José Pedro Serra Afonso

Bachelor in Computer Science and Engineering

Key-Value Storage for handling data in mobile devices

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática

Orientador: Hervé Miguel Cordeiro Paulino, Assistant Professor, NOVA University of Lisbon

Júri

Presidente: Paulo Orlando Reis Afonso Lopes
Vogais: João Coelho Garcia
Hervé Miguel Cordeiro Paulino

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

December, 2019

Acknowledgements

I would like to thank my thesis advisor, Hervé Paulino, for his availability and his continuous constructive feedback throughout this dissertation, and João Silva, for his constant support and presence throughout this whole work. I would also like to thank the DeDuCe Project (PTDC/CCI-COM/32166/2017) for allowing me to contribute to their impressive ongoing Edge investigation work, and for providing me with the required software to do so.

Finally, a special thanks to my family, who have blindly supported my education and always believed in me, and to my friends, who've motivated me to pursue my objectives and never give up.

Abstract

In the current era of technology, computers have shrinked to the point that more than half of the world population always carries one with them - their mobile devices. These are used in all sorts of different activities, constantly generating information that needs to be stored or processed somewhere. To cope with the huge amounts of data generated by all of these devices, applications have resorted to Cloud services to provide them with the much needed computational and storage resources, but as these remote infrastructures still represented a bottleneck communication wise, a new paradigm has been emerging, Edge Computing. Instead of processing and storing all the data in more distant cloud services, the data is spread among mobile devices and edge servers connected in a shared network.

In order to fully take advantage of the low latency times experienced in the Edge, applications still needed a distributed storage edge-oriented system, capable of handling the contents generated by all of these mobile devices. The current state-of-the-art storage systems are able to provide these applications with a storing platform that uses mobile devices or edge servers as data storing points, but neither uses both.

In this thesis we propose a Key-Value Edge Storage System named Basil, that uses both mobile devices and edge infrastructures as nodes of the system, capable of providing users from different locations with a cohesive and consistent distributed storage system. Furthermore, we will test our KV store against existing NoSQL storage models deployed in the edge, as well as its own performance while varying the number of nodes it relies on.

Keywords: Mobile Edge Computing, Key-Value, NoSQL, Peer-to-peer, Infrastructure, Distributed Storage

Resumo

Desde o aparecimento do primeiro computador, a tecnologia tem evoluído a um ritmo exponencial. Com estes avanços, os computadores tem ficado cada vez mais pequenos, até que hoje em dia, mais de metade da população mundial transporta sempre um consigo - os seus telemóveis. Para lidar com a vasta quantidade de dados gerados por todos estes dispositivos, as aplicações móveis recorreram a serviços *Cloud* para lhes fornecerem os recursos computacionais e de armazenamento necessários. Como estas infra-estruturas remotas ainda representam um entrave a nível de comunicação, um novo paradigma tem emergido, Computação na "Edge". Em vez de processar e guardar os dados em serviços *Cloud* longínquos, os dados estão espalhados pelos telemóveis e servidores na "Edge" que partilham a mesma rede.

Para aproveitar os baixos tempos de latência experienciados na "Edge", as aplicações ainda necessitam de um sistema distribuído de armazenamento orientado para a mesma, capaz de lidar com os conteúdos gerados por tantos dispositivos. Alguns sistemas de armazenamento, como o Cassandra ou o Krowd, são capazes de fornecer às aplicações uma plataforma de armazenamento que usa ou os dispositivos, ou os servidores na "Edge" como pontos de armazenamento, mas nenhum usa os dois.

Nesta tese propomos um sistema de armazenamento *Key-Value* orientado para a "Edge", construido em cima do Thyme, que usa ambos os telemóveis e servidores na "Edge" como agentes do sistema, capaz de fornecer a utilizadores de diferentes regiões um sistema de armazenamento distribuído coeso e consistente. Ademais, iremos testar o nosso sistema KV contra outros modelos de armazenamento *NoSQL* orientados para a "Edge", tal como a própria performance do nosso sistema em função dos seus nós de suporte.

Palavras-chave: *Mobile Edge Computing*, Key-Value, NoSQL, ponto-a-ponto, Infraestrutura, Armazenamento distribuído

Contents

List of Figures xiii

List of Tables xv

Listings xvii

1 Introduction 1

1.1 Context and Motivation . . . 1
1.2 Mobile Edge Computing . . . 2
1.3 Problem . . . 3
1.4 Proposed Solution . . . 4
1.5 Document Structure . . . 4

2 Thyme 5

2.1 Overview . . . 5
2.2 Thyme's API . . . 6
2.3 Operations . . . 7
2.3.1 Publish Data . . . 7
2.3.2 Subscription and Data Retrieval . . . 8
2.4 Replication . . . 9
2.5 Namespaces . . . 10
2.6 Thyme-GardenBed . . . 10
2.7 Using Thyme for our Work . . . 11

3 State of the Art 13

3.1 NoSQL . . . 13
3.1.1 NoSQL Models . . . 14
3.1.2 Why the Key-Value Model . . . 14
3.1.3 Comparison Dimensions . . . 15
3.2 NoSQL in Mobile Edge Networks . . . 19
3.2.1 Krowd . . . 19
3.2.2 Cassandra . . . 21
3.2.3 Ephesus . . . 22

3.3 Discussion 24

4 Basil **27**

4.1 Overview 27
4.2 Basil's API 28
4.3 Basil over Thyme 30
4.4 Managing the Keys 30
4.5 Hierarchical Namespaces 31
4.6 Operations 32
4.6.1 Inserting a new Value 32
4.6.2 Retrieving Values 33
4.6.3 Removing Values 34
4.6.4 Deleting a Key 35
4.7 Changes to Thyme 35
4.7.1 Linking and Unlinking existing objects 35
4.7.2 Enforced Removal 38
4.7.3 Unpublishing multiple objects 39
4.7.4 Status Update upon joining a new Cell 39
4.8 Maintaining Consistency 40
4.9 Implementation Details 41
4.9.1 Protocol Buffers 41
4.9.2 Searching for Objects Locally 42
4.9.3 Notifications Policy 42
4.10 Summary 43

5 Case Study: Class Quiz **45**

5.1 Overview 45
5.2 Application Flow 46
5.2.1 Professor operations 46
5.2.2 Students Operations 50
5.3 Summary 51

6 Evaluation **53**

6.1 Evaluation Methodologies 53
6.1.1 Absolute Evaluation 54
6.1.2 Comparing against Cassandra 54
6.2 Evaluation Environment 54
6.2.1 Simulation Case-Study 55
6.3 Absolute Evaluation Results 57
6.4 Comparing against Cassandra 60
6.5 Summary 62

7 Conclusions **65**
7.1 Conclusion 65
7.2 Future Work 66

Bibliography **67**

List of Figures

1.1 Mobile Edge Computing Architecture. Taken from [1] 2

2.1 Example of Thyme's publish and subscribe operations. The hash of the tags ("beach"and "summer") associated with the published image ("beach.jpg"), will determine the cells responsible for its metadata management. Taken from [29]. 8
2.2 Thyme's subscription notification & data retrieval process. Taken from [29]. 9
2.3 Overlapping Thyme *Namespaces*. Taken from [29]. 10

3.1 Wide-Column Store Data Model. 14
3.2 Document-Store Data Model 15
3.3 Key-Value Data Model. 15

4.1 Basil's proposed Architecture. 31
4.2 Thyme's Architecture. Adapted from [29] 31
4.3 Retrieval and publication of keys in the system. 32
4.4 Order in which the metadata filter and the selection algorithm are applied to the list of matched metadata files upon a subscription. 34
4.5 Example interaction between nodes upon a link operation of an object to the Tag B, previous already associated with the Tag A 36
4.6 Alternative explored for the implementation of the *link* operation using the same example from the Figure 4.5 37
4.7 Interaction between nodes upon an unlink operation of an object with the Tag B, previously associated with the Tags A and B 38
4.8 Interaction between nodes in the system, upon an ensured *unpublish* operation is issued. 39
4.9 Concurrent Put and Delete calls to the same Cell. 40
4.10 Concurrent put operations to the same key. 41
4.11 Inconsistent get operations to the same key. 41

5.1 Initial Application Home Screen. 46
5.2 Authentication with the university Gmail account. 46
5.3 Creation and Publication of a Question. 47
5.4 List of existing Areas in the System. 48

5.5 List of published questions under the Area "Matemática". 48
5.6 Creation of a Quiz. 49
5.7 List of the user's Quizzes. 49
5.8 Scanning a QR Code containing the Quiz's Key. 50
5.9 Screen upon starting an ongoing Quiz. 50
5.10 List of Quizzes done by a Student. 51

6.1 Example of a block of trace actions with a quiz action, its time span, the executing node, and the action's inputs. 55
6.2 Multiple mobile devices running Basil being simulated within a single machine 56
6.3 Multiple mobile devices being simulated within a single machine interacting with a Cassandra instance on another machine. 56
6.4 Number of messages received by the Nodes. 59
6.5 Workload inferred by Basil. 60
6.6 Cassandra Server measured metrics 61
6.7 Total number of bytes circulating through the system 62

List of Tables

3.1 Edge Stores comparison. 25

6.1 Load and Distribution of Active Messages 58
6.2 Load and Distribution of Passive Messages 58

Listings

Chapter 1

Introduction

1.1 Context and Motivation

In the last 20 years, the number of mobile devices has been escalating worldwide, reaching a density of 128 devices per 100 persons in developed regions [19]. These have been increasingly "smarter", and are no longer just a communication device, but a small portable computer which has become the world's favourite device to interact with the digital world.

With so many devices, capable of employing an horde of different applications, the amount of data being generated everyday has been growing accordingly, obliging mobile applications to adopt Cloud services [5], to help them cope with their storage needs. This continuous communication between mobile devices and remote data centers resulted in a high and expensive network load for mobile network operators. This has affected greatly affected online applications, where having low latency times is crucial to provide a satisfying and appealing experience to the users.

In an IoT era, where everyday object has been "smartified", the conditions to explore a more local and collaborative approach to Mobile Computing were significant enough to try a different approach. A new architecture, Mobile Edge Computing, which is capable of replicating the storage and data management services offered by the centralised Cloud servers, but without resorting to remote data centers. Using the local infrastructures to offer the same services, where mobile devices are also able to participate and support Edge applications, the mobile base stations' network load is greatly reduced, and the user's experience is greatly improved.

This new advances have motivated a change of paradigm, where the potential and boundaries for Mobile Edge Computing are still being explored, and a lot of challenges yet to be overcomed.

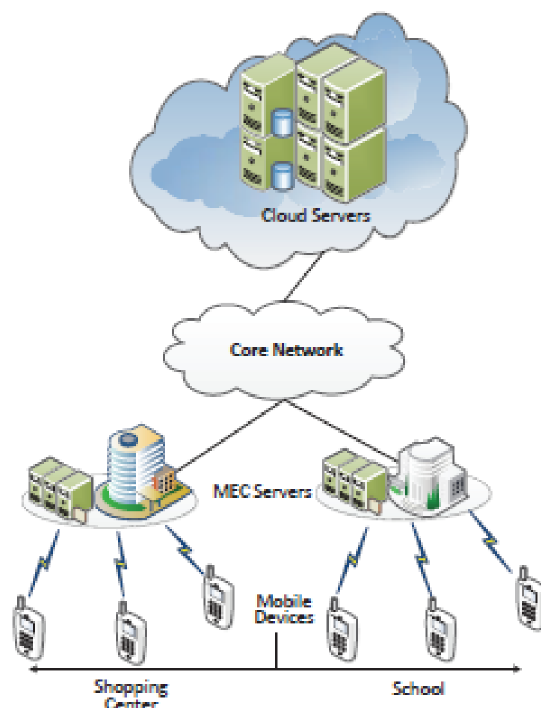

Figure 1.1: Mobile Edge Computing Architecture. Taken from [1]

1.2 Mobile Edge Computing

Mobile Edge Computing is an emerging technology that addresses the high network load experienced by mobile end users. It does so by shifting computational tasks from the data centers to a more local layer known as the Mobile Edge. The Mobile Edge has two main components (illustrated on the Fig. 1.1): Edge devices, that include all type of devices (both mobile phones and IoT devices) connected to a network; and the Edge Cloud, located in mobile base stations, which are responsible for controlling the network traffic, and are capable of hosting Cloud like applications.

There are mainly two approaches to this concept.

- **Offloading through the Edge**: Focuses on aggregating all the traffic directed towards Cloud services, and either forwards it to remote infrastructures or to Cloudlets, small-scaled cloud data centers composed by one or more computers deployed in the Edge Cloud. In doing so, the overall latency is decreased, as the distance between the device users and the Edge Cloud is smaller than the distance between device users and infrastructures Cloud, the consumption of bandwidth is reduced, and the response times are shortened, as the requests are now handled by more capable infrastructures.

- **Computing in the Edge**: On the other hand, this paradigm keeps the computational and storage tasks it in the Edge, by distributing the data among the local mobile

devices and Edge infrastructures.

This concept displays of the following advantages:

- **Deployment Agility**: Developers can quickly develop edge applications and deploy them where needed. Edge applications program the device to operate in the way each customer needs. For example, they can be programmed to only operate in the local network but a hybrid approach with computation done at Cloud infrastructures is also a viable solution.

- **Low response time**: As the data doesn't have to be deployed to infrastructure Clouds but rather distributed among the Edge Nodes, the response time from the system is greatly decreased.

- **Lower operating expense**: Although local network traffic is increased, network bandwidth is conserved, by processing the data locally instead of sending it to Cloud infrastructures.

Communication between Edge Nodes is essential, and it can be achieved using local network technologies, e.g., Wi-Fi, Wi-Fi Direct, Bluetooth and ad-hoc [9], or a combination of any of these [22, 24, 26]. Mobile nodes can either communicate directly with one another, through an access point, or they can communicate via their supporting Edge servers, when they're in different geographical regions.

Although this approach is not possible in some contexts, when the users don't share the same base station, the situations when they do, e.g., stadiums, concerts or university's campus, are worth trying for. The high density of mobile devices and the capable Edge servers can cooperate to form a powerful pool of resources, capable of producing reduced latency times, and an highly available distributed network of data storing replicas.

1.3 Problem

The problem that this thesis is addressing is the lack of storage support for mobile edge applications, capable of supporting large clusters of users in the same geographical space, who want to share content among them. Typical scenarios include venue events such as football games or music concerts, where the users could benefit greatly if they queried both each other for the desired contents, and Edge Infrastructures for popular items or content from another venues.

Most of the existing Edge Storage systems don't take advantage of the full possibilities that exist in the whole Edge layer, using only Edge Infrastructures to support their system, like Cassandra [15], or only a small number of mobile devices, like Krowd [10].

Although it is growing, the lack of research in this area, specially within NoSQL models, has been an obstacle when searching for the appropriate techniques to deal with the following challenges:

- How to distribute the network load in an even and fair way to the users?
- How to grant consistent data in a highly distributively Edge system?
- How to adapt the Key-Value data model to the Edge paradigm?

In the context of this thesis we addressed both the delete operation, and the global list of keys in an Edge environment.

1.4 Proposed Solution

In this thesis, we propose a Key-Value storage framework for mobile Android clients, that offers its users an highly available Edge Storage System, with an eventual consistency level and a persistent data model design.

Using the work developed by Pedro Vieira in his thesis dissertation [29], which includes the Thyme incorporation of infrastructure servers, and changes to the Thyme client, we propose a Key-Value abstraction layer, any possible necessary refinement updates to the existing client, and some small changes on the infrastructure component to cope with our system operations.

Moreover, besides the typical Key-Value operations, we also intend to implement some additional methods, in order to take full advantage of Thyme's architecture. To support all of them, we will need to keep track of the existing keys in the system, which we plan to do so, by keeping a shared collaborative key index on all the participating nodes of the system. Given this proposition, the challenges we expect to face are the following:

- How to partition the key space evenly amongst the system nodes?
- How to deal with popular keys hotspots?
- How to maintain and propagate a global consistent key index in an Edge environment?
- How to implement a typical KV delete operation in an Edge environment?

1.5 Document Structure

Following this Chapter, we present an overview of the Thyme system on Chapter 2. Afterwards, in the Chapter 3, we introduce the most relevant aspects that concern our work, and current state-of-the-art technologies and frameworks that make an attempt to solve the problem exposed in the Section 1.3. On the Chapter 4, we present our thesis work, followed by its case-study application on Chapter 5. Finally, we present the evaluation to which our thesis work was submitted on Chapter 6, followed by the conclusions derived from this work on Chapter 7.

Chapter 2

Thyme

In this chapter we start by presenting an overview of the Thyme system in the Section 2.1, followed by the list of its available operations in the Section 2.2, and their described implementation on the Section 2.3.

We continue this chapter by explaining Thyme's replication mechanisms on the Section 2.4, defining a *Namespace* on Thyme's framework in the Section 2.5, and how Thyme interacts with Edge Infrastructures on Section 2.6.

Finally, we close this chapter with the Section 2.7, where we detail the features already offered by Thyme that we used in our work, and the features we needed to cope with our solution.

2.1 Overview

Before starting the implementation of our Key-Value Storage, we must first understand the layer that it will be built upon, Thyme. A topic-based time-aware publish/subscribe system, designed specifically for mobile edge networks, that relies on peer-to-peer communication between mobile devices to form a persistent collaborative storage system. It is a unique system by two different reasons: its mobile edge oriented environment and its ability to refer to subscriptions within a time scope that can refer to the past.

Thyme is a distributed system, without a central unit to manage all the nodes and therefore requires a strong collaboration between the nodes to sustain a feasible system. In order to do so, all the nodes in the system share the same responsibilities and have no particular roles, meaning that they can either be a publisher, a subscriber or both.

2.2 Thyme's API

Along this dissertation, multiple references will be made about Thyme's operations, as they mainly compose most of the operations developed for this thesis work. In the following list, we present a description of Thyme's available methods signature.

`publish(dataItem, tags, description, opHandler)`

This method publishes objects in the system

`dataItem` – Object being published.

`tags` – Set of *tags* being associated with the object.

`description` – Description of the object, e.g. a small thumbnail or a brief text describing the inserted object.

`opHandler` – Implementation of the behaviour to be executed upon the operations' success or failure.

`subscribe(tags, startTime, endTime, notHandler, opHandler)`

This method allows users to issue subscriptions.

`tags` –*Tag* or set of *tags* being subscribed.

`startTime` – Starting time of the subscription's lifetime.

`endTime` – Ending time of the subscription's lifetime.

`notHandler` – Implementation of behaviour to be executed when notification is received from this subscription.

`opHandler` – Implementation of behaviour to be executed upon the operations' success or failure.

`unPublish(objectId, opHandler)`

This method unpublishes an object from the system.

`objectId` – Object's id being *unpublished*.

`opHandler` – Implementation of behaviour to be executed upon the operations' success or failure.

`download(metaData, dowHandler)`

This methods allows users to download items published in the system.

`metaData` – Object's metadata being downloaded.

`dowHandler` – Implementation of behaviour to be executed upon the operations' success or failure.

2.3 Operations

In this section we will be going through Thyme's operations most relevant for our work as we will be using them to compose our own storage's operations. So, it is crucial that we understand their syntax, behaviour and purpose.

2.3.1 Publish Data

The publish operation inserts a data object into the system under one or more topics, which are the object's associated tags. After a publish operation, a new metadata item associated with the inserted object is generated and spread by the publisher with the following fields:

- id_{obj}: The published object's identifier;
- T: A set of topics or tags for which the object will be stored under;
- s: A summarised representation of the shared object, e.g., a thumbnail of an image in the case of a photo sharing network;
- ts^{pub}: The object's publication timestamp to guarantee temporal perception;
- id_{owner}: The publisher's node identifier.
- L_{rep}: The set of all nodes that are in possession of this object, containing only the publisher node and his cell's neighbours at the time of the creation (explained with more detail at 2.4).

With the topology provided by the Cluster-Based Hash Table (CHT), this metadata item is then distributed to the cells responsible for the given tags in T, accordingly to the hash produced by each one of them as it is illustrated by the Figure 2.1. The selected cell will store permanently the metadata item, while the actual data object stays in the publisher's cell nodes. As it is the metadata that's being spread, instead of the object itself, the system's bandwidth and overall network resources are spared resulting in an overall more efficient system.

At the time of the writing of this document, Thyme only allowed read operations on the published objects, meaning that it wasn't possible to edit or update the objects already on the system. Ongoing work is addressing the ability to alter pre-published contents.

2.3.1.1 Unpublish Data

To remove values from Thyme, the authors implemented the inverse operation of *publish*, the *unpublish*. The same actions are replicated, but instead of adding new object, an existing one is removed. Upon an *unpublish* operation, the relevant object is no longer

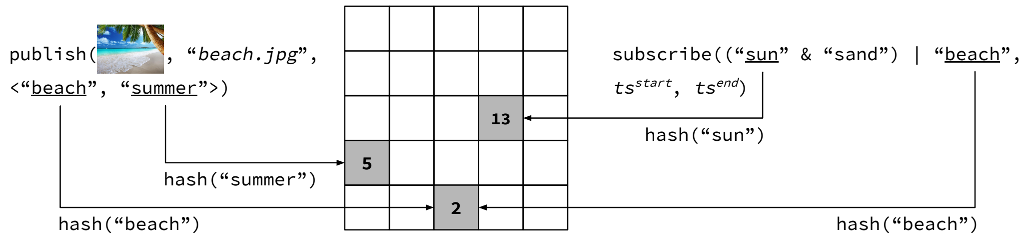

Figure 2.1: Example of Thyme's publish and subscribe operations. The hash of the tags ("beach"and "summer") associated with the published image ("beach.jpg"), will determine the cells responsible for its metadata management. Taken from [29].

included in incoming subscription requests, only being available for download if the downloading user has received a notification from previous subscription requests.

The metadata files from the object are deleted from the respective *tags* cells. The object itself is deleted from the active replicas, but not from the passive ones, who downloaded the object on purpose. Still, the object won't be available for future download requests, as its metadata files were indeed removed.

2.3.2 Subscription and Data Retrieval

Users can subscribe to existing tags by sending a message to the network containing the subscription metadata with the following fields :

- id_{sub}: The subscription identifier;
- q: A query's logic formula, where keyword conjunctions and disjunctions are allowed;
- ts^{s}: The initial timestamp of published data to be retrieved, where past references are allowed. To get all the data from the beginning of the system related to the specified query, users must specify with the parameter 0;
- ts^{e}: The timestamp's upper bound of published data to be retrieved, where future references are allowed. To receive all future data matching the query, ∞ should be used;
- id_{owner}: The requesting user identifier;
- $cell_{\mathrm{owner}}$: The identifier of the logical cell where the user is located.

From q, Thyme deduces what cells from the CHT shall receive the subscription request (as seen in the Figure 2.1), in a minimising way in order to reduce the amount of packets circulating through the system. Upon receiving the request, the nodes within the deduced cells are now responsible for storing and managing the active subscription list within the given time bounds, ts^{s} and ts^{e}, notifying the users with object's metadata when one is published, (as it is illustrated in Fig. 2.2). When the notified users wish to download a

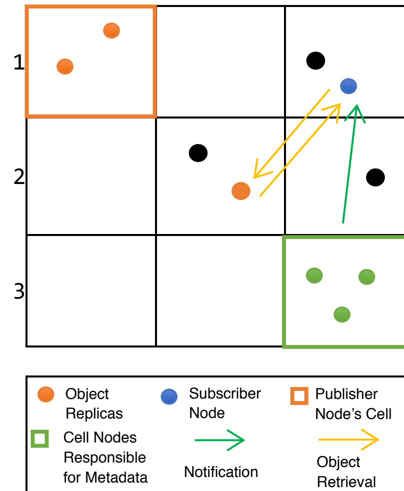

Figure 2.2: Thyme's subscription notification & data retrieval process. Taken from [29].

certain data item, they must send a data retrieval message to one the replicas, according to a replica selection algorithm, in the L_{rep} field. If the closest replica doesn't reply within a given threshold, possibly for going offline or leaving the network, the user's client will keep iterating the list of replicas, until one of them provides the requested data. After this, the receiving node is now part of the L_{rep} of the downloaded object.

2.4 Replication

Replication is a sensitive matter in edge peer-to-peer systems as it represents a trade-off between saving user's storage space and granting them high availability. Thyme's approach has both in consideration with the following mechanisms:

- **Active Replication**: After a publish operation, the new data object is broadcasted by its author to other nodes inside the same cell. This allows them to also reply to data retrieval requests for that object, relieving the author's network load and maintaining the object in the system in case the author leaves the network;

- **Passive Replication**: Users who download items also act as replicas by providing the remaining users with data retrieval points outside of its original publisher's cell. Once again, this contributes for the item's durability in the system and relieves the original cells from data retrieval requests.

The list of replicas for a particular object is saved on its metadata under the L_{rep} field, where each replica is identified by its id_{node} and $cell_{\text{node}}$.

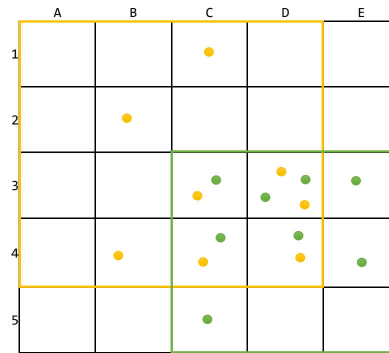

Figure 2.3: Overlapping Thyme *Namespaces*. Taken from [29].

2.5 Namespaces

A "*Namespace*" in Thyme's universe is defined as a domain for an employing application. Thyme allows for multiple, even overlapped, *Namespaces* to coexist in the same area without any conflicting issues, like synchronisation and shared resources problems, but can only belong to one *Namespace* at a time.

When initiated, Thyme can either join an existing *Namespace*, by scanning the packets being transmitted in the network, (e.g. "hello"messages), for a predefined amount of time, or, start his own *Namespace*. To do so, the user must specify multiple parameters such as its name, and how long it should be active. If he does, a unique identifier is generated, and a cluster based cell is created to accommodate the *Namespace*'s incoming mobile nodes.

Nodes that belong to different overlapped networks can collaborate between themselves to provide the whole system an higher availability and performance. These nodes will be able to store data objects and metadata from different *Namespaces* but can only reply to data requests from nodes that reside where the requested item lies (Fig. 2.3).

2.6 Thyme-GardenBed

Thyme-GardenBed [29] incorporates infrastructure servers, deployed in base stations located in the Edge Cloud layer (in the Fig.1.1), to complement the existing system composed of mobile nodes. These servers are ideally connected through wired means among themselves, where each one is responsible for managing a basic service set in different areas, thus allowing groups of clients from different locations to share data items between them. These nodes are also able to communicate with cloud services or external databases for long-term persistent storage or analytic purposes, if the infrastructure engineer decides to.

To sustain this multi-region system, this component is divided into two distinct pieces: the client and the server. The client shares subscription responsibilities with the local cells, and extends the subscription requests to all the other infrastructure servers in the

system. On the other hand, the server manages local subscription requests and download operations, and also processes incoming subscription requests from other infrastructure servers, replying them with the relevant local popular items from the current infrastructure.

To accommodate these functionalities, Thyme-Infrastructure's storage layer employs the following caching mechanisms:

- **Local Popularity Cache**: Stores the most popular items from the instances that are running on the current region. This allows for the local clients to download these objects directly from the infrastructure, instead of inquiring their neighbors for a replica, resulting in an overall more efficient and faster process. This cache will also be used to serve the subscription requests from other remote clients.

- **Prefetch Cache**: Cache that contains metadata and data items originated from remote clients. These items are periodically disseminated by other infrastructures, so that the current infrastructure local clients can access items from other regions.

- **Global Cache**: This cache moves the items that were initially in the *Prefetch Cache* and were later considered relevant (i.e. specifically downloaded) by at least one mobile client of the current infrastructure's node. The purpose of this cache, is to not overload the *Prefetch Cache* with irrelevant items, and preserve the relevant ones durability in the system. This way, future local users can discover and access items from other parts of the system.

2.7 Using Thyme for our Work

Thyme already offers the core features one needs in any Edge Storage System. Mechanisms like its active and passive replication techniques, its node clustering algorithm, and its *namespace* management are essential to any proper Storage System oriented for the Edge.

Besides its core features, Thyme also offers the main operations that any storage system must have: the ability to insert, retrieve and remove values from its system.

All these indispensable aspects were extended to our solution, but some Key-Value related features were still required. Features like the ability to inspect the current list of *tags* in the system, the ability to associate or disassociate *tags* from an existing object or the ability to query items from a given *tag*. Nevertheless, all of these details are explained with more detail on the Section 4.

Chapter 3

State of the Art

This chapter serves to give the readers an overall perception of the NoSQL landscape, with special attention to the Edge systems. We start with an overview of the database model, followed by the definition of the seven dimensions that characterize a NoSQL system in the Section 3.1.3. Along these dimensions, we exemplify with different techniques employed by current state-of-the-art technologies. On the Section 3.2, we study some Edge Storage systems, in order to learn their approaches to the previous defined dimensions. Finally, on the Section 3.3, we present some final remarks and conclusions from this research.

3.1 NoSQL

NoSQL is a Database Model that became popular in the beginning of the 1990's [16], where the traditional Relational Database (RDB) model could no longer sustain the great amount of data being generated by the emergence of the Internet in a viable way. This was due to the limits of vertical scaling of the data nodes and the complexity of horizontal scaling these same nodes in an efficient way. As RDBs were not designed to work in a distributed environment, the cost of join and transaction operations in this environment were too costly resulting in a decreased performance.

Instead of storing the data in a rigid structured way, the NoSQL model stores it in an adaptable and unstructured way that thrives in distributed environments due to its low degree of dependence between data records. In this era of IoT, where information is constantly being generated by so many different devices, the need to cope with it in an efficient way is essential. The NoSQL model is able to do so, by abstracting join operations from the storage level [7], as they are often too expensive, and leaving them

Row A	Column 1	Column 2	Column 3	Column 4
	Value	Value	Value	Value

Row B	Column 1	Column 2	Column 3	Column 4
	Value	Value	Value	Value

Figure 3.1: Wide-Column Store Data Model.

to the application level where the data can either be joined or denormalized. The first approach requires gathering data from several physical nodes so then we are able to join them, this results in an excessive effort for most contexts in decentralized systems. Denormalization on the other hand, involves replication of the data, or parts of it, in different physical nodes, which allows optimized queries on the denormalized attributes but also raises inconsistency problems between nodes holding the same attributes.

3.1.1 NoSQL Models

There are three main ways to prescribe a data layout for the stored data, that will dictate the replication and partition policies for the system:

- **Wide-Column or Column Families Model**: The data is stored by column (Fig. 3.1) and not by row as in the RDB model. With this independence between attributes of a single row, it is possible to apply data compression algorithms per column and distribute columns that are not often queried together across different nodes. Some notable Wide-Column stores are Amazon DynamoDB [3], Apache Cassandra [15] and Bigtable [6].

- **Document Oriented Model**: The data is stored in a document-oriented fashion (Fig. 3.2), where a document is a series of fields with attributes identified by an ID. This allows queries on the document's fields.

- **Key-Value Model**: The data is saved under a binary relation (Fig 3.3) between a Key and a set of Values, where the Key will be the unique identifier for the multiple Values. This allows for a distribution of the keys among a set of physical nodes.

3.1.2 Why the Key-Value Model

Our choice favours the Key-Value model, because this model thrives on Edge environments not only for its simple data model, allowing data items to be identified by a single

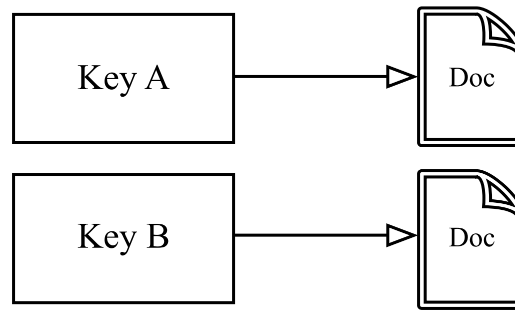

Figure 3.2: Document-Store Data Model

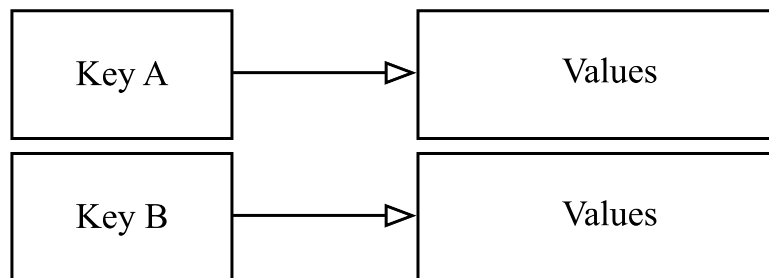

Figure 3.3: Key-Value Data Model.

key, but also for having the nature of the stored values transparent to the database, making data queries operations efficient, and the partition of they keys flexible.

This also reflects on the simple operations this model has to offer:

- *Get(key)*: Given a key, the database returns the values stored associated with the given key.

- *Put(key, value)*: Given a key and a value, the database either stores the value under the new key, or, if the key already exists, joins the new value to the list of values associated to the given key.

- *Delete(key)*: Given a key, the database deletes all the values associated to it from the system. This implementation may vary from case to case, as it can be a very destructive operation in a shared database.

3.1.3 Comparison Dimensions

Along this dissertation, we will compare our proposed Key-Value storage with other NoSQL storages using several dimensions. Along the next sections, we'll be defining these dimensions, where we'll refer some technologies and common techniques that are relevant for our work, and conclude them with the approach employed by our foundation layer - Thyme.

3.1.3.1 Persistence

This dimension refers to method of storing data employed by a system, to ensure that it remains accessible until it's deliberately removed. Some ways to do so include the use of indexes, files, databases and distributed file systems. Another alternative is to keep data in RAM and periodically make snapshots of them to persistent media.

Thyme does it by actively replicating its own data items (detailed in the section 2.4), making sure they are available until their owner decides otherwise. On the edge server, the data is also cached, and may be directed to external cloud databases if the system engineer desires to, enforcing even more the data's persistence in the whole system.

3.1.3.2 Replication

It's a technique used to guarantee high availability and durability, which consists on having data replicated on multiple physical nodes. It guarantees high availability because if the original node fails, the database can still provide the client with the desired data, which may also mean an increase of performance if writing or reading is permitted on the replicas, relieving some of the load on the original node. One of the most common examples of replication is the Master-Slave mechanism, where the Master is the original node that receives all the write operations and replicates them to its Slaves, the backup nodes. When writing operations are allowed on the Slaves nodes, the mechanism is called Master-Master. Depending on the replication technique being used, different levels of consistency can be guaranteed.

As seen in the Section 2.4, Thyme already addresses this dimension by actively and passively replicating the data upon its publication.

3.1.3.3 API

Here we just want to compare the type of programming interface used to access the database. For NoSQL databases, the most common is the HTTP protocol but there are other alternatives, such as database's own native clients in certain programming languages.

Although Thyme already has its own API, listed in the Chapter 2, our implementation will have a different API. Besides the typical Key-Value operations, we will also need a method to list all the keys in our storage. All of these will be implemented by calls to Thyme's API.

3.1.3.4 Implementation Language

This dimension refers to the programming language that the database is implemented with. In some cases, it may affect the developer's decision based on a required technology or personal preference.

Thyme was designed to be incorporated into any Android application, where the official language of development is Java, and so was the choice of the Thyme's authors.

3.1.3.5 Consistency

In a database there is a need to cope with concurrent reads and/or writes to be able to provide the clients with valid and integrate data. Of course, in a distributed database, maintaining Consistency is much more complex, and therefore more valuable, especially when performance remains unaltered. Two of the main types of consistency when handling a distributed database are:

Strong Consistency, which can be achieved by broadcasting all the data changes to all the replicas as soon as a write request is handled by one of them. All subsequent read/write request are put on hold while this process is being handled to ensure its atomicity. This is probably one of the most costly and inflexible techniques to achieve consistency, but a viable one in databases where data integrity cannot be risked.

And **Eventual Consistency**, which on the other hand is a more relaxed technique but with weaker consistency guarantees. Write requests are propagated to other replicas eventually, usually in periodic times, and don't prevent the replicas from handling other requests. It's a viable technique for databases that prioritise Availability over Consistency.

Other known Consistency models preserved in NoSQL stores include: Casual Consistency, Per-object Timeline Consistency and Parallel snapshot isolation [8].

Enforced by the Edge architecture, Thyme's communication model is incapable of coping with a state of Strong Consistency, as it would require long synchronised broadcast messages constantly being transmitted through the whole system. We will be exploiting Eventual Consistency techniques, what implications they might have on the system's metadata and what scenarios we must prepare for.

3.1.3.6 Sharding

To have an efficient distributed database, it must be able to partition the data among the computing nodes in a viable way, this is described as the Sharding technique. Using this technique, different data subsets are assigned to different nodes which allows the database to distribute the load among them. There are 3 main Sharding techniques [12]: Range-Sharding, Hash-Sharding and Entity-group Sharding.

Range-Sharding is done by partitioning data into ordered and contiguous value ranges, allowing the system to perform efficient search tasks. To do so, the system needs a master to manage assignments in a coordinated way as some shards may end more overloaded than others, so the system must be able to detect and resolve these types of situations. This technique is usually supported by Wide-Column stores like BigTable [6], HBase [13] or Hypertable [14] and Document stores like MongoDB [20], RethinkDB [21] and DocumentDB [2] [12].

Another technique to partition the data amongst various physical nodes is Hash-Sharding, where data is assigned to a shard according to a hash built from the primary key. This technique has the advantage of not requiring a master node to regulate the Sharding process and if the hash function produces an even distribution, it also guarantees an even distribution of the data amongst the shards. The lack of a master node though, results in unfeasible scans and thus only allowing lookups. Usually the shard assignment is determined by *serverid = hash(id) mod servers*, but in a system where nodes are constantly leaving and joining the system this can be a problem, as every time it happens, the data records need to be reassigned. A solution to this problem is to use consistent hashing [17] for the shard assignment, where records are not directly assigned to nodes but instead to logical partitions, thus allowing a greater flexibility for the database, as not all the data has to be reassigned when a node leaves or joins the system. This technique is used in Key-Value stores and in some Wide-Column stores like Cassandra [15] or Azure Tables [4] [12].

Finally, there's Entity-group Sharding, where the data partitioning is entity driven. There will be multiple partitions, where which one will be responsible for an entity or a group of entities. The sharding process can be either determined by the application (e.g. in G-Store [11] and Megastore [18]) or derived from transaction's access patterns (e.g. in Relational Cloud and Cloud SQL Server)[12]. Transactions between entity shared groups are optimized, but different entity-groups transactions require more complex transaction protocols.

As seen in the Chapter 2, Thyme employs a consistent hashing scheme by partitioning the data amongst a cluster of logical cells, based on the hash produced by the data's tags. However, the existing scheme doesn't take into account the workload that each cell is responsible for, resulting in an unfair high network traffic for the cell's respective mobile devices. Further in this document, we will explore the possibilities for partitioning an existing popular cell, in order to balance the workload sustained by all the devices.

3.1.3.7 Query Method

As the name suggests, this dimension looks to compare the methods to access the database and the list of different ways of accessing these methods through the database API. The methods are highly dependable from the database Sharding model, its data model and consistency guarantees.

Existing query methods range from basic primary key lookups supported by every NoSQL model, to complex filter queries, which are only supported by some models where the values stored aren't transparent to the database, e.g. Document-stores.

Although Thyme already has its own data retrieval operations, once again described with more detail in Chapter 2, these will have to be adapted to function within the Key-Value Model.

3.2 NoSQL in Mobile Edge Networks

In this section we will be exploring existing NoSQL models deployed at the edge layer. We will be iterating through their dimensions to see their approach in this type of environment, learn from them and later compare own Key-Value Storage's techniques with the ones studied in this section.

3.2.1 Krowd

Krowd [10] is a decentralized distributed Key-Value store framework for applications that share content between users connected in a network under the same access point. It was developed for users who are attending live events which impacted the authors design decisions. Krowd uses a distributed hash table (DHT) to store the key-value pairs amongst the nodes in the network, which uses a consistent hashing scheme to route operations based on the given key. It doesn't store the actual data but rather its URI (Universal Resource Identifier) under a key, which can be attributes of the content e.g. tags.

Besides the user's mobile devices and the network access points, no other infrastructures are used to support this system whose architecture is composed by the following three modules :

- **Discovery and Communications**: Responsible for announcing a device's presence, discovering other devices in the proximity, and communicating with them.
- **KRoute**: Given a key and data, this module is responsible for routing the data to the correct device. To do so, it uses *highest random weight hashing* which consistently maps the key to a unique device from the devices in the network.
- **KVStore**: Mounted on top of the KRoute component, this module handles remotes procedure calls (RPC) to the system. It also contains a local hash table to store the key-value pairs.

3.2.1.1 Persistence

Krowd is referred by its authors as a temporary infrastructure (since it was designed for live events) and as a result, its Persistence guarantees are rather weak. Whenever a new data item is inserted into the system, no events are triggered to preserve its durability and even though the new item can be downloaded by other users, they can't serve as data retrieval points for the downloaded item. Thus if a user shares an item and then leaves the network, this item won't be downloadable anymore.

3.2.1.2 Replication

Krowd doesn't employ any Replication technique. Users that download items from the network can't be denominated as replicas as they don't receive neither read or write requests.

3.2.1.3 API

Krowd's **KVStore** layer, which is responsible for handling remotes procedure calls (RPC) to the system, implements two operations:

- *Put(key, value)*: When the application stores a key-value pair, the **KVStore** module creates and serializes a RPC *put* object which is sent to the device responsible for the given key via the KRoute. When this device receives the RPC *put* object, its **KVStore** extracts the key-value pair and stores it in its local hash table.

- *Get(key)*: When the application requests a key, the **KVStore** module generates a *get* RPC object which is routed to the correct device via the **KRoute**. On the correct device, when **KVStore** receives this object, it extracts the key, checks with its local hash table, and responds with the associated value, a list of URIs.

3.2.1.4 Implementation Language

There is no reference to Krowd's implementation language in the authors paper[10].

3.2.1.5 Consistency

Krowd doesn't support neither delete or edit operations on the stored data, so the only consistency issues that may arise are when a *put* operation is performed on an existing key and when an user that performed a *put* operation leaves the network. When the first one happens, the new data's URI is appended to list of values associated with the given key on the device responsible for it. At the time of the developed solution, Krowd wasn't ready to accommodate churn (as it is noted by its authors in [10]), and so, no events are triggered upon the exit of a user from the network.

Krowd's consistency politics are eventual but its guarantees are rather weak.

3.2.1.6 Sharding

The **KRoute** module is responsible for partitioning the data amongst the devices in the network and as it was mentioned in 3.2.1, it uses a *highest random weight hashing* (also known as *rendezvous hashing*) scheme to do so. Each key is consistently mapped to a unique device from the list of all devices provided by the **Discovery and Communications** module. No actual data is distributed amongst the devices but rather the data's URI, used to locate the data's original publisher node.

3.2.1.7 Query Method

Users can query over the contents under a given key (by submiting a *get* operation), to which the system will reply with a list of URIs for those same contents. The URIs are then used to contact the devices that have the requested contents.

3.2.2 Cassandra

Cassandra [15] is a Distributed Wide-Column Store for managing very large amounts of structured data spread out across many infrastructure servers, while granting its clients with a highly available system with no single point of failure. Its data model can be described as a collection of tables, where each table is a distributed multi dimensional map indexed by a key.

Due to its locality awareness, as it tries to find the closest replica to deliver users with requested data, Cassandra can be a viable storage system for Edge environments.

3.2.2.1 Persistence

Cassandra approaches Persistence in two different ways:

- **Local Persistence**: Before writing the data to a disk, Cassandra registers the write operation into a commit log that provides durability and recoverability to the given data.

- **System Persistence**: To prevent data from getting lost, in case its original node fails, Cassandra also replicas the data amongst a number of servers configured by the system engineer.

3.2.2.2 Sharding

Cassandra partitions the data amongst its nodes by using consistent hashing, which selects a node from a fixed circular "ring", based on the hash produced by the data's key. The selected node is now the coordinator node for the given key, and all the keys that are in the region between it and its predecessor node in the "ring".

3.2.2.3 Replication

Cassandra allows for a custom configuration of the replication management. The number of hosts N that replicate each data item can be specified, which will be managed by the item's coordinator node. Besides replicating the data items, the coordinator node is also responsible for replicating the keys that fall within its range on N - 1 nodes in the "ring". Cassandra also provides the system engineer with three different replication policies: "Rack Unaware", "Rack Aware"and "Datacenter Aware".

3.2.2.4 Consistency

The consistency level granted by Cassandra is also user defined, and is impacted by the replication configuration chosen. The different consistency levels vary mainly the number of replicas for which the read and/or write requests are propagated to, which will directly impact the availability of the system and the data accuracy.

3.2.2.5 API

The Cassandra API consists of the following three simple methods:

- *insert(table, key, rowMutation)*
- *get(table, key, columnName)*
- *delete(table, key, columnName)*

Where *table* and *key* identify a row, and the *columnName* selects the desired field from the requested row.

3.2.2.6 Implementation Language

Cassandra was originally developed for Java environments, but has since been adapted to run on Python, C++, Go and Node.JS.

3.2.2.7 Query Method

Cassandra's only query method is the *get* operation, which retrieves a row's *column* identified by the row's *key* in the given *table*. Due to the sharding scheme employed, this operation is resolved on a one-hop lookup as all the nodes are aware of the system's topology.

3.2.3 Ephesus

Ephesus [23] is a decentralized distributed Key-Value store designed for mobile devices, that requires no other computing and/or storage resources besides those provided by the devices participating in the system. No Internet access is required to support this system, it only depends on the participating interconnected mobile devices. The authors classify it as an ephemeral system because it only exists as long as there are devices supporting it. Ephesus focus on providing its users with a available and churn resistant system, with a strong effort on data persistence as well.

3.2.3.1 Persistence

Due to the free movement of the mobile nodes, keeping a data item in this system may present a challenge. The network's topology may vary and some of the nodes may even temporarily fall outside the network's range. To make sure that the desired data items are still accessible within the system, Ephesus replicates these items on several devices also impacting positively the system's overall performance.

3.2.3.2 Replication

Ephesus employs a popularity-aware replication mechanism, combining active and passive replication, that prioritizes the most accessed data items granting them a higher replication level. Each data item has a responsible node that tracks the amount and the locations of the available replicas in the system, balancing the item's passive and active replicas. If the number of nodes passively replicating a data item (by downloading it) starts increasing, the responsible node will start to decrease the number of active replicas. On the other hand if the number of passive replicas starts decreasing, the number of active ones has to be raised.

3.2.3.3 Sharding

Ephesus sharding scheme is supported by a Distributed Hash Table, that allows nodes to have a partial knowledge of the network's topology and to store their own contents, reducing the overall broadcast search requests done in the system.

Replication wise, it would be unfair for the responsible nodes to handle all the replicas of a key. So, to balance the workload with the other nodes, the index of the values associated with a certain key are partitioned into smaller partial indexes, which its responsibility is shifted to other nodes. Whenever the partial indexes reach their fixed size limit, these are locally cached, allowing other peers to only keep requesting the last non-full partial index.

3.2.3.4 Consistency

The contents stored in the system are immutable, but their metadata is not. This design decision had some impact on the metadata's consistency to which the authors prepared for in the following scenarios: Whenever an *index* operation is issued, Ephesus needs to update its locally cached partial indexes in order to know which one it will insert the new metadata object. Since the partial indexes are cached, published metadata cannot be removed. This raises a problem since the removal of published contents is allowed, removing the data item from the DHT but keeping the corresponding entry in the index. To solve this situation, the system publishes a tombstone metadata object for the removed content in the next non-full partial index, guaranteeing the system an eventual consistency. Other relevant situations concern the *publish* and *delete* operations that are not executed atomically, leaving the index in an inconsistent state in case the relevant peers crash midway through the operation. The index is eventually consistent again, as the locally cached indexes are updated when actions are prompted over the relevant data items.

3.2.3.5 Query Method

To provide users with contents of the system, Ephesus has a *get* function which works in the following way: first the system checks if it is possible to serve the current operation with the local storage. If not, the request is routed to the DLS for a system-wide search. If the requested file is found, it is then locally stored by the requesting client.

3.2.3.6 API

Ephesus's supported operations are the following:

- *put(key, data)*: Stores *data* in the system with the identifier *key*;
- *get(key)*: Retrieves the data item identified by *key* if it's still in the system;
- *remove(key)*: Deletes the content associated to the given *key*, if it exists and was published by the issuing user;
- *list()*: Returns the list of contents currently published in the system;
- *index(metadata)*: Adds *metadata*, which consists of a *key* to the associated data and a summary of the data, to the system-wide index provided by the *list()* operation;
- *publish(key, data, metadata)*: Combines both *put* and *index* operations into one with their respective arguments;
- *delete(key)*: Calls both *remove* and *index* operations with the given parameters;

3.2.3.7 Implementation Language

Ephesus was implemented in *Java*.

3.3 Discussion

Throughout this chapter, besides defining the dimensions that characterise a NoSQL Storage, we performed a survey containing state-of-the-art NoSQL Storages that share some features with our envisioned system, compared in the Table 3.1. Unlike ours, none of them use mobile devices and Edge infrastructures simultaneously as nodes of the system, but we could still gather some relevant information from their approaches to the defined dimensions.

In Chapter 4, we present an overview of our system, our available operations and how we implemented them, the consistency issues that arose during our work, and other implementation details.

	Data Model	Supporting Nodes	Persistence	Sharding	Replication	Consistency	Query Method	Multiple Regions	Locality Awareness
Krowd	Key-Value	Mobile Devices	Local Storage	Rendezvous Hashing	*None employed*	*None guaranteed*	Get	No	No
Cassandra	Wide-Column	Edge Servers	Local Storage, Replication	Consistent Hashing	Ring (next N - 1)	User defined	Get	Yes	Yes
Ephesus	Key-Value	Mobile Devices	Local Storage, Replication	Consistent Hashing for Popular Tags, Active and Passive Replication for the data	Active, Passive	Eventual (metadata)	Get, List	No	No
Our Solution	Key-Value	Edge servers, Mobile Devices	Local Storage, Replication	Consistent Hashing for Popular Tags, Active and Passive Replication for the data	Active, Passive	Eventual (metadata)	Get, List, GetAll	Yes	Yes

Table 3.1: Edge Stores comparison.

Chapter 4

Basil

In this chapter we will present our proposed solution Basil, starting with a general overview of our system's features, followed by a detailed list of its available operations. On the Section 4.3, we explain how we integrate the Thyme system to fit our system's needs, and how we use it to handle Basil's *keys* on the Section 4.4. Over the Sections 4.6 and 4.7, we detailly explain our operations, and the modifications done to the current Thyme system to cope with Basil's needs. We close this chapter by examining possible inconsistent situations in our system on the Section 4.8, and list some of the most relevant implementation details on the Section 4.9.

4.1 Overview

Basil is a fully collaborative Edge Key-Value Store, designed for mobile devices in the same edge network who want to share any type of content between them. The data resides on the actual devices, and its persistence is guaranteed by Basil through passive and active replication mechanisms. Basil also accommodates users freedom of movement, and supports their departure from the system, making a strong active effort to assure no data is lost between these processes. As a collaborative system, all the mobile nodes in the system act like storing points, compelled to answer to read and write requests towards their stored objects, and to help other nodes integrate the system.

Although in this first approach the objects in the system are immutable, their metadata files aren't. The objects already published can still be linked to new keys or unlinked from previously associated keys. In order to manage the objects' metadata, the nodes are clustered, and each cluster is then responsible for handling the operations directed towards a group of keys.

We have also decided to introduce the concept of hierarchical namespaces to Basil, as we felt that it would broaden the system's features, thus making it more appealing for possible employing applications. For example, one powerful possibility would be an application capable of simulating a file system oriented for the Edge, where each directory is abstracted into a *key* at the Basil layer.

Besides interacting with each other, the mobile nodes can also communicate with infrastructures located at the Edge. The edge nodes act as central storing points for the most popular items, and can also communicate with another remote infrastructure nodes to broaden the system's geographical range. This allows nodes from different venues, but from the same *Namespace*, to share content between them. The infrastructure nodes can also make use of *Cloud* services to store files in a more persistent manner, or for other computing services.

The mobile nodes communicate directly with each other, via *Peer-to-peer* communication technologies, such as Bluetooth and Wi-Fi Direct, or indirectly through access points (via Wi-Fi).

4.2 Basil's API

Basil was built to integrate another system, just like we use Thyme, or to be directly implemented in a mobile application. And as so, in order to facilitate its integration, we will present a brief description of its available operations.

`put(key, value, opHandler)`

This method publishes an object in the system.

`key` – *Key* with which the object is being inserted under.

`value` – Object being inserted into the system.

`opHandler` – Implementation of the behaviour to be executed upon the operations' success or failure. In case of success, this is where the inserted object's id is retrieved from.

`putInAll(root, value, description, opHandler)`

This method publishes an object under all the *keys* derived from a given *root key*.

`root` – Root to be interpreted as a group of *keys* where the object is being inserted under.

`link(key, objectId, opHandler)`

This method links an existing object with a ney key.

`objectId` – Id of the object being linked.

`key` – New *key* being linked to the object.

`linkInAll(root, objectId, opHandler)`
This method links an existing object with a *root key*.

`root` – Root to be interpreted as a group of *keys* to which the object is being linked to.

`get(key, downloadHandler, filter)`
This method retrieves all the values published under a key.

`key` – *Key* from where the values are being retrieved.

`downloadHandler` – Implementation of the behaviour to be executed upon the downloads success or failure.

`filter` – Implementation of the metadata filter being applied to the matches values, before being sent back to the issuing user.

`getFromAll(root, downloadHandler, filter)`
This method returns all the values published under a directory.

`getUntil(key, endTime, downloadHandler, filter)`
This method returns the all the metadata files published under a key until a given timestamp.

`endTime` – Superior timestamp limit for the incoming metadata files.

`downloadHandler` – Implementation of the behaviour to be executed upon the downloads success or failure.

`remove(objectId, operationHandler, fromAllKeys)`
This method removes an object from the system.

`objectId` – Id of the object being removed.

`operationHandler` – Implementation of behaviour to be executed upon the operation success or failure.

`fromAll` – Flag indicating if the object should be removed from all the tags in system.

`unlink(objectIdr, key, operationHandler)`
This method unlinks a key from an object.

`key` – Key being dissociated from the object.

`unlinkFromAll(objectIdentifier, root, operationHandler)`
This method unlinks all the keys derived from a directory from an object.

`root` – Root to be interpreted as a group of *keys* to which the object is being unlinked from.

`delete(key, operationHandler, fromAll)`

This removes all the values published under a key by the issuing user.

`key` – Key being deleted from the system, as well as the values associated to it.

`operationHandler` – Implementation of behaviour to be executed upon the operation success or failure.

`fromAllKeys` – Flag indicating if the relevant values should be removed from the given key or permanently erased from all the keys in the system,

`listKeys()`

This method returns the global list of keys.

4.3 Basil over Thyme

Still, Basil is just a Key-Value Interface Layer that makes the bridge between the employing application and the Thyme system, thus making use of the latter's Storage, Replication and Routing mechanisms (explained in the Chapter 2 and illustrated on Fig. 4.1 and 4.2) to conduct its own operations.

Basil adapts Thyme's Publish/Subscribe data model, by transforming *tags* into *keys*, and shapes Thyme's message-based architecture by immediately triggering all the notification messages in a transparent way to its users, thus providing them with an abstraction of an asynchronous Key-Value database. Besides these adaptions, some new operations were added to the current Thyme system in order to comply with the Key-Value model, such as the ability to link and unlink a Tag from an existing object, which are detailly explained in the sections below. To avoid *ghost* objects in the system, i.e. objects without tags (and therefore unreachable), the original object's cell started being responsible for keeping track of their objects tag counter.

Regarding Thyme's time-awareness subscription properties, we have decided to not include them in Basil's retrieval operations. This is because we wanted to preserve the Key-Value paradigm as an asynchronous storage system, and abstract it from its foundation layer, a message-based synchronous system. And as so, all the subscribe operations issued at the Thyme layer always have a time window between $-\infty$ and the current moment.

4.4 Managing the Keys

In order to fully take advantage of the Key-Value data model, one should be able to see all the existing *keys* in the system before performing any operation. In order to achieve this, as the current Thyme version doesn't disseminate *tags*, we decided to reserve a *tag*, "KEYS", just to store our *keys*.

As soon as Basil is fully initialised, it executes a private operation, *list()*, which consists of a *subscribe* operation to the reserved tag *"KEYS"*, with a time span of [- ∞,+∞]. Upon

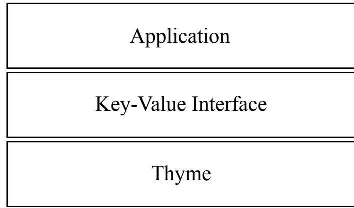

Figure 4.1: Basil's proposed Architecture.

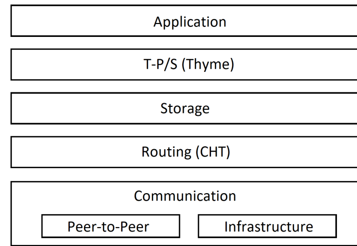

Figure 4.2: Thyme's Architecture. Adapted from [29]

receiving any notification, the notification handler is programmed to extract the *key* from the received metadata files description field, and store it in Basil's local *key* set. By having this permanent subscription, we prevent duplicates from existing on the global list, inspecting our own local list before publishing a new *key*.

Regarding the list's contents, everytime a *put* or a *link* operation is issued, the user's local *key* list is queried for the inserted *key* to check if an extra *publish* operation is necessary. The inverse verification is also made upon a *remove*, *unlink* or *delete* operation, issuing an extra *unpublish* operation if the user has no more objects published under a *key* that he published on the global list in the first place.

To minimise the overhead induced by this mechanism, the *key* objects are simple strings, stored as the description of a null object. This way, the *List* operation is a 1-hop operation, (instead of a 2-hop), allowing the clients to communicate directly with the responsible cell to retrieve the global list, as it is illustrated on the Fig, 4.3.

At the infrastructure level, we have decided to instantly promote the KEYS to a popularity level, such that, it is always broadcasted between different venues from the same world. In the end, we have a global list of *Keys* from all the Basil instances, with a eventual consistency level.

Also, in another attempt to reduce the number of duplicates on our global *key* list and keep it precise, an extra unpublish operation is issued to remove the given *key* from the list, if the user has published it on the global *key* list in the first place.

4.5 Hierarchical Namespaces

An hierarchical namespace, (not to be confused with Thyme's *Namespace*), is the set of leaves contained in a *root key*. For example, the leaves contained in the *root key* *"FCT\DI\AC\2019"* are the following set of *keys* : [*FCT\DI\AC\2019*, *FCT\DI\AC*, *FCT\DI*, *FCT*].

These namespaces are able to introduce an hierarchy concept to Basil, where *keys* can either be a folder or a sub-folder. In order to cope with this abstraction, we implemented the following operations: *putInAll*, *linkInAll*, *unlinkFromAll* and *getFromAll*. For each operation, the given *root key* is interpreted as a hierarchical namespace, deriving the

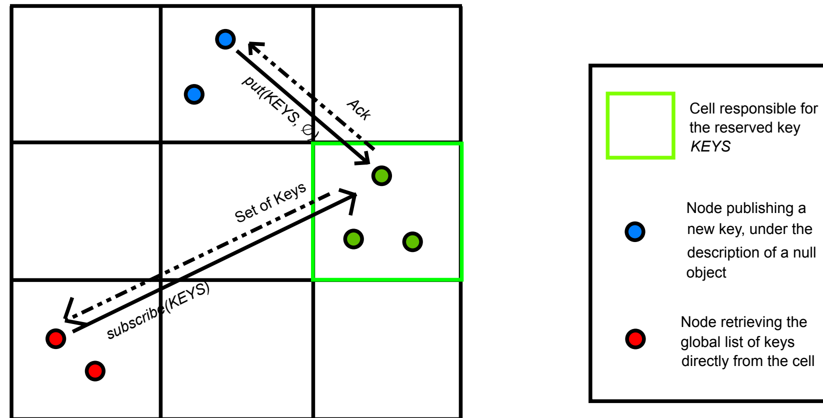

Figure 4.3: Retrieval and publication of keys in the system.

respective *key leafs*, which are then passed on to the generic operations and are interpreted as a normal set of *keys*.

4.6 Operations

The operations implemented in Basil are what really defines our work. In the following sections, a detailed analysis will be presented for each operation, as well as their biggest implementation challenges, changes to Thyme's existing operations and messages, and new operations added.

4.6.1 Inserting a new Value

Besides the typical Key-Value *put* operation behaviour, where a new object is inserted into the system, our implementation hides another secondary mechanism to accommodate its P/S foundation layer.

This operation starts by issuing a *publish* operation in Thyme with the inputs passed to the *put* operation. Upon its successful completion, if the given *key* doesn't exist in the user's local *key* list, a second *publish* operation is issued to the global *key* list. In case the second *publish* operation fails, up until two extra *publish* operations are issued to the global *key* list until one completes successfully.

In case the object being published is smaller than a stipulated threshold, we have decided to store it as a description of its own metadata file. Just like the scenario described on the Fig. 4.3, this optimisation allows for the direct download of the object from its *keys* cell, being stored within its own metadata file. This way, the workload imposed by Basil onto its nodes is decreased, as the download operation now only requires one *op* instead of two, and no active replication mechanisms are necessary to persist the small object published.

The operations is considered unsuccessful if the publication of the *value* fails, or if the publication of the new *key* fails after three tries.

4.6.1.1 Linking an existing Value with a new Key

If an object already exists in the system, and we just want to associate it with a new set of *keys*, there is no need to republish it. Just like the operation *link* in the operative system *Unix* [28], where an existing file can be linked to a *pointer* file, we *link* an existing object with new *keys* instead of publishing it again.

The *link* operation can either be called explicitly by the programmer, or it can be called inside the *put* operation if the user has already published or received the inserted object. We only check if the object exists locally, because the cost of searching for an object in a distributed Edge network like Thyme would be unbearable. It would require querying through all the cells in the network for the given object, with the possibility of extending the search to another venues. Not only would this congest the network with heavy queries, it would also delay *link* and *unlink* operations, forcing the users to wait until the search be completed.

4.6.2 Retrieving Values

As in any normal Key-Value database, the values from a given *key* can be retrieved through the *get* operation. Ours is no different. The operation *get* starts by issuing a *subscribe* operation at Thyme, automatically triggering all the received notifications, downloading the objects as soon as their metadata files are received. But if the objects are small enough to be included in their own metadata files, their respective notifications aren't triggered, and the object is extracted directly from the respective metadata.

We've added the possibility to perform selective queries through the filtering of the matched metadata files at the *keys* cell. The filter can either be directly applied to the matched metadata files based on a metadata attribute (e.g. the description field), or it can be a selection algorithm (e.g. random selection). If both are applied, they follow the order demonstrated on the Figure 4.4.

As of right now, there are two possible selection algorithms:

- **Random Selection**: Where a random number, or a specified number of files are randomly selected from the matched metadata files of a subscription request.

- **Partial Selection**: Where the first *n* published objects are selected from the matched metadata files of a subscription, where *n* is a number specified by the user.

To take advantage of Thyme's time-awareness subscription properties, we've included the *getUntil* operation that allows users to handle directly the matched metadata files from objects yet to be published. Referring to future objects can be a powerful feature, and we didn't want to miss out the opportunity to include it in our system's abilities.

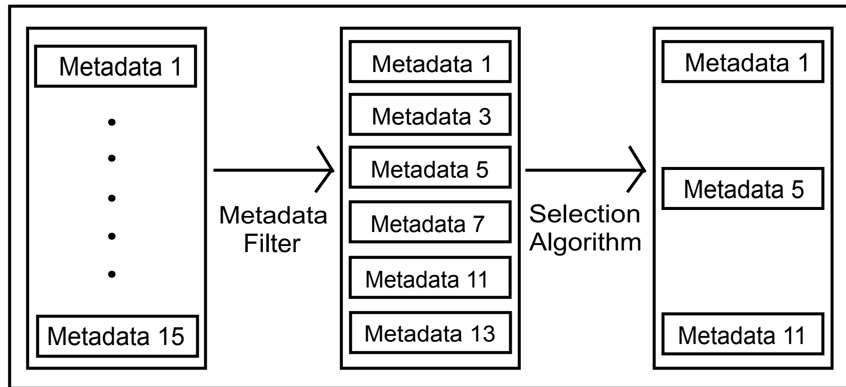

Figure 4.4: Order in which the metadata filter and the selection algorithm are applied to the list of matched metadata files upon a subscription.

To cap of our retrieval operations, we conclude with the *getFromAll* operation, which as mentioned in the Section 4.5, retrieves all the matched objects published under the *leaves keys* derived from a given *root key*.

4.6.3 Removing Values

Removing a value in a typical distributed Key-Value database, erases it from circulating in the respective network. In Basil, this behaviour is similarly replicated with the *remove* operation.

Underneath, an *Unpublish* operation is issued at the Thyme level, which deletes all the metadata files associated to the object identified by the *objectId* in its initial linked tags cells, preventing the object from being included in future subscriptions. But, if the same object has already been linked to new *keys* by another user, that same object will still be available for download under those new *keys*.

If the object's owner wants to indeed remove the object from the entire system, the actual list of *keys* associated must be retrieved first, to issue all the necessary *unpublish* operations. As this assertion has an higher communication cost comparing to a normal *remove* operation, we have decided to make it optional by including a *fromAllKeys* flag in the *remove* operation's signature. This way, the users can remove their own objects from the *keys* they associated it with, or they can ensure that the object is properly deleted from the system.

4.6.3.1 Unlinking an existing Value from a Key

Just like we made it possible to link an existing value to a new *key*, we also made its inverse operation possible, unlink an existing value from an already associated *key*. But unlike the *link* operation, where any user can link an existing object with any *key*, the *unlink* operation can only be performed by the object's owner, or if the unlinked *keys* were linked by the issuing user in the first place.

4.6.4 Deleting a Key

Typically, on a Key-Value database, a *delete* operation erases all the values associated with a given *key*. But we thought that this behaviour would be too destructive on an edge collaborative storage like Basil, as it would allow users to eliminate any value in the system, even if they weren't the original publishers. And so, as a compromised solution, our *delete* operation only removes the values from a given *key* that were published by the issuing node in the first place.

Just like in the *remove* operation, where the users can ensure that the objects are indeed removed from the system, the same decision is granted in the *delete* operation. In order to so, before issuing the *delete* message, the node requests for the actual list of *keys* from the objects being deleted, to assure that all their metadata files are indeed erased from the system.

4.7 Changes to Thyme

In order to support Basil's operations, new ones were added to Thyme's API, and some of the existing operations were slightly modified. These not only required the introduction of new messages and data structures to Thyme's system, it also called for a revision over some of the existing ones.

Over the following subsections we'll be detailly explaining the newly added changes, as well as their purpose for our solution.

4.7.1 Linking and Unlinking existing objects

Until our implementation, Thyme didn't support the ability to add or remove *tags* from a published object. In order to so, first we needed some kind of structure to keep track of the object's set of *tags*. And so, to the storage layer we added a new structure, a *Map<ObjectId, Set<Tag>> tagsCounter*, responsible for keeping track of the objects' tags, maintained by the object's original cell. Whenever a new node enters a cell, it automatically starts replicating this structure, which is transferred by its new cell's peers. We chose the object's original cell to keep track of its list of linked *tags* because, by placing the object's *tag* counter closest to its original source, we are reducing the traffic between different cells. To optimise the initial structure propagation, if the object is being actively replicated, (which is also done at the publisher's cell as previously stated in the Section 2.4), we merge the active replication message with object's tag counter initial message.

To Thyme's *publish* operation, we've also added a *boolean* flag, indicating if a *tag* counter should be kept for the object being published, preventing it from being linked or unlinked with further *tags*. This was useful for our reserved *tag "KEYS"*, whose *key* objects won't be associated with any other *tag*, and allows the responsible nodes to have a reduced workload.

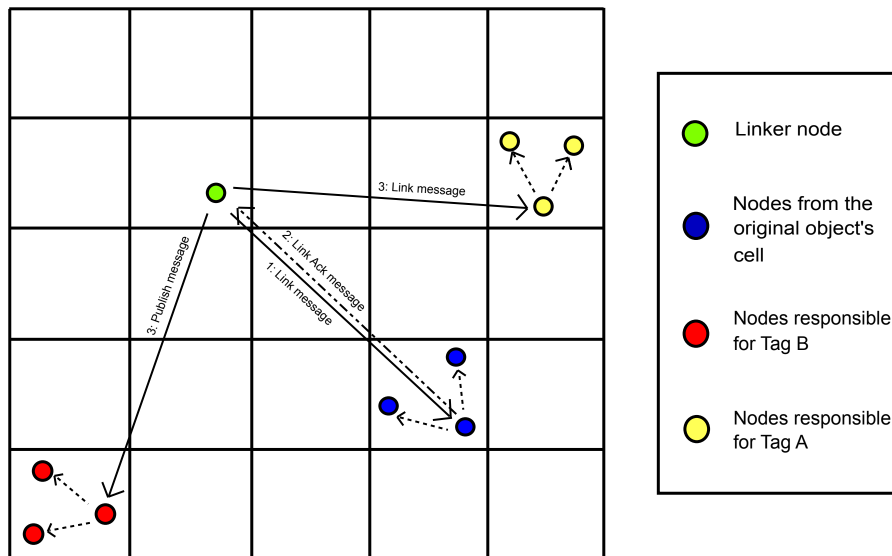

Figure 4.5: Example interaction between nodes upon a link operation of an object to the Tag B, previous already associated with the Tag A

4.7.1.1 Link Implementation

The *link* operation starts with the linking node sending a *link* message to the original object's cell containing the following fields:

- A *NamespaceId* identifying the Namespace where the object resides;
- An *ObjectId* identifying the object to link;
- A set of *tags* to be added to the object's metadata.

On the original cell, upon receiving this link message, the receiving node will both update its respective object's tag counter and broadcast this message to its cell's peers so they can do the same. After this, the receiving node will send back an acknowledgement link message containing 4 items:

- The relevant object's identifier;
- The new *tags* linked to the object;
- The already associated *tags*;
- And the correct updated metadata for the relevant object.

Now, the first node is responsible for both rerouting the link message to the previously associated tags cells, where the receiving nodes will update their object's metadata, and issuing a publish message with the newly updated metadata to the object's new tags.

On the Fig. 4.5, it is illustrated the interaction between nodes from different cells, when an object that was already associated with the *tag* A is being linked with the *tag* B.

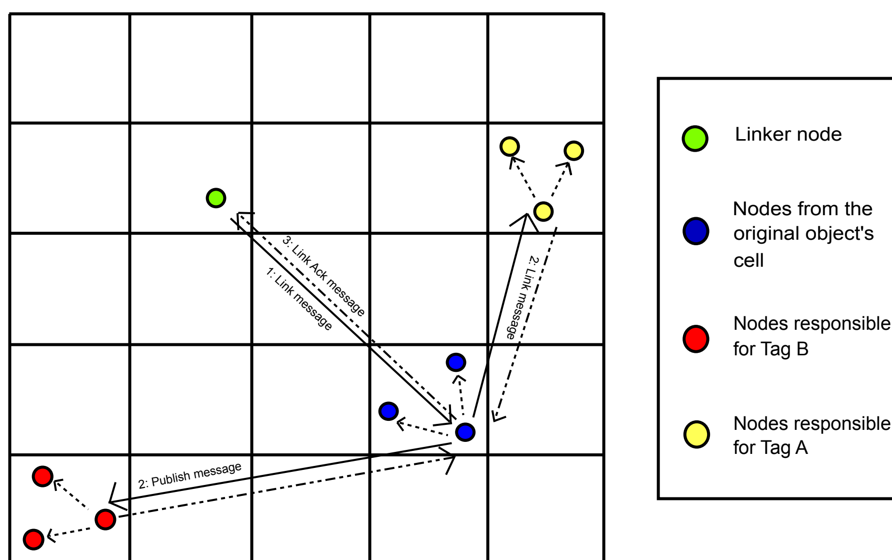

Figure 4.6: Alternative explored for the implementation of the *link* operation using the same example from the Figure 4.5

Another possibility we explored, was to shift all the message handling to the object's original cell, to route the collateral publish and link messages. As it is depicted on the Fig. 4.6, the nodes responsible for managing the object tags, would have to issue the necessary *publish* and *link* messages themselves. In the end, in order to balance the workload between all the nodes in the system, we decided to not pursue this option as it would be unfair to the nodes handling the most popular objects, who would have an heavier workload compared to the remaining nodes.

4.7.1.2 Unlink Implementation

The *unlink* operation is very similar to the *link* operation implementation wise. It starts with the unlinking node sending an *unlink* message to the object's original cell containing the following fields:

- A *WorldId* identifying the Namespace where the object resides;
- An unique *ObjectId* to identify the relevant object;
- And a set of *Tags* to be dissociated from the object's metadata.

A random node on the object's original cell, upon receiving this message, will update the object's tag counter, and disseminate this message to its cell peers so they can do the same. Then, the same node will send an acknowledgement message back to the unlinker node containing the same *WorldId*, *ObjectId* and *Tags*. Besides these fields, this message also contains another set of tags of which the object is also associated with.

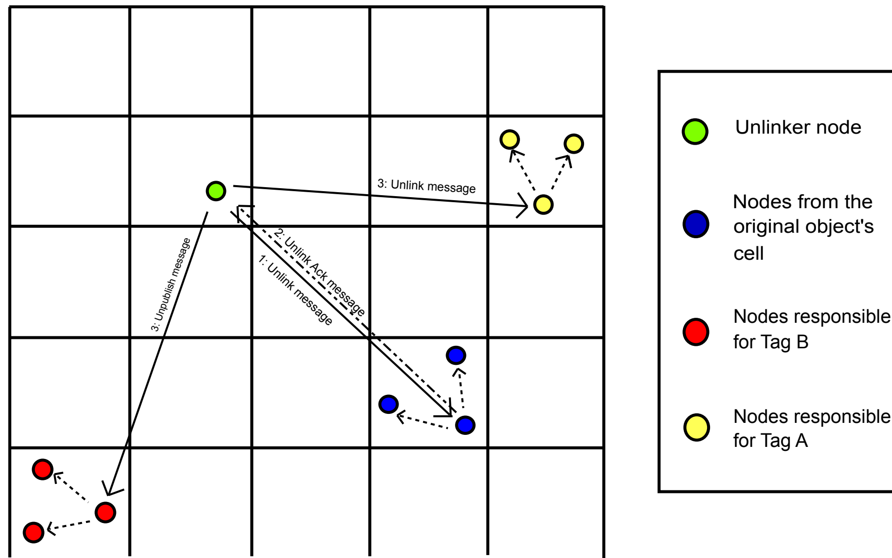

Figure 4.7: Interaction between nodes upon an unlink operation of an object with the Tag B, previously associated with the Tags A and B

The unlinker node, upon receiving this acknowledgement message, is now responsible for rerouting the unlink message to each of the tags cell that the object was already associated with, so that the cells can update their metadata files, and an unpublish message to the removed tags to stop including the object in the following subscriptions.

On the Fig. 4.7, it is illustrated the interaction between nodes from different cells, when an object that was already associated with the *tags* A and B, is being unlinked from the *tag* B.

4.7.2 Enforced Removal

With the ability to add *tags* to published objects, a revision needed to be made to Thyme's *unpublish* operation. Before our work, the set of *tags* associated with an object were immutable, and therefore, an *unpublish* message to each of the associated *tags* was enough to prevent its metadata files from being included in future subscriptions. But, with the set of the object's *tags* now being mutable, issuing an *unpublish* message for the initial *tags* cell could not be enough. To resolve this issue, we could either disseminate the updated metadata files, which would require keeping track of all the nodes' lists of retrieved metadata files to update them everytime a link/unlink would be issued, or an extra message could be issued to the object's original cell querying for their current list of *tags*. We opted for the second option, as we believe that compared to the first, it would induce the system in a lower communication effort to guarantee the removal of an object.

As it is shown on the Fig. 4.8, the enforced removal starts with the issuing node requesting the current list of *tags* associated with a given object with a message containing the following fields :

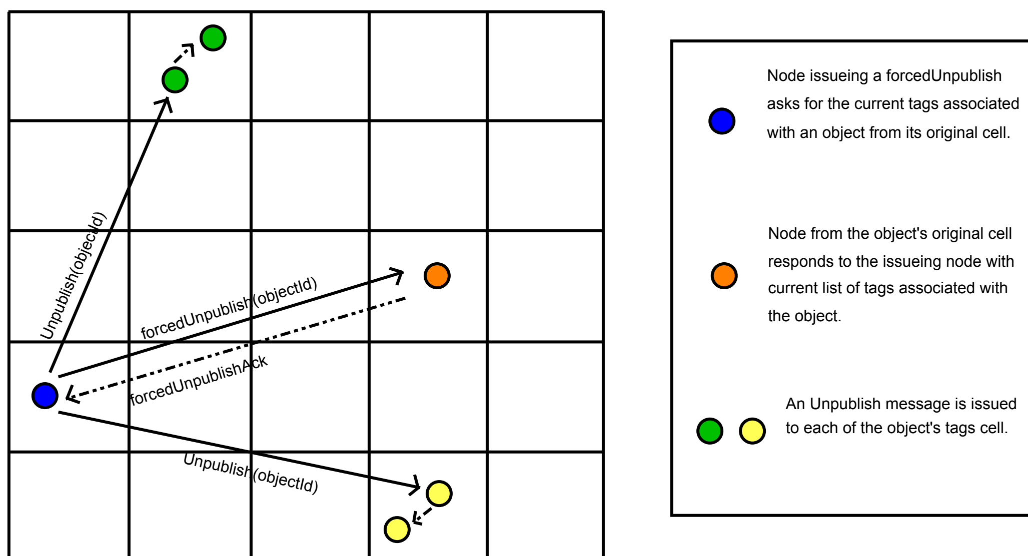

Figure 4.8: Interaction between nodes in the system, upon an ensured *unpublish* operation is issued.

- A *WorldId* identifying the Namespace where the object resides;
- And an unique *ObjectId* to identify the relevant object;

Finally, upon being returned the object's current list of associated *tags*, the operation is carried out as a normal *Unpublish*.

4.7.3 Unpublishing multiple objects

Removing multiple objects in Thyme would implicit issuing that many *unpublish* messages. But if all those objects are published under the same *tag*, one aggregated *unpublish* message should suffice. As that is the case for our *delete* operation, we have decided to add a new message type to Thyme's array of messages, the *aggregatedUnpublish* message. This new message instead of referring to only one object, as the *unpublish* message (referenced in the Section 2.3.1.1), refers simultaneously to multiple objects at a time.

With this new message type, once again we are able to reduce the number of messages circulating through the network, making an active effort to improve our system's usability.

4.7.4 Status Update upon joining a new Cell

With the introduction of the new data structure *tagsCounter* at the storage layer, (responsible for keeping track of the objects associated list of *tags*), a revision had to be made to the current *UpdateStatus* message.

The *UpdateStatus* message is a message returned to the nodes who just joined a new cell, so that they can aid its neighbour nodes with the cell's incoming requests. As soon as they enter a cell, they ask another random node for the cell's state, which until our

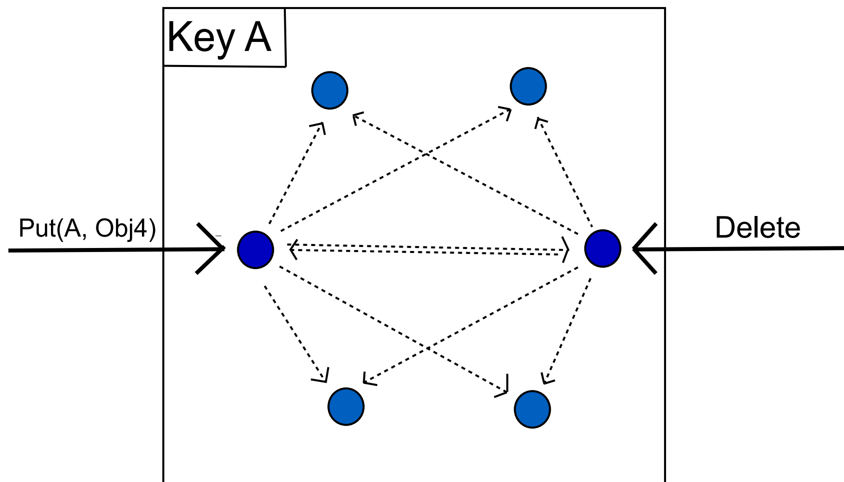

Figure 4.9: Concurrent Put and Delete calls to the same Cell.

implementation only included the list of received metadata files from the cell's tags, and the list of published files being actively replicated in that cell. To this two fields, we added the list of associated *tags* from the objects originally published on the relevant cell.

With this addition, the newly joined nodes are now able to respond properly to incoming *link*, *unlink* and *enforced remove* operations.

4.8 Maintaining Consistency

Keeping consistency in a distributed Key-Value database can be challenging, specially when multiple nodes can insert and remove multiple values simultaneously under the same *key*. One of the concerning scenarios at the start of our solution, was when a *delete* and *put* operation would be issued to the same *key* simultaneously by different nodes, that it could result in a completely different outcome depending on which operation would be handled first.

To exemplify this scenario, let's consider there are three objects published under a *key* **A**, and as demonstrated in the Fig. 4.9, a *put* and a *delete* operation are simultaneously received by two different nodes in the cell. The two following scenarios could happen:

- A node receives the *put* propagation message before the *delete* propagation message, and its values change in the following way: *[Object1, Object2, Object3]* $\xrightarrow{put}$ *[Object1, Object2, Object3, Object4]* $\xrightarrow{delete}$ *[]*
- A node receives the *delete* propagation message before the *put* propagation message, and his values change in the following way: *[Object1, Object2, Object3]* $\xrightarrow{delete}$ *[]* $\xrightarrow{put}$ *[Object4]*

This could be a problematic situation if the *delete* operation would actually delete all the *key*'s values, as opposed to our implementation where it only deletes the user's own

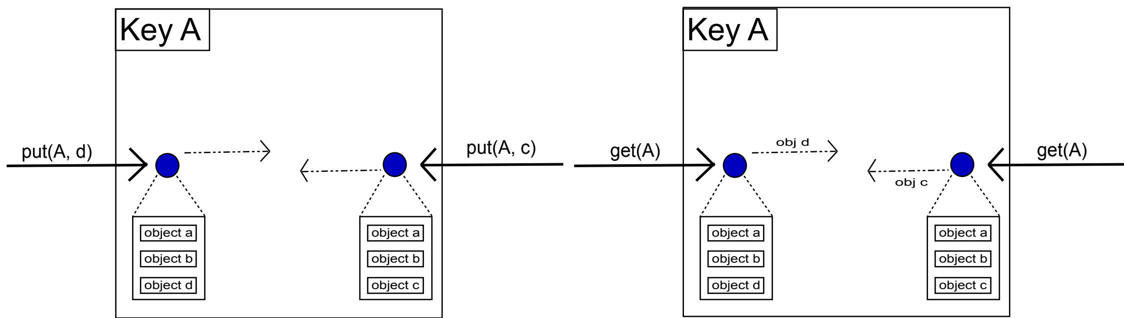

Figure 4.10: Concurrent put operations to the same key.

Figure 4.11: Inconsistent get operations to the same key.

values. And even so, if two *put* and *delete* messages sent from the same node happened to be handled by two different nodes inside the same cell, we still wouldn't have an inconsistent state. This is because Basil's operations are based on Thyme's own message types, where a *put* originates a *publish* message, and a *delete* originates an aggregated set of *unpublish* messages. And as stated in the Subsection 2.3.1.1, to issue an *unpublish* message, one must have the id of the object being removed. As this id is only returned upon the object's successful publication, this conflicting scenario would never arise.

The only inconsistent scenario that could eventually happen, is the one illustrated on the Figures 4.10 and 4.11, where a *get* operation is issued to two different nodes in the same cell, midway the propagation of two different objects. From the example demonstrated, the left *get* would return the objects **a**, **b** and **d**, and the right *get* would return the objects **a**, **b** and **c**.

This scenario although unlikely, may occur as there is no order preserved inside the cells, allowing for an immediate propagation of the metadata files as soon as they are received. This discards the necessity to enforce strong consistency policies, as eventually all the nodes in the same cell converge to the same state.

4.9 Implementation Details

There are some implementation details that weren't fully explained in the previous sections, but were essential for the development of our solution. In order to understand how Basil's operations were implemented, we will review the technologies and mechanisms employed throughout our system.

4.9.1 Protocol Buffers

The previous authors of Thyme already used Protocol Buffers to serialize their messages, and noted that when compared to other serialization mechanisms and formats, such as XML, JSON and Java Serialization, this mechanism fitted better their needs[25]. Following

Algorithm 1: Object Search in the Storage Layer

```
Result: Object's Metadata
1  int i = 0;
2  while i < PublishedFiles.size() do
3  |   auxObject = PublishedFiles.get(i);
4  |   if auxObject == object then
5  |   |   return auxObject.Metadata;
6  |   else if auxObject.size() == object.size() then
7  |   |   if byteComparison(auxObject, object) then
8  |   |   |   return auxObject.Metadata;
9  |   i++;
10 end
```

their justified decision, we decided to also use Protocol Buffers to update and create the messages listed in the previous sections.

Protocol Buffers is a serialization mechanism developed by Google, which is universal to any platform and programming languages, allowing distinct hardwares and software solutions to use it interchangeably. Besides this adaptive characteristic, this mechanism is also one of the fastest, beating XML by 20 to 100 times during transmissions, and its generated packets are much smaller than the packets generated by other serialization mechanisms.

All this mechanism needs, is a *.proto* file for each message, listing its fields and their respective data types, which still include a large array of options, allowing its compiler to build the message's prototype.

4.9.2 Searching for Objects Locally

To perform a *link* or *unlink* operation, Basil needs to confirm that the pertinent object exists in the system. Although we discarded the option to search through the whole network, we still search locally with the implementation in Algorithm 1.

This implementation starts by comparing all the published object references from the given world, and compares it to the given reference (on the line 4). If no match is found, it proceeds to compare all the objects with the same size as the object being searched, (line 6), by fragmenting the objects into chunks of 64 bytes and comparing them one by one, stopping when a pair of chunks aren't equal, (performed by the auxiliary function *byteComparison* in the line 7). If all the chunks are indeed equal, then the object's metadata is returned (on the line 8). In case none of the published files match with the given object, this process is repeated for the received files.

4.9.3 Notifications Policy

This concept was firstly introduced to Thyme's subscriptions to make selective queries possible. In order to materialise this concept, a new *NotificationPolicy* object was added

to the current subscription containing two fields:

- An *Integer* representing the number of items to be retrieved;
- And an *Enum* representing the notification policy itself. As of right now, the two possible policies are: *PARTIAL* and *RANDOM*

As of right now, only the metadata files from the past are filtered. If the subscription has a time window in the future, all the incoming matched metadata files will be sent regardless.

4.10 Summary

In this Chapter we presented Basil, the *framework* developed in the context of this thesis. Besides listing all its operations, explaining their parameters and behaviour, we also explained how we made use of its foundation layer, Thyme, changes we made to existing operations, and the new operations we added to Thyme's API.

Finally, we made an analysis of Basil's consistency, focused on how the cells would handle problematic concurrent operations, and if they would infer the system on an inconsistent state.

On the next chapter we will present our case study application, a Quiz developed to be possibly integrated in our university's lectures, that will be our primary evaluation target.

Chapter 5

Case Study: Class Quiz

In this chapter we will be presenting the mobile Android application developed as a case study for Basil. We start by presenting an overview, explaining the application's purpose and available functionalities on Section 5.1. Following this, on Section 5.2 we describe all the operations available for the two types of user supported, professor and student, alongside with its implementation details employing Basil.

5.1 Overview

Class Quiz is an application for Android devices that allows students to participate in live class quizzes created by authenticated professors.

Professors can publish quizzes formed from a set of previously published questions in the system, to which the students can join via the scanning of a *QR* code. The answers submitted by the students are then collected and reported back to the professor, who can group them by student or by question. Upon the quizzes completion, these are automatically corrected. The students can then inspect their correction, as well as the grades obtained in each one of them.

Class Quiz extends Basil to conduct its operations, and will serve as a proving target for our developed solutions, as a well as our comparing target against another Edge storage.

Class Quiz can be installed on any Android device with the operating system version 4.1 (*Jelly Bean*) or higher. Due to Thyme's nature, it doesn't require any Internet connection, only relying on *Peer-to-peer* communication technologies (such as Bluetooth and Wi-Fi Direct), and communications via an Access Point (using Wi-Fi).

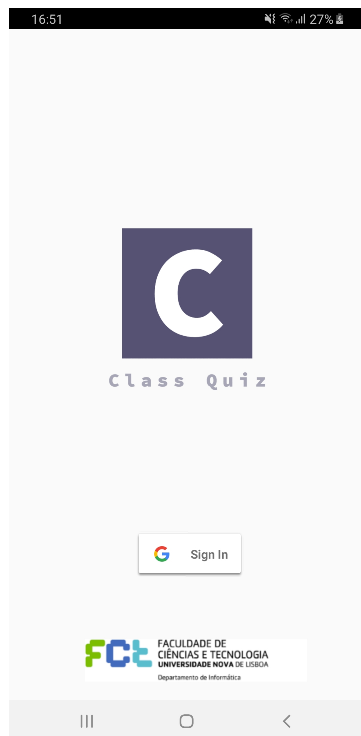

Figure 5.1: Initial Application Home Screen.

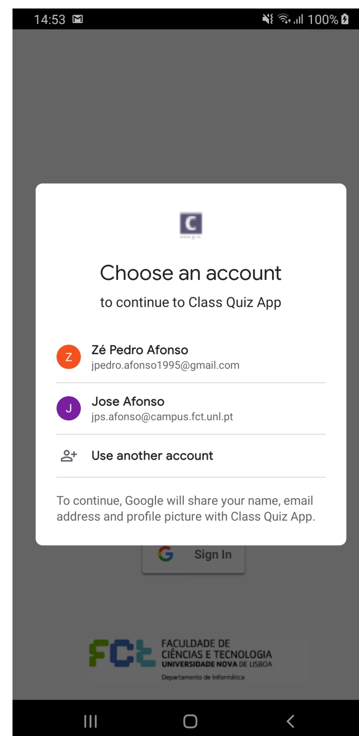

Figure 5.2: Authentication with the university Gmail account.

5.2 Application Flow

The application flow starts with a simple *LogIn* activity (show on Fig. 5.1), which will only authenticate accounts from the FCT official domain. The users will be authenticated either as a Student or as a Professor. Depending on their roles, they will both have a specific list of available operations.

5.2.1 Professor operations

If the user is authenticated as a Professor, he will be immediately prompted with a list of possible operations, necessary to the creation and publication of a Quiz. The following are:

- Create questions under one or more areas;
- Associate areas to existing questions;
- List all the areas and questions published on each of them;
- Create and Edit quizzes;
- Start a quiz by publishing it;
- Inspect the answers submitted to a quiz;

Figure 5.3: Creation and Publication of a Question.

5.2.1.1 Publishing a Question

Before creating a *Quiz*, the Professor should make sure there are enough questions published to form one. As shown in the Fig. 5.3, a question is composed by the following fields:

- **Question** : The Question itself.
- **Area** : The area it is being published under.
- **Right Answer** : At least one right answer for the inserted question.
- **Wrong Answer** : At least one wrong answer for the inserted question.

After the mandatory fields are filled, the question is then published onto the system through a *put* call in Basil, and is ready to be associated with future Quizzes.

The list of existing Areas, and questions published under each one, are then also available in the Professors interface with the activities displayed on the Figures 5.4 and 5.5, which issues a *get* operation for each area in the global key list. The professors can either *remove* a question, if they were the ones to publish it, *link* existing questions to new areas, and also *unlink* them from other areas linked by them.

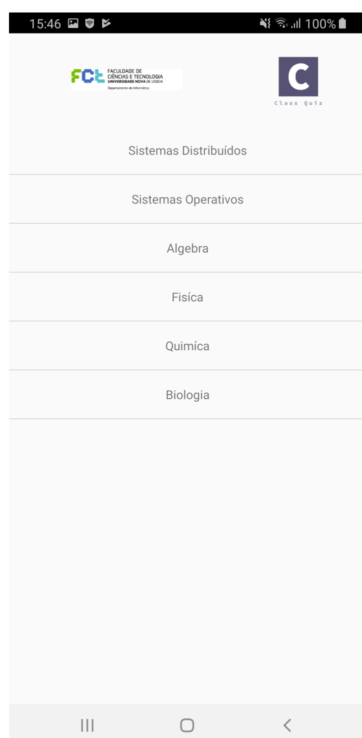

Figure 5.4: List of existing Areas in the System.

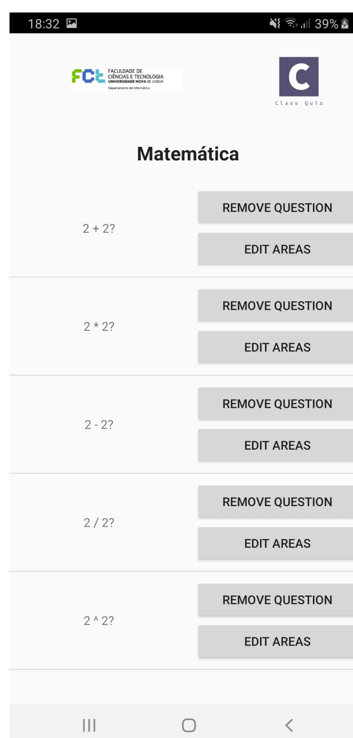

Figure 5.5: List of published questions under the Area "Matemática".

5.2.1.2 Generating a Quiz

Unlike the questions, the quizzes must be created first before being published. This way the professors can prepare the Quizzes before the classes, and publish it at a more convenient time. As seen in the Fig. 5.6, the mandatory fields to generate a new Quiz are the following:

Department : Department from which the Subject belongs to, and will be published under.

Subject : Subject in which the Quiz is being evaluated, and will be published under.

Quiz Name : Name identifying the Quiz.

Area and Number of Questions : At least one area to be associated with the Quiz, and the corresponding number of questions being asked in the quiz.

After built, the quizzes can either be edited or published as illustrated on the Fig.5.7.

Before being published, the Quizzes are wrapped into a *JSON* object containing: the Quiz's name, the Quiz's answers key (where the answers from the students will be published on), and a map for the areas associated with the Quiz and its corresponding number of questions. This *JSON* object is then published with a *put* call onto Basil, under the

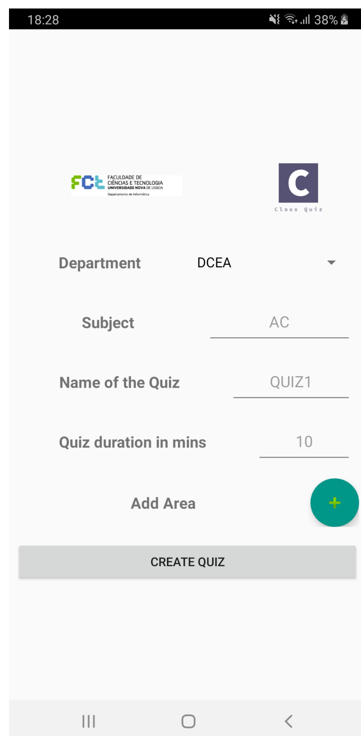

Figure 5.6: Creation of a Quiz.

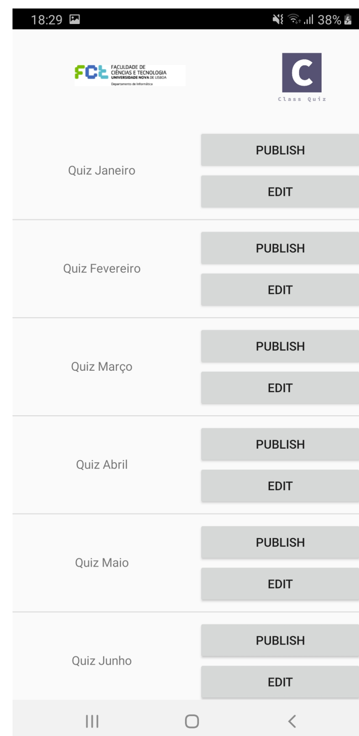

Figure 5.7: List of the user's Quizzes.

Quiz's *key* which is created with the Quiz's name, department and subject. After published, this object can now be downloaded by the quiz's attending students, and will be interpreted at the application layer.

Also, as soon as a professor publishes a quiz, the application automatically issues a *getUntil* operation for the quiz's answers *key*, defining the superior timestamp limit as the current's timestamp plus the quiz's duration. This way, only the answers published within the quiz's duration will be considered for the professor's quiz correction.

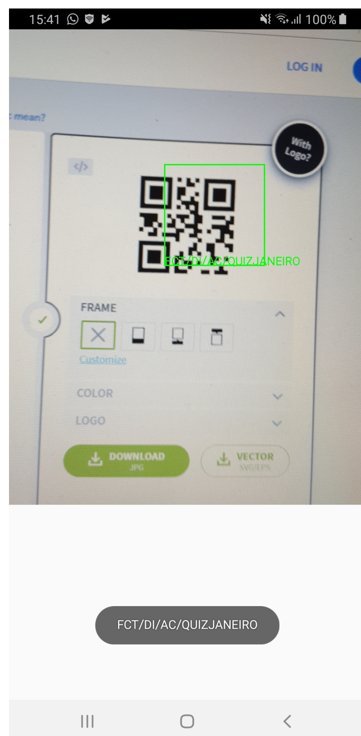

Figure 5.8: Scanning a QR Code containing the Quiz's Key.

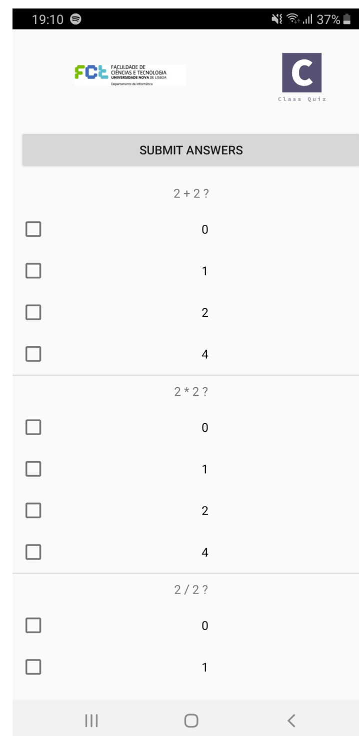

Figure 5.9: Screen upon starting an ongoing Quiz.

5.2.2 Students Operations

Upon being authenticated as a Student, the users are immediately prompted with a list of possible operations that allows them to either join an ongoing quiz, or inspect results obtained in past quizzes.

5.2.2.1 Joining a Quiz

To start a Quiz, the Students scan a QR Code through the mobile application, as it is illustrated on the Fig. 5.8, which will contain the *Key* under which the Quiz is published under.

Upon downloading the Quiz object from the scanned *Key*, the application will issue a *get* operation with a *random* metadata selection filter for each of the Quiz's areas with its corresponding number of Questions, ensuring that each student will probably have a different quiz. As soon as all the questions are downloaded, the students can start answering the Quiz, as it's shown on the Fig. 5.9.

As soon as the questions are solved and submitted, the respective answers are published under the Quiz's answers *key*. As the answer itself is such a small object, (composed only by the student's name, the question identifier and the list of submitted answers), it is published as a description of a null object. This later allows the responsible professor to retrieve the answers directly from the answer's key metadata files, instead of downloading them as a value, who can still group them by student or question.

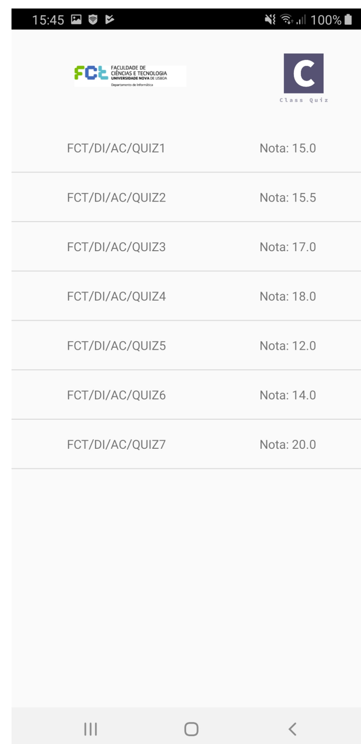

Figure 5.10: List of Quizzes done by a Student.

After participating on a Quiz, the students can later review them, as it is illustrated on the Fig. 5.10, being able to check which questions they answered wrong and right, and the grades obtained in each of the quizzes. The correction for each student is done locally, as the downloaded questions in the start of the quiz already contain the respective solutions.

5.3 Summary

Throughout this Chapter, we presented *Class Quiz*, the Android mobile application developed as a case study for Basil. This application allows professors to create and publish quizzes, as well as the questions that form them. The students on the other hand, are able to join any ongoing Quiz, as long as they scan its corresponding QR code, and can later review the Quizzes they participated on.

The application as of right now, is oriented just for our university members, only authenticating UNL FCT registered users, and joining the FCT *World* by default. But with some simple authentication adjustments, this application could easily be integrated into any university classes.

On the next chapter, we will present the evaluation done to our implemented system, and to the application developed as our case study. We will also present the evaluation results, and a constructed analysis derived from these.

Chapter 6

Evaluation

This chapter begins with a description of the methodologies used to characterise and evaluate our proposed solution, and how we plan to compare it against a database with edge awareness, Cassandra. In Section 6.2, we present our simulation environment, the use-case generated for testing our system, and how we implemented the employing application using Cassandra. On Sections 6.3 and 6.4, we present and discuss the results obtained with both Cassandra and Basil, while explaining the nature of such results. Finally, on Section 6.5, we reflect on the more appropriate contexts for the employment of our solution, and what distinguishes it from Cassandra.

6.1 Evaluation Methodologies

To validate our solution, we need to submit it to a variety of tests that simulate a real-world edge environment. Throughout these tests, besides the number of active quizzes, we will be ranging the number of participating nodes in the system to further test and compare the scalability of our system. We conducted our tests to answer the following questions:

- How much data does Basil need to generate?
- How does the network load vary?
- How scalable is Basil?
- How demanding it is for the mobile devices to use Basil?
- In which scenarios should engineers consider Basil?

Besides performing an absolute evaluation to our system, we will also compare it to another Key-Value Edge storage system, and further evaluate the scenarios in which our solution prevails.

6.1.1 Absolute Evaluation

To purely evaluate Basil as a storage system, we will start by measuring its available operations performance one by one. We are interested in the number of messages sent per operation call, and the average load carried out by a message. These two metrics will allows us to calculate the average load induced by an operation onto to the nodes in the system. Besides evaluating the operations' performance, we will also be measuring the workload imposed by Basil onto its participating nodes, which is defined as the number of messages sent, received and processed by a node.

We will then compare Basil's performance using only mobile nodes, against its performance while also interacting with an Edge infrastructure.

6.1.2 Comparing against Cassandra

Basil is the only Key-Value Edge oriented Storage framework that uses both mobile nodes, and infrastructure nodes as active components in the system. And so, we wanted to compare it against one of the most prevailing NoSQL storage at the moment, Cassandra, which also has some Edge-oriented implementation details, (referenced in the Section 3.2.2).

We are expecting Cassandra to prevail over Basil whenever only one quiz is being carried out, as the more powerful infrastructure can outperform the weaker mobile devices. But as the number of attending students per quiz, and the number of active quizzes is increased, Cassandra's performance will surely decay, as the solo infrastructure will be flooded with requests. Whilst in Basil, as each quiz is contained in its geographical space, for our case-study a classroom, its performance is independent from the number of ongoing quizzes. Still, Basil's performance will most likely also decay when the number of attending students is increased, as the proportional increasing number of circulating messages will slow the executed operations.

We are expecting that, from a certain number of mobile nodes and ongoing quizzes, Basil will prove itself as the more scalable system, against the solo Cassandra infrastructure. To prove so, we will be comparing both systems latency times, for read and write operations, and hopefully find the pivotal point where Basil prevails over Cassandra.

6.2 Evaluation Environment

To truly evaluate Basil and Cassandra we would need a large quantity of mobile devices to mimic a real world Edge scenario. Unfortunately, as this scenario is unfeasible, we will

```
ADDQUESTION$|$582.574914$|$0$|$DI/AC/QUIZ0$|$DI/AC/hxojhyjcfr$|$3
ADDQUESTION$|$588.841853$|$0$|$DI/AC/QUIZ0$|$DI/AC/qbcsghjdfg$|$3
ADDQUESTION$|$589.262598$|$0$|$DI/AC/QUIZ0$|$DI/AC/mqdezayujs$|$3
ADDQUESTION$|$594.371015$|$0$|$DI/AC/QUIZ0$|$DI/AC/srlcbpiokr$|$3
```

Figure 6.1: Example of a block of trace actions with a quiz action, its time span, the executing node, and the action's inputs.

resort to a simulator to simulate multiple mobile devices engaging and interacting with each other in an Edge-like network.

Regarding our simulator, we used a custom trace-based simulation framework, previously developed in Java by some of Thyme's authors, where each mobile device is simulated on a single thread on the same machine, all contained in the same network layer. To do so, the simulator offers a network layer capable of supporting logical dissemination of messages between any number of virtual nodes within a single machine. Also, some of Thyme's logic which depends on Android-specific classes and dependencies had to be exchanged with the simulator's counterparts.

Each trace is a *txt* file, which consists of some simulator specific actions, such as the spawn of the mobile nodes, and multiple quiz actions (e.g. *CREATE QUIZ*, *CREATE QUESTION* or *ANSWER QUESTION*) Each action is associated to a specific node, a relative time span, and the actions' inputs as it is illustrated in the Fig. 6.1. As the time span is relative, the trace's execution can be sped up or slowed down, without compromising the actions order.

For the Cassandra tests, in order to produce real-world evaluation results, we hosted the infrastructure server on a different machine from the machine simulating the mobile devices. Regarding the simulated devices, depending on the system being evaluated, each one is either running a Basil instance, or a simple Cassandra client. For Basil, we won't be measuring the latency times, since the mobile devices will all be simulated in the same machine, thus not producing real-world values.

The machines used to run our simulations, hosting both the mobile devices and the Cassandra server, had the following specifications:

- **CPU** : 2 Intel Xeon E5-2620 v2 (with Hyper-Threading)
- **RAM** : 64 GB
- **Network Adapter** : 2 NIC Intel Corporation I210 Gigabit Network Connection

6.2.1 Simulation Case-Study

To directly compare Basil and Cassandra's performance, we had to submit them to the exact same traces, where each one contains all the crucial operations for the realisation of one or multiple quizzes.

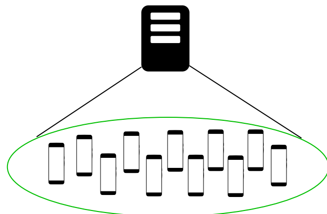

Figure 6.2: Multiple mobile devices running Basil being simulated within a single machine

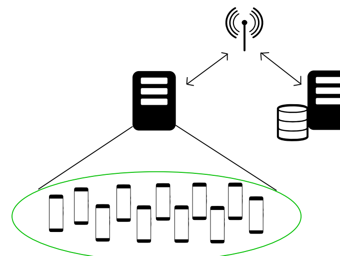

Figure 6.3: Multiple mobile devices being simulated within a single machine interacting with a Cassandra instance on another machine.

It starts with a professor node preparing a quiz, by publishing multiple questions under different areas. For all our quizzes, we published three different questions per five different areas each, resulting in fifteen questions overall per quiz. After the questions are published and linked to a quiz, the quiz itself is published, and the students start to retrieve it, alongside with its respective associated questions. After each question is answered by all of the students, the professor node retrieves all the submitted answers in order to correct it.

Regarding the quiz temporal structure, we reserved ten minutes for the creation and publication of the quizzes, twelve minutes for the students to retrieve and answer the questions, and one minute for its correction

6.2.1.1 Cassandra Implementation

In order to produce trustworthy evaluation results, we didn't try to emulate Basil's logical structures in Cassandra, and instead implemented what we think was the best approach to Cassandra's data model. As soon as the cluster is instanced, four tables are created: a quiz table, an areas table, a questions table and an answers table.

A **Quiz** Table, where each quiz object contains:

- The quiz's tag, as its primary key;
- And the number of questions per different area, represented by a map.

An **Area** Table, where each area object contains:

- The area's name, as its primary key;
- And a set of questions ids associated to it.

A **Question** Table, where each question object contains:

- An id, as the question primary key;
- The question itself;
- A set of associated areas;

A set of possible answers;

And a set of correct answers.

And finally, An **Answer** Table, where each answer object is composed by:

The id from the question answered,

The quiz's tag containing the question answered,

The answer's author,

And the answer it self.

The question's id, quiz's tag and author's name altogether identify uniquely each answer submitted, thus composing the answers primary key.

All the subsequent operations executed are then translated into simple *inserts*, *updates*, and *selects* for the creation, modification and retrieval of the quiz's objects respectively.

6.3 Absolute Evaluation Results

The following graphics refer to Basil's own performance when sustaining the realisation of a single Quiz, with a different number of participating nodes. In every simulation scenario, we decided to use seven *cells*, one for each of the quiz's *keys*, one cell for the quiz, one for the quiz's answers, and 5 cells for the questions for each different area. This way, each cell is only responsible for a *key*, minimising Basil's network traffic and reducing the nodes workload.

In order to measure our solution's scalability, we looked at the workload imposed onto the participating nodes derived from utilising the system, depicted in the Fig. 6.5, and the total number of messages received by the nodes, illustrated on the Fig. 6.4.

In order to provide a more complete evaluation for our system, we categorised the messages (and their associated workloads) into two types:

Active messages, which refers to all the messages originated from user's input, e.g. first *link* message towards a cell;

And ***Passive*** messages, which refers to all the messages required from the nodes to support the System, e.g. secondary *link* propagated inside a cell.

Analysing Fig. 6.4, we can see that the number of all messages exchanged throughout the system is mainly composed by passive messages, with an average *68 %* distribution of all messages, while the active messages have an average of *29 %*.

Regarding the messages fluctuation, we can see that there is a sudden increase in the number of passive messages sent from the simulation with 40 nodes to the simulation with 80 nodes. This rapid increase of passive messages derives from the substantial increase of *New Location* messages, which are propagated throughout the network after a

Table 6.1: Load and Distribution of Active Messages

Nodes	Publish		Subscribe		Download		New Location	
11	148	76Kb	37	6Kb	75	7Kb	84	41Kb
21	268	140Kb	71	12Kb	256	23Kb	620	307Kb
41	559	297Kb	186	31Kb	642	59Kb	3030	1806Kb
81	1001	542Kb	398	66Kb	1391	128Kb	12635	8604Kb

	Total number of messages received
■	Total load of the messages received

Table 6.2: Load and Distribution of Passive Messages

Nodes	Publish		Subscribe		New Location	
11	5	2Kb	20	3Kb	70	34Kb
21	330	172Kb	255	42Kb	3084	1524Kb
41	1272	673Kb	1186	197Kb	20181	12Mb
81	15738	8535Kb	5577	925Kb	152433	104Mb

successful download, informing the responsible cells of the new replicas location. This abrupt increase can be explained with the fact that on the simulation with 40 nodes, there may have been more cells responsible for multiple *keys* than on the simulation with 80 nodes, which would also raise the probability of the object's *keys* sharing the same cell, and thus requiring less *New Location* messages per download.

Analysing the Table 6.1, we can observe that the *New Location* messages is one of the biggest stake in the messages distribution. Its abnormally high volume is due to the fact that these messages are sent whenever an item is downloaded, and the item's respective original cell nodes are constantly updating the item's metadata with new locations. But in the scenarios with a smaller number of nodes, we note that these messages don't have the same message volume. This is because the probability of a node requesting for an item he is already passive replicating is higher than the scenarios with more nodes, and therefore, not needing to re-download it, no further *New Location* messages need to be issued. This particular result raises a relevant problem for scenarios with particular popular items, which should be attacked in the next iterations of this thesis work. One possible solution could be to define a threshold for the number of locations associated to a item's metadata, which should be low enough to prevent congested cells, but high enough to still make this item easy to download.

Regarding the Table 6.2, we can see that, by far, the biggest share in the Passive messages distribution is represented by the *New Location* messages. This is because, in our

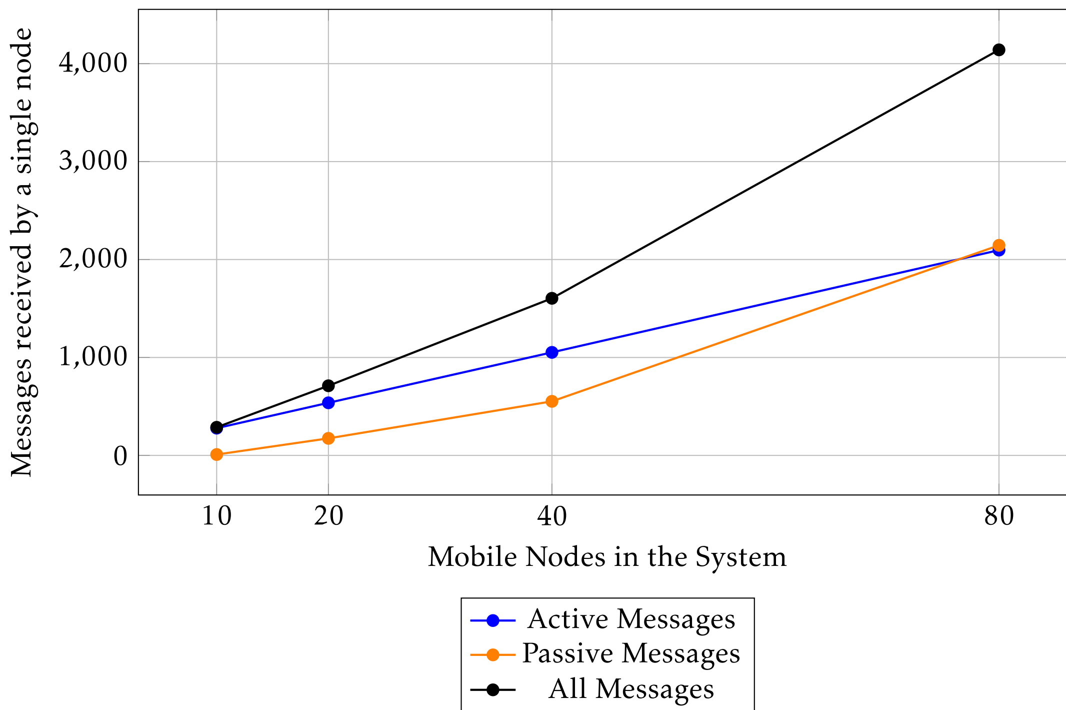

Figure 6.4: Number of messages received by the Nodes.

case-study, the download operation is the most frequent one, with each student issuing it 16 times per quiz, directly reflecting the number of passive messages broadcasted inside each download item's keys cell.

From the graphic 6.4 we can observe that the nodes using Basil, due to its high co-operativity properties, are required to be in constant communication with each other in order to keep the system's state consistent and cohesive. Consequently, the number of messages exchanged in the whole system increases almost exponentially when the number of participating nodes is incremented. Although most of the traffic is still contained in the logical cells, this still represents a great communication load for nodes to endure, specially for the most popular cells which handle a larger number of requests. But it's a price that distributed systems must pay to abdicate from centrality, specially in Edge environments whose systems are solely sustained by mobile devices.

Varying the number of cells could have produced better results, as this would've widen the hash space and distributed the nodes in an evener way. This would positively reflect on the number of passive messages sent, as the cells would certainly contain fewer nodes. For future iterations, this thesis work would benefit greatly of being able to dynamically calculate the optimum number of cells given the number of nodes, and analysing the network traffic inside each cell.

Still, on the busiest case scenario with 80 nodes, a node receives on average 120 messages per minute, while on the smallest case with only 10 nodes, the average drops to only 3 messages per minute.

Regarding the workload required from the nodes in the system, we can see that it

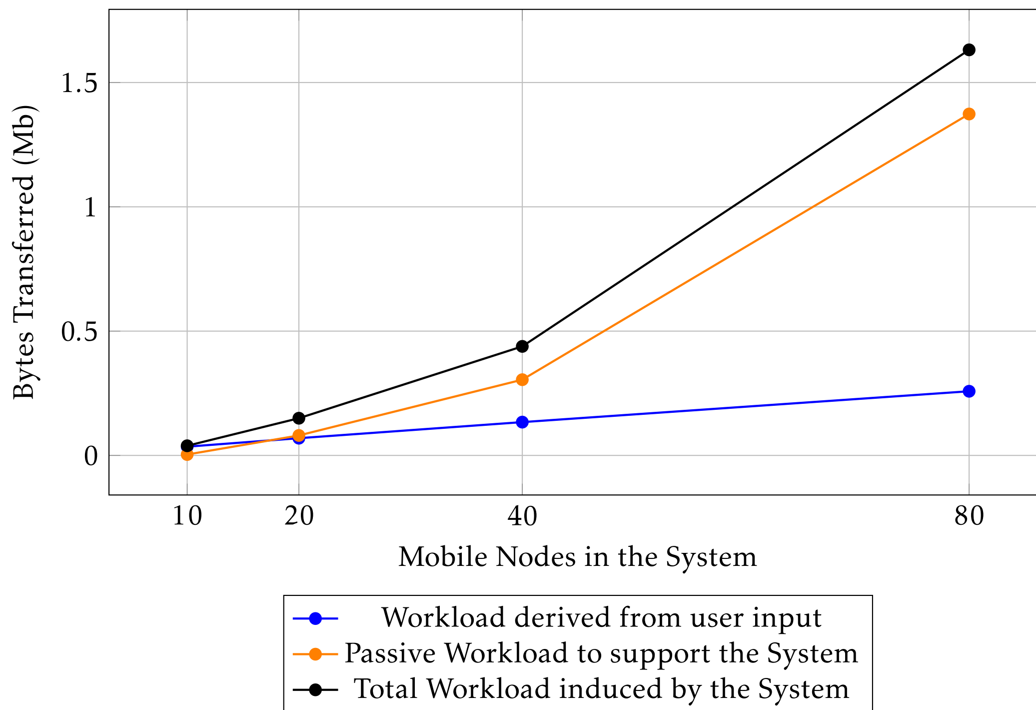

Figure 6.5: Workload inferred by Basil.

follows the same pattern as the graphic illustrated in the Fig. 6.4. The passive workload inferred onto to the nodes increase abruptly from the 40 nodes test simulation case to the 80 nodes case, once again, as a consequence from the high throughput of passive messages registered, required to keep the state of the logical cells consistent.

On the heaviest case scenario with 80 nodes, a node transfer on average 77 Kb per minute, while on the smallest case with only 10 nodes, the average drops to only 1 Kb per minute. If we combine the data obtained from both the graphics in the Figures 6.4 and 6.5, we can derive that for a scenario with 80 nodes, each node receives on average 120 messages per minute, with an average of 642 bytes per message. As for the smaller scenario with only 10 nodes, each node receives on average 3 messages per minute, with each message averaging a size of just 333 bytes.

Before comparing Basil and Cassandra evaluation results on the next section, one should take into account that Basil's performance is independent from the number of simultaneous ongoing quizzes, as the corresponding network traffic is contained in the respective classroom, and unless the quizzes instances share the same infrastructure (as in Cassandra), only the participating nodes will be affected by the corresponding quiz's workload.

6.4 Comparing against Cassandra

Whilst on the previous graphics we were interested in the metrics measured on the mobile nodes, on Cassandra we are more interested in the variation of behaviour measured at the

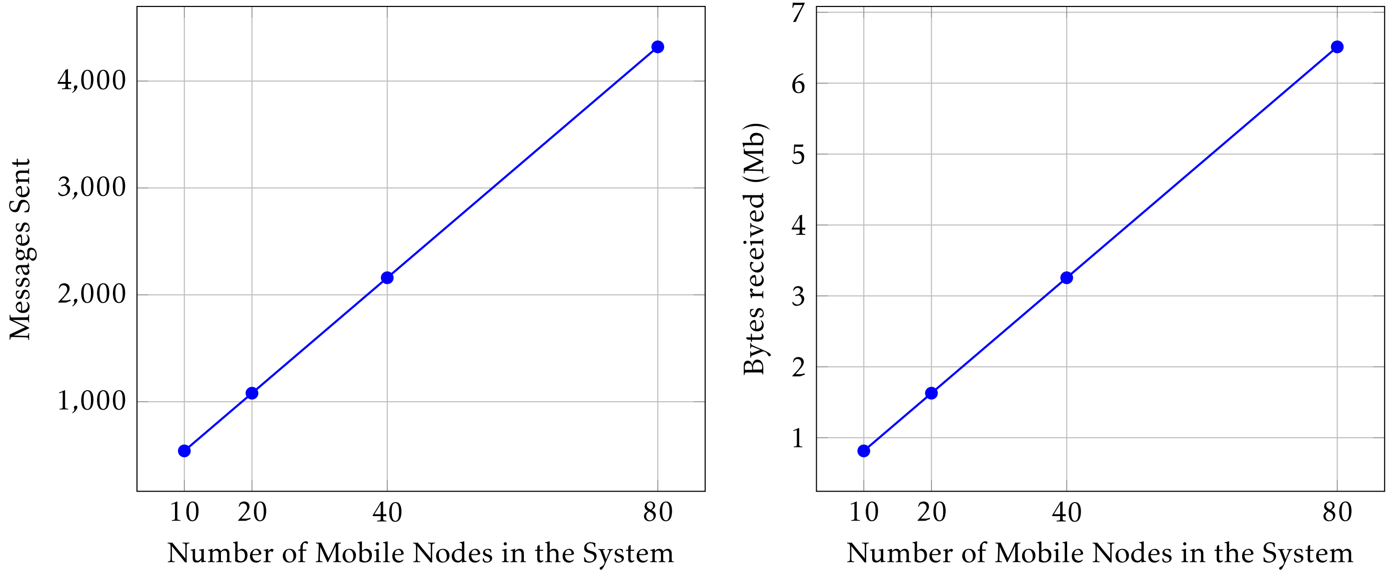

Figure 6.6: Cassandra Server measured metrics

infrastructure server (depicted in the Fig. 6.6), as these remain constant for the mobile Cassandra clients. To directly compare the two storage systems, we choose the evaluation results obtained from the simulations with 80 nodes, as we believe it is the scenario closest to the real world.

When comparing the results obtained from using Basil and Cassandra to support a single Quiz, we can observe that the single Cassandra infrastructure outperforms the multiple collaborative mobile devices. This is mainly because Cassandra relies on a single central node to process and storage all the data, and unlike Basil, no additional passive messages are required to sustain the system, greatly reducing the network traffic.

But when comparing the two storage systems performance supporting multiple concurrent quizzes on the Fig. 6.7, we can observe that Basil's workload stays constant throughout the duration of multiple quizzes, as opposed to Cassandra, whose workload is directly proportional to the number of ongoing quizzes. As mentioned before, this happens because Basil's network traffic is contained in its own geographic venue, in this case the respective quiz's classroom, and Cassandra's network traffic is all routed to the central infrastructure handling the quizzes generated data.

We can also notice, that employing Basil as the quizzes supporting storage system, as opposed to Cassandra, starts being a viable alternative when the number of simultaneous quizzes is equal or bigger than 19, which in the application's context is a very plausible number. This is of course, if the deciding criteria is the traffic and load endured by the nodes in the system.

Regarding the latency times experienced in both cases, we can only reference other articles as we didn't get the opportunity to simulate real-world experiences in the context of this thesis work. For Basil, we will reference the evaluation results obtained in Pedro

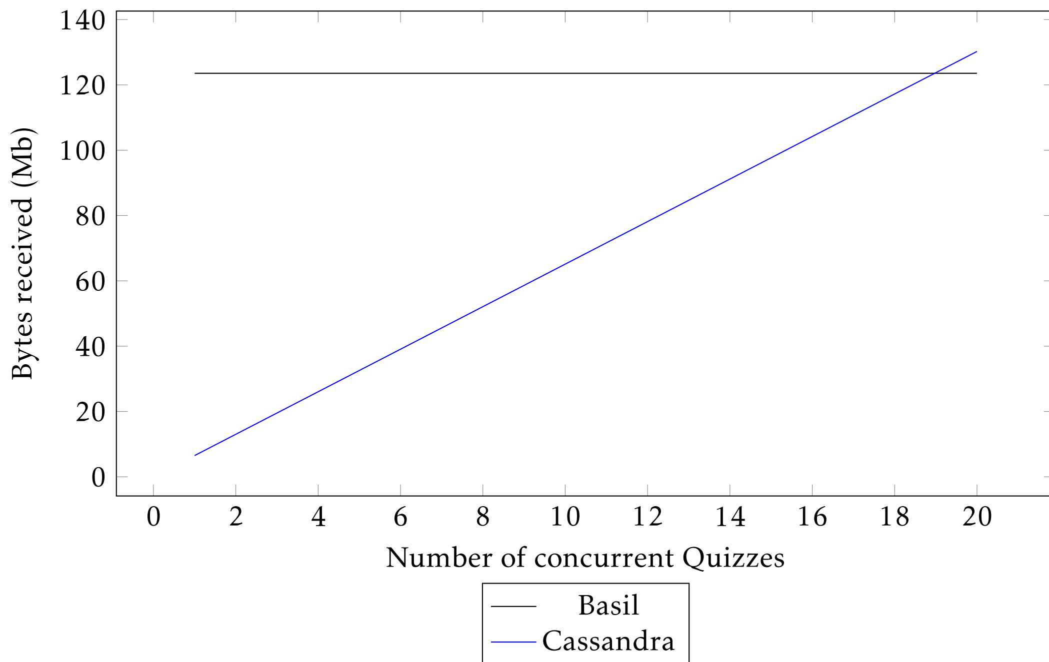

Figure 6.7: Total number of bytes circulating through the system

Vieira's master thesis [29], where Basil's supporting system, Thyme, got an exhaustive performance depiction. In its thesis, Pedro measured the latency times sustained in a system solely composed by mobile nodes, and the same system integrated with an Edge infrastructure. For the first scenario, the average latency times of all the operations was registered with a duration of 375 ms, while on the second scenario with an edge infrastructure, the same average dropped to 160ms per operation.

As expected, the more powerful Cassandra infrastructures are able to outperform the cluster of weaker mobile devices for a single Quiz instance. But as the number of instances is incremented, Cassandra's throughput and latency times are expected to decay, if of course, no infrastructure nodes are added to Cassandra's ring of servers, while Basil's performance remains constant. To reproduce the same linearity with Cassandra, a server would have to be allocated for each Quiz instance, guaranteeing that each instance's network traffic were isolated from each other.

6.5 Summary

When comparing the two frameworks, there is no doubt that Cassandra is the one who presents the most prominent results performance wise, being the Data Storage technology of election for the great majority of Tech Companies [27]. What Basil can offer that Cassandra can't, is a flexible, stable and horizontally scalable framework for edge environments, independent from remote and/or local infrastructures to support an employing application.

In the end of the day, if an edge application requires a system to be constantly delivering data within a single instance, serving multiple requests concurrently while still delivering at a steady and fast pace, Cassandra should be its choice. On the other hand, if the application's engineers are looking for an adaptable mobile-oriented system, so independent that it's able to function on a peer-to-peer network, and/or in a shared local network, while still delivering decent performance results, then Basil is one serious option to consider.

Conclusions

In this chapter we present some of the conclusions derived from this dissertation and some possible future improvements to our developed system.

7.1 Conclusion

In this dissertation, we present our developed Key-Value Storage for Edge environments, Basil, alongside with its founding layer, Thyme. We start by presenting the latest's core features and available operations, explaining how will we accommodate them to Basil's requirements.

We then present our approach to the Key-Value model for Edge networks based on a message-oriented system, where we propose an implementation to allow the association and disassociation of keys from existing objects, and how are the keys managed and dissipated throughout the whole system. The major contribution from this thesis's work, is a fully functional Key-Value Storage to handle data generated from mobile devices, without resorting to external services or servers to mitigate the workload from the mobile devices. Although the data doesn't leave the devices, the system is also prepared to cope with an Edge infrastructure to alleviate some of the workload from the devices, and to broad the system's geographical range.

Following Basil's chapter, we present our developed application, which was also the use case for our evaluations, in which we exemplify a plausible employment of our system, listing its available functionalities to further display Basil's wide array of possible implementations.

Ultimately, analysing the results obtained from the previous chapter, we can conclude that our system demonstrates a greater horizontal scalability when employed by an typical Edge application, comparing it to another NoSQL Edge oriented infrastructure-based.

Due to the fact that Basil's communication is local and contained to its geographical space, we also derive what scenarios most favour Basil, and what criteria should employing engineers look for when considering Basil as their Edge Data Storage System.

7.2 Future Work

While our system is fully functional, we still believe that there are some implementations that could benefit the system's overall performance and usability. Although these implementations weren't initially planned, their materialisation would still enrich our framework, making it more appealing for possible employing applications. From those, we list the following:

- Disseminate *keys* removal, so that the system's nodes could have a constant consistent list of *keys*. In Thyme, it would require some notification mechanism for active subscriptions, whenever an object is unpublished.
- Eliminate possible inconsistent *get* operations, via a viable secondary mechanism to assure consistent queries.
- Reduce the payload originated from Basil's own system messages
- Further evaluate Basil in a real-world scenario to collect latency times and compare it to another Edge oriented storage

These are some of the possible areas of improvement, in case this dissertation work would be continued, or integrated with another system, that would develop Basil into a more complete system.

Bibliography

[1] A. Ahmed and E. Ahmed. "A Survey on Mobile Edge Computing." In: Jan. 2016. doi: 10.1109/ISCO.2016.7727082.

[2] *Amazon DocumentDB*. Amazon. url: https://aws.amazon.com/documentdb/.

[3] *Amazon DynamoDB*. Amazon. url: https://aws.amazon.com/dynamodb/.

[4] *Azure Tables*. Microsoft Azure. url: https://azure.microsoft.com/en-us/services/storage/tables/.

[5] M. T. Beck, M. Werner, S. F. L. Maximilian, and T. Schimper. "Mobile Edge Computing: A Taxonomy." In: (2014). doi: http://dx.doi.org/10.1002/andp.19053221004.

[6] *Bigtable*. Google. url: https://cloud.google.com/bigtable/.

[7] A. Corbellini, C. Mateos, A. Zunino, D. Godoy, and S. N. Schiaffino. "Persisting big-data: The NoSQL landscape." In: *Inf. Syst.* 63 (2017), pp. 1–23. doi: 10.1016/j.is.2016.07.009. url: https://doi.org/10.1016/j.is.2016.07.009.

[8] A. Davoudian, L. Chen, and M. Liu. "A Survey on NoSQL Stores." In: *ACM Comput. Surv.* 51.2 (Apr. 2018), 40:1–40:43. issn: 0360-0300. doi: 10.1145/3158661. url: http://doi.acm.org/10.1145/3158661.

[9] U. Drolia, R. Martins, J. Tan, A. Chheda, M. Sanghavi, R. Gandhi, and P. Narasimhan. "The Case for Mobile Edge-Clouds." In: *2013 IEEE 10th International Conference on Ubiquitous Intelligence and Computing and 2013 IEEE 10th International Conference on Autonomic and Trusted Computing, UIC/ATC 2013, Vietri sul Mare, Sorrento Peninsula, Italy, December 18-21, 2013*. 2013, pp. 209–215. doi: 10.1109/UIC-ATC.2013.94. url: https://doi.org/10.1109/UIC-ATC.2013.94.

[10] U. Drolia, N. D. Mickulicz, R. Gandhi, and P. Narasimhan. "Krowd: A Key-Value Store for Crowded Venues." In: *Proceedings of the 10th International Workshop on Mobility in the Evolving Internet Architecture, MobiArch 2015, Paris, France, September 7, 2015*. 2015, pp. 20–25. doi: 10.1145/2795381.2795388. url: https://doi.org/10.1145/2795381.2795388.

[11] *G-Store*. url: http://g-store.sourceforge.net/.

[12] F. Gessert, W. Wingerath, S. Friedrich, and N. Ritter. "NoSQL database systems: a survey and decision guidance." In: *Computer Science - R&D* 32.3-4 (2017), pp. 353–365. DOI: 10.1007/s00450-016-0334-3. URL: https://doi.org/10.1007/s00450-016-0334-3.

[13] *HBase*. Apache. URL: https://hbase.apache.org/.

[14] *Hypertable*. URL: http://www.hypertable.org/.

[15] A. Lakshman and P. Malik. "Cassandra: a decentralized structured storage system." In: *Operating Systems Review* 44.2 (2010), pp. 35–40. DOI: 10.1145/1773912.1773922. URL: https://doi.org/10.1145/1773912.1773922.

[16] N. Leavitt. "Will NoSQL Databases Live Up to Their Promise?" In: *IEEE Computer* 43.2 (2010), pp. 12–14. DOI: 10.1109/MC.2010.58. URL: https://doi.org/10.1109/MC.2010.58.

[17] F. T. Leighton and P. W. Shor, eds. *Proceedings of the Twenty-Ninth Annual ACM Symposium on the Theory of Computing, El Paso, Texas, USA, May 4-6, 1997*. ACM, 1997. ISBN: 0-89791-888-6.

[18] *Megastore*. Google. URL: https://perspectives.mvdirona.com/2011/01/google-megastore-the-data-engine-behind-gae/.

[19] *mobiThinking. Global mobile statistics 2013 part a: Mobile subscribers; handset market share; mobile operators*. mobiThinking. 2013. URL: https://mobiforge.com/research-analysis/global-mobile-statistics-2014-part-a-mobile-subscribers-handset-market-share-mobile-operators.

[20] *MongoDB*. URL: https://www.mongodb.com/.

[21] *RethinkDB*. URL: https://www.rethinkdb.com/.

[22] J. Rodrigues, E. R. B. Marques, L. M. B. Lopes, and F. M. A. Silva. "Towards a middleware for mobile edge-cloud applications." In: *Proceedings of the 2nd Workshop on Middleware for Edge Clouds & Cloudlets, MECC@Middleware 2017, Las Vegas, NV, USA, December 11 - 15, 2017*. ACM, 2017, 1:1–1:6. DOI: 10.1145/3152360.3152361. URL: https://doi.org/10.1145/3152360.3152361.

[23] J. A. Silva, R. Monteiro, H. Paulino, and J. M. Lourenço. "Ephemeral Data Storage for Networks of Hand-Held Devices." In: *2016 IEEE Trustcom/BigDataSE/ISPA, Tianjin, China, August 23-26, 2016*. 2016, pp. 1106–1113. DOI: 10.1109/TrustCom.2016.0182. URL: https://doi.org/10.1109/TrustCom.2016.0182.

[24] J. A. Silva, J. Leitão, N. M. Preguiça, J. M. Lourenço, and H. Paulino. "Towards the Opportunistic Combination of Mobile Ad-hoc Networks with Infrastructure Access." In: *Proceedings of the 1st Workshop on Middleware for Edge Clouds & Cloudlets, MECC@Middleware 2016, Trento, Italy, December 12-16, 2016*. Ed. by R. Martins and H. Paulino. ACM, 2016, p. 3. URL: http://dl.acm.org/citation.cfm?id=3022873.

[25] J. Silva, H. Paulino, J. M. Lourenço, J. Leitao, and N. Preguica. "Time-Aware Reactive Storage in Wireless Edge Environments." In: *Proceedings of the 16th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services, Houston, United States, November 12-14, 2019*. ACM, 2019.

[26] A. Teófilo, D. Remédios, J. M. Lourenço, and H. Paulino. "GOCRGO and GOGO: Two Minimal Communication Topologies for WiFi-Direct Multi-group Networking." In: *Proceedings of the 14th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services, Melbourne, Australia, November 7-10, 2017*. Ed. by T. Gu, R. Kotagiri, and H. Liu. ACM, 2017, pp. 232–241. DOI: 10.1145/3144457.3144481. URL: https://doi.org/10.1145/3144457.3144481.

[27] U. of Toronto. *Apache Cassandra NoSQL Performance Benchmarks*. https://academy.datastax.com/planet-cassandra/nosql-performance-benchmarks. 2012.

[28] I. University. *Create a symbolic link in Unix*. https://kb.iu.edu/d/abbe. [Last modified on 2019-08-27 08:51:18]. 2019.

[29] P. Vieira. "A Persistent Publish/Subscribe System for Mobile Edge Computing." Master's thesis. FCT NOVA, Nov. 2018.

2019 **Key-Value Storage for handling data in mobile devices** **José Afonso**

José Pedro Serra Afonso

Bachelor in Computer Science and Engineering

Key-Value Storage for handling data in mobile devices

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática

December, 2019

José Pedro Serra Afonso

Bachelor in Computer Science and Engineering

Key-Value Storage for handling data in mobile devices

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática

December, 2019

Copyright © José Pedro Serra Afonso, Faculdade de Ciências e Tecnologia, Universidade NOVA de Lisboa.
A Faculdade de Ciências e Tecnologia e a Universidade NOVA de Lisboa têm o direito, perpétuo e sem limites geográficos, de arquivar e publicar esta dissertação através de exemplares impressos reproduzidos em papel ou de forma digital, ou por qualquer outro meio conhecido ou que venha a ser inventado, e de a divulgar através de repositórios científicos e de admitir a sua cópia e distribuição com objetivos educacionais ou de investigação, não comerciais, desde que seja dado crédito ao autor e editor.

José Afonso

2019

Key-Value Storage for handling data in mobile devices

2019