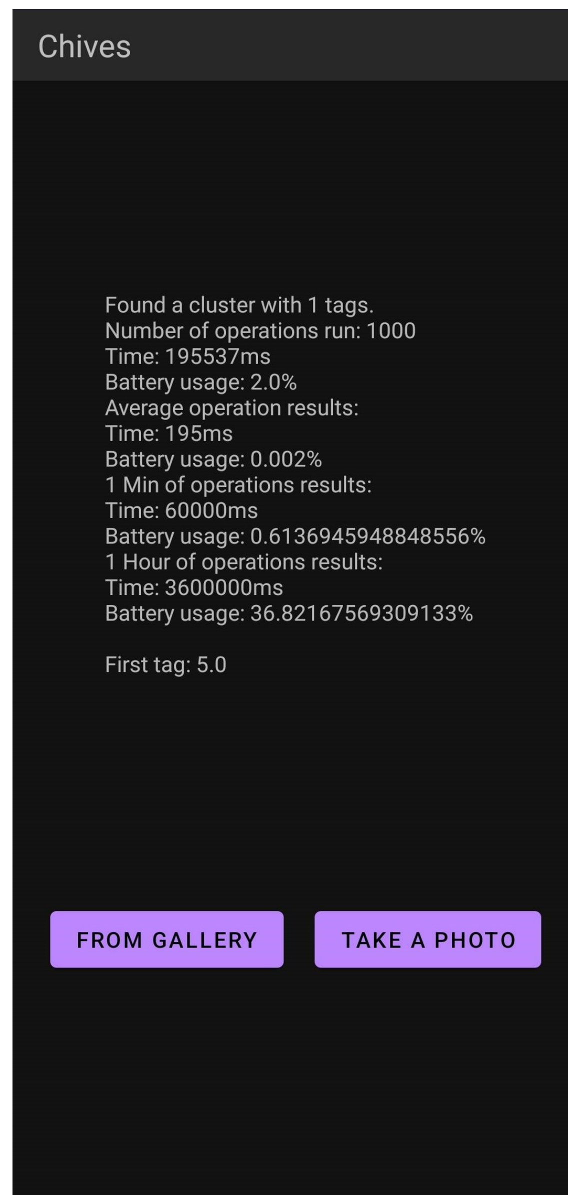

Figure 4.4: Testing Time and Battery Consumption

4.3.3 Summary

After analyzing the experimental results of our solution, we could answer the questions proposed in section 4.1:

Accuracy *How accurate is our solution at presenting results?* Our solution presents correct and precise results when a user tries to search for a cluster in the index, having our experimental results showed 100% accuracy in retrieving such clusters. To distinguish between an indexed person and a non-indexed person, we have displayed the best presets a programmer can choose for their solutions.

Efficiency *How fast is our solution?* Our solution presented fast results, not surpassing five times the average human's frame rate perception.

Efficiency *How much energy is consumed during the execution of our solution?* Our solution consumes low values of energy. Even if a user proceeds to perform constant searches for one hour straight, not even 50% of the phone's battery would be consumed.

5 Conclusion

During this chapter, we will conclude our dissertation. We start by presenting some conclusions on our solution, in section 5.1, followed by some topics that we propose as future work, providing some guidance, on section 5.2.

5.1 Conclusions

In this dissertation, we have presented Chives, a solution to face-based searches on photos within an edge computing environment, centered on an index of face clusters.

The presented solution is composed by two main components: an edge server and a region of mobile phones. The edge server is responsible for caching published photos by the mobile phones and proceed to retrieve face's features from the photos, cluster and index them into a Cluster Index. Each mobile phone is responsible for retrieving this index and allowing the user to search for photos giving a photo of a face as an input.

This implementation is not yet finished as it still needs published photos to be appended to the tags of the clusters generated using its faces, in order to provide such photos to the user on the mobile phone.

We conclude that our solution presents accurate and efficient results on the proposed topic. Future developers using our API must choose the implementation parameters wisely, depending on the purpose they desire for their implementation.

Further below, we will present some future work on topics that could enhance our solution.

5.2 Future Work

Although we have developed a solution that works and allows indexing faces on the edge server and searching for photos giving such faces on the mobile phone. We consider that further work can be done in order to enhance our solution. In this section, we describe some topics that can be worked on, in order to improve this dissertation's solution.

5.2.1 Integration With Oregano

As we want to allow our solution to work on every device, we must consider all types of devices a user may have. Some users might not have the most recent devices with power capacity to perform FcE and FtE, or may have less storing capacity that allows them to download the full index. With that in mind, we propose to integrate the Face Searching Service within Oregano, so that users with older devices can ask the edge server to perform the searching computation on their behalf. This can be done using the Subscription with Computation (SubC) property of Oregano, as referred in subsubsection 2.1.3.3.

A user with an old phone, when performing a face search operation, would trigger a SubC that would publish the input photo with a *tag* where such user expects to receive the output, followed by a subscription to such *tag*. The edge server, upon receiving such publication, would compute the searching operation, and publish the output on the *tag* specified by the mobile phone.

5.2.2 Merging and Diverging Clusters

As the index is being computed, on the edge, there may be cases where a person is indexed on different two clusters as if it was two different persons. Some other cases might group two persons on a single cluster. This happens when such people are similar in facial features or when the clusterer performs its clustering with inaccuracies.

In order to solve this issue, the mobile phone may ask the users for feedback on whether a cluster is grouping the right faces, or even, an implementation on the edge can be performed to tackle such problem. In either cases, the environment must be able to handle these changes on clusters. When a cluster merges with another or diverges into two of them, the index must be updated and, if needed, so do the *cluster tags* on the photos.

Handling a cluster merge can be done by simply appending projections and *tags*, without requiring changes on the published photos. When the merged cluster is retrieved on a search, all the *cluster tags* are subscribed, resulting in the retrieval of all the photos of the previously two clusters. Handling a cluster diverge might be a more complex task. One possible solution would be unlinking all the photos that had the previous *tag*, and proceed to perform the linking operation distinctly with the two new *tags*.

Bibliography

[1] P. S. Almeida, A. Shoker, and C. Baquero. "Efficient State-Based CRDTs by Delta-Mutation". In: *Networked Systems - Third International Conference, NETYS 2015, Agadir, Morocco, May 13-15, 2015, Revised Selected Papers*. Ed. by A. Bouajjani and H. Fauconnier. Vol. 9466. Lecture Notes in Computer Science. Springer, 2015, pp. 62–76. DOI: 10.1007/978-3-319-26850-7_5. URL: https://doi.org/10.1007/978-3-319-26850-7%5C_5 (cit. on p. 17).

[2] C. Baquero, P. S. Almeida, and A. Shoker. "Making Operation-Based CRDTs Operation-Based". In: *Distributed Applications and Interoperable Systems - 14th IFIP WG 6.1 International Conference, DAIS 2014, Held as Part of the 9th International Federated Conference on Distributed Computing Techniques, DisCoTec 2014, Berlin, Germany, June 3-5, 2014, Proceedings*. Ed. by K. Magoutis and P. R. Pietzuch. Vol. 8460. Lecture Notes in Computer Science. Springer, 2014, pp. 126–140. DOI: 10.1007/978-3-662-43352-2_11. URL: https://doi.org/10.1007/978-3-662-43352-2%5C_11 (cit. on p. 17).

[3] C. Baquero, P. S. Almeida, and A. Shoker. "Pure Operation-Based Replicated Data Types". In: *CoRR* abs/1710.04469 (2017). arXiv: 1710.04469. URL: http://arxiv.org/abs/1710.04469 (cit. on p. 17).

[4] A. Barreto et al. "PS-CRDTs: CRDTs in highly volatile environments". In: *Future Generation Computer Systems* 141 (2023), pp. 755–767. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2022.12.013. URL: https://www.sciencedirect.com/science/article/pii/S0167739X22004186 (cit. on p. 17).

[5] C. C. Baseca et al. "An IoT service-oriented system for agriculture monitoring". In: *IEEE International Conference on Communications, ICC 2017, Paris, France, May 21-25, 2017*. IEEE, 2017, pp. 1–6. DOI: 10.1109/ICC.2017.7996640. URL: https://doi.org/10.1109/ICC.2017.7996640 (cit. on p. 2).

[6] S. Burckhardt. "Principles of Eventual Consistency". In: *Found. Trends Program. Lang.* 1.1-2 (2014), pp. 1–150. DOI: 10.1561/2500000011. URL: https://doi.org/10.1561/2500000011 (cit. on pp. 14, 16).

[7] U. Drolia et al. “Cachier: Edge-Caching for Recognition Applications”. In: *37th IEEE International Conference on Distributed Computing Systems, ICDCS 2017, Atlanta, GA, USA, June 5-8, 2017*. Ed. by K. Lee and L. Liu. IEEE Computer Society, 2017, pp. 276–286. DOI: 10.1109/ICDCS.2017.94. URL: https://doi.org/10.1109/ICDCS.2017.94 (cit. on pp. 18, 22, 23).

[8] *Ericsson Mobility Visualizer - Mobility Report*. Nov. 2022. URL: https://www.ericsson.com/en/mobility-report/mobility-visualizer (cit. on pp. 1, 2).

[9] D. Guo et al. “HDS: A Fast Hybrid Data Location Service for Hierarchical Mobile Edge Computing”. In: *IEEE/ACM Trans. Netw.* 29.3 (2021), pp. 1308–1320. DOI: 10.1109/TNET.2021.3058401. URL: https://doi.org/10.1109/TNET.2021.3058401 (cit. on pp. 18, 21).

[10] H. Jacobsen. “Topic-based Publish/Subscribe”. In: *Encyclopedia of Database Systems*. Ed. by L. Liu and M. T. Özsu. Springer US, 2009, pp. 3127–3129. DOI: 10.1007/978-0-387-39940-9_1208. URL: https://doi.org/10.1007/978-0-387-39940-9%5C_1208 (cit. on p. 8).

[11] G. Klas. “Edge Computing and the Role of Cellular Networks”. In: *Computer* 50.10 (2017), pp. 40–49. DOI: 10.1109/MC.2017.3641649. URL: https://doi.org/10.1109/MC.2017.3641649 (cit. on p. 2).

[12] A. van der Linde, J. Leitão, and N. M. Preguiça. “Δ-CRDTs: making Δ-CRDTs delta-based”. In: *Proceedings of the 2nd Workshop on the Principles and Practice of Consistency for Distributed Data, PaPoC@EuroSys 2016, London, United Kingdom, April 18, 2016*. Ed. by P. Alvaro and A. Bessani. ACM, 2016, 12:1–12:4. DOI: 10.1145/2911151.2911163. URL: https://doi.org/10.1145/2911151.2911163 (cit. on p. 17).

[13] N. Mittal and F. Nawab. “CooLSM: Distributed and Cooperative Indexing Across Edge and Cloud Machines”. In: *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*. IEEE, 2021, pp. 420–431. DOI: 10.1109/ICDE51399.2021.00043. URL: https://doi.org/10.1109/ICDE51399.2021.00043 (cit. on pp. 18, 19).

[14] F. Nunes. “Armazenamento Reativo e Persistente para Ambientes Mobile Edge Computing”. MA thesis. NOVA School of Science and Technology, NOVA University Lisbon, 2022 (cit. on pp. 3, 6, 7, 10).

[15] P. E. O’Neil et al. “The Log-Structured Merge-Tree (LSM-Tree)”. In: *Acta Informatica* 33.4 (1996), pp. 351–385. DOI: 10.1007/s002360050048. URL: https://doi.org/10.1007/s002360050048 (cit. on p. 19).

[16] C. Pereira. “ Dynamic Content-based Indexing in Mobile Edge Networks”. MA thesis. NOVA School of Science and Technology, NOVA University Lisbon, 2021 (cit. on pp. 3, 12).

[17] R. Potharaju et al. "Helios: Hyperscale Indexing for the Cloud & Edge". In: *Proc. VLDB Endow.* 13.12 (2020), pp. 3231–3244. DOI: 10.14778/3415478.3415547. URL: http://www.vldb.org/pvldb/vol13/p3231-potharaju.pdf (cit. on pp. 18, 21).

[18] N. M. Preguiça. "Conflict-free Replicated Data Types: An Overview". In: *CoRR* abs/1806.10254 (2018). arXiv: 1806.10254. URL: http://arxiv.org/abs/1806.10254 (cit. on pp. 14–17, 36).

[19] P. Read and M.-P. Meyer. *Restoration of motion picture film*. Elsevier, 2000 (cit. on p. 53).

[20] P. Sanches et al. "Data-Centric Distributed Computing on Networks of Mobile Devices". In: *Euro-Par 2020: Parallel Processing - 26th International Conference on Parallel and Distributed Computing, Warsaw, Poland, August 24-28, 2020, Proceedings*. Ed. by M. Malawski and K. Rzadca. Vol. 12247. Lecture Notes in Computer Science. Springer, 2020, pp. 296–311. DOI: 10.1007/978-3-030-57675-2_19. URL: https://doi.org/10.1007/978-3-030-57675-2%5C_19 (cit. on pp. 3, 6, 10).

[21] M. Shapiro et al. "Conflict-Free Replicated Data Types". In: *Stabilization, Safety, and Security of Distributed Systems - 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings*. Ed. by X. Défago, F. Petit, and V. Villain. Vol. 6976. Lecture Notes in Computer Science. Springer, 2011, pp. 386–400. DOI: 10.1007/978-3-642-24550-3_29. URL: https://doi.org/10.1007/978-3-642-24550-3%5C_29 (cit. on pp. 15, 16).

[22] J. Silva. "Data Storage and Dissemination in Pervasive Edge Computing Environments". PhD thesis. Universidade NOVA de Lisboa, 2021 (cit. on pp. 7, 8).

[23] J. A. Silva, P. Vieira, and H. Paulino. "Data Storage and Sharing for Mobile Devices in Multi-region Edge Networks". In: *21st IEEE International Symposium on "A World of Wireless, Mobile and Multimedia Networks", WoWMoM 2020, Cork, Ireland, August 31 - September 3, 2020*. IEEE, 2020, pp. 40–49. DOI: 10.1109/WoWMoM49955.2020.00021. URL: https://doi.org/10.1109/WoWMoM49955.2020.00021 (cit. on pp. 3, 6–9, 11).

[24] J. A. Silva et al. "It's about Thyme: On the design and implementation of a time-aware reactive storage system for pervasive edge computing environments". In: *Future Gener. Comput. Syst.* 118 (2021), pp. 14–36. DOI: 10.1016/j.future.2020.12.008. URL: https://doi.org/10.1016/j.future.2020.12.008 (cit. on pp. 3, 6, 8, 11).

[25] J. A. Silva et al. "Time-aware reactive storage in wireless edge environments". In: *MobiQuitous 2019, Proceedings of the 16th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services, Houston, Texas, USA, November 12-14, 2019*. Ed. by H. V. Poor et al. ACM, 2019, pp. 238–247. DOI:

10.1145/3360774.3360828. URL: https://doi.org/10.1145/3360774.3360828 (cit. on p. 8).

[26] A. Taivalsaari and T. Mikkonen. "On the development of IoT systems". In: *Third International Conference on Fog and Mobile Edge Computing, FMEC 2018, Barcelona, Spain, April 23-26, 2018*. IEEE, 2018, pp. 13–19. DOI: 10.1109/FMEC.2018.8364039. URL: https://doi.org/10.1109/FMEC.2018.8364039 (cit. on p. 2).

[27] J. Xie et al. "COIN: An Efficient Indexing Mechanism for Unstructured Data Sharing Systems". In: *IEEE/ACM Trans. Netw.* 30.1 (2022), pp. 313–326. DOI: 10.1109/TNET.2021.3110782. URL: https://doi.org/10.1109/TNET.2021.3110782 (cit. on pp. 18, 19, 23).