Pedro Miguel Castanheira Sanches

Aluno Nº 41599 (MIEI)

Distributed Computing in a Cloud of Mobile Phones

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática

Orientador: Hervé Miguel Cordeiro Paulino, Prof. Auxiliar, Universidade Nova de Lisboa

Júri

Presidente: Carmen Pires Morgado
Vogais: Rolando da Silva Martins
Hervé Miguel Cordeiro Paulino

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Setembro, 2017

Distributed Computing in a Cloud of Mobile Phones

Este documento foi gerado utilizando o processador (pdf)LaTeX, com base no template "unlthesis" [1] desenvolvido no Dep. Informática da FCT-NOVA [2].
[1] https://github.com/joaomlourenco/unlthesis [2] http://www.di.fct.unl.pt

Aos meus pais e irmão.

Acknowledgements

I am thankful to my thesis adviser, Hervé Paulino, for his guidance throughout the elaboration of this thesis.

I am also thankful to my family for their support and patience for the past five years.

Abstract

For the past few years we have seen an exponential growth in the number of mobile devices and in their computation, storage and communication capabilities. We also have seen an increase in the amount of data generated by mobile devices while performing common tasks. Additionally, the ubiquity associated with these mobile devices, makes it more reasonable to start thinking in a different use for them, where they will begin to act as an important part in the computation of more demanding applications, rather than relying exclusively on external servers to do it.

It is also possible to observe an increase in the number of resource demanding applications, whereas these resort to the use of services, offered by infrastructure Clouds. However, with the use of these Cloud services, many problems arise, such as: the considerable use of energy and bandwidth, high latency values, unavailability of connectivity infrastructures, due to the congestion or the non existence of it. Considering all the above, for some applications it starts to make sense to do part or all the computation locally in the mobile devices themselves.

We propose a distributed computing framework, able to process a batch or a stream of data, which is being generated by a cloud composed of mobile devices, that does not require Internet services. Differently from the current state-of-the-art, where both computation and data are offloaded to mobile devices, our system intends to move the computation to where the data is, reducing significantly the amount of data exchanged between mobile devices.

Based on the evaluation performed, both on a real and simulated environment, our framework has proved to support scalability, by benefiting significantly from the usage of several devices to handle computation, and by supporting multiple devices submitting computation requests while not having a significant increase in the latency of a request. It also proved that is able to deal with churn without being highly penalized by it.

Keywords: Mobile Edge Cloud Computing, Stream Processing, Distributed Computing, Android

Resumo

Nos últimos anos, temos visto um crescimento exponencial no número de dispositivos móveis e nas suas capacidades computacionais, de armazenamento e de comunicação. Nós também vimos um aumento na quantidade de dados gerados pelos dispositivos móveis enquanto executam tarefas do dia-a-dia. Além disso, a ubiquidade associada a estes dispositivos móveis, torna mais razoável começar a pensar num uso diferente para os mesmos, onde começarão a agir como uma parte importante na computação de aplicações mais exigentes, ao invés de confiar exclusivamente em servidores externos para o fazer.

Também é possível observar um aumento no número de aplicações mais intensivas em termos de recursos, sendo que estas recorrem ao uso de serviços, oferecidos por infraestruturas Cloud. No entanto, com o uso destes serviços, vários problemas surgem: o considerável uso de energia e de largura de banda, valores de latência elevados, indisponibilidade de infraestruturas de conectividade, devido a congestão ou à não existência das mesmas. Considerando o que foi mencionado, para algumas aplicações começa a fazer sentido efectuar parte ou toda a computação localmente nos próprios dispositivos móveis.

Nós propomos uma framework de computação distribuída, capaz de processar um conjunto ou um fluxo de dados, que está a ser gerado por uma cloud composta por dispositivos móveis, que não requer a utilização de serviços de Internet. Diferentemente do estado de arte atual, onde a computação e os dados são enviados para dispositivos móveis, o nosso sistema pretende mover a computação para onde os dados estão, reduzindo significativamente a quantidade de dados trocados.

Com base na avaliação realizada, tanto em ambiente real e simulado, a nossa framework provou suportar escalabilidade, beneficiando significativamente do uso de vários dispositivos para lidar com a computação, e sendo capaz de suportar a submissão de vários pedidos de computação, sem aumentar significativamente a latência de um pedido. Também provou ser capaz de lidar com churn sem ser altamente penalizada por isso.

Palavras-chave: Computação em "Clouds" Movéis na Periferia, Processamento de Fluxo, Computação Distribuída, Android

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Mobile devices are changing our everyday life style. Whether we use our mobile devices to check our emails, access social networks or use them on more demanding applications, like gaming, the reality is that mobile devices changed, and are still changing core aspects of our daily style. According to Cisco, the number of mobile devices and connections has grown from 7.6 billion in 2015, to 8.0 billion in 2016 [6]. Moreover, according to that same source, by 2021, the expected number of mobile-connected devices will be around 11.6 billion, increasing the number of mobile devices per capita to 1.5.

Mobile devices have also grown in terms of computational power and storage. Users feel the need for more powerful mobile devices, in order to achieve a better user experience when executing increasingly demanding applications [15]. For the past years, at least one new CPU core design has been released each year, one more advanced than its predecessor [15]. We are thus witnessing a paradigm shift, where these devices' resources are starting to be used for local computation, instead of relying on external servers.

Alongside with the fast increase rate in computational power of mobile devices, mostly smartphones, the amount of data which needs to be handled is also increasing, due to the rise of the amount of data each user generates. Besides, the number of applications that rely heavily on resource usage also have been increasing substantially. In the past few years this has been handled by offloading data and tasks of the applications to infrastructure Clouds, also simply known as Clouds. Clouds host services that provide computing resources to the users in a ***'pay as you go manner'*** [22]. These services are usually hosted in large data centers owned by big companies such as Amazon and Google [10].

One of the problems of using Cloud services through mobile devices is that they still use a significant amount of energy and bandwidth. Meanwhile, the period of time it takes from the moment a specific data is sent to be processed in the Cloud to the moment the output is returned, is way too considerable in order to be used by applications that

require almost an immediate response user interaction [10]. This can happen because the access to the Cloud infrastructure is weak or non existent. Also, in many cases the amount of data that is sent is way too large and sometimes not all information is relevant.

Given the increasing generation of data by the users and the exponential growth of mobile devices, in terms of number and computational power, it begins to make sense for some applications, not all, to process data in the edge/periphery, where they were generated, instead of offloading them to the Cloud. For such, one possible solution consists of the aggregate use of multiple geographically nearby mobile devices' resources.

1.1 Motivational Example

Consider Alice, that is attending a party. During the party, Alice will be taking photos with other people using her own mobile phone, and other people will also be taking photos with Alice using their own mobile phones. By the end of the party, Alice decides to use an application that allows her to see all the photos taken during the party and choose the ones she wants. To use this application she will have to upload her photos of the party to the Cloud, so others can use the same service. After uploading her photos, Alice decides that she wants all photos where she shows up. Instead of going through all photos, one by one, she uses one of the features of the application: facial matching algorithm. This feature allows for the recognition of an individual based on its facial features. Since facial matching algorithms consume an high amount of energy and take long time to execute [29], this computation is offloaded to the Cloud.

Even though, the application used in this example seems to be very practical, useful and flexible to support many different scenarios, this is not entirely true. One of the problems that arise in this example, is the amount of data that needs to be uploaded and downloaded to a "folder" of the service responsible for the facial matching algorithm, while using the application. The user has to upload some of the photos, maybe all, she photographed during the party, transmitting already a large amount of data through the network. After applying the face matching algorithm over the collection of all photos from the party, the photos selected by the algorithm will be downloaded to the user's mobile phone, transmitting once again, a large amount of data through the network. This can be a problem if the user does not have access to a WiFi network, due to congestion or non existence of it, forcing her to use other mobile technologies like 3G/4G, which implies spending a certain quantity of money, depending on the mobile data plan, to have access to the Internet. In addition, even if the user has access to a Wi-Fi network, the network could be unnecessarily overloaded with data that could be treated locally.

Another problem that arises from using this application is the time it takes to complete the entire process of getting photos the user wants. This time, known as latency of the service, can potentially be high, when the distance between the Cloud infrastructure and the end user is of several hundreds to thousands of kilometers, when compared with a more local solution, where the service latency will solely depend of the processing time.

In this application, response time is not a core aspect to its right functioning, because even with a slow response time the results will still be the ones expected. The same does not happen in applications that require real time response, like augmented reality applications [39] or applications that warn users for an immediate event [10].

One other problem needs to be taken into account, and that is the previous setup of the service. In order to make this service available to all users, it is necessary to configure the service before the party, for example, by creating a public folder where all the photos will be uploaded, by providing this folder to the Cloud service responsible for the facial matching algorithm and ensuring that all users are aware of the folder where they will need to upload these photos for.

The last problem that is not considered in the example described is related to network connectivity. In this example it is assumed the constant availability of infrastructure connectivity, however in a real life situation, this does not always happens. An example of a scenario would be a concert, located in a remote location, such as in the middle of a farm field, where it would not exist Wi-Fi access-points. Although, people could choose to use 3G connections, this has a financial cost associated to it. Another example, would be if this concert was located in a wide area, with a considerable capacity size, like a stadium or mall, where, even though a connectivity infrastructure is available, its services might not be available because of the weak or degraded signal caused by congestion, which is a consequence of an overloaded infrastructure [7].

Based on all of the problems described, to certain applications it starts to make sense to partially or fully compute data in a more local way, while also allowing users to use services offered by infrastructure Clouds, locally, while having fast response times, decreasing the amount of data transferred through the network and without being bothered whether there is an available infrastructure connectivity or not. This can be possible by using a distributed computing system for clouds of mobile phones.

1.2 Computing at the Edge

Edge Computing is in the base of all the concepts addressed in this section. It basically focuses on the idea of performing computation closer to or where the data is.

Edge Cloud services provide computational resources and data storage closer to the end users, allowing those users to have access to services that a Cloud would offer, while consuming very little bandwidth and having a very fast response time [10]. This occurs because Edge Cloud services are localized at the edge of the network, with that being, the proximity which exists between the end user devices and the edge of the network is less, when compared to the proximity between the end user and infrastructure Clouds. This will reduce the latency experienced, consequently, decreasing the consumption of bandwidth and decreasing the response time. Some of these Edge Cloud services are prepared to forward data, to be processed, by more capable infrastructure Cloud services.

One way of dealing with the high latency, inherent to offloading tasks to Cloud infrastructures, is through the use of Cloudlets. Cloudlets are formed by a computer or a cluster of computers, that are rich in resources and can be trusted, located near or alongside Wi-Fi access points [16]. This proximity grants mobile devices a new source of computation near them, reducing high latency and power consumption. In this way, resource expensive applications, like augmented reality applications [39], that require a real time interactive response, are no longer a problem, since mobile devices can simply send tasks to Cloudlets to be processed and wait for the results.

Extending the concept of Cloudlet, Fog Computing, also addresses high latency and slow response time problems. Fog Computing can be seen as a group of devices, with wireless capabilities, that communicate with each other and the network in order to process tasks and storage data, without the use of Cloud infrastructures [11]. These devices are virtualizations with their own data storage, computing and communication facility [19].

Once again, proximity to mobile users, is a core aspect of Fog Computing. Most of these systems are deployed in locations with a large number of mobile users in the vicinity, providing fast computing and data storage services due to this proximity and wireless connections. Although the services and computing that these Fog Computing devices offer can be done locally, when connected to the Internet, a device can always choose to forward data to be processed by more capable infrastructure Cloud services, in order to manage its resources [19].

With the significant growth of the number of mobile devices per capita, and with the creation of the concept of Edge Clouds, a new technology arose, Mobile Edge Clouds [7]. The Mobile Edge Cloud concept refines the previously mentioned concept of Edge Clouds, where instead of also having the computation closer to the data, the computation is done directly where the data is. Mobile Edge Cloud services are provided by a cluster of mobile devices, that individually may not be able to handle most of the tasks to process, however, when organized as a cluster, these devices are able to do so with ease [7]. The cluster formed by these mobile devices empower the overall system, in terms of computation, allowing tasks that are not very computational demanding, to be processed locally, at the edge, instead of being sent to the Cloud. In terms of storage, the total amount of data that can be stored will increase, for example, if we consider 4 mobile devices with 10 GB of available data storage each, the overall system will provide a total amount of 40 GB of data storage. In terms of computation, the total amount of data that can be processed at the same time will increase or the time it takes to process a certain amount of data will be much lower. If we consider 4 mobile devices, Samsung Galaxy S6 with 4 A57 cores 2.1GHz + 4 A53 cores 1.5GHz [15], the overall system will offer a total of 32 cores. Imagining that with a mobile device, 8 cores, we can process for example 20 photos in 10 seconds, with 4 mobile devices we can possibly process those same 20 photos in 2.5 seconds, or another possibility, we can process 80 photos in 10 seconds. This can be very useful in the example described in Section 1.1.

The communication between mobile devices that take part of the cluster is possible by using local network communication, e.g., Wi-Fi, Wi-Fi Direct, Bluetooth and ad-hoc [7]. Similar to Edge Clouds, network connectivity is a very important aspect in these systems. The need for an infrastructure or a reliable infrastructure is no longer an obstacle for these systems, since they can still be used in locations where the connectivity is weak, congested or not consistent [7].

1.3 Problem Statement

This thesis aims to address a problem, which is, the creation of a system capable of processing a batch or a stream of data, that is being generated in a network formed by mobile devices, without needing to resort to Internet services.

The existing work in distributed computing over mobile devices, allows for the distribution of computation and data to other mobiles devices, similar to what happens in a cluster of server nodes. The model that we are proposing in this thesis, conceptually different, proposes a system where the computation is performed where the data is. This means that, the computation is offloaded to every node where the specific data is and it will be up to these nodes to perform such computation. With this solution we pretend to reduce the amount of data transferred between devices. Considering the above stated and the context of the example presented in section 1.1, we wish to provide mobile devices' the ability to process computational intensive tasks, with the help of a network composed of multiple devices in the vicinity, by offloading work to some of those nodes. This way, users would experience short latency values, would not require Internet services and would no longer need to transfer large amounts of data when offloading work.

In order to devise and implement a system like the one proposed in this thesis, many challenges arise:

- How to find the data to be computed in a network of mobile devices?
- How to distribute the computation to the mobile devices?
- How to detect and deal with the entrance and exit of mobile devices from the network, (churn)?
- How to design this system to be generic, in order to support a wide variety of mobile applications?
- How to devise a system that is able to apply a computation to an unbounded dataset, i.e. to data freshly generated by the devices that compose the system?

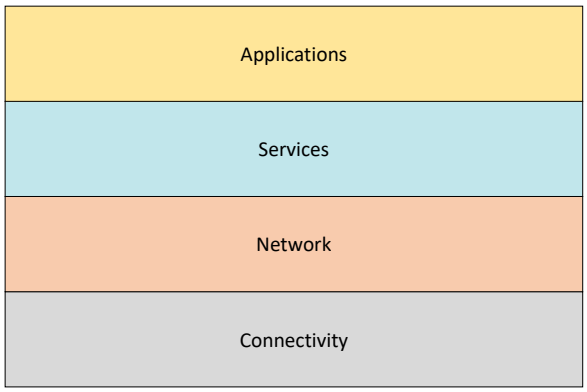


Figure 1.1: Hyrax's middle ware stack.

1.4 Solution

The objective of this thesis is to design and implement a distributed computing framework, capable of processing batches and streams of data, being generated by mobile devices in a network, without resorting to services on the Internet. The system presented in this thesis is part of a larger project, Hyrax[1], whose main objective is to change the paradigm on mobile devices, where instead of using them as only an input medium for computing and storage in external sources, mobile devices can be used to form a Cloud that uses the computing and storage capabilities as well as their proximity to each other, in order to benefit the owners of the devices. Hyrax project is composed of several teams of different institutions, where each teams helps to develop a complete set of middle ware, focusing on diverse areas like connectivity, network, services and applications, as observed in Figure 1.1. Our thesis falls into the Services layer, contributing with a distributed computing service, abstracting itself from details about other layers, such as whether the network supports single hop or multi-hop communication.

Aiming to address the problems enumerated in Section 1.3, our system must be able to process data that has already been stored and shared between devices in the network, as also be able to process data that is being produced in real time. Data sharing and dissemination is assured by Thyme [3, 27], a time-aware storage and dissemination system with publish and subscribe mechanisms, that stores and shares data in a network of mobile devices. This system is being developed in the Hyrax project context, also being part of the Services layer.

Our solution presents a programming and execution model based on the manipulation of datasets, following what was proposed in [23]. These data sets correspond to a *Mobile Dynamic Dataset (MDDS)* that organise data logically. Each *MDDS* is associated with a tag. Using the example given in Section 1.1, in that example we would have at least two tags, a result tag where Alice's results would be stored, i.e. '#my_results', and the other would be a tag used to publish the photos taken during the party, i.e. '#this_party'.

[1] http://hyrax.dcc.fc.up.pt/

The *MDDSs* are composed of various data which may be distributed by different mobile devices. Whenever a new computation is performed over a *MDDS* a new one is generated being associated with a result tag.

In what concerns the scheduling of the computation, we leverage on Thyme to obtain the location of all the files published in the network, allowing our *framework* to choose which nodes should process data.

In terms of how to deal with churn, the storage system provides the ability to mobile devices to download data from other devices, creating replicas of the original data in each mobile device that downloaded it. The storage system, also offers the possibility of configuring the active replica feature. In both cases, it is necessary to choose which node will be assigned the computation, detect if the computation was successfully finished and if not, then it is necessary to reassign the failed task to another device that is in possession of one of the replicas.

Based on the system described above, we intend to address all the problems enumerated in Section 1.3, presenting a solution that is considerably different from the existing state of the art.

1.5 Contributions

In this thesis we devise a distributed computing framework, capable of processing batches and streams of data, in a network composed of mobile devices.

- Definition of a distributed computing model targeting networks formed exclusively by mobile devices.
- Design and implementation of an Android prototype for the framework proposed in this thesis.
- Performance evaluation of the prototype to be developed, based on scalability, time dependent on the environment tested, resource usage, cost of message exchange and how churn is handled.

1.6 Publications

We were able to publish one article from the work developed on this thesis:

- "Computação Distribuída em Redes Formadas por Dispositivos Móveis", [24] INForum, where we presented the prototype implemented and some evaluations.

1.7 Report organization

This thesis is composed of five other chapters besides this one. Chapter 2 starts by explaining in more detail the concepts of Fog Computing and Mobile Edge Cloud Computing. It

then addresses several state-of-the-art solutions, while presenting a summary table and a critical analysis about them. It finishes with distributed computing frameworks for cluster environments that support data stream applications. Chapter 3 starts by giving a description of the system and its workflow. It then presents the system's model, API and Architecture. In Chapter 4 we address what technologies were used and some implementation details regarding the API and the Architecture. In Chapter 5 we present and discuss the experimental evaluation of the prototype devised. In Chapter 6 we conclude this thesis by presenting what was achieved and future work.

Chapter 2

Related Work

In this Chapter we present pertinent concepts for our system and address some of the existent related work. The two first Sections, 2.1 and 2.2, explain in more detail what are Fog Computing and Mobile Edge Cloud Computing, since this thesis presents a system which is related to both concepts, being part of the scope of the latter. The third Section 2.3 presents the state-of-the-art in Mobile Edge Computing, explaining it in more detail and drawing conclusions about it. The fourth Section 2.4 presents different distributed computing frameworks to process data streams on clusters of server nodes.

2.1 Fog Computing

Fog Computing is a relatively new idea/concept, which makes it hard to find an unanimous definition among different papers. Even though, a consensus is not achieved, Fog Computing is considered by most as an extension of Cloud Computing, where the Cloud is closer to the ground [2, 19, 28]. This proximity to "the ground", is the key aspect of Fog Computing, in which a group of heterogeneous devices, with wireless, some with storage and computation capabilities, communicate with each other, to process tasks and store data without the need for a Cloud infrastructure. Although there is no necessity for a Cloud infrastructure, Fog Computing is considered by many to be a system composed of three layers, as observed in Figure 2.1, Mobile, Fog and Cloud, where the role of the Fog layer is to work as a bridge, between Mobile and Cloud [19]. The role played by this Fog layer exhibits an idea of duality, where the fog nodes can offer storage, computation, data and application services, through wireless communications like Wi-Fi or Bluetooth, using its own resources. But at the same time, these fog nodes can communicate with the Cloud, over the Internet, offering more capable resources in terms of computation, storage and services.

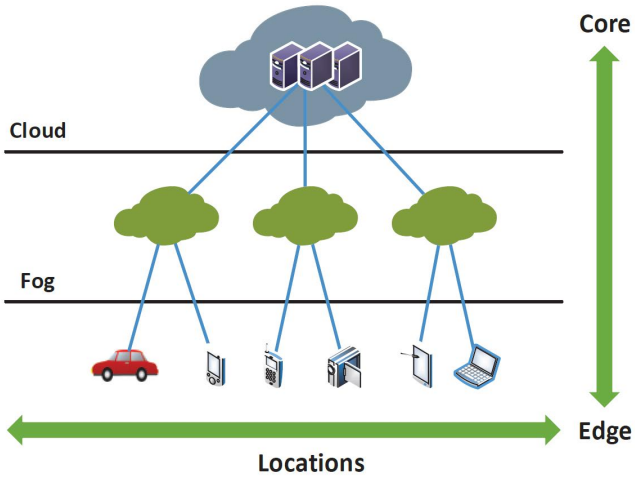


Figure 2.1: Fog Computing three level hierarchy (adapted from [28])

A fog layer is formed by devices, called fog nodes, which are deployed in locations with a large number of mobile users in the vicinity, like shopping malls, stadiums, factories, etc. These nodes can be any device with computational, storage and communication facility, like servers, routers, mobile devices, etc [5]. Besides being able to connect to the Cloud and to mobile devices, these nodes can also communicate among each other, to increase the overall system performance, through data routing [19].

There are three major motivations for the use of Fog Computing, which the current Cloud Computing model can not address, such as: location awareness, low latency and bandwidth cost. Bandwidth cost can be expensive, when considering a scenario where a mobile user wants to know an information about a statue on a museum, but instead of obtaining that information from close by fog nodes, the mobile user is forced to use the cellular network to retrieve that information, which was uploaded beforehand, from a Cloud.

In terms of location awareness, taking into account a scenario where a large museum, divided into three different physical areas, where each physical area corresponds to a different subject, if we were to discover a specific information about a piece of art in one of those areas, instead of retrieving that information from the Cloud, one could simply get the specified information through the fog node associated with that area, improving its quality of service.

Low latency is also present in the previous scenario, since the mobile user chooses to retrieve the information from a fog node which is closer to the mobile user, instead of using the Cloud. This proximity decreases the latency value, granting a faster response time and the possibility of using applications that require almost real time responses. This is one of the reasons why Internet of Things (IoT) is highly associated to fog computing. The current Cloud model does not support the requirements of IoT, mainly because, in most cases, the distance between the IoT devices and the Cloud infraestructures are too large, therefore the latency experiencied is too high. Due to the high latency, some

of the data generated in IoT can not even be considered, since by the time that data is processed, the requester device might be in a different state, where that data is no longer relevant. One example of this situation is given in [5], where in a chemical vat the temperature is reaching an alarming level and a correction is needed to be taken immediately. Considering the time it takes to transfer the temperature information to the Cloud and to receive a response, indicating which action should be performed, using the Cloud may not be the best solution for this system.

2.2 Mobile Edge Cloud Computing

Similar to Fog Computing, Mobile Edge Cloud Computing is an idea/concept for which it is difficult to find an unanimous definition, or even a coherent definition among different papers, mainly because it is an idea that comes from a mixture of other ideas like Edge Cloud Computing and Mobile Cloud Computing. Like many others, [18], when trying to present a definition for Mobile Edge Cloud Computing usually tends to one of two possible sides, or Mobile Cloud Computing or Edge Cloud Computing, in the case of the article in question this tends to Edge Cloud Computing. Another problem associated with the definition of Mobile Edge Cloud Computing arises from the different definitions for Mobile Cloud Computing. As explained in [8], a survey that addresses these subjects, there are three common different definitions for Mobile Cloud Computing. It can be seen as running a certain mobile application on an infrastructure Cloud, using the mobile device like a thin client. One other view, which is actually closer to the definition of Fog Computing where the Cloudlet is a fog node, is related to offloading computationally demanding tasks to a nearby Cloudlet, which is connected remotely to an infrastructure Cloud over the Internet, using the mobile device as a thin client to the Cloudlet. One other common definition for Mobile Cloud Computing, also used in [7], is based upon the idea of having a cloud comprised by mobile devices, connected over Wi-Fi, Bluetooth or ad-hoc, where these mobile devices share their resources in order to process certain tasks, store certain data and sense the environment for computing purposes. This last approach, where there is a cloud formed by mobile devices that processes data, is the one we consider to provide the best definition for Mobile Edge Cloud Computing.

Similar to Fog Computing, Mobile Edge Cloud Computing has location awareness and low latency as motivations for its use. Besides these two, there are also three other motivations, network connectivity, the increasing amount of resources in mobile devices and privacy.

The increasing amount of resources in mobile devices, makes them more powerful, allowing the usage of their resources in a mobile cloud, in order to use mobile applications that are computationally intensive. This approach changes how mobile devices are viewed, making them part of the workers instead of simply doing requests [7, 8].

Location awareness can be seen as a motivation for the use of Mobile Edge Cloud

Computing, precisely context aware services and personalized experiences [8], since mobile devices are capable of sensing the environment around them, and with a cloud of mobile devices, more and different sensing capabilities will be used, providing a deeper understanding of the environment where the user is, allowing for those services and experiences.

Low latency can be achieved when using Mobile Edge Cloud Computing simply because instead of offloading a certain task to an infrastructure Cloud, this task is offloaded to nearby mobile devices which are much closer to the mobile user, significantly reducing the latency value. The same applies when retrieving the results of the said task that was previously offloaded. It is possible to observe this behaviour in the example described in 1.1. In that case, instead of bringing the data towards the computation, basically offloading those photos to an infrastructure Cloud, the computation is brought towards the data, which means, it is done by a group of mobile devices, whose owners are attending the event, thereby reducing the latency value significantly.

Network connectivity is also a motivation for the use of Mobile Edge Cloud Computing, in the sense that, when in a scenario where there is no network connectivity, a mobile user will still be able to use services similar to the ones offered by an infrastructure Cloud, since the mobile device is part of a cloud composed of mobile devices which are able, in a group, to process tasks and store data. This lack of network connectivity can be caused by the non existence of network infrastructures, due to a user being located in a remote place [7] or due to a natural disaster that destroyed most of the network infrastructures [8]. It can also be caused by the congestion of some network infrastructures which may lead to the impossibility of using Cloud infrastructure services [7].

Privacy can also be seen as motivation for the use of Mobile Edge Cloud Computing by some, as users are no longer forced to trust Cloud infrastructure service providers their data, simply because the data is stored and processed by mobile devices in the vicinity. At the same time, some could argue that the privacy subject still arises in these scenarios, where now instead of trusting in Cloud infrastructure service providers, the user would have to trust mobile devices in the vicinity. In these cases, the way the data dissemination and computing system are built, influences greatly the matter of offering privacy to the end user, being ultimately the responsibility of the developer to provide some privacy and of the user in choosing which data he/she should or not share.

2.3 Computing in mobile (Android) device networks

In this subsection we aim to present different existent solutions that address some of the motivations discussed earlier. In Section 2.3.1, we will firstly give an overall description of all the systems. We will then, address the system model, the programming model and the execution model of the solutions proposed, while providing a critical analysis about the systems in study. In Section 2.3.2, we will address implementation details and the workflow of the solutions proposed.

The solutions being discussed in this section are part of the same scope of frameworks which support Mobile Cloud Computing application development: **Hyrax** [30], **Serendipity** [25], **MClouds** [20], **Service-Oriented Heterogeneous Resource Sharing (SOHRS for short)** [21], **FemtoClouds** [14], **Honeybee** [9], **P3-Mobile** [26].

We will proceed with a brief description and comparison of all these systems:

- **Hyrax**, can be seen as motivation for the work done in this thesis. Hyrax is a Mobile Cloud Computing infrastructure which adapted an already existent Cloud Computing platform, in this case Apache Hadoop MapReduce [37], to a mobile environment, where local resources are used for storage and computational processing purposes.

- **Serendipity** is a system developed with the objective of offering mobile devices in need of some computation capability, remote computational resources that are granted by other mobile devices in the vicinity. This system works towards decreasing the computing time and energy consumption of certain application's tasks. This system also has in consideration the intermittent connectivity between mobile devices and locality awareness.

- **MClouds** are clouds composed of mobile devices that offload computation to other mobile devices or a fixed Cloud, when a certain task can not be completed on a single device because of the lack of resources. In this system, a mobile device that wishes to offload computation to a mCloud is considered a master *mDev*, while the mobile devices that offer themselves to compute certain subtasks, slave *mDevs*.

- **SOHRS** proposes an architecture and mathematical framework for heterogeneous resource sharing via service oriented utility functions. These services are used through applications installed in mobile devices and are composed by several tasks.

- **FemtoClouds** are mobile clouds composed of several mobile devices located at the edge, based on the idea of a Cloudlet. Different from the approach used on Cloudlets, the computation is all done by the mobile devices, whereas the managing of the femtocloud and the distribution of multiple tasks to the mobile devices is done by a fixed device/control device.

- **Honeybee** is a mobile edge cloud composed of mobile devices that share resources between them for computation purposes. In this system the master node is known as a *delegator node* which has a job queue, from where multiple workers will steal jobs. Also, it is considered that a task is divided into several jobs.

- **P3-Mobile** is a parallel computing model, based on another parallel peer-to-peer P3 model, which was adapted to work on a mobile environment. The architecture of P3 Mobile is based on services, having four main services that define it: *Link service*, *Network service*, *Storage service* and *Parallel Processing service*.

2.3.1 Critical Analysis

In this subsection we will present an overview and analyze, critically, the proposed solutions, taking into consideration Subsection 2.3.2 and the related work summary Table 2.1.

Solution	Communication Protocol	Computation	Organization	Able to deal with Churn?	Programming Model	Data Processing Model	Obtaining Results
Hyrax	Wi-Fi	Work giving	Master slave	Yes	Task-based	Batch	Hadoop Distributed file system
P3-Mobile	Wi-Fi (Also supports Wi-Fi Direct and Bluetooth)	Work stealing/searching	Master slave	Yes	Task-based	Batch	Send
Serendipity	Wi-Fi	Work giving	Master slave	Yes	Task-based	Batch	Send
MCloud	Wi-Fi	Work giving	Master slave	Yes	Task-based	Batch	Send
SOHRS	Wi-Fi, Bluetooth	Work giving	Master slave	Not addressed	Task-based	Batch	Send
FemtoCloud	Wi-Fi	Work giving	Master slave	Yes	Task-based	Batch	Notification
Honeybee	Wi-Fi Direct	Work stealing/searching	Master slave	Yes	Task-based	Batch	Send
Ours	Wi-Fi, Bluetooth, Wi-Fi Direct	Work giving	Peer-to-Peer	Yes	Streaming	Batch and Streaming	Notification and Distributed File System

Table 2.1: Related work summary table.

For better clarity, this subsection is subdivided into three topics:Programming Model, System Model and Execution Model, where defining characteristics of the multiple solutions will be addressed.

Although the solutions proposed present a lot of differences between them and between our solution, in what concerns the systems themselves and the goals to achieve with those systems, there is one common objective, shared by all: devising a system capable of providing distributed mobile computing.

2.3.1.1 Programming Model

How applications are programmed Not all the studied systems detail the programming model. The ones that do, expose a framework that requires the concrete implementation of some methods and allow the programmer to override them by default behaviour of others. These methods define how a task should be divided and executed, and are called by the framework, abstracting a whole set of details which should not matter to an application developer that wishes to develop an application that supports distributed mobile computing. The systems addressed are implemented in Android. Only **Hyrax**, **P3-Mobile**, **Serendipity** and **Honeybee**, address how applications should be programmed when using their solutions.

Programming model All of the studied systems follow a task-based programming model which raises some limitations to the type of applications we want to support. The model in question, focus in applying computation to a set of data and returning the results. With our solution, when work is given/distributed to some device, the computation consists in applying multiple sequential transformations/operations over a dataset creating a new one after each operation. By following a streaming programming model approach, we are able to apply an operation to a dataset, publish the results, and in the next work distribution, apply computation over the published results.

Data processing model Just like the programming model, the data processing model is also not discussed explicitly in most of the solutions, nonetheless, based on the descriptions of those solutions, we could conclude that all of them follow a batch data processing model. This approach presents some limitations to what we pretend to offer with our solution because it can only deal with applications that follow a batch data processing model, differently from our solution, that besides supporting batch it also supports streaming of data, covering a larger base of distributed mobile computing applications. Since we can support both batch and streaming of data, our solution can handle tasks that execute for a predefined period of time, computing every new data that arrives to the mobile device and returning the result of that computation. In order to achieve the same goal with a solution that only supports batch data, the common approach would be the creation and submission of a new task every time new data arrives to the mobile device.

2.3.1.2 System Model

Organization of the systems The majority of the solutions encountered display some similarities, in terms of: 1) offloading computation to a cloud composed of only mobile devices, 2) the usage of a master slave execution model and 3) the nonexistent necessity of an internet connection. However, Serendipity is the only one to actually combine all these three characteristics. The rest of the solutions exhibit some small design differences in at least one of the three characteristics. In **Hyrax** and **FemtoCloud**, the cloud is composed of both mobile and stationary devices. **MClouds** and **SOHRS** are able to establish an Internet connection to offload some computation to a fixed Cloud. **P3-Mobile** and **Honeybee** present a master slave execution model which is different from the ones used in other solutions, since both of these solutions display a work-sharing parallel programming model. With this being said, none of the solutions addressed in this subsection exhibit the typical Peer-to-Peer execution model since all of them at one point have the idea of one master node responsible for different coordination features than the ones its peers have.

We do not consider the Master Slave model to be the best suited model for this thesis' proposed distributed mobile computing system, therefore, our solution uses a Peer-To-Peer (P2P) model. The Master Slave model raises some problems when we think of

a system with only one master node that fails, which will consequently jeopardize the mobile cloud, being a single point of failure. Even in systems, like **P3-Mobile**, where there is at least two coordinators per coordination cell, when these two fails, the coordination cell will loose valuable meta data for management purposes.

One other problem associated with the Master Slave model, that is present in the majority of the solutions, even though is not addressed explicitly, except for solutions like **Honeybee** and **P3-Mobile**, where it is said that the cloud is dismantled after the master node has its task fully computed, is the persistence of the mobile cloud. With the dismantle of the cloud, when another device chooses to form a new cloud, it is not guaranteed that the old master node will join this new cloud, which means that the data of the old master node will not be available for computation and the same happens to the data generated by the computation done for the old master node's task. In our solution, the cloud stays live as long as it has mobile devices in it. This means that, even if one of the nodes that requested help to compute some task, leaves the cloud after the completion of its task, both the data of this node and the data generated by the computation of its task, can potentially be replicated in our distributed file system. This will help to produce better outputs for future tasks and it will likely decrease the amount of computation done, since some nodes can probably desire to compute tasks that have already been computed or wish to apply some sort of filter over a group of data that was generated by the computation of an old task.

Another problem related to the Master Slave model, that was referred above, is the amount of information that needs to be known by the master node about the slave nodes. In our proposal, this information is way less which grants some flexibility to our system, allowing us to deal with different types of mobile devices without prior contact with them. This lead us to another less positive aspect of the Master Slave model, which is the distribution of tasks. This distribution process will require extra resources to be performed, which may add more load on the master device, as in **SOHRS**, where the master node chosen is the most suitable mobile device to serve as master, generally correlated with hardware power. This may raise some questions in terms of fairness.

Communication Protocol The vast majority of the solutions addressed uses Wi-Fi as the communication protocol. Only **SOHRS**, allows for both Wi-Fi communication protocol and Bluetooth communication protocol. The usage of Wi-Fi can present some limitations to those systems, specially when confronted with examples like the one described in 1.1, where there is a possibility of having no infrastructure connectivity due to the lack of Wi-Fi access points or network congestion. Both **P3-Mobile** (only if the application developer decides to use this technology) and **Honeybee** support Wi-Fi Direct as the communication protocol, although, in **Honeybee's** case (since it is the only solution that actually has an implementation done with Wi-Fi Direct, it is the only one we address), the network formed is quite small, since it is limited by the maximum number of connections allowed per device, which is eight connections, as stated in **Honeybee's** paper. Although, it is not

addressed specifically, it is possible to conclude, based on the descriptions of the systems, that all of them presents WLAN type of networks, where the amount of devices supported is usually very reduced. Our solution is designed to support Wi-Fi, Bluetooth and Wi-Fi Direct, which ultimately will help to deal with the limitations presented above.

Churn All of the solutions, can handle the entrance and departure of devices from their clouds, (churn), with exception of **SOHRS**, where this subject is not addressed. The solutions that do address it, follow different approaches as to how to detect when a node enters/leaves the cloud or what to do when churn happens. There is one solution, **Hyrax** which displays a slightly different behavior than the rest, simply because the mechanisms to deal with churn are offered by Hadoop, which means that there was no necessity of changing or creating the mechanism from scratch. Like the other solutions, Hadoop is able to re-execute failed tasks on other nodes and replicate data blocks to other nodes. The main difference is Hadoop's capability of executing the same task on different nodes, in order to increase the chance of success and reduce the execution time.

Relatively to the detection, the vast majority uses heartbeat and resource discovery mechanisms to detect churn, with exception of **MClouds**, that propose an interesting and advantageous approach, where besides using resource discovery mechanisms, it also uses certain events to inform that a specific node is leaving the cloud. These events can be associated with lack of resources or when a user starts moving, which is detected by the mobile device's sensors. This can reduce significantly the amount of communication done between nodes, during the process of verifying which nodes are alive, because it will not need to have a mechanism like heartbeat, running on a short period of time.

Similar to the approaches of other systems, our solution uses heartbeat mechanisms and timeouts in order to detect churn. Some of this mechanisms are granted by Thyme, while others were created exclusively to fit our requirements. Although there is a mechanism, assured by Thyme, that is constantly running in order to update information regarding neighbours, the rest of the mechanisms only execute when certain operations occur and focus specific nodes that are intermediary in those operations, trying to reduce this way the amount of messages exchanged between nodes.

About on what to do when churn happens, **P3-Mobile** and **Honeybee** use a similar approach that involves adding back to the task pool any task which has not been computed successfully. Both **Serendipity** and **FemtoCloud** use algorithms where they try to prevent unfinished tasks, caused by churn, by distributing the right task to the best available device to process it. When this prevention is not enough and the tasks have not yet been completed, the master node is responsible for reassigning these tasks to new devices. The same happens to **MClouds**, where besides having the master node re-assigning tasks, the master node can also decide whether to offload these uncompleted tasks to a fixed cloud or not.

We leverage on Thyme to replicate input data and metadata to other mobile devices in the network, allowing us to support churn even if the devices that originally stored

the data leave the network, since other devices will be in possession of that data. Besides this replication mechanisms, we also reassign failed tasks to other nodes. As for the other solutions, like **FemtoCloud** and **Serendipity**, where the master nodes choose to distribute the unfinished tasks to other slave nodes, we consider to have some negative aspects related to how computation is shared, which will be addressed in Subsection 2.3.1.3. **MCloud's** approach uses a fixed Cloud service to process unfinished tasks, which exhibits some limitations when there is no infrastructure connectivity available or the mobile data plan is expensive.

2.3.1.3 Execution Model

Computation Sharing The majority of the solutions discussed in this subsection chooses to have a master node that distributes all of the computation to its slave/worker nodes. This can be seen as an asset, specifically because of solutions like, **Serendipity**, **SOHRS** and **FemtoCloud**, where the nodes chosen by the master, to receive a certain work, are usually the best nodes to perform that work, reducing the amount of time and sometimes energy it takes to compute some tasks. This choice is made based on algorithms and information taken from the mobile devices which take part of the mobile clouds. Although these approaches raise some good points, they also present some negative aspects if we consider the amount of information required to make the best decision possible. This information implies a communication between the devices involved in the computation of tasks, prior to the execution of those tasks, something that can be very costly in terms of the amount of processing power necessary to do so and energy spent, but also in terms of time. Time that is a valuable asset, especially in these systems where the entrance and exit of mobile devices can happen at any time, which can cause the re-execution of some tasks that were not completed, since the mobile device responsible for them has left the cloud.

As already presented in Chapter 1, our solution addresses the sharing of computation in a conceptually different approach, where computation is performed where data is located, rather than distributing tasks and the corresponding data to any device in the cloud. This means that only mobile devices that have the input data, will actually perform computation over that data. Although similar to some related work solutions, where there is a node that distributes tasks to other nodes, our solution possibly implies fewer communications between the devices that compose the mobile cloud, for two main reasons. Firstly, since the nodes that process data are solely the devices that have that data, the number of eligible nodes to receive work is significantly less when compared to some of the solutions priorly presented, therefore the node distributing the tasks will have less possible targets to choose from. The other reason is associated to the fact that the node that distributes tasks, does that based on metadata that he already owns, since Thyme assures the creation and replication of metadata inside specific cells when a file is published. This means that, to distribute work, there is no necessity of communication

with the eligible devices besides the messages that actually send the work to be done. Ultimately, that implies less communication, even if there is communication due to replication of metadata, the metadata is only replicated between the devices of a cell and not all devices which compose the cloud. These details will be further explained in Chapter 3.

Task division Of the solutions that address this subject in their correspondent papers, only **P3-Mobile** displays a dynamic task division, in the idea that a task is divided into smaller parts solely when some node asks for work or work is given to that node. The other solutions that do address task division, choose a static approach to do so.

Obtaining results Relatively to how results of computations are obtained by the devices that asked for that computation, the majority of the solutions chooses to have their worker/slave nodes sending the results to the master node, except **Hyrax** that stores them in the Hadoop distributed file system, and **FemtoCloud** that chooses to notify the master node. What differentiates our solution from the others, is that our system supports both publishing the results, storing them in the distributed file system, and notifying the device that requested the computation about its results. With this, we are able to add an extra flexibility to our solution, since it is possible to choose the approach that bests suits the application, instead of always choosing one approach that could spend unnecessary resources. When compared to other solutions, if we had to send systematically results to devices, we would need to establish various communications to transmit files, potentially wasting a large amount of resources, if the device that requested computation was not interested in all results. This does not happen in our solution, independently if the results are published and stored in the distributed file system or a notification is sent to the requester device. Both of these approaches, store the results in the distributed file system and allow the requester device, after receiving information regarding the results, to download only the results that interests it, avoiding wasting resources.

What differentiates our two approaches of each other, is fundamentally the number of devices that are notified about the results and the number of messages sent/received, since when publishing the results, all devices that subscribed to the result tag will be notified about each one of the results, against only having the requester device receiving a single notification about its results. By storing the results in the distributed file system, the persistence of the files in the cloud is assured by Thyme's replication mechanisms, which ultimately can be very beneficial if we consider that two mobile devices might want to compute the same files, or one mobile device might want to apply some computation over a group of data resulting of a prior computation. In both of these cases, the amount of resources and time spent while handling a request, might decrease significantly.

2.3.2 Implementation details and workflows

In this subsection we discuss each of the solutions proposed in more detail, addressing both the architecture and workflow of the systems in question. Although these systems are different from what we propose in terms of distributing computation, it still makes sense to go into more detail about each one of these solutions.

Hyrax Since Hyrax adapted Apache Hadoop MapReduce, the same model in terms of what processes are running on a cluster/mobile cloud is followed, where there is only one *NameNode* and one *JobTracker* process running on the master node and there is only one *DataNode* and one *TaskTracker* process running in each slave node. The *NameNode* is responsible for obtaining information about the location of each block of data and conveying this information to the slave nodes, instructing them from (or to) where to read (or write) data. The *JobTracker* is responsible for determining how tasks are distributed by all *TaskTrackers* and for coordinating the execution of tasks for each job. The *TaskTracker* is responsible for executing all tasks assigned to it. The *DataNode* is responsible for managing the disk space allocated in the mobile device/slave node for storing data blocks for the Hadoop Distributed File System (HDFS) [35].

Mobile devices/slave nodes are also responsible for sending job requests to the mobile cloud through the client application. This client application interacts with *NameNode* and *JobTracker* instances running on the master node, in order to process some job.

Since the *NameNode* and the *JobTracker* run on the master node, and this master node is a fixed device, the only code that was actually ported to Android was the *DataNode's* and *TaskTracker's* code in order to run on the mobile devices. With this port changes were made, specifically, the *TaskTracker's* code was modified to launch a new thread every time a new task was assigned to it by the *JobTracker*. This task is then launched by calling the main method of the child's class. One other change made was on the client side code, in order to support dynamic class loading. The code responsible for launching a job was modified to compile job's code into .dex format before sending that code to the slave nodes.

P3-Mobile As already stated, P3-Mobile's architecture is based on services, having four main services that define it.

The *Link service* is responsible for handling the communication between nodes, where this communication can be implemented by the application developer that is using the P3-Mobile framework, according to the technology desired (eg. Wi-Fi, Wi-Fi Direct, Bluetooth). This service supports sending/receiving of messages and files, where the application developer, only needs to provide the data to be sent and the destination of that data.

The *network service* is responsible for creating and managing the P3-Mobile network. This network is based on a tree format, where each node of this tree is a *coordination*

cell. These *coordination cells* are composed by multiple nodes/mobile devices, that may be either *coordination nodes* or *worker nodes*. The difference between the two is that the *coordination node* has extra responsibilities. A *coordinator node* is responsible for knowing the identifier values of all nodes in its cell and the identifier values of the *coordinator nodes* of its parent cell. It is also responsible for knowing all files and directories of the distributed file system of its cell. Besides the responsibilities already presented, the coordinator node is also responsible for knowing the state of all worker nodes in its cell and working as an entry point for communication.

The *storage service* handles the distributed file system of the P3-Mobile network. For the developer of the application, this distributed file system is displayed as a disk with maximum capacity of all mobile device's storage size. The *coordinator node*, is once more a key component in this service, as it saves meta data off all files in its cell. This service allows to save and delete files in the distributed file system, always maintaining multiple replicas and consistency between those replicas and its meta data.

The *parallel processing service* manages all the distributed parallel task computation. Whenever a task first starts it triggers the creation of the network. The process of finding work is responsibility of every node, since to find work, a *worker node* asks one of the *coordinator nodes* of its cell, if there is any work to be done. If there is work available and the current running task can be divided, the task is then divided between the *worker node* and the *coordinator node*. If there is no work, the *coordination node* asks for more work around all nodes of its cell. If no work is available in the cell, the *coordinator node* asks for more work to the *coordinator nodes* of its parent cell.

Serendipity A job in this system can be composed of several *PNP-blocks*. These *PNP-blocks* are a combination of a pre-process program, a certain number of parallel task programs and a post-process program. The pre-process program prepares the input data for parallel computation, dividing this data into equal parts and distributes that prepared data to each task. The post-process program processes the output of every task and combines them into one file. Both of these programs execute on the node that wants to process the given job, which is also called an *initiator device*, while all the tasks run on other devices (though they can also be executed in the *initiator device*).

As for the system's architecture, one Serendipity node is composed of a *job engine process*, a *master process* and several *worker processes*, as observed in Figure 2.2. Every node also has its own profile, which is created by running benchmarks where the computing speed and energy consumption are tested. Besides having its own profile, for every mobile device encountered, the Serendipity node stores the other mobile device's profile and the contact that happened between the two. Because of this, the process of task allocation is possible.

In terms of the system workflow, an user that wants to submit a job, he first needs to submit a script to the *job engine*, specifying the *job Directed Acyclic Graph (DAG)*, where each vertex is a *PNP-block*, the programs, execution profiles and input data. If the

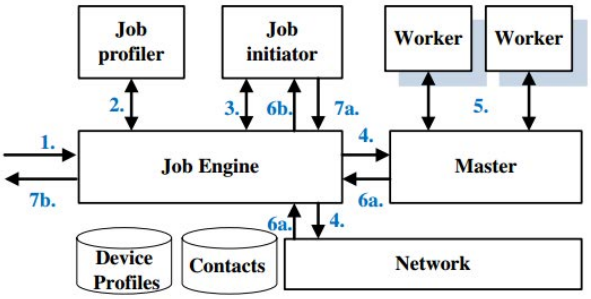


Figure 2.2: Serendipity node's architecture (taken from [25])

script has all the correct and necessary info, a *job initiator* is launched. With this, the pre-process will execute over the *PNP-blocks* launching them and assigning a time-to-live, a priority and a worker for every task. These tasks can be done locally or allocated to other remote mobile devices if the *job engine* so decides. This decision is made based on the meta data that two mobile devices exchange when a contact occurs, like their profiles, remaining energy and carried tasks. The job engine is also responsible for scheduling task execution in the local master of a Serendipity node using an algorithm which determines the job priority. Whenever a master receives a task from the *job engine* it launches a worker to process that task. When the said task finishes, the output is sent to the *job initiator*, which will wait for all tasks of a job to finish and then merge all outputs and return the final results.

One important aspect of this system is how they focused on presenting three different versions of algorithms for tasks allocations. These three versions allow them to handle different types of scenarios. The first algorithm was designed to target scenarios where it is possible to predict all future contacts between mobile devices and there is a *control channel* for coordination purposes. Using the first algorithm it is possible to choose the nodes where a certain task would take fewer time to execute. The second algorithm targets scenarios similar to the ones targeted by the first algorithm, but in this case, there is no *control channel* that would allow to schedule tasks in advance. To deal with this, the second algorithm uses *computing on dissemination*, which allows to disseminate tasks to mobile devices that are encountered, if the computation on those mobile devices takes less time than the computation made locally, instead of assigning tasks to mobile devices in advance. The third algorithm was designed to target scenarios where the lack of knowledge about future contacts and the lack of a *control channel* is a reality. It is very similar to the second one, but since future contacts are unknown, it focus on decreasing the amount of time it takes for the last task to return a result to the *job initiator*, which was observed that it would decrease the completion time of a *PNP-block*. This is done by assigning the last task to a mobile device which is very close to the *job initiator* node.

This system also takes in consideration the amount of energy spent executing a task. The way they do this is through the use of an utility function as replacement for the time

variable on each of the three algorithms discussed above. This utility function uses the energy consumption values of all nodes and the remaining energy value in each node.

VisionMClouds Mobile devices in this system are called *mDevs* and tasks are called *mTasks*. A *mTask* can be executed on a single mobile device if this task only needs few resources to be completed. A *mTask* can also be divided into smaller subtasks, independent from one another, which are distributed among the devices that form the mCloud.

The MCloud solution, addresses the idea of compensation for mobile devices that offer their resources to others' computation. This compensation is financial and it takes into consideration some aspects like the completion time of the task being processed, the number of slave *mDevs* and the amount of energy spent by the slaves.

In terms of the workflow, when a mobile device wishes to offload computation to a mCloud it has first to go through a discovery process, where it broadcasts a message announcing the intention of creating a mCloud to process a certain task. If there is not enough mobile devices or any to execute all subtasks, those can be partial or totally offloaded to a fixed Cloud. Any slave mDev that wants to help with the computation of the subtasks, responds to the masters mDev's request with its unique identifier. After receiving and finishing the computation of a subtask, the slave *mDev* responsible for this subtask, sends the output to the master *mDev* that will then wait for all outputs of subtasks and merge them all in one result returning that result.

SOHRS Relatively to the system architecture, a group of mobile devices connected to each other, form a local cloud. In this local cloud there is one *local resource coordinator (LRC)* and multiple nodes/mobile devices that will provide or ask for resource sharing. The *LRC* is responsible for managing the resources in the local cloud and assigning tasks to nodes or to a fixed Cloud if necessary. Since the *LRC* has such an important role in the correct operation of the local cloud, it is chosen carefully from a group of mobile devices, while taking into consideration multiple aspects like, the quality of the connection to the other mobile devices, the CPU performance and battery life time.

In terms of the workflow, when a node requires resources it sends a message to the *LRC* specifying the type of resources and how much does it need, what tasks and how much does it need to complete a service, its MAC and IP address. The *LRC* then assigns the tasks to some of the nodes, using that node's utility value for the service as factor of choice and informing the nodes chosen, of which and how much tasks do they need to execute. The node that is currently outsourcing its tasks, has the responsibility of informing the nodes chosen by the *LRC* about necessary information to execute each task. When a node finishes the execution of a task, it will send the results to the node that is outsourcing the tasks, which will then construct the service based on the results of the tasks.

The key aspect of the system being described above, is the utility value used as a factor of choice when assigning tasks to nodes. This utility value is obtained by the utility function based on the latency of a certain service. The latency of a given service is considered to be the amount of time it takes since the first task of that service begins to run until all the tasks of that service are finished. Considering the heterogeneity of the resources, which can be CPU performance, Internet connection, connection speed, etc, it was fundamental to devise a way of comparing all type of resources. The authors of the system have chosen latency as a way of dealing with this heterogeneity, since all of the resources help decrease the latency of a service.

FemtoClouds The mobile devices have a service always running called *femtocloud client*. This service is responsible for gathering information about the device and receiving input from the user with the objective of creating a profile of the user, which is crucial for the task distribution process. This information is sent to the control device which is responsible for estimating the amount of time a user stays in the femtocloud, managing the femtocloud, choosing which devices are eligible to join the femtocloud and distributing multiples tasks to these devices.

In the FemtoCloud system it is considered that the users are available to offer their resources, if in exchange they receive some kind of financial gain. It is also considered that the tasks can be sent to the control device individually or in groups/batches.

In terms of the architecture of the system, both type of devices display a module architecture type. The *client femtocloud* presents modules which focus on retrieving data from the mobile devices, about the computational capability and availability of the device, creating a profile with that data which will then be sent to the control device. The control device presents modules which focus on estimating data about the several mobile devices in the femtocloud, based on the profiles received, making sure of a better task distribution. This better task distribution is granted by the task scheduler, in the control device, which uses the several heuristics shown in the femtocloud paper, in a way of maximizing the useful computation of the cluster.

In terms of the implementation, the control device offers an interface to the task originators, allowing to receive code to be executed, input data of the tasks and results of computations. The communication between mobile devices and control device uses persistence TCP connections and the control device can act as a Wi-Fi hotspot.

The *femtocloud client* allows the user to insert data about which and how much resources does it wishes to share, and which information of the device can be shared with the control device. After doing this process, the *femtocloud client* is able to connect to the Wi-Fi network that was created by the control device. Whenever a *femtocloud client* finishes a task, it saves the outputs from that task and notifies the control device about the conclusion of the said task, starting then executing another task. After this, the control device can download the outputs of the task from that mobile device, which will allow the *femtocloud client* to erase all data associated with the task in question.

Honeybee In terms of workflow, whenever the program starts, the *delegator node* splits its task to be computed, into multiples jobs and adds them to its *job queue*. Afterwards the *delegator node* will start consuming the jobs, only if it can contribute for the computation of those jobs. The *delegator* will then start searching for worker nodes. When a worker node establishes a connection with the *delegator node*, it will be able to steal jobs from the *delegator node's job queue*. When the *delegator's job queue* gets empty, the *delegator node* will begin to steal jobs from slower nodes and it will add them back to its queue in order to be stolen by faster nodes. Whenever a worker node finishes executing jobs it has stolen, it will send the results back to the *delegator node*. When all jobs complete, the *delegator node* sends a termination notification to all worker nodes in the cloud.

In terms of implementation, the framework is composed of three components, *Application*, *Job Handling* and *Communication*. The *Application* component acts as an interface between the developer's application and the rest of the components. The *Job Handling* component is responsible for splitting a task in multiples jobs, adding them to the *job queue*. It is also responsible for executing jobs locally, steal jobs from worker nodes and assuring fault tolerance mechanisms. The Communication component handles all of the communications between every node in the cloud.

Conclusion Based on the implementation details and workflow presented in this subsection, we can further conclude that these solutions present a different approach from ours, specially in terms of following a master slave organization, where there is a master node responsible for managing the network and distributing tasks to the best suited nodes. Although different from our solution, it is important to emphasize that some of the decisions of these solutions are interesting and can be taken into consideration for future work. Namely, the idea of compensation and fairness that is not addressed in this thesis. Also, the usage of a fixed Cloud alongside with the system we propose, in situations where computation can be very intense and spend too much resources, for a cloud of mobile devices. Of all of the solutions above described, **P3-Mobile** and **Honeybee**, are the ones that most distinguish from our approach in regards to sharing computation, nonetheless, their approach of work stealing/searching, can potentially reduce significantly the amount of information exchanged before work sharing, since in the approaches, there is no necessity of having a master node choosing and distributing tasks to the best nodes. With the reduction of information exchanged, the communication between nodes will also reduce, saving mobile devices' resources. We could, as future work, complement our computation distribution mechanism by adding the idea of local distribution of computation, at the cell level, inspired by the approach followed by **P3-Mobile** and **Honeybee**. It is also important to refer, the way tasks are divided in **P3-Mobile**, could potentially inspire mechanisms in our system that would subdivide tasks that are computational intensive and have many items to be processed, in subtasks that would be distributed to neighbours of the node in charge for handling the task.

2.4 Data Stream Distributed Computing Frameworks

Our solution aims in implementing a distributed computing framework able to process a stream of data. In this sense, we studied different distributed computing frameworks for cluster computing, in order to understand how these frameworks handle a stream of data.

There are multiple distributed computing frameworks available for application developers, such as **Apache Spark** [33], **Apache Flink** [36], **Apache Hadoop** [31], **Apache Storm** [34]. Of these four frameworks, we only chose to focus on **Apache Spark** and **Apache Flink** because they are closer to what we intend to offer in our solution's data processing model. There is still another framework that we will address, Lei Yang's et al. **Partitioning Framework** [40], that is different from the aforementioned because it falls in the Mobile Cloud Computing context. It is important to address [40], because it is one of the few frameworks that supports data stream in a mobile environment. We will now proceed in detailing both Apache Spark and Apache Flink.

Apache Spark is a distributed computing framework for clusters that handles large-scale data processing, where its processing time is faster than that of Hadoop MapReduce when done in both memory and disk. Like Hadoop MapReduce, Spark also grants automatic fault tolerance [33]. This system offers an API to *Java*, *Python*, *Scala* and *R*, application developers, offering at the same time, a batch engine that also supports stream processing [42].

One of the key aspects that differentiates **Apache Spark** from others frameworks, are the *Resilient Distributed Datasets (RDDs)*, which consist in a collection of objects partitioned by several nodes of the cluster, which are subject to parallel computations similar to the ones done with MapReduce [41]. These RDDs can be stored both in memory and in an external storage, like the *Hadoop Distributed File System (HDFS)*. One fundamental feature of RDDs, is their capability of fault tolerance through lineage. Lineage allows for the reconstruction of faulty RDDs, by re-executing the original operations like *map, reduce, groupBy* and others, on the dataset which produced the RDD that failed [42].

The **Apache Spark** framework also offers built-in libraries. Of all the libraries and APIs offered by **Apache Spark**, *Spark Streaming* is the library that we will discuss in more detail, since it concedes the capability of stream processing and data stream to the **Apache Spark** framework, granting fault tolerance.

Spark Streaming, as observed in Figure 2.3, handles streaming computation as a group of batch computations in small time intervals. The input data received is stored during a time interval, in order to be computed by operations like *map* and *reduce*, generating new datasets after the time interval reaches the end. Those datasets can be stored as RDDs. All of this is possible because of the creation of a new processing model, *Discretized Streams (D-Streams)*. A *D-Stream* represents a continuous stream of data, coming directly from a client that keeps feeding them or from an external storage, as *HDFS*, where the stream is fed periodically with data of the file system. *D-Streams* are composed by a sequence

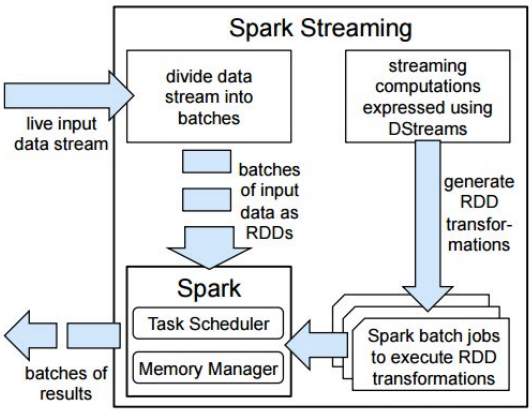


Figure 2.3: High level overview of the Apache Spark system (taken from [42]).

of RDDs, and like RDDs, when a transformation like *map* is applied to them, a new *D-Stream* is created [42].

Apache Flink is a distributed computing framework for clusters, which presents a stream programming model and supports both distributed data stream and batch data processing, granting fault tolerance.

The **Apache Flink** framework gives a clear distinction between *unbounded datasets* and *bounded datasets*, in which the first consists in infinite datasets that are continuously appended, and the second consists in finite datasets. This systems has a streaming execution model where the processing is continuous as long as the data keeps being produced. Different from **Apache Spark**, where there is a batch engine that also supports streaming, in **Apache Flink's** framework, *bounded datasets* are considered to be a special case of *unbounded datasets*, where the stream of data is finite. This allows both types of datasets to be handled by the engine in a very similar way, with only minor differences. The same thing happens with two APIs offered by the framework, *DataStream* API that handles *unbounded datasets*, and *DataSet* API that handles *bounded datasets*, where both are very similar and exhibit small variations [36].

Relatively to the workflow of a program in **Apache Flink**, it first starts by creating a *DataStream/DataSet*, depending on what type of program is being developed. *DataStreams* are created from various *data sources*, like socket streams, files, Java collections and different types of streaming connectors, such as the one used for Apache Kafka [32]. *DataSets* are also created from various *data sources*, like files or collections. In both types of programs, transformations such as *map* or *filter*, and others that can vary based on the type of program, generate a new *DataStream/DataSet*. These can be stored or consumed by *Data Sinks* and returned. These *Data Sinks*, according to the type of program *DataSet/DataStream*, forward the results to files, standard output such as command line, and in case of *DataStreams*, they can also forward the results to sockets or external systems through streaming connectors.

Apache Flink is able to provide fault tolerance through a checkpoint system, that

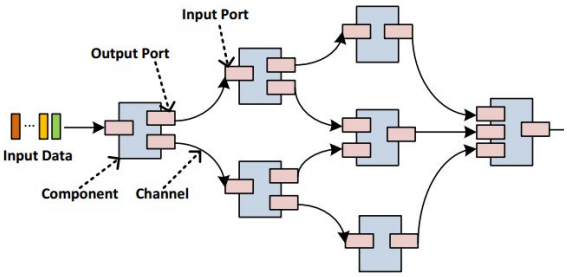


Figure 2.4: Framework Partitioning's application model (taken from [40]).

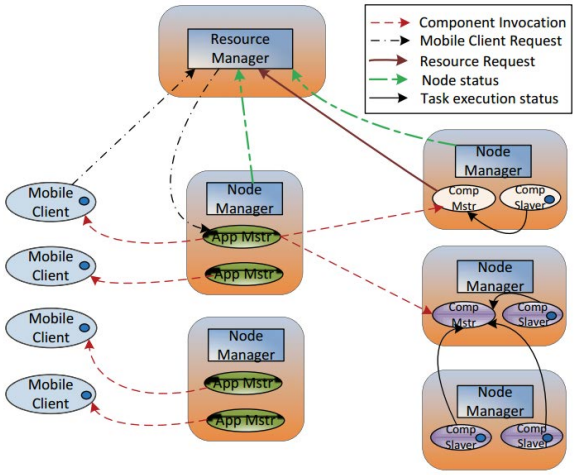


Figure 2.5: Framework Partitioning's application framework overview (taken from [40]).

stores data related to *data sources*, *data sinks* and *states*. Every faulty execution can be re-executed with a specified delay, in order to handle possible timeouts of external systems.

Lei Yang's et al. **Partitioning Framework** [40] provides mobile application developers the possibility of developing data streaming applications that support adaptive partitioning in run time and distributed execution, where the main focus is to enhance the throughput of those applications. This framework fits in the Mobile Cloud Computing category, in which fixed Clouds are used to augment the execution of mobile applications.

In terms of the application model of the framework, data stream applications are represented as *directed acyclic dataflow graphs*, as observed in Figure 2.4. These are composed of multiples *components* and *channels*. The *components* execute functional operations over data and are generally mapped into threads or processes. The *channels* are responsible for transporting data between *components*, and are implemented by TCP sockets, shared memory or persistent storage. The first *component* is known as the *entry node*, since it is responsible for processing all the input data that arrives from the mobile devices' sensors. The final *component* is knows as the *exit node* and it is responsible for generating output data.

Relatively to the workflow of a data stream application that uses the **Partitioning Framework**, it first needs to have software modules both in the fixed Cloud and in the mobile device. When the application is launched in a mobile device, a request is sent to the *Resource Manager* that is part of the software module in the Cloud. The *Resource Manager* assigns an *Application Master* to the mobile device/*mobile client*. After this, multiple informations like, CPU capability, workload of the device, network bandwidth, will be exchanged between the *Application Master* and the *mobile client*, in order to feed data to the genetic algorithm used to partition *components* in an optimal way.

The *components* which are responsibility of the *mobile client* are initiated as different threads, whereas the *components* that are responsibility of the fixed Cloud are launched as *Components-as-a-Service*. In the fixed Cloud's side, the *Resource Manager* is in charge of resource allocation, where communication is established with *Node Managers*, for status updates, and with *Component Masters* that instruct *Node Managers* for the necessity of launching/terminating *Component Slaves*, according to the computation required. For better understanding of the workflow described, Figure 2.5 shows the overview of the application framework.

It is still important to refer that the *Application Master* is in continuous communication with the *mobile client* for data exchange between the fixed Cloud and the *mobile client*, and for message exchange regarding possible readjustments of the partitions made, based on the mobile device's conditions which may have changed during execution.

Conclusion With the description of the systems presented in this section, it is possible to conclude that several of the concepts here addressed, influenced to some extent the design and implementation decisions of our solution. One of the key concepts addressed in this section, is the idea of a *dataset*. A *dataset* can be seen as a group of data, distributed across multiple mobile devices, where this data is not modified. Although, data is no modified inside a *dataset*, it allows for computation to be done over one, generating a new *dataset*.

Another important aspect, addressed in this subsection, and that influenced our solution, is the idea of the *bounded and unbounded datasets*, where the *bounded datasets* are considered to be special cases of *unbounded datasets*. Similar to some of the solutions presented, our framework is also able to support both batch and data stream processing, while offering an API that works with both types of data processing.

In terms of fault tolerance, we do not consider the checkpoint system used in **Apache Flink**, to be a suited mechanism to our solution, since it uses checkpoints to reset the system to a previous state, something that does not goes towards our solution's design.

In general, our solution handles different types of problems/scenarios that the other solutions proposed in this chapter, were not able to address, focusing on performing computation where the data is. Nonetheless, certain features of the frameworks addressed in this section, can be seen as an influence to some parts of our solution.

Chapter 3

Oregano

In this chapter we present an initial overview of our system, Oregano, a distributed computing *framework* capable of processing batches and streams of data, without resorting to services on the Internet. With our solution, application developers will be able to develop a wide variety of applications that are part of the Mobile Cloud Computing environment. With this system we intend to address the group of problems enumerated in Section 1.3, something that no other related work was able to do the way we do.

In Section 3.1, we will give an overview/description of our system, its features and how one could use it in a example like the one described in Section 1.1. In Section 3.2, we will proceed by explaining our system's model. Additionally, in Section 3.3, we will present our *framework's API* and how to use it. Lastly, in Section 3.4, we will describe our system's main components and interactions.

3.1 System description

In order to support local processing of data that has been or is being shared by mobile devices, Oregano must provide functionalities to: i) process data that has already been shared and stored in one or more devices in the network and ii) process data streams that are being produced in real time. The ability to share and store data is assured by Thyme, a time-aware storage and dissemination system with publish and subscribe mechanisms, addressed in more detail in Section 3.2.

The data processed by Oregano is logically organized by *Mobile Dynamic Dataset (MDDS)*. Each *MDDS* is associated with a *tag*, similar to the ones used in social media, e.g. '#party'. Based on this relation between a *tag* and data, Oregano offers two operations: i) publish data items with a *tag*, followed by the pre-process of those data items; and ii) subscribe to data items published with a specific *tag*, associating a computation request

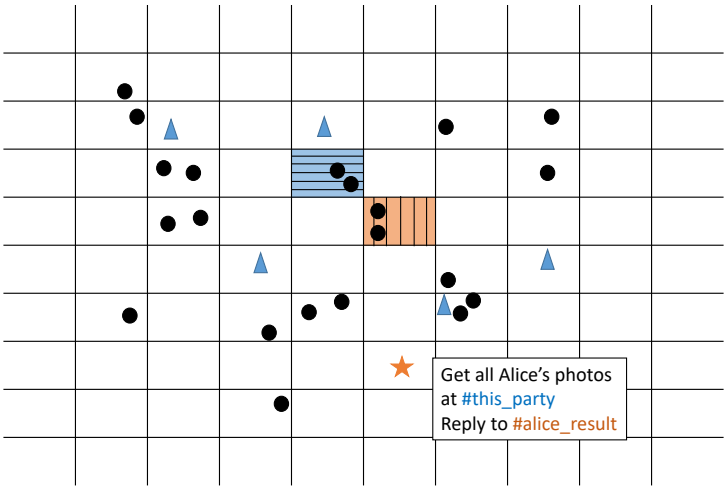


Figure 3.1: System description of a party scenario.

to the subscription. The subscription has a time interval associated to it, that can cover past, present and future. This means that our system offers the ability to process data that was published in the past, even if the original mobile device that published the data left the network. Furthermore, it allows the processing of data that will be published in the future. Whenever data is published with a *tag* that has computation associated to it, this said data will be processed. This feature grants data stream computation capability to our system.

Besides supporting the process of data that might have been published or it is still to be published, our system offers the ability of pre-processing every file before being processed. This pre-processing stage may be performed eagerly, whenever a data item is published, in order to reduce the amount of work to be performed by the mobile device, later in the future.

It is important to state that pre-processing and processing data can correspond to a wide variety of different operations/actions, like identifying a person in an image through facial recognition algorithms, finding specific words/patterns in different texts, modifying audio samples, etc... Possible examples of pre-processing for each one of the operations aforementioned could be: extracting faces from a photo; replace character/expression for other character/expression, like punctuation for white spaces; trimming the first 5 seconds of each audio sample.

In order to explain our solution in more detail, we will give an overview of the workflow of our system and how can an application developed with our *framework* be used in the party example presented in Section 1.1.

Figure 3.1 illustrates the workflow of our system, where it is possible to observe multiple elements. The grid displayed is not an actual physical grid, but a virtual one created by the storage system, which will be further on explained in Section 3.2. All the dots, triangles and star in this figure correspond to mobile devices. The star is Alice's mobile device, that wishes to find all photos of her that were published with *tag* '#this_party'.

The triangles correspond to mobile devices of people that took photos of Alice. The dots correspond to the rest of mobile devices in the party.

Consider that Alice and all the attendees of a party, instead of using the application described in the example, are using an application developed with Oregano, with which users are able to share photos with each other and obtaining photos where a given person appears. During the course of the party multiple users and in specific the triangles, would publish photos that they took, with the *tag* '#this_party'. By the end of the party, Alice would decide to retrieve all of the photos where she shows up, taken during the party. In order to provide such functionality, Alice, through the application, would subscribe with computation to the *tag* '#this_party', specifying an Oregano service that allows obtaining photos where a given person is present. This service would be used to process all photos published with the *tag* '#this_party', which, from Oregano's point of view, are logically seen as a *MDDS*.

After the subscription, the mobile devices with elements composing the *MDDS* bound to *tag* '#this_party', represented by the triangles, execute the service on every photo published with *tag* '#this_party'. The processing would consist of applying a facial recognition algorithm on all photos. After processing, all photos where Alice is, would be published with the result *tag*, '#alice_result'.

3.2 System model and Architecture

In our framework a *Service* is the core of all processing done on data. *Service* is designed to support all types of data, as it is the responsibility of the application developer using our *framework*, to decide the types of data to use and the processing/computation to be done. We also extended the idea of *Service* by devising a *Service with pre-process*, which aims to add a preliminary step of pre-processing all published data. The decision of using this preliminary step is totally entitled to the application developer.

An application may make use of multiple Oregano *Services* that are installed in a device. *Services* have unique global identifiers which will be used to choose what *Service* will process some data. The target data files/objects to be processed in these *Services* are files/objects that are independent of each other and do not follow any particular order. What this means is that if we imagine a situation where five files/objects are published with the same *tag*, processing them all in a group or one by one individually, the final result must be the same. A *Service* processes files/objects that compose a *MDDS*.

In our *framework*, we assume that a *Service* is priorly installed by the user so that we do not send code to be executed. Oregano allows the registration, replacement and removal of these *Services* in order to persist, change and remove them. How this is done will be further detailed in Subsection 4.2.2.

The system proposed in this thesis is to be executed by a network of mobile devices, where each one of these devices communicate with each other through wireless technologies. These devices have no movement restrictions, meaning that they can move to any

location, even if that means moving to a location where they are no longer in range of the network, leaving the same. To ensure the accuracy and ultimately the success of the system, all devices must have their clocks synchronized.

Each one of the devices that compose our system's network, run the stack/layer architecture observed in Figure 3.2. It is important to highlight once more, that this *framework* is a large system and is being developed in the context of the *Hyrax* project, using different layers/sub-systems that have been or are being developed by other teams in the same context. The main focus of this thesis addresses a portion of the overall system, concretely the computation layer, OREGANO.

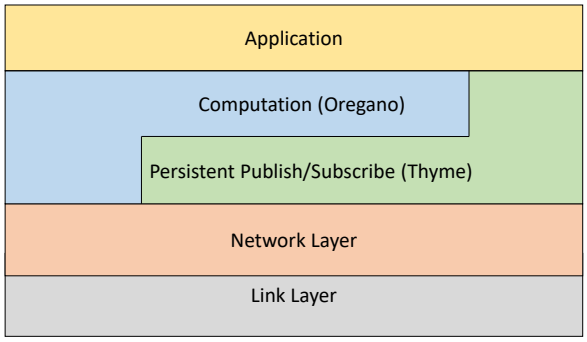


Figure 3.2: Overview of our system's architecture.

Network. The network component is ensured by the first two layers, Link and Network. Both of these layers are a research direction alone and are not the scope of this thesis. OREGANO uses them as a service. Through these layers it is possible to form a network between multiple mobile devices, allowing their communication using three different types of communication technologies: Wi-Fi, Bluetooth and Wi-Fi Direct. The network component also supports single-hop and multi-hop communication between devices.

Persistent Publish/Subscribe (Thyme). This layer consists of a distributed storage and dissemination system, time-aware, with publish/subscribe mechanisms, able to handle failure of multiple mobile devices, through replications strategies. One of the distinguishing aspects of this storage system is that it is time-aware. This storage system enables subscription by setting a start time and an end time, thus specifying that only files/objects published during this time interval should trigger a notification. This allows you to obtain data that was published in the past, as long as there is at least one mobile device with that data in the network. Thyme [4] uses a Cell-Based Geographic Hash Table [1] that divides the space where the system is to be used in a grid of equally-sized cells, like the one showed in Figure 3.1, whereby several mobile devices are distributed according to their current location. This system follows the concept of *tags* for its publish and subscription operations. Through dispersion functions, a *tag* is mapped in one of the cells of

the grid, being that all devices in a certain cell are responsible for storing subscriptions and notifying other devices that new objects have been published. Additionally, these devices also store meta data associated with each published file with one of the *tags* that are mapped to the current cell. In this layer when an object/file is published it is associated with at least one *tag*. The mapping mechanism is also observable in Figure 3.1, where the *tag* '#this_party' is mapped to the cell with the horizontal lines and the '#alice_result' *tag* is mapped to the cell with the vertical lines, meaning that only the devices inside of each one of these cells store meta data associated with objects published with the correspondent *tags*.

Computation (Oregano). The computation layer corresponds to the *framework* presented in this thesis. Oregano allows the distributed computing of data, with the main purpose of moving the computation to where the data is. Oregano will use the network layer in order to send and receive relevant messages to the system logic. It also interacts with Thyme, depending on many of the latter's features, like data storage with fault tolerance by replication. In this way it is possible to deal with incoming and outgoing mobile devices from the network, i.e., *churn*, since operations on the data can be performed on any replica. In addition, the existence of replicas allows to balance the distribution of computation among them, thus avoiding overload on some mobile devices. At the same time we can potentially observe faster computation times if more devices process less data. Similarly, Oregano depends on the meta data stored by Thyme, since it is crucial for the entire process of scheduling computation and return of results.

Application. As the name implies, this layer concerns applications that use the Oregano *framework*. The application layer interacts with Oregano, using the *API* offered, and it can also interact with Thyme, in order to use some of its unique features.

Every device that executes an application developed with Oregano, can have different and multiple roles during the lifetime of the said application. A mobile device can play the role of: i) Client; ii) Scheduler; and iii) Computing. These three roles can also be seen as three components that compose the computation layer, interacting with each other to assure the successful conclusion of a *computation request*. Each of these components may operate in different devices or simultaneously on the same device. This means that a mobile device is able to play one, two or even three different roles at the same time. All mobile devices in our system's network have these three components, being that in a default/basic scenario, each component belongs to a different mobile device. An example of a default scenario where every component is being used is observed in Figure 3.3. This Figure describes the same example of the party already addressed in Section 1.1 and is based on the already presented Figure of the description of the workflow shown in Figure 3.1.

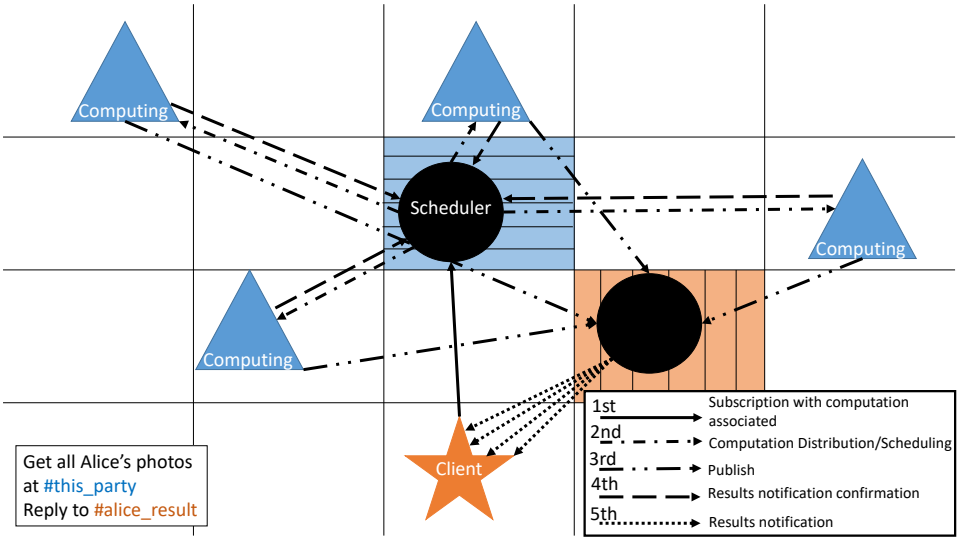


Figure 3.3: System components workflow on a party scenario.

In Figure 3.3, the star corresponds to Alice's mobile device that will be playing the *Client* role and it will be in charge of requesting the computation. The triangles correspond to every device that took pictures of Alice and published them with the *tag* '#this_party'. Theses devices play the *Computing* role, as they will be responsible for processing all photos. Since the *tag* '#this_party' is mapped to the cell with the horizontal stripes, the mobile device inside this cell will be responsible for storing all meta data of all photos published with *tag* '#this_party'. This is the reason why the mobile device in question will play the *Scheduler* role, as it is the only device in Figure 3.3 that has all the information to schedule *tasks*. Lastly, the *tag* '#alice_result' is mapped to the the cell with the vertical stripes, being that the mobile device inside of this cell will be responsible for storing meta data of all the published photos with *tag* '#alice_result', and notifying Alice's mobile device, the *Client* component, about the results.

3.3 *APIs*

In this Section we will present our *framework's APIs*, explain some important concepts for a better understanding of our system and give some small and simple examples on how to use these *APIs*. OREGANO's *APIs* can be divided in two parts: i) Service development (Subsection 3.3.2), which consists on the implementation of a *Service* that will process data; and ii) Invoking service (Subsection 3.3.3), which consists on invoking a priorly implemented *Service*, through publish/subscriber with computation operations.

3.3.1 DataItem concept

The objects that compose a *MDDS* must abide to the *DataItem* interface, which defines the set of operations that all data items communicated in OREGANO and Thyme must

offer. A *DataItem* forces applications developers to implement two operations that allow the serialization and deserialization of an object. Besides the definition of a *DataItem* we also devised the definition of a *Pre Processed DataItem*, *PPDataItem*. The interface of these classes can be observed in Tables 3.1 and 3.2. The main difference between the two is that the latter has a third operation that returns the original input value of a *DataItem* that was pre processed. This can be helpful in situations where a *DataItem* was pre processed but the result to be returned should be the value of the *DataItem* before being pre processed, an example of this will be presented in Subsection 3.3.2, more precisely in Listing 3.1.

Table 3.1: Abstract Class *DataItem*

byte[] toByteArray()	Returns the byte array representation of the data item
static <T **extends** DataItem> T fromByteArray(**byte**[] bytes)	Constructs and returns a data item of type *T* from an array of bytes

Table 3.2: Abstract Class *PPDataItem* that extends *DataItem*

Input getOriginalInput()	Returns the original input value of this data item before being pre-processed

3.3.2 Service development *API*

The definition of a service is simple and allows the usage of any type of data. The application developer only needs to implement one operation, **process**, being the second one, **preProcess**, optional depending whether the developer desires to pre process data. The **process** operation processes any *DataItem* that belongs to a *MDDS*, independently if our *framework* is handling this *DataItem* as corresponding to a batch or stream of data. Each device processes *DataItems* of the *MDD* that it has in its possession, being that the *MDD* is passed to the **process** operation as a *MDDStream*. It is important to emphasize, as it is observed in Figure 3.4, that in conceptual terms all *DataItems* published with the same *tag* form a *MDDS*, however, in practical terms, each mobile device that will process data joins the correspondent *DataItems* to be processed inside a *MDD*, being the *MDD* used as data source by the **process** operation.

Based on the characteristics of the target data *Services*' process, we decided to follow a streaming programming model. This type of programming model allows to perform operations/transformations over a set of data originating a changed one, but still keeping the characteristics of the target data, which is something of our interest as it suits the type of applications our *framework* was designed for. An example of this would be, in the case of the motivational example presented in Section 1.1, if Alice decided that additionally to the requirement that the resulting photos must contain her face, they would now also have to contain Bob's face, Alice's friend. One can simply apply a second transformation

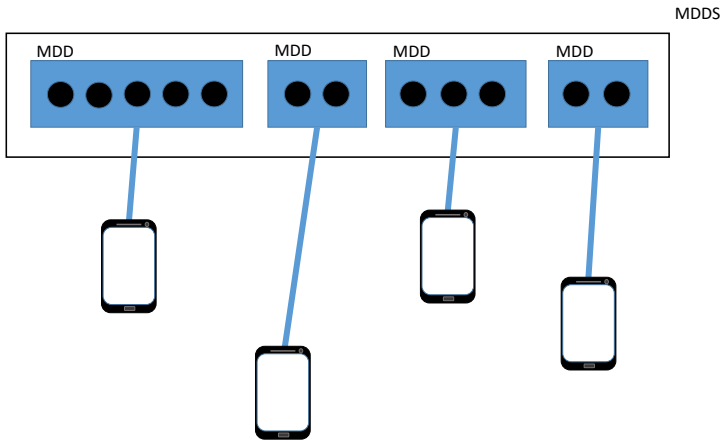


Figure 3.4: *MDDS* data partition.

on the first set of results to find Bob's face, so that only the photos where Alice's and now Bob's face is present would be returned as result.

In order to allow application developers to develop *Services* that follow a streaming programming model, we created the idea of *MDDStream*. A *MDDStream* is simply a stream that has as data source a *MDD*, offering the ability to apply transformations over data in a *MDD* without requiring in-memory data structures to store intermediate data. Besides supporting the more recurrent functions that operate on sequences of elements, like *map*, *reduce*, *filter* and others, *MDDStreams* also offer a *persist* operation that allows application developers to store/persist intermediate results, so that these may be reused by subsequent runs, without requiring to re-process data to get these intermediate results once again. Due to its size and relevance, only part of *MDDStream's* interface is presented in Table 3.3.

Table 3.3: Interface of a *MDDStream*

IMDD<DataItem> getPersistedResult(**String** identifier)	Returns an intermediate persisted result, identified by identifier
IMDDStream<T> persist(**String** identifier)	Stores intermediate result with identifier

The header of the **process** operation is:

- `MDDStream<Output> process (MDDStream<Input> stream, List<Args> serviceArgs)`

where

`stream` is the *MDDStream* upon which transformations will be applied in order to return the expected outcome, by the application developer. The data source from where this *MDDStream* was created, is formed by *DataItems* that may or may not have been pre processed.

`serviceArgs` is a collection of *DataItems*, service arguments, that can potentially be used, depending on how the application developer implemented his/her *Service*. An example of a service argument in a **process** operation would be, the extracted faces to find in a set of photos.

Operation **process** outputs the result of the process as an *MDDStream*. The result objects also follow the same characteristics of the input values that originated the stream.

The **preProcess** operation consists in a computation applied to a single *DataItem*, at the time of its publishing or before being processed. In Oregano, **preProcess** is only used if the application developer defined a *Service with pre-process* instead of a *Service*. The main goal of the **preProcess** operation in our *framework* is the ability to balance the workload, as it can be used to pre-process *DataItems* at publish time instead of pre-processing them all at once before processing them.

The header of the **preProcess** operation is:

- `PreProcessedInput preProcess (Input value, List<Args> serviceArgs)`

where

`value` is a *DataItem* that will be pre processed and the result of that pre process persisted in order to be used when **process** is called by the *framework*.

`serviceArgs` is a collection of *DataItems*, service arguments, that can potentially be used, depending on how the application developer implemented his/her *Service*. An example of a service argument in a **preProcess** operation would be, the size that is used to resize the extracted face from the photo received as input.

Operation **preProcess** outputs the result of the preprocessing, a *PPDataItem*. Besides containing the computed value, this type of object also stores the original input value allowing the application developer to use it if so he/she desires, in the *process* operation.

For an easier understanding of how a *Service* should be implemented, we will present the implementation of a simple text pattern finding *Service* in Listing 3.1. This *Service* receives texts, pre processes them by replacing certain characters by others specified by the user, e.g ',' for '', or 'a' for 'b', and creates a suffix tree with all the words on a received text. Besides pre processing the received texts, the service will also process all resultant suffix trees returning the original text of each tree if all the patterns specified by the user exist in that tree. In the **preProcess** operation, serviceArgs contain the specified characters and in the **process** operation, serviceArgs contain the specified patterns. It is also important to state that in Listing 3.1, `OreganoString` is a *DataItem*, and `OreganoSuffixTree` is a `PPDataItem`.

3.3.3 Invoking service *API*

This second part of Oregano's *APIs* enables applications to invoke *Services*. To do this, Oregano provides two operations, **publish** and **subscribe**. Both of these operations have computation associated with them.

Listing 3.1: *Service* text pattern finding example

```
1
2       @Override
3       public OreganoSuffixTree preProcess(OreganoString input,
4                       List<OreganoString> serviceArgs) {
5           String inputAux = input.getValue();
6           for(int i = 0; i < serviceArgs.size() ; i = i + 2){
7               inputAux = inputAux.replace(serviceArgs.get(i).getValue(),
8                       serviceArgs.get(i+1).getValue());
9           }
10          List<String> words = Arrays.asList(inputAux.split(" "));
11          SuffixTree<Integer> suffixTree = new SuffixTree();
12          for(String word : words){
13              suffixTree.putIfAbsent(word,0);
14          }
15          return new OreganoSuffixTree(suffixTree,input);
16      }
17
18      @Override
19      public IMDDStream<OreganoString> process(IMDDStream<OreganoSuffixTree>
20                                                  stream,
21                                              List<OreganoString> serviceArgs) {
22          return stream.filter(tree -> treeContainsArgs(tree.getValue(),
23                                                      serviceArgs))
24                  .map(tree -> tree.getOriginalInput());
25      }
26
27      private boolean treeContainsArgs(SuffixTree tree,
28                                      List<OreganoString> serviceArgs){
29          int count = 0;
30          for(OreganoString arg : serviceArgs){
31              if(tree.getKeysContaining(arg.getValue()).iterator().hasNext())
32                  count++;
33          }
34          return count == serviceArgs.size();
35      }
```

In order to explain OREGANO's **publish** and **subscribe** it is necessary to present the concept of time-aware publish/subscribe with persistence, provided by Thyme. Both of these mechanisms are similar to a regular publish or subscribe, however they associate a time stamp/interval to them. In terms of the publish operation, whenever a *DataItem* is published with some *tag*, a meta data object is created to store/keep information about the *DataItem* being published. The information that is stored includes a time stamp of the publication moment. This meta data is stored by the mobile device that published the *DataItem* and it is sent to all devices in the cell where the *tag* used during the publish is mapped, as already stated in Section 3.2. Since the time stamp of a published *DataItem* is stored, this enables subscriptions where the user can specify a time interval filtering which *DataItems* should trigger a notification. Basically, only *DataItems* that have been published between the start time and the end time, with some *tag*, are of interest to the user that subscribed. Both start time and end time can specify timestamps in the past or in the future, as long as the end time is higher than the start time. This raises

a problem, especially in systems like ours, where mobile devices can enter or leave the network at any time. Therefore, there was the necessity of granting persistence to these publish/subscribe mechanisms. By replicating meta data, subscriptions and *DataItems*[1], the persistence is granted, as it allows users to retrieve *DataItems* that could have been published by a mobile device that already left the network.

Regarding Oregano's **publish** and **subscribe** operations, the idea of time-aware and persistence is also assured, in part due to the use of Thyme's **publish** and **subscribe**. Oregano's **publish** consists of a regular publish of a *DataItem*, which in case of success triggers the pre-process of the published *DataItem*, and the result of the pre-processing is stored. Whenever Oregano's **publish** is called, the pre-processing is done eagerly rather than only pre-processing the *DataItem* before processing it.

The header of the **publish** operation is:

- `void publish (DataItem object, Tag tag, byte[] description, ObjectOperationHandler handler, boolean replicate, UUID serviceID, List<DataItem> serviceArgs)`

where

– `object` is the *DataItem* to be published.

– `tag` is the *tag* to which the *DataItem* will be associated with.

– `description` is a description of the *DataItem*. This can be a shorter representation of the *DataItem* being published, like a *thumbnail* of an image or a *title* of a text.

– `handler` is a callback which will inform the user if the *DataItem* was successfully published or not.

– `replicate` field that will specify if the *DataItem* being published should be replicated or not.

– `serviceID` the identifier of the *Service* to be used to pre-process the *DataItem*.

– `serviceArgs` is a collection of service arguments, which will be passed to the *Service* and used according to the application developer defined implementation of a *Service*, more precisely the operation **preProcess**.

Oregano's **subscribe** consists of a subscribe with computation associated with it. This subscription allows the processing of a batch or stream of data, notifying the device that called this operation about the results. This **subscribe** operation is the core of our *framework*, since it starts a *computation request* triggering a large group of actions in our system, in order to guarantee the data processing. It is important for a better understanding of the arguments received by the **subscribe** operation, to state that a *computation request* can be stopped and continued. This, along with the actions and their workflow

[1] In what regards *DataItems*, replication may be turned on or off by the application

will be explained in more detail in Section 3.4. The processing is assured by the **process** operation discussed in Subsection 3.3.2.

The header of the **subscribe** operation is:

- void subscribe (Tag inputTag, Time startTime, Time endTime, ResultHandler handler, UUID serviceID, List<DataItem> serviceArgs)

where

– `inputTag` is a *tag*, which specifies that every *DataItem* published with this *tag* between the startTime and the endTime should be processed.

– `startTime` is a starting time, which determines that only *DataItems* published with the inputTag after this startTime should be processed.

– `endTime` is an ending time, which determines that only *DataItems* published with the inputTag before this endTime should be processed.

– `handler` which will return notification results, present information regarding the state of a *computation request* and allow to control the *computation request* (stopping or continuing it).

– `serviceID` the identifier of the *Service* to be used to process all *DataItems* published with inputTag between the startTime and the endTime.

– `serviceArgs` is a collection of service arguments, which will be passed to the *Service* and used according to the application developer defined implementation of a *Service*, more precisely the operation **process**.

For an easier understanding on how to call OREGANO's **publish** and **subscribe** we will present an example of a possible usage for the text pattern finding *Service*, implemented in Listing 3.1. The example omits certain implementation details, such as both handlers and OREGANO's initialization, as they will be discussed later in Chapter 4. The example in question describes a scenario where an user that is attending an event publishes a text, at 5:10 p.m. to describe an opinion about something using a particular tag. This action is observed in Listing 3.2. In this same event, at 5:30 p.m., a second user subscribes with computation to the tag used by the first user, in order to find all texts that may have been published with a certain pattern. This action is observed in Listing 3.3. It is also important to state that in both above mentioned Listings, `0reganoString` is a *DataItem*.

3.4 Workflow

As already stated in Section 3.2, devices that execute an OREGANO application can have multiple roles during its execution: i) Client; ii) Scheduler; and iii) Computing. These roles can also be known as the three main components of OREGANO. Each one these

Listing 3.2: Example of using the **preProcess** operation of the text pattern finding *Service*

```
1    Oregano oregano = getOreganoInstance(...);
2    OreganoString object = new OreganoString("Alice had a lot of fun at the" +
3                                " party she attended yesterday. Five stars.");
4    Tag tag = new Tag("party");
5    byte[] description = "Alice party opinion".getBytes();
6    ObjectOperationHandler handler = createObjectOperationHandler();
7    UUID serviceID = getTextPatternFindingServiceId();
8    List<OreganoString> serviceArgs = new ArrayList<OreganoString>();
9    serviceArgs.add(new OreganoString("."));
10   serviceArgs.add(new OreganoString(""));
11   oregano.publish(object,tag,description,handler,true,serviceID,serviceArgs);
```

Listing 3.3: Example of using the **process** operation of the text pattern finding *Service*

```
1    Oregano oregano = getOreganoInstance(...);
2    Tag tag = new Tag("party");
3    Time startTime = new Time("17:00:00");
4    Time endTime = new Time("18:00:00");
5    ResultHandler handler = createResultHandlerHandler(...);
6    UUID serviceID = getTextPatternFindingServiceId();
7    List<OreganoString> serviceArgs = new ArrayList<OreganoString>();
8    serviceArgs.add(new OreganoString("Alice"));
9    oregano.subscribe(tag,startTime,endTime,handler,serviceID,serviceArgs);
```

components is composed by multiple subcomponents that belong to other layers of our system, some of which are used and others extended to support Oregano's needs. In Subsection 3.4.1, we will present a description of each one of the components. In Subsection 3.4.2, we will present the workflow of our two main operations, **publish with pre-process** and **subscription with computation**.

3.4.1 Components Description

Client The *Client* component is responsible for managing the application's subscriptions with computation, as well as all messages associated with each subscription stored, like failure, acknowledgements and result notification messages. A subscription request with computation triggers two Thyme subscriptions: the first is done on the *tag* specified on the request and the second is done on a result *tag*, generated internally by our *framework*, which will offer, to the mobile devices that will process data, the means to publish the results notifying the *Client* about these results.

The *Client* component is also responsible for controlling the arrival of results and offering users the ability to *stop* or *continue* a *computation request*. From the moment a *computation request* is submitted, the *stop* option is available at any time. The availability of the *continue* option is dependent of many factors like, the total amount of data already processed; the amount of data to be processed; and if new elements have been published. The *continue* option was devised with the objective of allowing application developers to control the amount of results to be received. In scenarios where the input *MDDS* is

composed by a large amount of *DataItems* and the process operation usually produces a similar amount of results, it made some sense to offer the results in smaller batches as it allows application developers to choose if enough results were already produced or if a specific result was already found. At the same time, with the *continue* option, it is possible to avoid situations where a *computation request* would be occupying a great amount of computational resources because of the large number of *DataItems* to be processed. This can cause longer wait times for other *computation requests* to be handled if a large amount of devices are submitting *computation requests* at the same time. With this *continue* option, more mobile devices are able to submit/continue *computation requests* without having to wait longer. It is important to state that the *Client* is the only component responsible for the evolution of its *computation requests*. Both *Scheduler* and *Computing* component do not save any state on current *computation requests*, therefore these components do not distinguish between a *continuation* or a submission of a *computation request*.

The *Client* is also responsible for all publishes with pre-process. Furthermore, the *Client* allows the registration, replacement and removal of *Services*.

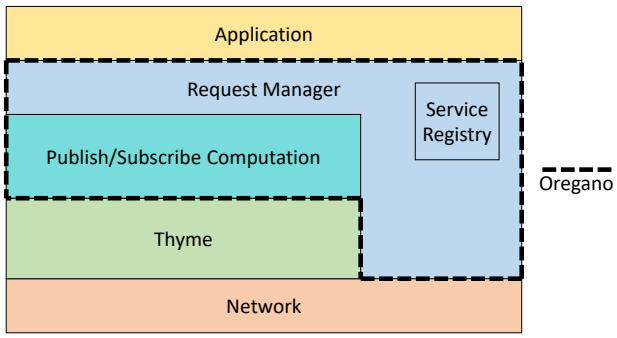


Figure 3.5: *Client's* subcomponents stack.

As observed in Figure 3.5, the *Client* is composed by multiple subcomponents. The *Application* component interacts with *Request Manager* to submit *computation requests* and at the same time to receive feedback and control those *computation requests*. The *Application* also interacts with *Request Manager* in order to register, replace or remove a *Service* defined by the application developer. This interaction usually occurs the first time an application is used, with the objective of persisting a *Service*. In the interest of registering, replacing or removing a *Service*, *Request Manager* interacts with *Service Registry* component because it is the *Service Registry* responsibility of managing and saving all *Services*, besides initializing them at boot time.

The *Request Manager* component is also in charge of managing all *computation requests*, its acknowledgement, failure and result messages. It is also in charge of storing all of *computation requests'* info, and control the logic behind *stop* and *continue* operations.

The *Request Manager* interacts with the *Network* to send and receive messages, and interacts with a *Publish/Subscribe Computation* component in order to subscribe to *tags* and receive notifications of results. Since there was a necessity of supporting a different type of subscription and handle publishes in a different way, we extended some operations of Thyme and created a *Publish/Subscribe Computation* component. This component is responsible for the subscriptions and publish operations. Note that, although we had the need to create this subcomponent, most of the publish/subscribe logic is still managed by Thyme, as we only extended some operations.

The *Request Manager* is also responsible for the **publish with pre-process** operation. The publish is done through the *Publish/Subscribe Computation* and preprocessed by the *Request Manager*. In order to pre-process *DataItems*, *Request Manager* interacts with *Service Registry* to obtain the instance of the specified *Service*.

Scheduler The *Scheduler* component is responsible for managing all subscriptions with computation that it receives, scheduling *tasks* among the devices that have the data. A device is chosen as *Scheduler* of a given subscription with computation if it is positioned in the cell to which the request's *tag* maps to, meaning that the device chosen has all the meta data associated with the subscribed *tag*. In this way, this device will have access to a wide variety of information about the *DataItems* to be processed, being able to choose the devices that will process the data. The *Scheduler* component chooses a parametrizable x number of *DataItems* to be processed at one time, forcing the application to decide if the next x number of *DataItems* should be processed or not. This approach is the same, whether the target data of a *computation request* is a batch of data (start time and end time are both times in the past) or a stream of data (end time is a time in the future). It is important to note that from the moment a subscription with computation is submitted until the moment it stops, the *computation request* can be handled as a data stream in some specific moments and a batch of data in other moments. Whether Oregano opts for handling the *computation request* as a data stream or a batch of data is dependent on the amount of data available to be processed, at a given time, and if the end time is a time in the future. With the objective of explaining in what moments Oregano handles a *computation request* as a data stream or a batch of data, an example scenario will be given: imagining a situation where mobile devices publish a total of 12 photos at 17:10 with *tag* '#this_party'. At 17:20 some other device subscribes with computation to *tag* '#this_party', specifying the start time at 17:00 and the end time at 18:00. Considering that the maximum amount of *DataItems* to be processed at once is set to 10, the *Scheduler* will handle the *computation request* as a batch of data. Once all the results corresponding to the 10 processed elements arrive to the *Client* device, the *continuation* option will be selected. Now the *computation request* will be handled as data stream, where the remaining 2 *DataItems* will be processed and the *Scheduler* will schedule new *tasks* as new *DataItems* are published until a maximum of 10 *DataItems* are processed. Figure 3.6 demonstrates this process.

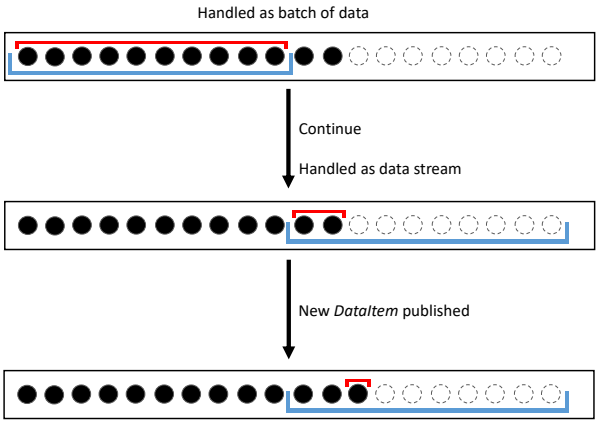


Figure 3.6: Continuation of a *computation request* diagram.

The *Scheduler* is also responsible for rescheduling failed *tasks*. A *task* scheduled for execution in a device that holds (part of) the data to process may successfully conclude its execution or fail. A *task* is considered to fail if the device to which the said *task* was sent, does not send an acknowledge message, stops sending *heartbeats*(also known as Task Status messages in our system) or sends a message notifying about some error that might have occurred. When the device to which a *task* was sent, does not have the specified *Service*, it will send an error message to the *Scheduler*. The *task* succeeds if all the data was processed without any error.

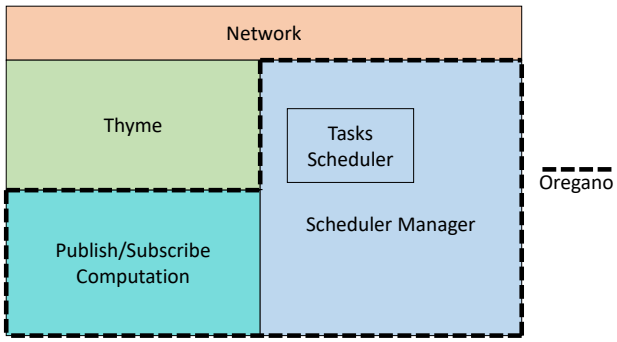


Figure 3.7: *Scheduler's* subcomponents stack.

The *Scheduler* is composed by multiple subcomponents, as observed in Figure 3.7. Contrary to *Client's* subcomponents stack presentation where the *Application* is the first subcomponent, *Scheduler's* entry layer is the *Network*, because scheduling actions are triggered by the reception of remote messages. The *Network* interacts with Thyme through the reception of subscriptions with computation and publish messages, which will consequently interact with the *Publish/Subscribe Computation* component. The

Publish/Subscribe Computation is the same as the one described in the *Client* paragraph of this Section, however, in the context of the *Scheduler*, this subcomponent has the responsibility of handling subscriptions with computation and publish operations, triggering different handlers on the *Scheduler Manager*.

The arrival of a subscription with computation to the *Scheduler Manager* triggers a new process of scheduling and distributing *tasks*. The actual process of scheduling is assured by *Tasks Scheduler*, based on the meta data obtained from Thyme. This subcomponent can be defined/implemented by the application developer, allowing the developer to customize the scheduling process to one that meets the needs of his/her application. Both sequence diagrams of a subscription with computation and a continuation of a subscription with computation being handled by a *Scheduler* device can be observed in Figures 3.8 and 3.9, respectively.

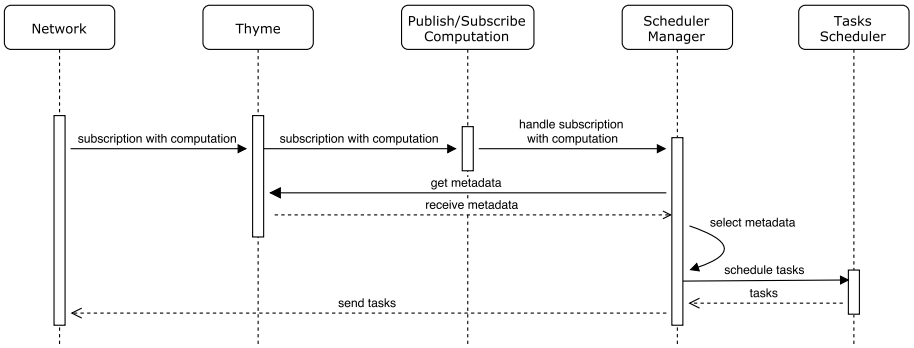


Figure 3.8: Sequence diagram of a subscription with computation handled by the *Scheduler* component.

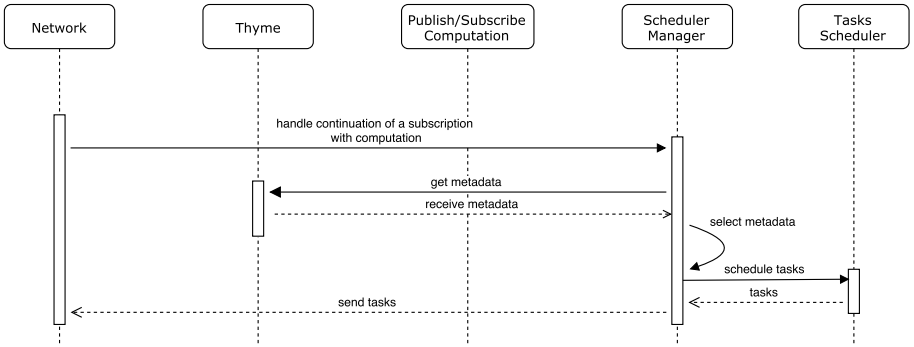


Figure 3.9: Sequence diagram of a continuation of a subscription with computation handled by the *Scheduler* component.

When a publish with a *tag* that has computation associated to it arrives to the *Scheduler Manager*, depending on whether the subscription with computation that is affected by this publication is being handled as a data stream, the *Scheduler Manager* tries to send a notification to the *Client* component of the device that requested the computation, stating that a new *DataItem* has been published and that this *DataItem* will or will not be processed. Until the maximum amount of *DataItems* to be processed is not reached, the

Scheduler Manager schedules a *task* related to the newly published *DataItem*. When the maximum amount of *DataItems* is reached the *Scheduler Manager* will no longer handle the *computation request* in question. The sequence diagram of a publish, handled by the *Scheduler* component can be observed in Figure 3.10.

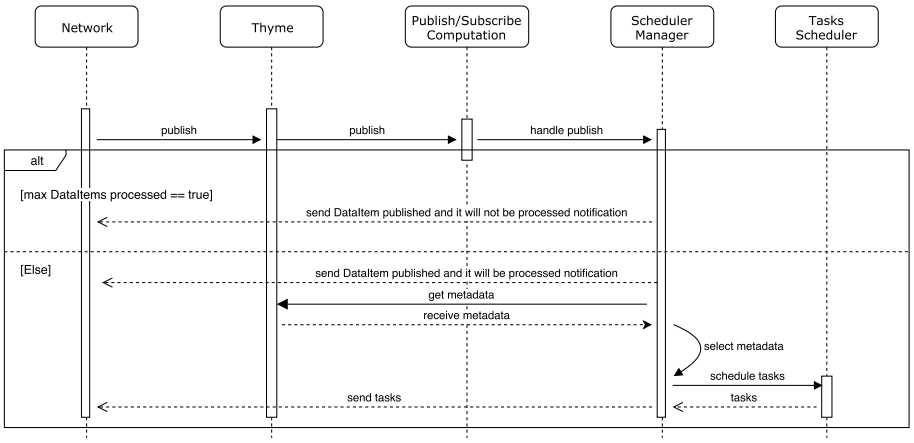


Figure 3.10: Sequence diagram of a publish with a *tag* with computation associated, handled by the *Scheduler* component.

Computing The *Computing* component is responsible for creating a *MDD* with the *DataItems* locally stored in Thyme. Through the service identifier and the service arguments passed in the *task* message exchanged between the *Scheduler* and the *Computing* component, the latter will execute the corresponding *Service*, providing it the data to be processed and the service arguments. Upon execution of the *Service*, this component will decide whether to publish the results with a result *tag*, or directly notify the *Client* device that requested the computation, on the whereabouts of the result data so that the user can download them to his/her device.

Computing is composed by multiple subcomponents, as observed in Figure 3.11. Similar to the *Scheduler's* subcomponents stack, *Computing's* entry layer is the *Network*, because computing actions are triggered by the reception of remote messages. The *Network* interacts with the *Executor* through the reception of stop and *tasks* messages and by sending failure, acknowledgement, success and notification result messages. The *Executor* is responsible for executing and managing all *tasks* that might have arrived to a device, being responsible for interacting with Thyme in order to obtain the *DataItems* that are stored locally and need to be processed. In order to use the *Services* defined by application developers, the *Executor* interacts with *Service Registry* to obtain instances of the specified *Services*. *Executor* also interacts with *Result Delivery Strategy* and *Result Delivery* components to handle the results. The *Result Delivery Strategy* can be implemented by the application developer and is responsible for choosing whether the results should be published or whether a notification should be sent directly to the requester's device. *Result Delivery* is the actual component that performs the actions chosen by *Result*

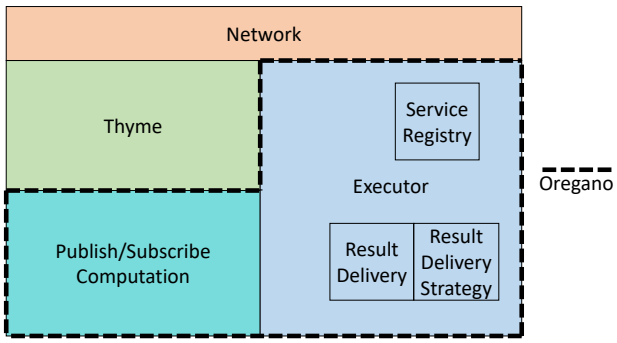


Figure 3.11: *Computing's* subcomponents stack.

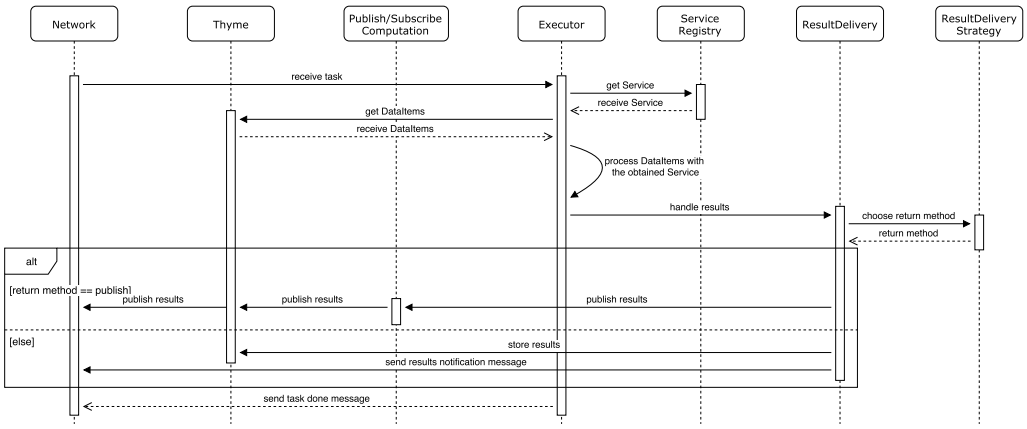


Figure 3.12: Sequence diagram of a *task* being handled by the *Computing* component.

Delivery Strategy. This component will interact with *Publish/Subscribe Computation* if the results are published. It will also interact with Thyme if a notification is sent, since the result data needs to be stored in Thyme to be able to download them. The sequence diagram of a *task* being handled by the *Computing* component, in a situation where it is assumed that there are no faults, can be observed in Figure 3.12.

3.4.2 End-To-End Operations Workflow

Based on the description of the subcomponents discussed in Subsection 3.4.1, we can now present the workflow of our two main operations, **publish with pre-process** and **subscription with computation**.

In terms of the **publish with pre-process**, this operation is responsibility of a single component, *Client*, and consists, firstly, in a publish requested by the *Application* and handled by *Publish/Subscribe Computation* as a way of publishing a *DataItem* with a specified *tag*. If the publish operation concluded successfully, the *Request Manager* will obtain a service from *Service Registry*, by using the service identifier specified in the **publish with pre-process** operation request, and the *Request Manager* will pre process

the corresponding *DataItem*. The result of the pre processing is persisted so that the *DataItem* does not need to be pre processed again.

In terms of the **subscription with computation**, this operation depends on the 3 main components to succeed, as observed in Figure 3.13. This operation starts in the *Client* component, through an interaction between the *Application* and the *Request Manager* subcomponents. The *Request Manager* starts by generating and subscribing to a result *tag*, and subscribing with computation to the specified input *tag*. The subscriptions are assured by *Publish/Subscribe Computation*. Once a subscription is received by a *Scheduler* device, and relayed to the *Publish/Subscribe Computation*, an handler is trigged in the local *Scheduling Manager* to start the whole scheduling process.

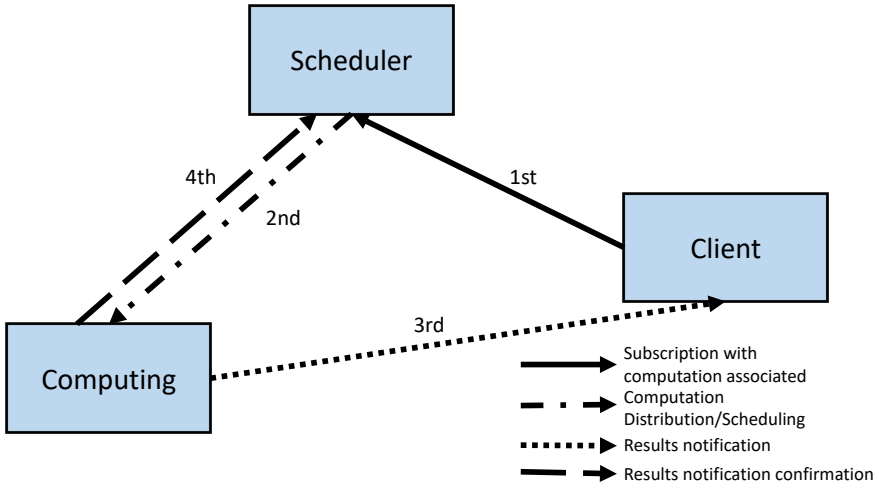


Figure 3.13: Overview of our system's components and the workflow of a subscription with computation.

The subscription request contains the number of *DataItems* already processed. Taking into consideration this information and all the meta data associated to the specified input *tag*, obtained through Thyme, a parametrizable maximum number of *DataItems* will be chosen to be processed. Each one of this *DataItems* has an unique identifier. All of the meta data corresponding to the *DataItems* chosen are passed to *Task Scheduler*, in order to decide which devices will process which *DataItems*. *Tasks Scheduler* will return a group of *tasks* that need to be sent to the target devices. From this four different scenarios can arise:

i) if no *tasks* were returned by *Tasks Scheduler* and the ending time of the request is less than the current time, it means that there are no more elements to be processed therefore an error/stop message stating this, will be sent to the *Client* device;

ii) if no *tasks* were returned by *Tasks Scheduler* but the ending time of the request is larger than the current time, it means that currently there are no more *DataItems* to be processed but in the future more *DataItems* can be published. In this case

the request received will be handled as a data stream. From now on, every time a *DataItem* is published, a notification is sent to the *Client's* device stating that a new *DataItem* was published and it will be processed, followed by the scheduling of a *task* associated to the newly published *DataItem*. Once the maximum number of elements to be processed is reached, a message is sent to the requester's device *Client* component stating that a new *DataItem* was published but it will not be processed. This will inform the user that no more *DataItems* will be processed until he/she uses the *continue computation request* option offered by the *Client* component.

iii) if *tasks* were returned by *Tasks Scheduler* but the total number of *DataItems* to be processed, in all *tasks*, is less than the maximum number of elements to be processed and the ending time of the request is larger than the current time, it means that the *tasks* will be sent to their destination and only then the request will be treated as a data stream. From there on, every new *DataItem* that is published, a new *task* will be scheduled in order to process the newly published *DataItem*, until the maximum number of *DataItems* to be processed is reached.

iv) if *tasks* were returned by *Tasks Scheduler* and the total number of *DataItems* to be processed, in all *tasks*, is equal to the maximum number of elements to be processed at once, and there is still more *DataItems* to be processed in future requests, the request is handled as batch of data. All the *tasks* are sent to their destinations.

Every time *tasks* are sent, the *Scheduler Manager* starts waiting for *heartbeat* signals/task status messages corresponding to that *task*, in order to reschedule the said *task* in case of failure. These *tasks* contain specific information, and it is important to refer that they contain a set of identifiers of the *DataItems* to be processed and the *Service* identifier to be used to process those *DataItems*.

Once a *task* message reaches the target's device *Computing* component, the *Executor* will try to obtain the specified *Service* from *Service Registry*. After that, the *Executor* will create a *MDD* with the *DataItems*, that are stored locally by Thyme, specified by the set of object identifiers passed in the *task* message. While using the obtained *Service* to process the newly created *MDD*, an *heartbeat* mechanism is constantly running, notifying the *Scheduler* device. When the execution of the *Service* terminates, the *Executor* passes the results to *Result Delivery*, which will consequently ask *Result Delivery Strategy* for the chosen return method. After this action, a message confirming the notification of the results is sent to the *Scheduler's* device.

As soon as the requester's device *Client* component is informed of the state of the results, either by publication of results or a results notification message, the *Request Manager* forwards this information to the *Application*. Based on this, the Application can trigger a *continuation* of a *computation request*. The *continuation* operation can only occur if the maximum number of *DataItems* to be processed was actually processed, and in case of data stream besides this, a message stating that a new *DataItem* was published

and it will not be processed needs to be received first. The *continuation* of a *computation request* is handled in part like a freshly submitted *computation request*, but instead of the communication between a *Client* and *Scheduler* being a responsibility of the *Publish/-Subscribe Computation* of each one of these components, it is assured directly by the *Request Manager* and the *Scheduler Manager*. So when a *computation request* is *continued*, a new computation request message is sent by the *Request Manager* of the *Client* device to the *Scheduler Manager* of a *Scheduler* device and handled like a normal/first request.

3.5 Summary

In this chapter we presented OREGANO, a distributed computing *framework* capable of processing batches and stream of data without using Internet services. We started by giving an overview of our *framework* and the underlying layers that allows us to focus solely on the distributed computing part. We proceeded by presenting certain concepts relevant for a better understanding of OREGANO and the *API* offered by our *framework*. Regarding the *API*, we discussed the different operations available and gave some examples on how to use the *API*. We then addressed OREGANO's architecture, where we presented and addressed the three main components of our *framework*, *Client*, *Scheduler* and *Computing*. In the next chapter we will address certain implementation details regarding: i) the technologies used to develop OREGANO, ii) an OREGANO *Service* and iii) OREGANO's main components.

Chapter 4

System Implementation

This chapter addresses several implementation details of Oregano's *framework* prototype. Firstly we will present the technologies used in the implementation of our *framework* in Section 4.1. We will then address several implementation details regarding *Service* development, life cycle and how it is invoked, in Section 4.2. Lastly, in Section 4.3, we will discuss certain implementation details of our system's main components.

4.1 Technologies used

Oregano was written in Java language and developed targeting Android platforms. Due to the decision of developing a *framework* that follows a stream programming model, we decided to use Java 8 streams [38]. Since only on Android 7.0 (API level 24), Java 8 streams feature became supported, Oregano and all applications developed with Oregano require a minimum SDK Version of API level 24 [13].

Oregano has a wide variety of messages that are exchanged between mobile devices that form the network of our system. Since our *framework* was developed targeting wireless networks formed exclusively by mobile devices, there was a need of using technologies that allowed us to exchange messages quickly and efficiently, where only the data exclusively needed was sent/received. Therefore we decided to use Protocol Buffers [12], as it allows us to serialize structured data in a faster and smaller way, compared with other solutions like XML and Java serialization. Also, the fact that only the information needed is exchanged, make Protocol Buffers a better solution for our *framework*, instead of Java serialization where extra data regarding Class information is encoded into bytes. More over, the simplicity of usage of Protocol Buffers is an advantage, as it is only required to the developer to define structured data in a *.proto* file and consequently compile it to create a class that abstracts all the parsing and encoding/decoding of a message.

Besides, the ability of extending structured data format creating new ones but supporting both old and new formats, in regard to reading and writing, has already proven to be an asset to our *framework*. An example of the usage of this ability, is our OreganoSubscription message that can be used as a regular/normal subscription or a subscription with computation, by specifying certain fields as required fields (shared by both types of message), and other as optional fields and repeated fields (used only in subscriptions with computation messages).

4.2 Service implementation details

In this Section we will present certain implementation details regarding *Service* development, how are we able to avoid sending code to be executed through the network and what type of control and feedback does an application developer has when invoking *Services*.

4.2.1 Service development

Regarding *Service* development, there are a few implementation details that need to be addressed in order for a better understanding of our *framework*: i) how do we support different types of data; ii) how do we use previously defined *Services* to process/execute *tasks* and iii) what is a result in our *framework*.

As already addressed in Section 3.2, our *framework* has the idea of *Service* and *Service with pre-process*. In order for application developers to implement/define a *Service* they need to extend a *Service* or a *Service with pre-process* and implement four or five abstract methods, depicted in Tables 4.1 and 4.2, respectively.

Table 4.1: Abstract Class *Service* where Input, Args and Output extends *DataItem*.

abstract Class<Input> getInputClass()	Returns the input Class
abstract Class<Args> getArgsClass()	Returns the service arguments Class
abstract IMDDStream<Output> process(IMDDStream<Input> inputStream, List<Args> serviceArgs)	Processes the inputStream returning a result stream
abstract DataItem outputToDescription(Output output)	Returns the correspondent description of the output DataItem
RunningTask newTask(IMDD<BytesDataItem> mdd, **byte**[][] serviceArgs, Set<ObjectIdentifier> objectIdentifiers)	Creates and returns a new RunningTask that will process the data in mdd

When using Protocol Buffers there is a gain in performance, the size of the messages sent in the network decreases, but certain information regarding the transmitted objects is lost. In this context, it is up to the application developer to provide this information through the implementation of the methods **getArgsClass** and **getInputClass/getOriginalInputClass**. By using these methods, Oregano is able to support different types of data, since they return the input/original input and service arguments

Table 4.2: Abstract Class *ServiceWithPP* which extends *Service*, where PreProcessedInput extends *PPDataItem*, and Input and Args extend *DataItem*.

abstract Class<Input> getOriginalInputClass()	Returns the original input Class
void setWillPreProcess(**boolean** value)	Specifies if this service needs to pre process all input elements before processing them
abstract PreProcessedInput preProcess(Input i, List<Args> serviceArgs)	Pre processes object i returning the result of the pre process
void preprocess(Input i, List<Args> serviceArgs, ObjectIdentifier objectIdentifier)	Calls preProcess implemented by the framework user, pre processing i, adding the result of the preprocess to a mdd that will be persisted

classes used by some *Service*, allowing us to build *DataItems* from bytes through the **fromByteArray** method of the *DataItem* class presented in Table 3.1, using Java Reflection. We require to build *DataItems* from bytes, because service arguments are sent through the network in form of bytes, the data to be processed (input/original input) is also stored in Thyme in form of bytes and it can still be replicated to other mobile devices, being sent as bytes through the network.

The execution/process of *tasks* is mainly assured by the **newTask** method in Table 4.1. When using a *Service with pre-process*, the **newTask** method prepares the data to be processed, by obtaining the corresponding pre-processed *DataItems*, if they exist, or in case they do not, uses the **preProcess** method to pre process data. The pre process only occurs in this phase if the method **setWillPreProcess** is called priorly by the *Application*. Besides preparing data to be processed, the **newTask** method returns a new RunningTask which consists on a *Callable* object that will be submitted to a Java *ExecutorService* and executed as an asynchronous task in the *Computing* component in order to process data. The **call** method of this *Callable* uses the **process** operation implemented to process data.

In terms of the results of processing data, they are returned by the call method of a RunningTask. A result in our *framework* consists on a tuple composed of a *MDD* with the result *DataItems* and a collection of the corresponding descriptions of each result *DataItem*. These descriptions are generated by the **outputToDescription** method, which has the objective of allowing application developers to return a smaller representation of each result *DataItem*, like the thumbnail of an image or the title of a text. A description is associated with the meta data of a result *DataItem* and it reaches the mobile device that requested computation through the result notifications, independently of which mechanism was chosen to notify the *Client* about the results. This can be helpful as it offers users a preview of the results allowing them to know which results are available to be downloaded. If an application developer is not interested in implementing this method he/she can return a null value informing our *framework* to ignore the description.

Initially presented in Subsection 3.3.2, a *MDDStream* is a wrapper class for a Java 8 stream, sharing a similar *API* to the Java 8 stream *API*, additionally offering methods to persist and obtain persisted *DataItems*. *MDDStream* also offers a method that allows the

creation of the tuple returned by the **call** method of a RunningTask.

4.2.2 Service life cycle management

An application developed with our *framework* can have multiple *Services* and is able to use *Services* developed for other applications. This means that any *Service*, independently of the application to which the *Service* was developed to, can be used by any other Oregano application to process data. This can be helpful in scenarios where in a social location where multiple Oregano applications may be in use by different mobile devices. An example scenario could be: in a particular social site there are two Oregano applications being used, the application we have been using as an example in this thesis, a photo-sharing application, and an application to find missing persons. Both of these applications share the same *Service*, a facial recognition *Service* and share the same *DataItems* that are of interest to both. Although scenarios like this are rare, where two different applications publish *DataItems* of interest to both applications, Thyme's replication mechanisms could eventually replicate *DataItems* of one application to the other, making the device to which data was replicated, eligible for computing/processing data.

Oregano is able to offer this ability because of the *Service Registry* component, presented earlier in Subsection 3.4.1. The *Service Registry* offers methods to register, replace, remove and get *Services*. All of these methods receive an identifier key of a *Service*, additionally the register and replace methods also receive a *Class<? extends Service>* object which represents the *Service* to be registered/replaced. Since a Java *Class<T>* implements *Serializable*, we use Java serialization to persist *Services* in the external storage of an Android mobile device. This means that an application only needs to register its *Services* once, such as when the application first runs. At the same time, by following this approach we avoid sending code through the network to be executed.

The *Service Registry* component also assures that every time an application is booted, all previously persisted *Services* are deserialized and instantiated by using Java Reflection. In an application only exists one unique instance of each one of the persisted *Services*, since to actually process data using *Service's* **process** operation, our *framework* only requires calling the **newTask** method which returns a Callable that is submitted to a Java *ExecutorService* and executed as an asynchronous task.

4.2.3 Service invocation

An Oregano *Service* is invoked by applications through Oregano's **publish with pre-process** and **subscribe with computation** operations. Both of these operations, besides receiving as arguments pertinent informations for the successful conclusion of both types of operations, they also receive handlers, observed in Subsection 3.3.3, that offer feedback and control to the application developer regarding the operations in question.

The **publish with pre-process** operation receives an *ObjectOperationHandler*, devised for Thyme's publish operation, which is an interface that requires implementation of two

methods that are triggered when a successful or failed publish occurs. In our publish we create a new *ObjectOperationHandler* that is passed to Thyme's publish and allows us to know when a determined *DataItem* was published with success, thus initiating the pre-processing of this *DataItem*. The last action that is done in our publish is to trigger the corresponding method of the *ObjectOperationHandler* passed as argument, in order to notify the user about the success or failure of the publication.

In our *framework* a subscription with computation creates a *ServiceRequest*, which will consequently create a *job*. Whenever a *computation/service request* is continued a new *job* is created. This is the reason why we stated earlier in Subsection 3.4.1, that the *Client* component is the only component responsible for the evolution of its *computation requests* and both *Scheduler* and *Computing* do not distinguish between a submission of a *computation request* or a continuation. The *Scheduler* component is only aware of the existence of *jobs* and the *Computing* component is only aware of the existence of *tasks*. A *job* is composed of multiple *tasks*, depending where the data is. If when handling a *job* the selected group of object identifiers of the *DataItems* to be processed can be divided in two different sets of *DataItems* depending on the location, then two *tasks* will be created and sent to the corresponding mobile devices.

It is important to detail what information a *job* and a *task* contains, to better understand its purposes, and how the main components of Oregano use them to complete operations successfully. A *job* has informations regarding:

- its own identifier, the identifier of the *computation/service request* to which it is associated with and the identifier of the *Service* to be used when processing data.
- the input and result *tags*.
- the start and the end time stamp specified by the *Application* when submitting a new subscription with computation.
- last time stamp received from a *job* acknowledge indicating when the previous *job* started being handled and the total number of *DataItems* already processed. Both values avoid *Scheduler* from keeping state regarding past *computation requests* that it had already handled.
- the address of the mobile device that submitted the subscription with computation.
- the service arguments to be used by the *Service* when processing data.

A *task* has informations regarding:

- the identifiers of the *computation/service request* and the *job* to which it is associated with; its own identifier; an identifier created by the *Scheduler*, specifically for the *job* being handled; and the identifier of the *Service* to be used when processing data.

- a collection of object identifiers that specify which objects must be processed by the device receiving this *task*.
- the address of the mobile device that submitted the subscription with computation and the address of the device that scheduled this *task*.
- the result *tag*.
- the service arguments to be used by the *Service* when processing data.

The **subscription with computation** receives a *ResultHandler*, Table 4.3, that allows application developers to control a *computation request* and obtain information regarding the state and results of a request.

Table 4.3: Interface ResultHandler.

void onResult(ObjectIdentifier objectID, **byte**[] description, **List**<Address> dataLocation, TagExpression resultTag)	Called when a result notification arrives
void onFailure(**String** reason)	Called when the request could not be accomplished because an error was experienced
void onServiceRequestDone()	Called when all the expected result notifications arrived, allowing for a continuation of a subscription with computation
void onServiceRequestStarted(IServiceRequestController serviceReqController)	Called when this computation request was acknowledged

The method **onFailure** is called when a *ServiceRequest* fails, stopping every action related to this subscription with computation. This method can be triggered from the moment a subscription with computation starts till the moment an user stops this subscription.

The method **onResult** is triggered every time a result notification arrives to a device that subscribed with computation to some *tag*, independently if it arrives in form of a publish or a result notification message. From the arguments received an user is able see a preview of a result *DataItem* and download a result by using one of the addresses in dataLocation and calling Thyme's download operation that allows users to download *DataItems*.

The method **onServiceRequestStarted** is triggered when a *job* is acknowledged by a *Scheduler* component of a device located in the cell to which the input *tag* was mapped. This happens every time a subscription with computation is submitted and when a *ServiceRequest* is continued. This method will notify the user that his/her request is being handled, passing as argument an object that provides methods that return the number of *DataItems* already processed and the total number of *DataItems* that were published with a input *tag* between a specified start time and end time. The object passed as argument, also provides a continue and stop methods that allows users to control their subscriptions with computation.

Finally, the method **onServiceRequestDone** is triggered every time a *ServiceRequest* can be continued, informing the user that it is now possible to continue a *ServiceRequest*. This includes scenarios where a *job* is being handled as a data stream and a maximum number *x* parametrizable of *DataItems* were processed, therefore a message is sent from a *Scheduler* to a *Client* stating that no more *DataItems* will be processed until a continue of *ServiceRequest* is requested by the user.

4.3 Architecture implementation details

In this Section we will address implementation details regarding each one of our *framework's* main components. We will also detail how is Oregano able to handle the entrance and exit of mobile devices in the network, i.e churn, in each one of the components. Furthermore, we will explain how does an application developer initializes Oregano.

4.3.1 Instantiation

Oregano's initialization is dependent on Thyme's initialization, as it supports the creation of multiple Oregano instances like Thyme. Each one of these instances has a different world associated to them. This idea of world was devised by Thyme's developers, and it consists on the definition of a geographical space for the Cell-Based Geographic Hash Table, earlier presented in Section 3.2 and observed in Figure 3.1. It is not possible to exchange messages between worlds. This idea of several worlds offers application developers extra flexibility when creating their applications, allowing multiple different applications to be used in the same social location but in different worlds, if so they desire. At the same time, an application can be developed with multiple purposes, justifying the individualization between worlds but still using the same distributed system.

To initialize an Oregano instance an application must provide as arguments an Android *Activity*, *Context*, and a *Bootstrap* object devised for Thyme. The *Bootstrap* object allows applications to create new worlds or join existent worlds, after the search for other mobile devices in the vicinity terminated. When creating a new world, an application can specify: i) the center of the world in GPS coordinates (it is usually the location of the mobile device creating the world); ii) distances in meters from the center of the world to the north, to the south, to the east and west of that point; iii) size in meters of the cells; iv) the name/identifier of the world, so others can join the world being created; v) the duration in milliseconds of the world being created.

The network used by Oregano is initialized by Thyme, where the type of communication technology is chosen based on the active technology on the mobile device. When multiple communication technologies are active Thyme gives priority to Wi-Fi, then Bluetooth and lastly Wi-Fi Direct.

Regardless of the number of Oregano instances created, an application will only have a single instance of each one of our three main components. All of these instances

are initialized by the first Oregano instance.

Still regarding Oregano's initialization, every first time an Oregano is instantiated by some application, the application can specify a path to a properties file which has values that allow to customize certain behaviours of the main components. If no path is specified, Oregano uses a properties file created through Java Properties, with default values defined by us. These values correspond to different system and messages timeouts, maximum numbers of different types of errors/failures supported, maximum number of *jobs* and *tasks* to be handled by a mobile device at the same time and maximum number of elements to be processed at once.

4.3.2 Client

The *Client* is responsible for instantiating a *Service Registry*. Similar to the three main components, there is only one *Service Registry* for all instances of Oregano in an application, and this instance is shared by the *Client* and by the *Computing*.

Whether it is a newly submitted subscription with computation or a continuation of a subscription with computation, both operations require certain actions to take place in order to notify the user, through the **onServiceRequestStarted** method observed in Table 4.3, that its request has started successfully. Both operations require two acknowledge messages. Besides these messages, a newly submitted subscription also requires that both the subscription to the input and result *tag* succeeds. When either of the subscriptions fails or does not succeed during a specified time interval in the properties file, the *computation/service request* fails and the user is notified through the **onFailure** method, observed in Table 4.3. Regarding the acknowledges, one of the acknowledges consists on an oregano acknowledge which objective is to notify the sender, in this case the *Client*, that the *job* sent has been successfully received and that it will be handled in the future. The second required acknowledge consists on a job acknowledge which main objective is to notify the *Client*, that its *job* will now be handled. The job acknowledge has information regarding the total number of *DataItems* that exist in the *MDDS* to which the input *tag* is associated with, the number of *DataItems* that will be processed, a time stamp indicating when the *job* started to be handled and whether the *job* in question is being handled as a data stream or a batch of data.

Although the application is able to request a continuation of a subscription with continuation at any time, through one of the methods offered by the object passed as an argument in method **onServiceRequestStarted** observed in Table 4.3, this method will only perform a continuation and return true if certain, already addressed conditions, are met. Whenever a continuation of a subscription with computation is done, a new *job* is created and sent to one of the devices in the cell to which the input *tag* was mapped.

Churn The *Client* component is able to partially deal with churn, i.e entrance and exit of mobile devices from the network. In the *Client* component, we take a more "passive"

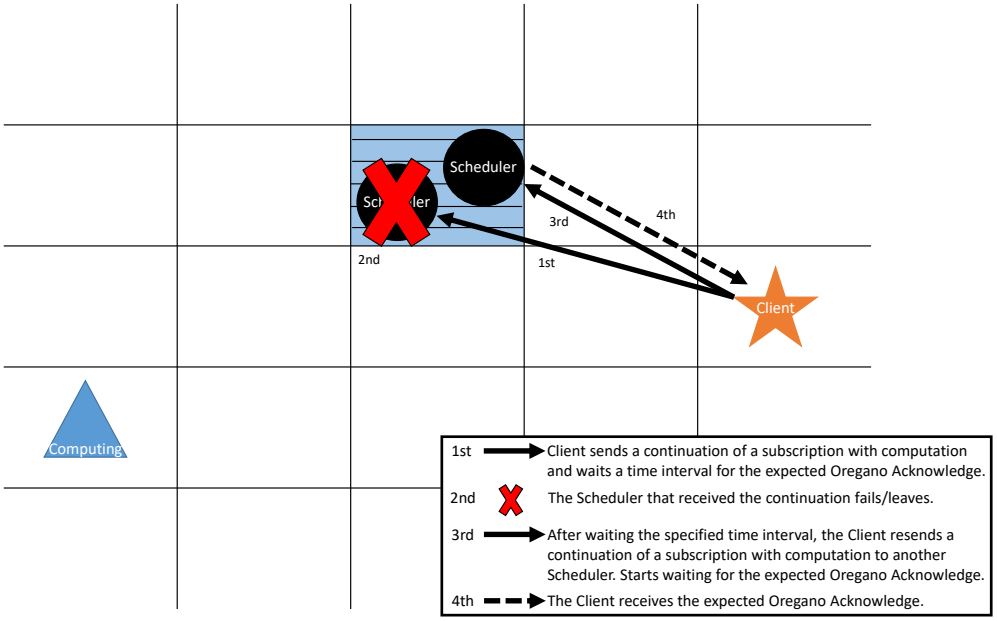


Figure 4.1: Scenario example of the churn experienced between a Client node and a Scheduler node.

approach to how we handle churn, in that we focus on telling the user that an error has occurred therefore his/her *computation request* will be stopped. An example of this was already presented in this Subsection, where we stated that the *Client* waits a specified amount of time for the subscriptions of the result and input *tag* to succeed, and when that does not happen, the user is notified. Also, regarding cases of continuations of *computation requests*, an oregano acknowledge needs to arrive between the same time interval used for the subscriptions, by the end of the time interval, if no oregano acknowledge arrived, the *job* that was not acknowledged will be sent to some other device located in the same cell of the device that did not acknowledged. This process is repeated a maximum of three times, after that the user will be notified about the failure of his/her request. An example of a scenario of churn experienced between a *Client* node and a *Scheduler* node, when a continuation of a *computation request* is attempted is depicted in Figure 4.1.

4.3.3 Scheduler

The *Scheduler* component is responsible for instantiating a *WorkThreadPool* class, which consists on a wrapper of a Java ThreadPoolExecutor. There is only one instance of a *WorkThreadPool* for all Oregano instances in an application, and this instance is shared by the *Scheduler* and *Computing* component. The main purpose of this *WorkThreadPool* is to allow the assignment of threads to handle *jobs* and *tasks* that arrive at each of the two components, while limiting the number of threads created. Through the properties file, an application developer can specify the maximum number of *jobs* and the maximum number of *tasks* to be handled at the same time by the threads in the thread pool, limiting

the size of the thread pool. By default, these maximum numbers are correlated to the number of available processors to the Java virtual machine, e.g. if four processors are available, then a maximum of two *tasks* and two *jobs* are handled at the same time.

Whenever a *job* arrives to a *Scheduler*, whether it was sent directly, continuation of a *computation request*, or a subscription with computation was received, the newly arrived *job* is enqueued in a queue of *jobs* and an oregano acknowledge is sent to the *Client's* device. Only when there is an available thread in the *WorkThreadPool*, this *job* is dequeued and handled.

By using the meta data offered by Thyme and the information *job* contains, regarding specifically the number of *DataItems* already processed and the last time stamp of when a previous *job* was handled, the *Scheduler* component is able to select the meta data associated to each *DataItem* that was not processed yet. The maximum number of *DataItems* to be processed at once, specified in the properties file is also taken in consideration. The selected meta data is passed to the *Tasks Scheduler* subcomponent, as already addressed in Subsection 3.4.1. The *Tasks Scheduler* corresponds to an interface with only one method to be implemented by an application developer (if so he/she decides) and it has the sole purpose of choosing which mobile devices are going to process which *DataItems*, based on the meta data received, returning a set of *tasks* to be sent to the corresponding mobile devices. An application developer can set his/her *Tasks Scheduler* at any time during the execution of an application. Only after all *tasks* are created, a *job* acknowledge is sent to the *Client's* device.

Similar to the *Client* component, the *Scheduler* also requires two acknowledges: i) an oregano acknowledge which indicates that a *task* was received successfully; and ii) a *task* acknowledge, that will trigger an *heartbeat* mechanism addressed below.

Churn The *Scheduler* is able to deal with churn through *heartbeat* mechanisms and consequent reschedule of failed *tasks*. When handling a *job*, if at least one *task* is created, sent to some device that has data to be processed and the corresponding *task* acknowledge arrives, the *Scheduler* starts an *heartbeat* mechanism that expects to receive a task status message stating that the corresponding *task* is currently running/being handled. The *heartbeat* time interval and the total number of missed task status running messages can be specified in the properties file. If some *Computing* mobile device becomes unreachable, therefore the maximum number of missed running messages is reached, the *Scheduler* reschedules the failed *task* sending it to a different mobile device that also has the data to be processed. After a maximum number of failed *tasks*, specified in the properties file, the same quantity of *DataItems* that were not processed due to the failure of the *tasks*, is chosen and scheduled in order to be processed. If no more *DataItems* are successfully processed and the end time of the request is reached, the *Scheduler* notifies the *Client* which consequently stops a *computation request* and notifies the user. An example of a scenario of churn experienced between a *Scheduler* node and a *Computing* node, when scheduling *tasks* is depicted in Figure 5.11. In this scenario it is assumed that every

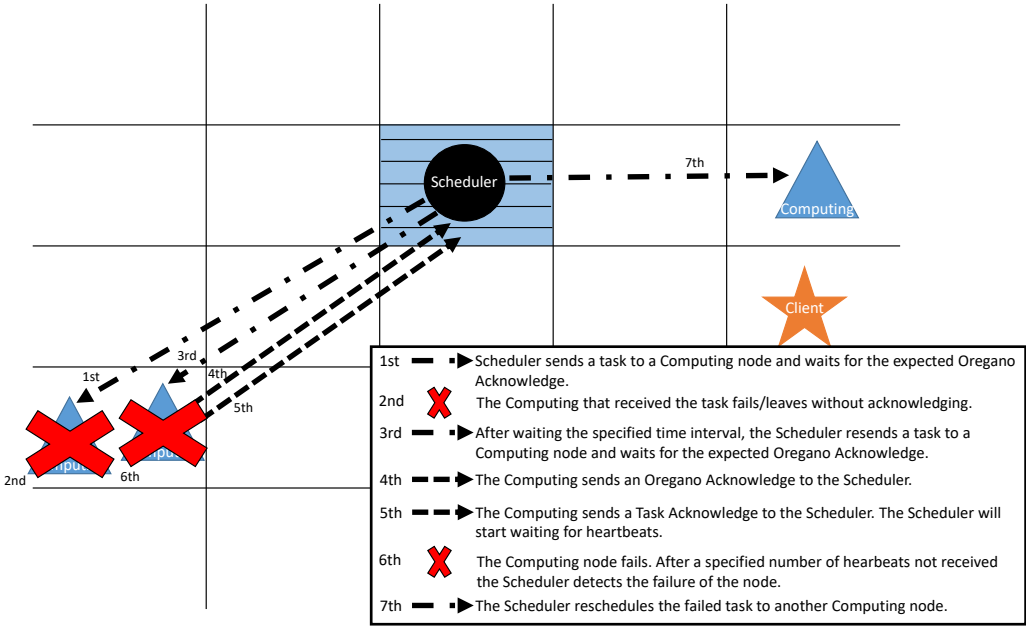


Figure 4.2: Scenario example of the churn experienced between a Client node and a Scheduler node.

Computing node has the same *DataItems* therefore they are eligible for processing the *task* being scheduled/rescheduled.

4.3.4 Computing

As already stated in Subsection 4.3.3, the *Computing* component also uses a *WorkThreadPool* in order to have a limited number of threads handling *tasks* received. Similar to the *Scheduler*, whenever a *task* arrives it is enqueued in a queue of *tasks* and an oregano acknowledge is sent to the *Scheduler*. Only when there is an available thread in the *WorkThreadPool*, this *task* is dequeued and handled.

When the *Computing* starts handling a *task*, it first checks whether the service identifier passed in the *task* is registered in *Service Registry*. If the corresponding *Service* is not found, an error message is sent to *Scheduler* component, forcing a reschedule of this *task*. If the *Service* is registered, the *Computing* creates a *MDD* with all the *DataItems* to be processed that are stored in Thyme. Only after the creation of this *MDD* a *task* acknowledge is sent to the *Scheduler*. After that, the *Computing* component calls the **newTask** method, discussed earlier in Subsection 4.2.1, which returns a *Callable* that is submitted to a Java *Executor Service*. During the computing of data, a *task* running message is sent periodically to the *Scheduler* device, proving the liveness of the mobile device that is processing data.

After the conclusion of computing of data, the results are passed to the *ResultDelivery* subcomponent which interacts with *ResultDeliveryStrategy* in order to choose and act upon the decision of publishing the results or sending a result notification message to

the *Client* device, as already addressed in Subsection 3.4.1. The *ResultDeliveryStrategy* corresponds to an interface that can be implemented by the application developer which has the sole purpose of choosing the adequate method to notify the user regarding the results. Similar to *Tasks Scheduler*, it can be set at any time during the execution of an application.

Churn The *Computing* component is not affected directly by churn, in the sense of when comparing to both *Scheduler* and *Client* where these components interact between each other and with the *Computing* component, both components are dependent on others' components actions and expect the arrival of certain messages to proceed with the *computation request*, this is not *Computing's* reality. As this component only sends messages and does not expect any other messages like acknowledges. If any of the messages sent by *Computing* do not reach a target destination, because the target has became unreachable, the *Computing* is not affected. It will still do any action regardless whether there are devices expecting for its messages or not, considering that the *Scheduler* did not send any stop message to the *Computing* component regarding this said *task*.

4.4 Summary

In this chapter we addressed certain relevant implementation details regarding Oregano. We started by presenting the different technologies used in the implementation of our *framework*. We then proceeded by addressing certain implementation details regarding a *Service*, namely the different methods offered by this class. We also detailed how does the life cycle of a *Service* is treated in our *framework*, as well as presented other details regarding the invocation of a *Service*, different handlers and the attributes of a *job* and a *task*. We followed by explaining the instantiation of Oregano. We finished by giving some implementation details regarding the three main components *Client*, *Scheduler* and *Computing*. Furthermore, we addressed how each one of the component handles churn.

In the next chapter we will present and address the results of the evaluation performed on Oregano, both on a real environment and on a simulated environment, using different metrics.

Chapter 5

Experimental Evaluation

This chapter presents and discusses different results obtained through experiments done with Oregano's *framework*, on a real environment using mobile devices, and on a simulated environment using a simulator developed for testing. In Section 5.1 we will present what metrics we intend to evaluate. In Section 5.2 we explain what technologies and software were used to test our *framework*. In Sections 5.3 and 5.4 we will discuss the results obtained in different scenarios.

5.1 Evaluation Metrics

With the tests to be discussed in this chapter, we intend to observe the behaviour of our *framework* in different scenarios and its ability to offer application developers a reasonable distributed computation alternative to some Internet Services. In order to do this, we performed tests in a real environment and in a simulated one.

For the real environment we developed an application implemented with our *framework* that was tested on a Wi-Fi network formed by 6 mobile devices. We focused on testing different metrics:

- the latency of the operations offered by Oregano, in different scenarios.
- the scalability of our *framework*, related to the increase of computation requests submitted at the same time interval and the distribution of workload.
- the energy cost of our operations and mechanisms.

For the simulated environment we developed a simulator that is able to create a large number of simulated nodes on the same machine, which are able to communicate between each other on a wired network. The simulator allowed us to control certain scenarios in

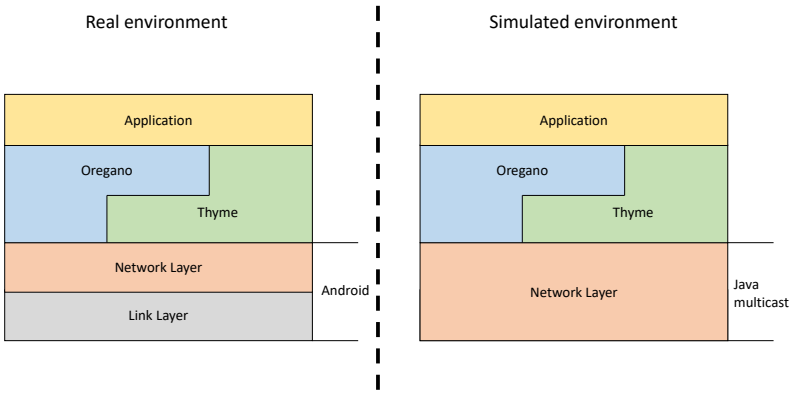


Figure 5.1: System architecture used on a real environment and on a simulated environment.

order to observe Oregano's response/behaviour to particular situations that would be harder to test on a real environment and, at the same time, increase the number of operations and nodes tested. We focused on testing different metrics:

- the workload in terms of messages exchanged between nodes in our system.
- the ability to support the entry and exit of nodes from the network, i.e churn.

Figure 5.1 shows our system's architecture on a real environment(Android devices), and on a simulated environment(simulator). Both architectures are similar, although the layers responsible for communication are different in each version, where the real environment architecture targets Android devices and wireless technologies, while the simulator architecture uses Java multicast to communicate between all the nodes created by the simulator on a wired network. Besides communication, small details regarding localization were also changed in the simulator, although these changes are related to Thyme and not Oregano.

5.2 Study Cases

In order to test our *framework* in a real environment, we developed a facial recognition Android application that was implemented with Oregano, which allowed us to apply significant computation to data. We defined an Oregano *Service* with the ability of applying facial recognition to images, using other images received as service arguments to specify the person(s) to be found in the photos to be processed. This *Service* uses JavaCV [17] library, a Java wrapper for video processing libraries like OpenCv, in order to apply facial recognition to images. The **preprocess** operation consists on extracting every face in a photo being preprocessed. The **process** operation goes through a stream of *Pre Processed DataItems* composed by the original photo and every extracted face from that photo, and applies facial recognition on every face returning the original photos where

Mobile Device	Motorola Moto G (2nd gen)	Motorola Nexus 6
CPU	Quad-core 1.2 GHzCortex-A7	Quadcore 2.7 GHz Krait 450
RAM	1 GB	3 GB
Storage	8 GB	32 GB
Connectivity	Wi-Fi 802.11 b/g/n	Wi-Fi 802.11 a/b/g/n/ac
OS	Lineage 14.1 (Android Nougat 7.1.2)	Android Nougat 7.1.1
Battery	Li-Ion 2070 mAh	Li-Po 3220 mAh

Table 5.1: Specifications of the mobile devices used for testing

the corresponding person's face was extracted. Besides receiving images of people to find as service arguments, the facial recognition service also receives a percentage of the confidence level the algorithm must respect when finding people in a set of data.

The facial recognition application offers the ability to publish photos with or without pre process and subscribe with computation to specific *tags* in order to process data.

It is also important to note that each test to be covered in the sections below, regardless of the number of objects processed, used only two different photos, one being used to specify which person is to be found and the other a two person photo, where only one person corresponds to the person to be found.

For the simulated environment we implemented another Oregano *Service*, less intensive computation wise, since we did not had the objective of measuring latencies but instead we intended to address specific scenarios that did not targeted the actual computation done by a *Service*. This second *Service* consists on a text pattern finding *Service* that returns texts that match with the specified pattern to be found.

5.3 Mobile Devices Tests

The tests performed on a real environment used 6 mobile devices, 3 of each type as observed in Table 5.1, connected to the same Wi-Fi hotspot and had the main objective of observing the latency of our **subscription with computation** operation, the scalability of our *framework* and the energy cost of the operations offered by Oregano. The tests varied the number of mobile devices used based on the scenario being tested. Regarding the Wi-Fi hotspot, a laptop with a Intel(R) Dual Band Wireless-AC 3160 802.11ac adapter was used.

All tests, except for the energy cost ones, were performed by using different Python scripts that issued commands to the mobile devices, using Android Debug Bridge(ADB) through USB.

We performed 5 different tests each one with different goals. In the two first tests, we had the objective of observing the latency experienced when subscribing with computation in different scenarios, as well as the scalability of our *framework*. The third test focus on showing and justifying the utilization of the **publish with pre-process** operation. The last two tests have the objective of allowing us to draw conclusions regarding battery

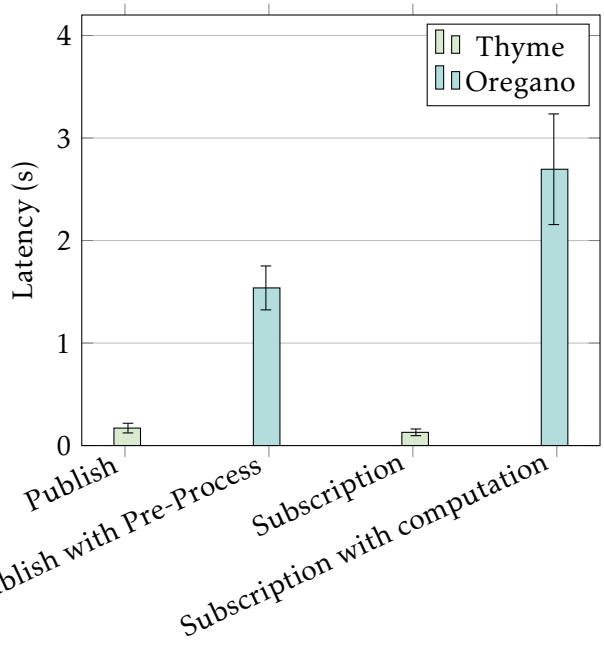


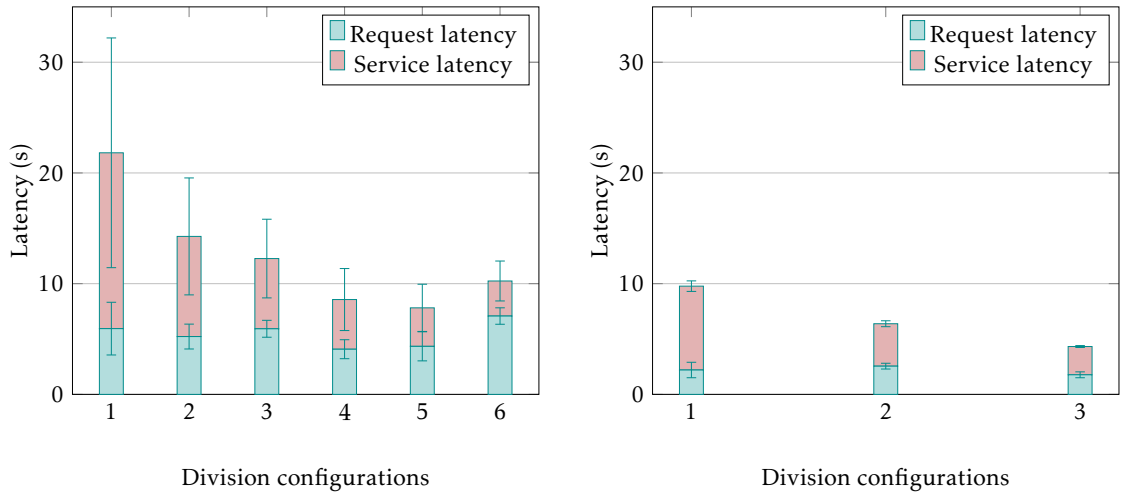
Figure 5.2: Oregano overhead in relation to Thyme's operations.

usage. In all tests, we decided to have a world that only had one cell where all devices were in it, mainly due to the reduced number of devices and the tests that required certain devices to play multiple roles.

Before addressing the tests and its values, it is important, for contextualization purposes, to present the comparison of the latencies observed when performing Thyme's regular publish and subscribe operations, with Oregano's **publish with pre-process** and **subscription with computation** operations. Figure 5.2 shows the comparison between latency values of both systems, where as expected Oregano's latency values are greater. When comparing both publish operations it is important to understand that the **publish with pre-process** latency value consists on the pre-process of an image plus a regular Thyme's publish. In terms of the **subscription with computation** operation, the value presented consists on two Thyme's regular subscribes plus all the logic behind the coordination/management of a computation request and the processing of five photos.

5.3.1 Latency and Scalability

In the first test we had the goal of observing how does having different mobile devices processing data correspondent to one single subscription with computation, impact the latency of that operation. Therefore we initially tested 6 different scenarios, where the base was the same, a device wishes to have data computed, so it subscribes with computation to a specific *tag*. In all scenarios, 30 photos, all equals, were published with pre-process to a specific *tag* without replication. On the first scenario only one device published all 30 photos, being the only one that was able to compute those photos; on the second, 2 devices published each one 15 photos, therefore both were able to process part



(a) Heterogeneous network

(b) Homogeneous network

Figure 5.3: Division of 30 files for x mobile devices

of the data; on the third scenario, 3 devices published each one 10 photos; on the fourth, 2 devices published each one 8 photos and the other 2 published 7 photos each one; on the fifth scenario, 5 devices published each one 6 photos; lastly on the sixth, 6 devices published each one 5 photos.

In all of these scenarios a single subscription with computation was handled, computation wise, by several mobiles devices balancing the workload. It is also important to note that we changed the parametrization, so that only 30 files were processed at once. Also in all tests, we consider the latency of a subscription with computation the time interval starting from the moment a subscription with computation was submitted to the moment a *service request* done is called, indicating that all 30 files were processed and in order to process more files, one has to continue a subscription with computation.

In Figure 5.3a is possible to observe the results of the first test. In this Figure, we divided the total latency in two different latencies, where the *Request latency* corresponds to the average amount of time taken by our *framework* to handle a subscription with computation without considering the time used to process data, and the *Service latency* corresponds to the average amount of time it took each mobile device to process a varied amount of data with the facial recognition Service.

It was possible to observe that the average amount of time to handle a subscription with computation with our *framework*, without considering the amount of time to process data, does not vary significantly, being around 6 seconds in all scenarios. The same can not be stated regarding the *Service latency*, as it shows that it takes in average around 15.8 seconds for a single device to process 30 photos, while 5 photos only take 3.1 seconds to be processed in average. Also, and as observed in Figure 5.4, the facial recognition *Service* is computational intensive being highly dependent on the hardware, since for the Nexus 6 it takes them around 8-7 seconds to process 30 photos, while for the Moto G it takes them 30 seconds to process 30 photos. These greatly disparate values between the Moto

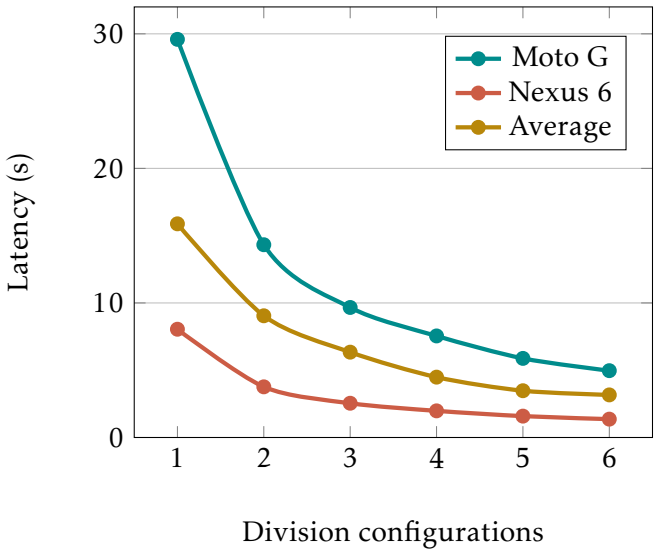


Figure 5.4: *Service latency* experienced in 3 different scenarios where 30 files were processed by *x* mobile devices on an heterogeneous network.

G and the Nexus 6 devices justify the large deviation pattern observed in Figure 5.3a, and because of that, all tests done afterwards and presented in the rest of these chapter were only performed using Nexus 6 devices, as it allowed us to obtain meaningful values and that depended of less variables.

Still regarding Figures 5.3a and 5.3b, in all tested scenarios it was possible to observe a slight variation in *Request latency* values. Although we expected a slight variation in the values observed, since computation wise the whole process of submitting a *service request* and scheduling *tasks* is not very intensive, we also expected a small increase regarding *Request latency* values because more devices would need to interact between each other to conclude a *service request* with success, something that was only noticeable when 6 devices had to process data. The variation observed can be explained by different hardware, specially when the *Scheduler* node is chosen randomly from the cell to which the input *tag* was mapped to, so in Figure 5.3a the different division configuration scenarios present an average of observations where some of these observations had Moto G device as a *Scheduler* node and others had Nexus 6 as a *Scheduler* node. One is able to observe the difference between using one type of device or the other by comparing division configuration scenario 1 of Figure 5.3a with scenario 1 of Figure 5.3b, even tough scenario 1 of Figure 5.3a includes observations where the type of device used changed from observation to observation.

The Figure 5.3b shows the first test performed, where only the first 3 scenarios were tested, as we only had 3 mobile devices to distribute data. Similar to what is observed in Figure 5.3a, the *Request latency* does not vary significantly when 1, 2 and 3 mobile devices are used to process data associated to a single subscription with computation. In the 3 scenarios the average *Request latency* corresponds to 2 seconds. Also similar

to Figure 5.3a, the tests performed on the homogeneous network showed that only the *Service latency* varied significantly. The total latency of a subscription with computation alternates from 9.7 seconds when only one device is computing data, to 4.4 seconds when three devices processed data, resulting in a 54% decrease of the latency of a subscription with computation.

The first test allowed us to demonstrate the scalability of our *framework*, regarding distribution of workload. When we have more devices able to process data related to a single subscription with computation, we observe a decrease in the total latency of that request. We were able to apply facial recognition on 30 pictures, each one with two people in it, taking only 4.4 seconds when 3 devices process data, something that we consider to be a reasonable and positive latency, for a facial recognition *Service*. Also, as we expected, the total latency of a subscription with computation is highly dependent on the *Service latency*, since the *Request latency* does not vary significantly and it is short.

In the second test we had the objective of understanding how does having multiple devices submitting subscriptions with computation one after another affected the latency of a subscription with computation. We increased the number of subscriptions with computation done in each scenario, starting at 1 subscription in the first scenario to a maximum of 18 subscriptions in the fifth scenario. In this test, it is important to note that we modified the parametrizations to only allow processing of 5 files at once, and at most one device could only handle 2 tasks at a time. Also, each one of the subscriptions submitted one after another, was submitted with a 2 seconds time interval between each other.

The Figure 5.5 shows us the results of the second test. The values observed in this Figure correspond to the highest latency value between all the subscriptions with computation submitted in the same time interval one after another. The error bars shown correspond to the standard deviation between different observations of the same scenario.

When only one subscription with computation was submitted it was possible to observe that the maximum latency experienced corresponds to 2.6 seconds, increasing slightly to 3.4 seconds when 18 subscriptions with computation were submitted during the same time interval. This value can be justified partly by the parametrization used, regarding the maximum number of jobs allowed to be handled by a *Scheduler*. Since we only allowed 2 at the same time in each device, and in total, on the fifth scenario each device handled 6 jobs, this may have forced other jobs to wait a small amount of time for their turn. It only influences partly because the subscriptions with computation were submitted one after another, with separations of two seconds. The slight increase observed allows us to conclude that our *framework* is not highly affected by the number of subscriptions with computation submitted one after another, proving its scalability regarding supporting an increase amount of workload, while using the same number of mobile devices.

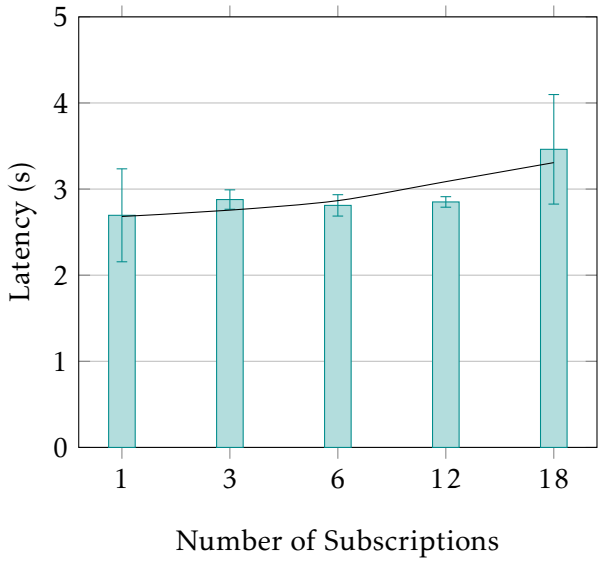


Figure 5.5: Maximum latency of a subscription with computation when several subscriptions with computation are submitted one after another in the network

5.3.2 Publishing with pre-process

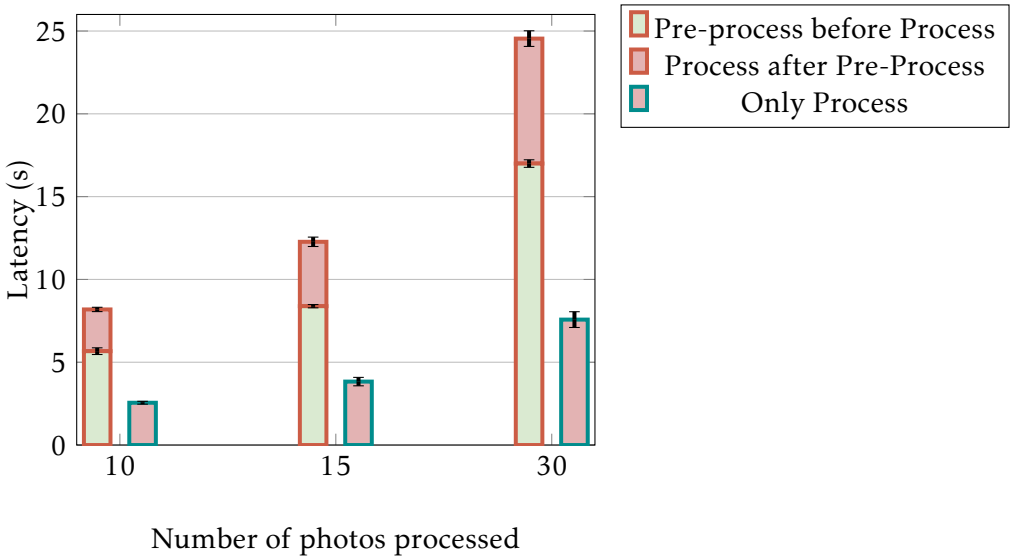


Figure 5.6: Comparison of *Service latency* with and without pre-process

This third test had as objective to show how much the publish with pre-process influences the *Service latency* of a subscription with computation. We tested 6 different scenarios where the number of *DataItems* to be processed by a single device varied from 10 to 15 and finally to 30. In 3 of the scenarios, the files were eagerly pre-processed (all files were published with pre-process). On the other 3 scenarios, the files were lazily pre-processed, meaning that they were only pre-processed right before the actual process of data.

In Figure 5.6 we observe the results of the third test. In this figure the first bars correspond to the *Service latency* when a pre-process was required to be done before the process operation. The other bars correspond to *Service latencies* in scenarios where each file was published with pre-process, therefore no pre-process was required before actually processing.

As it is possible to observe, the latency of the process operation did not varied significantly between the 2 types of scenarios. What actually influenced the *Service latency* of a subscription with computation, was the pre-process required to be done on each object that was going to be processed, before processing them. In all scenarios where a lazy approach was followed, the amount of time spent on pre-processing every file at once before processing them, increased the *Service latency* by 3 times. On one hand this proves that publishing with pre-process can potentially be an alternative if the user intends faster response times, although, pre-processing every *DataItem* published can be a waste of computational resources if those *DataItems* will never be processed. On the other hand, by lazily pre-processing, the pre-process will also only be done once, but it will guarantee that no computational resources are wasted, since only *DataItems* that are going to be processed at least once are pre-processed. This last approach has its drawbacks, the first time a dataset is processed, the *Service latency* of the determined request will be longer, which means that it may not be the best approach if the data to be processed is processed only once. Both of the approaches have their advantages and disadvantages depending on what context they intend to be used, it will be the responsibility of the application developer to choose the one that best suits his/her application.

5.3.3 Battery cost

The fourth and the fifth test allowed us to measure the energy cost in Joules of each one of our operations. This is important as the context of our *framework* understands that the mobile devices that form our network have limited resources, being one of those the amount of battery available. In both tests we changed the parametrization to only allow 5 *DataItems* to be processed at once.

In the fourth test we intended to observe the amount of energy spent in Joules when a **publish with pre-process** is performed and a **subscription with computation** is submitted. Also associated with each one of these operations there are other mechanisms that are triggered in other mobile devices to handle the operations performed, therefore we also require testing the energy cost of the corresponding handling of the operations performed.

The results of the fourth test can be observed in Figure 5.7. As it is possible to observe the **publish with pre-process** is the operation that spends the most energy, around 6.64 Joules. This is mostly caused by the fact that this operation is done on a very short period of time, around 1.5 seconds but it performs a computational intensive operation of extracting faces of an image. The mechanism to handle a publish is granted by Thyme,

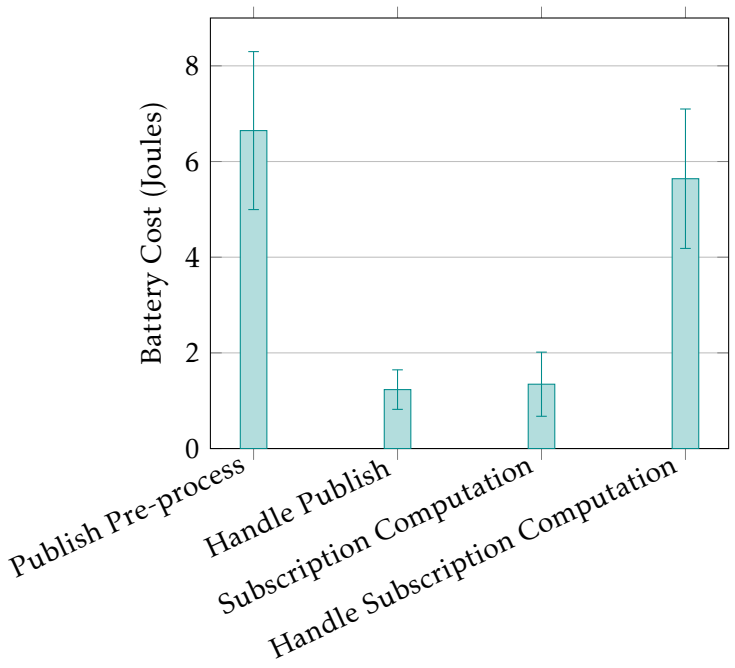


Figure 5.7: Battery cost in Joules when performing single operations

and it is only displayed with the objective of offering the full view of the energy cost of a publish with pre-process.

Regarding the **subscription with computation** operation and its handling mechanism, as we expected, handling a subscription with computation is more energy costly than submitting it and receiving notifications of its results. Although handling a subscription with computation implies processing *DataItems*, which in this case is more computational intensive than just pre-processing them, the time it takes to process *DataItems* is larger than publishing them with pre-processing, therefore handling a subscription with computation is still inferior regarding energy cost.

As the values did not offered enough information to help us understand the actual cost of using an application implemented with Oregano, we performed a fifth test. In this test we did multiple sequential operations to analyse how much energy is spent when a mobile device performs each one of the operations for 1 minute straight. The operations were: i) a regular publish for comparison purposes; ii) a publish with pre-process; and iii) a subscription with computation. Since this only tested the battery cost of submitting a subscription with computation, we also tested in two different devices the cost of handling the scheduling of tasks and the cost of processing those tasks.

The results of the fifth test can be observed in Figure 5.8. We started by testing the amount of energy spent by a Nexus 6 when in stand by only connected to a Wi-Fi hotspot for a minute. This value corresponds to 40 Joules. It is also possible to observe that launching our application and leaving it in stand by doing nothing besides exchanging some messages, required by Thyme to maintain the network, only spends 20 Joules more than when the device was in stand by. In Figure 5.8, Delta corresponds to the extra

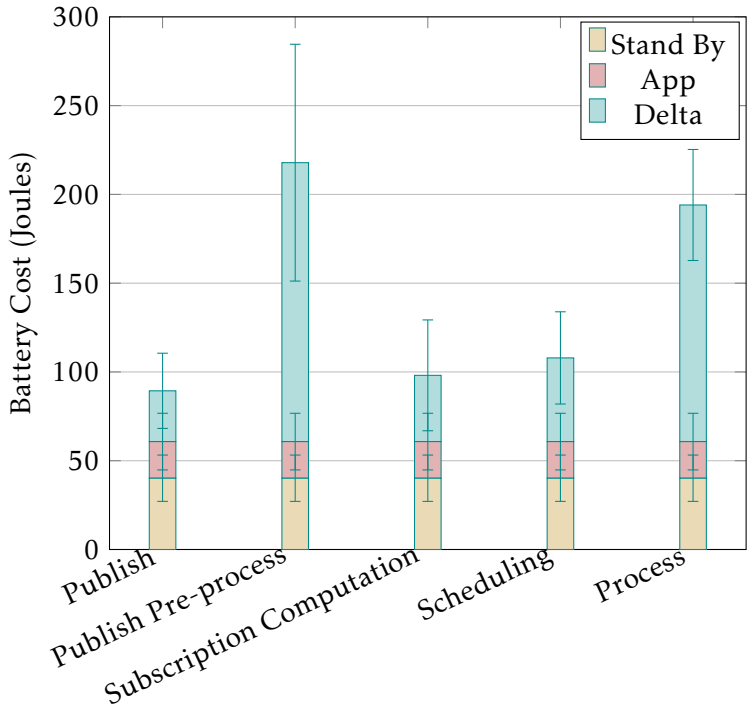


Figure 5.8: Battery cost in Joules when performing operations for one minute

amount of battery spent without considering the implicit cost of being in stand by and having the application opened.

Submitting various subscriptions with computation for a whole minute spent around 98 Joules, a delta of 38 Joules, which we consider to be a reasonable amount of energy spent considering that we submitted 15 subscriptions with computation and that 98 Joules correspond to about 0.22% of the battery of a Nexus 6. As we expected, when a device had to schedule tasks, a slight higher value of battery cost was observed when compared with submitting a subscription with computation, since the process of scheduling is not very computational intensive. When scheduling for a whole minute, 107 Joules were spent, a delta of only 47 Joules, which is a reasonable value considering that 107 Joules corresponds to 0.24% of the battery of a Nexus 6. When processing for one minute straight, as expected, we observed the second highest battery cost value, 194 Joules, a delta of 133 Joules. This value corresponds to about 0.44% of the battery of a Nexus 6, which we consider to be a positive value considering that a facial recognition algorithm was applied to 75 photos.

It is important to understand that, on ours system's network, a mobile device can play different roles during a time interval, which means that although we presented each one of the three battery cost values separately, Subscription Computation, Scheduling and Process, the whole process of a subscription with computation, on 3 different mobile devices, spent a total of 400 Joules. If we imagine a scenario where a mobile device is used for two hours and a half (150 minutes), constantly being used as a *Client*, or a

Scheduler or *Computing*, an average of 133 Joules (0.3% of the battery of a Nexus 6) would potentially be spent per minute, which after 150 minutes would correspond to about 45% of a Nexus 6. This value is quite reasonable, considering the type and duration of consecutive usage.

We can also observe that the publish with pre-process spent a total of 220 Joules, a delta of 157 Joules, which is a reasonable value considering that 220 Joules correspond to about 0.5% of the battery of a Nexus 6 and we were able to publish with pre-process 40 photos. Despite this, we can conclude that publishing with pre-process must be used wisely, on one hand, it allows users to experience reduced latency values when subscribing with computation, on the other hand it spends energy that is not justified if the published object is not processed eventually, opening space to other options, like lazy pre-processing.

5.4 Simulator Tests

For the simulated environment we aimed to observe the behaviour of our *framework* on controlled scenarios of churn, allowing us to test the consequences of a node leaving the network in specific scenarios. We also aimed to observe how the number of nodes in a cell influences the workload in our system in terms of messages exchanged. In order to test these behaviours we performed four tests. In the first test we varied the number of subscriptions with computation submitted and observed the number of messages sent/received by all the nodes in our system. In the second test we used different scenarios varying the number of nodes in a cell. In the third and fourth test we used different scenarios varying the percentage of nodes that left a specific cell and the influence of this in the success of a subscription with computation.

5.4.1 Number of messages

In the first test we submitted 1, 10, 20, 50 and 100 subscriptions with computation with the objective of showing the different types and number of messages that are necessary to be exchanged between mobile devices in our system's network, in order for a successful conclusion of a subscription with computation. In this test, each node only submitted one subscription with computation, starting with only 1 node created to a maximum of 100 nodes. For a single computation request a total of at least 9 messages are exchanged between devices in our system's network: 1 Subscription Computation, 4 Acknowledges, 1 Task, 2 Task Status and 1 Result Notification. For our tests, we considered that only one device had data to be processed, therefore only 1 task was sent. Also, as the *Service latency* was small, only 2 task status were sent. Additionally we decided that the *Client* would be notified about its results by a Result Notification message instead of using a publish mechanism, which would increase the number of messages sent, considering that

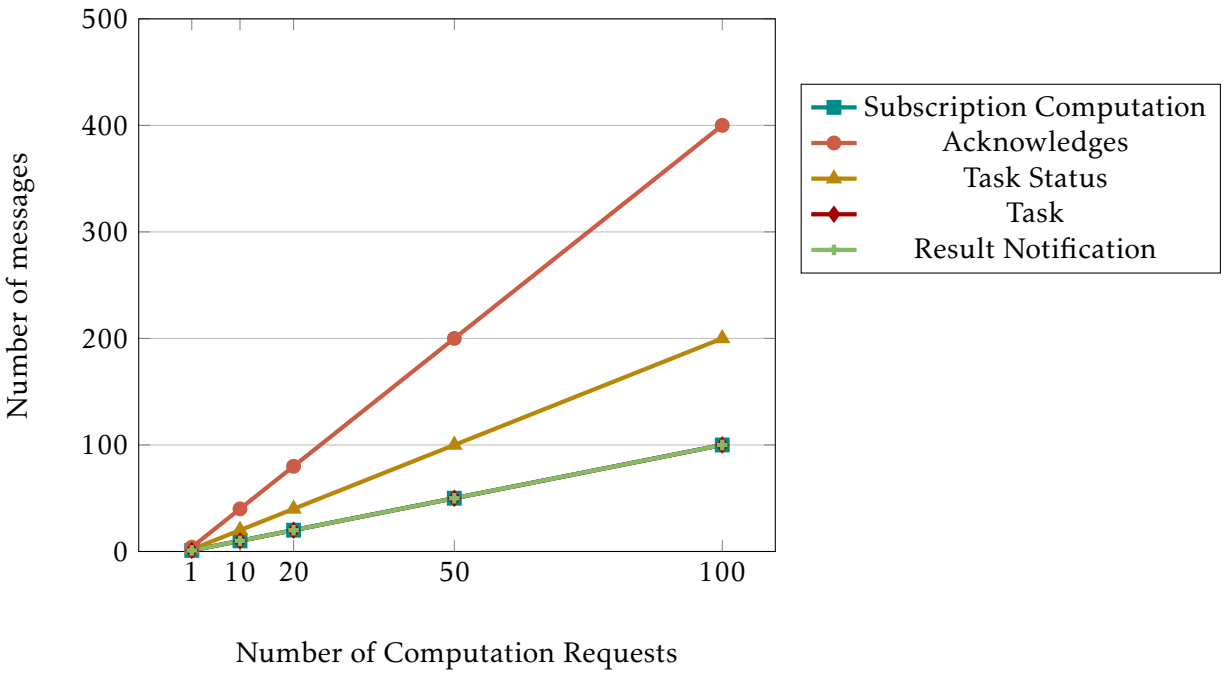


Figure 5.9: Number of messages sent/received per number of Computation requests submitted

each result item is published individually triggering multiple notifications on the *Client's* device.

The results of the first test can be observed in Figure 5.9. It is important to note that, although it is not visible, the number of Result Notification messages is equal to the number of Task and Subscription Computation messages. Based on this Figure, it is possible to observe that the total number of messages exchanged for handling an increasing number of computation requests grows linearly. This proves that the number of computation requests submitted during a specific time interval will not influence the number of messages exchanged negatively, by sending/receiving more messages than the ones necessary to conclude a subscription with computation successfully. This will always be true if the premisses presented in the paragraph above remain true and we consider that there are no flaws either in terms of messages not received or mobile devices that may have left the network while a subscription with computation was happening and they were key elements necessary for its completion. Although, in scenarios where failures occur, the number of messages exchanged increases because there are messages that need to be resent to other nodes, the increase experienced does not depend on the number of computation requests submitted.

5.4.2 Number of nodes in a cell

In the second test, we varied the number of *Scheduler* nodes in a cell with the objective of observing how the number of *Scheduler* nodes in a cell influenced the number of messages

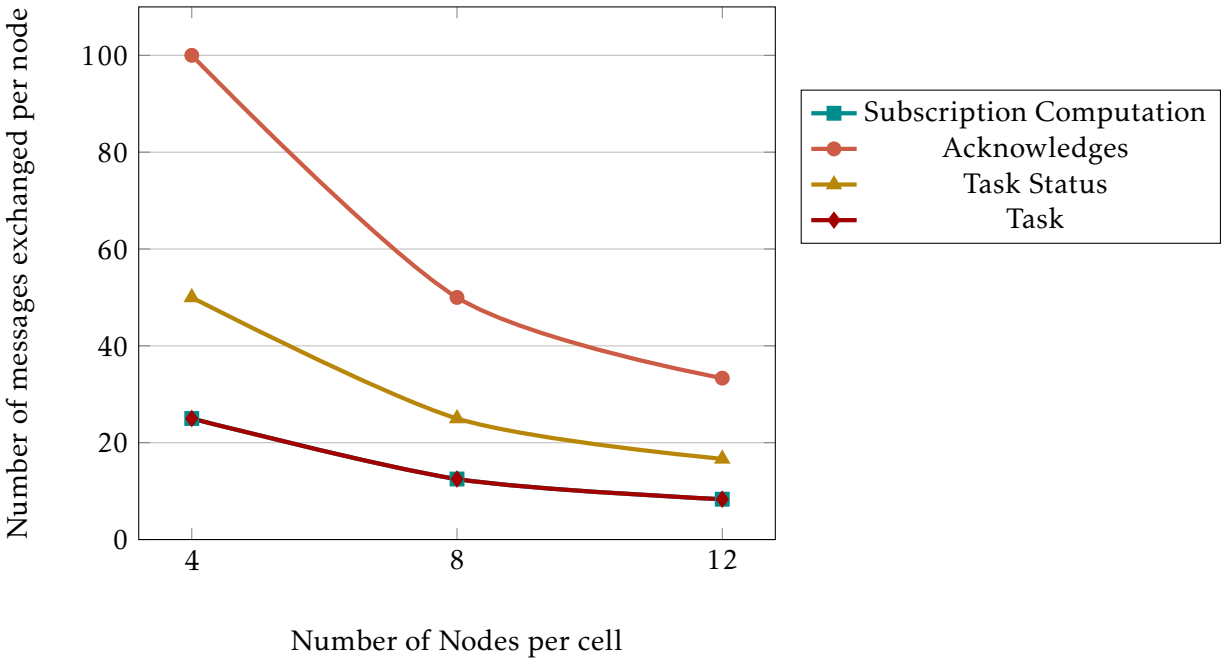


Figure 5.10: Number of messages sent/received per *Scheduler* node in a cell

each one of the nodes received/issued. We tested 3 scenarios where 100 subscriptions with computation were submitted with a specified input *tag*, and the number of nodes in the cell to which the input *tag* was mapped varied between 4, 8 and 12 nodes. In all scenarios, the world created had 4 cells and 18 nodes in it. One of the cells had a node playing the role of a *Client*; Another cell had one node which was only used to handle possible result tag subscriptions mapped to the cell; In the other two cells, the number of nodes varied, where in the *Scheduler* cell there were 4, 8 and 12 nodes, the remaining nodes occupied the *Computing* cell. It is also important to mention in order to understand the results more clearly, that with a subscription with computation are associated different types and numbers of messages which are exchanged between nodes in order for a successful conclusion of the request. Regarding the *Scheduler* nodes, for a single computation request: 1 Subscription Computation is received, 2 acknowledges are sent, 2 acknowledges are received, at least 1 task is sent and at least 2 tasks status messages are received. For our tests, we considered that only one device had data to be processed, therefore only 1 task was sent. We also considered that the *Service latency* was short, therefore only 2 task status were sent.

The results of the second test can be observed in Figure 5.10. It is important to state that, although it is not visible in this Figure, the number of Subscription Computation messages is equal to the number of Task messages for 4, 8 and 12 nodes. Based on this Figure it is possible to observe a decrease in the number of messages exchanged by each node in a cell to which an input *tag* of a subscription with computation was mapped to. This decrease results from workload sharing, since there are more nodes in the cell which are available to handle computation requests. The number of Subscription Computation

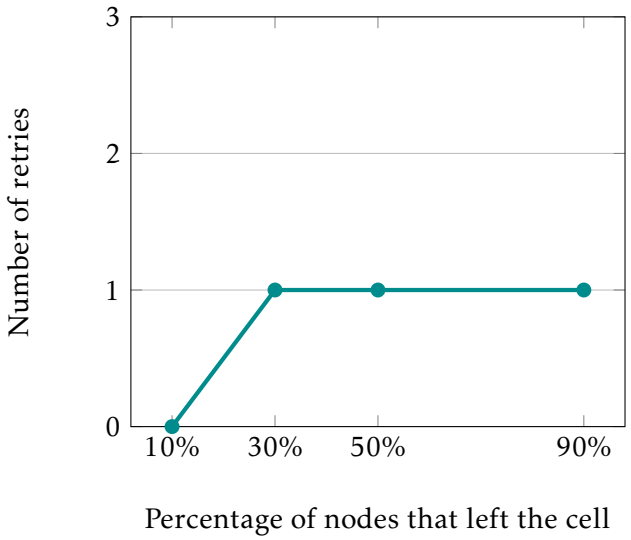


Figure 5.11: Number of retries done when 20 sequential continues of subscription with computation were performed

messages received by each node in the cell starts by being 25 when there are only four nodes in a cell, and ends up being 8-9 messages per node in a cell. As we expected, having more devices in a *Scheduler* cell helps the amount of subscriptions with computation each node has to handle, consequently reducing the amount of messages received/issued by each node.

5.4.3 Churn

Regarding the third test, we varied the percentage of *Scheduler* nodes that randomly left a cell. We tested different scenarios that ranged from 0% of the nodes leaving the cell to a maximum of 90% of nodes leaving the cell. If nodes leave a cell to which an input *tag* of a previous subscription with computation was mapped to, this may cause the impossibility of continuing a previous computation request. The test consisted of trying to perform 20 continuations of subscriptions with computation until an error stopped the subscription with computation, while observing the number of retries done before the subscription with computation failed. In this test, the world created had 4 cells and 16 nodes in it. One of the cells had a node playing the role of a *Client*; Another cell had one node which was only used to handle possible result tag subscriptions mapped to the cell; The *Scheduler* cell had 10 nodes and the *Computing* cell had 4 nodes.

The third test showed us that it was possible to request 20 continuations of a subscription with computation before an error stopped the computation request, when 0% to 90% of the nodes left the cell. This did not allowed us to fully understand the impact of churn in our system. Therefore, we decided to observe the number of retries done by the *Client* when trying to perform a continuation of a subscription with computation. These retries happen when a *Client* node tries to perform a continue, sending a message to a random

node in the *Scheduler* cell to which the input *tag* of the request was mapped to, and this message is not acknowledged.

The results of the third test can be observed in Figure 5.11. This figure shows us that 0 retries were done when 10% of the nodes left the cell and only one retry was needed to actually perform 20 continuations of a computation request, when 30%, 50% and 90% of the nodes left the cell. This shows us that our *framework* is able handle churn experienced between the *Client* and the *Scheduler* nodes, without being highly negatively influenced by churn. Our churn support mechanism is limited to a maximum of three attempts to request the continuation of a computation request to any node in the determined cell. In our test, only one continuation manifested problems, being that only one attempt of three possible attempts was used, highlighting Thyme's ability to detect which nodes are still active and which nodes are not.

Regarding the fourth test, we decided to remove devices from a cell, which for test purposes we identify as the cell in which all nodes intended to process data are located. Similar to the third test, we removed nodes randomly and gradually starting by removing 0% of the nodes, to a maximum of 90% of nodes removed. The test consisted of performing 50 subscriptions with computation and observe the percentage of operations that succeeded. We also observed the number of retries performed while scheduling *tasks*. This test allowed us to analyse our support to churn between the *Scheduler* and the *Computing* components. It is also important to state that, for this test a few premisses were taken in consideration: i) we chose to support a maximum of 10 failed tasks before choosing more *DataItems* to be processed or if there are no more *DataItems* before sending an error message to the *Client*; ii) only one node published all the data to be processed; iii) all the publishes were done with active replication enabled, which means that all files published were replicated to every node in the cell where the node that published the files is located; and iv) we used a default implementation of a *Tasks Scheduler* where the addresses of the nodes able to process *DataItems* were chosen randomly.

In this test, the world created had 4 cells and 16 nodes in it. One of the cells had a node playing the role of a *Client*; Another cell had one node which was only used to handle possible result tag subscriptions mapped to the cell; The *Scheduler* cell had 4 nodes and the *Computing* cell had 10 nodes.

The fourth test showed us that Oregano is able to perform 50 subscriptions with computation with success, while 0% to 90% of the nodes in the *Computing* cell were being removed randomly. For this test, we considered that a subscription with computation succeeded, when the maximum amount of files specified in the parametrization were processed, allowing for a continuation of a subscription with computation. Similar to the third test, we were not able to understand clearly how did churn influenced Oregano, therefore we decided to test the number of retries done, when scheduling *tasks*. These retries happen when a *Scheduler* node sends *tasks* to be handled by *Computing* nodes, but no acknowledges are received, an error occurs while handling a *task*, or the *heartbeat* messages stop being received because the *Computing* node left the network.

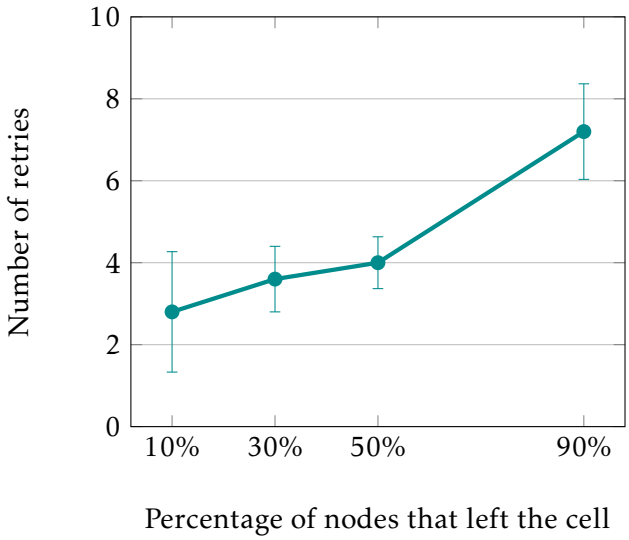


Figure 5.12: Number of retries/(tasks schedules) done when 50 subscriptions with computation were performed

Figure 5.12 shows the results of the fourth test. The figure shows us that an average of 2.8 retries were performed when 10% of the nodes left the cell, having increased as more nodes were removed from the network, to an average of 7.2 retries when 90% of the nodes left the cell. These values are due to two main reasons:

- Although Thyme is able to detect which nodes are alive in the network, it does not update the data location in each meta data, which means that when OREGANO, specifically the *Scheduler* retrieves meta data to schedule *tasks*, these meta data can have addresses of nodes that already left the network, which was the case in our observations. By locally saving the address of each node that failed to conclude a *task*, the *Scheduler* was able to avoid scheduling and rescheduling *tasks* to devices that were not available in our system. The main problem of this approach is its local aspect, since only the *Scheduler* device which realized that a certain node is no longer available, is aware that it should not choose the said device to process *tasks*.

- The nodes that left after acknowledge or before acknowledging, also contributed for the values observed.

These values allow us to conclude that our *framework* is able to deal with churn experienced when an interaction between the *Scheduler* and *Computing* components occurs, because all operations were completed with success. Also, taking in consideration that between the 10% scenario and the 90% scenario only a 4 retries increase was observed, allow us to conclude that OREGANO is not highly affected by churn, being able to recover from failures without exchanging too many messages. Nonetheless, there is a necessity of improving our churn support mechanisms, although a change to Thyme would probably

be required to deal with the meta data limitation experienced. Also, it is important to understand that, independently of the results, even if we improved our churn support mechanisms or used a different one, churn would still undermine our system in situations where the device who published data left the network before there was enough time to replicate that data.

Although in both tests we were able to handle churn, there are still some scenarios that our *framework* is unable deal with. Specifically when after a *Scheduler* node acknowledges the *Client* device that its request is being handled, and both the *Scheduler* device and the device/s to which the *task/s* was/were submitted, leaves the network. This will cause the *Client* node to wait endlessly for the results of the computation or until the end user cancels the request, not allowing further continues. In order to observe Oregano's behaviour when this scenario happened, we performed a fifth test where we divided per *Scheduler* and *Computing* cell an equal amount of nodes, and submitted subscriptions with computation until the scenario described occurred, while randomly and gradually removing each node of the two cells. In this test, the world created had 4 cells and 22 nodes in it. One of the cells had a node playing the role of a *Client*; Another cell had one node which was only used to handle possible result tag subscriptions mapped to the cell; The *Scheduler* cell had 10 nodes and the *Computing* cell had 10 nodes.

We were able to observe that, in average, 1 in each 98 subscriptions with computation submitted experienced the situation being tested. The deviation pattern measured was 56 subscriptions with computation, something that show us the randomness of this situation. Taking this in consideration and the number of times the situation in question occurs, the values observed are reasonable. Nonetheless, we think that there is still the necessity of improving our churn support, by devising mechanisms more dependent on our layer, instead of being partly dependent on Thyme's ability to detect and update information regarding devices liveness.

In both third and fourth churn tests, the values observed were positive and corresponded to what we expected, it is relevant to point that our *framework* is still influenced negatively by churn, specifically in regards to the latency experienced and the number of messages exchanged. In terms of latency, our system detects that a failure has happened if specific messages do not arrive to their supposed location during a determined amount of time. Therefore, usually when an error occurs, the specified time out time has already passed, increasing significantly the latency of a computation request. Since we allow the parametrization of these timer/time out values, the amount of time spent to detect a failure can be controlled by the user of our *framework*, allowing her/him to potentially reduce the extra latency in case of *services requests* that may experience some failures.

In regards to the number of messages exchanged, for every new attempt of rescheduling a *task* or resending a job to another node, a minimum of two messages are sent, one to stop a previous failed *task* or job and other to send a *task* or a job. Two other messages can be received, which are acknowledges stating that a *task* or job was received and it is going to be handled. These retries will increase the overall number of messages sent/received

by some nodes.

5.5 Summary

In this chapter, we addressed the evaluation done on our *framework*, Oregano. We tested our system on a real environment using Android mobile devices, and on a simulated environment with a simulator that shares a very similar layer architecture with our Android *framework*.

On the real environment, we evaluated different metrics that allowed us to prove the scalability of Oregano in terms of workload distribution, exhibiting a *Speedup* of 2.26 when 3 mobile devices helped processing data associated with the same request. Oregano also proved to support scalability in terms of not being highly affected, latency wise, when handling multiple requests at the same time. Finally, on the real environment we were also able to present reasonable battery cost values, when a mobile device takes part of Oregano's network.

In terms of the simulated environment, we evaluated different metrics that allowed us to prove the linearity, related to the number of messages exchanged and the increase in the number of computation requests submitted at the same time. Furthermore, we also tested different churn scenarios that enabled us to show our system's ability to handle churn, being able to conclude almost every operation successfully, with few retries needed.

In the next chapter we will present a conclusion of our dissertation, also addressing some of the future work that could be considered for our *framework*.

Chapter 6

Conclusion

In this chapter we will present some conclusions regarding our system, what was done and some of the limitations that exist. We will also present some suggestions that would allow someone to potentially address these limitations in the future.

6.1 Conclusion

In this thesis we presented a distributed computing *framework* capable of processing batches and streams of data, that are being generated by a network formed exclusively by mobile devices, without resorting to Internet services. With our framework, we aim to offer application developers the ability to devise Android applications that are able to function on different scenarios where the network may be constrained or unavailable, limiting the access to Internet services, by using other mobile devices in the vicinity to process data. At the same time, the proximity of the mobile devices to each other when compared to Cloud services, allows us to offer reduced latency times.

What distinguishes Oregano from other related work is our approach on sending data to be processed and the ability to compute files that may have been shared or are still being produced and shared in real time. In our *framework*, only mobile devices that have the data to be processed are the ones that actually process that data, avoiding sending data over the network every time computation is required by some mobile device. The ability to share and store data is offered by Thyme, a time-aware storage and dissemination system with publish and subscribe mechanisms.

Based on all aspects presented above, Oregano offers two different operations, a publish with pre-process and a subscription with computation. The publish with pre-process allows to publish data with a specific *tag* followed by the pre-processing of the object being published. The subscription with computation allows to process data that

has been or will still be published with a specified *tag*.

Both pre-process and process are defined by an application developer, as we devised a simple and generic API that allows Oregano to support the processing of a wide variety of data types.

Based on the experimental evaluation done on the real environment, we were able to state that our *framework* supports scalability, in terms of benefiting significantly with the usage of several devices to handle computation, and at the same time, our evaluation also showed us that by increasing the number of subscriptions with computation submitted in the network at the same time does not necessarily mean significant increases in the latency time of a subscription with computation.

The experimental evaluation done on the simulator showed us that our system is able to handle the entrance and exit of devices of the network, i.e churn, to some extent, although there are still some extra features/mechanisms that, if implemented, could allow a better churn support.

Overall we were able to address the problems proposed to be solved in this thesis, as we devised a functional Android prototype of a distributed computing *framework* capable of processing batches and data streams, without resorting to Internet services.

6.2 Future Work

As stated earlier, we were able to develop a functional Android prototype of our *framework*, although there are still some issues that could be improved in the future.

Related to churn, our solution presents some limitations in certain situations, specifically between the *Client* and *Scheduler* node, since the support to churn between the interaction of the two components consists on a simple retry of a failed continuation a specified maximum amount of times. A direction of future work could consist of a consistent distributed queue located in each cell, allowing multiple mobile devices in a cell to get jobs from this queue. This could also be used for the interaction between the *Scheduler* component and the *Computing* component, being that in this case, instead of jobs, the queue would be composed by tasks. It is important to note that there is already work done on this direction and that Oregano was designed considering this premiss.

Some other direction of future work that could be followed consists of the ability to use infrastructures to help in the computation of data, specifically in cases where the computation is highly intensive consuming lots of resources. This would be a feature to be used alongside the work we already developed.

Some other possible future work would consist in reducing the amount of the repetitive computations that currently can exist in our system. In cases where multiple mobile devices subscribe with computation to the same tag at the same time, a possibility of future work could be to modify the *Scheduler* in order to coordinate two or more subscription with computation requests to the same tag, avoiding sending tasks which trigger the processing on the same objects twice.

Bibliography

[1] F. Araújo, L. E. T. Rodrigues, J. Kaiser, C. Liu, and C. Mitidieri. "CHR: A Distributed Hash Table for Wireless Ad Hoc Networks". In: *25th International Conference on Distributed Computing Systems Workshops (ICDCS 2005 Workshops), 6-10 June 2005, Columbus, OH, USA*. IEEE Computer Society, 2005, pp. 407–413. ISBN: 0-7695-2328-5. DOI: 10.1109/ICDCSW.2005.48. URL: https://doi.org/10.1109/ICDCSW.2005.48.

[2] F. Bonomi, R. A. Milito, J. Zhu, and S. Addepalli. "Fog computing and its role in the internet of things". In: *Proceedings of the first edition of the MCC workshop on Mobile cloud computing, MCC@SIGCOMM 2012, Helsinki, Finland, August 17, 2012*. Ed. by M. Gerla and D. Huang. ACM, 2012, pp. 13–16. ISBN: 978-1-4503-1519-7. DOI: 10.1145/2342509.2342513.

[3] F. Cerqueira, J. A. Silva, J. M. Lourenço, and H. Paulino. "Towards a Persistent Publish/Subscribe System for Networks of Mobile Devices". In: *Proceedings of the 2Nd Workshop on Middleware for Edge Clouds & Cloudlets*. MECC '17. Las Vegas, Nevada: ACM, 2017, 2:1–2:6. ISBN: 978-1-4503-5171-3. DOI: 10.1145/3152360.3152362. URL: http://doi.acm.org/10.1145/3152360.3152362.

[4] F. Cerqueira, J. A. Silva, J. M. Lourenço, and H. Paulino. "Um Sistema Publicador/-Subscritor com Persistência de Dados para Redes de Dispositivos Móveis". In: *Comunicações do INForum 2017*. 2017.

[5] Cisco. "Fog Computing and the Internet of Things: Extend the Cloud to Where the Things Are". In: (2015). URL: https://www.cisco.com/c/dam/en_us/solutions/trends/iot/docs/computing-overview.pdf.

[6] Cisco. *Cisco VNI Mobile Forecast (2016 – 2021)*. 2017. URL: http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/mobile-white-paper-c11-520862.html.

[7] 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*. IEEE Computer Society, 2013, pp. 209–215. ISBN: 978-1-4799-2481-3. DOI: 10.1109/UIC-ATC.2013.94.

[8] N. Fernando, S. W. Loke, and J. W. Rahayu. "Mobile cloud computing: A survey". In: *Future Generation Comp. Syst.* 29.1 (2013), pp. 84–106. DOI: 10.1016/j.future.2012.05.023.

[9] N. Fernando, S. W. Loke, and W. Rahayu. "Computing with Nearby Mobile Devices: a Work Sharing Algorithm for Mobile Edge-Clouds". In: (2016). URL: https://www.researchgate.net/profile/Seng_Loke/publication/301719756_Computing_with_Nearby_Mobile_Devices_a_Work_Sharing_Algorithm_for_Mobile_Edge-Clouds/links/572f0fb408ae744151901f4c.pdf.

[10] D. Georgakopoulos, P. P. Jayaraman, M. Fazia, M. Villari, and R. Ranjan. "Internet of Things and Edge Cloud Computing Roadmap for Manufacturing". In: *IEEE Cloud Computing* 3.4 (2016), pp. 66–73. DOI: 10.1109/MCC.2016.91.

[11] L. M. V. Gonzalez and L. Rodero-Merino. "Finding your Way in the Fog: Towards a Comprehensive Definition of Fog Computing". In: *Computer Communication Review* 44.5 (2014), pp. 27–32. DOI: 10.1145/2677046.2677052.

[12] Google. *Protocol Buffers. Developer Guide*. https://developers.google.com/protocol-buffers/docs/overview.

[13] Google. *Use Java 8 language features*. https://developer.android.com/guide/platform/j8-jack.html.

[14] K. Habak, M. H. Ammar, K. A. Harras, and E. W. Zegura. "Femto Clouds: Leveraging Mobile Devices to Provide Cloud Service at the Edge". In: *8th IEEE International Conference on Cloud Computing, CLOUD 2015, New York City, NY, USA, June 27 - July 2, 2015*. IEEE, 2015, pp. 9–16. ISBN: 978-1-4673-7287-9. DOI: 10.1109/CLOUD.2015.12.

[15] M. Halpern, Y. Zhu, and V. J. Reddi. "Mobile CPU's rise to power: Quantifying the impact of generational mobile CPU design trends on performance, energy, and user satisfaction". In: *2016 IEEE International Symposium on High Performance Computer Architecture, HPCA 2016, Barcelona, Spain, March 12-16, 2016*. IEEE Computer Society, 2016, pp. 64–76. ISBN: 978-1-4673-9211-2. DOI: 10.1109/HPCA.2016.7446054.

[16] Y. Jararweh, L. A. Tawalbeh, F. Ababneh, and F. Dosari. "Resource Efficient Mobile Computing Using Cloudlet Infrastructure". In: *IEEE 9th International Conference on Mobile Ad-hoc and Sensor Networks, Dalian, MSN 2013, China, December 11-13, 2013*. IEEE Computer Society, 2013, pp. 373–377. ISBN: 978-0-7695-5159-3. DOI: 10.1109/MSN.2013.75.

[17] *JavaCV*. https://github.com/bytedeco/javacv. Retrieved 1 August 2017.

[18] G. I. Klas. "About Mobile Edge Cloud Computing - An Introduction". In: (2015). URL: http://yucianga.info/wp-content/uploads/2015/11/15-11-08-About-Mobile-Edge-Cloud-Computing-%E2%80%93-An-Introduction-v1.pdf.

[19] T. H. Luan, L. Gao, Z. Li, Y. Xiang, and L. Sun. "Fog Computing: Focusing on Mobile Users at the Edge". In: *CoRR* abs/1502.01815 (2015).

[20] E. Miluzzo, R. Cáceres, and Y.-F. Chen. "Vision: mClouds-computing on clouds of mobile devices". In: *Proceedings of the third ACM workshop on Mobile cloud computing and services*. ACM. 2012, pp. 9–14.

[21] T. Nishio, R. Shinkuma, T. Takahashi, and N. B. Mandayam. "Service-oriented heterogeneous resource sharing for optimizing service latency in mobile cloud". In: *Proceedings of the first international workshop on Mobile cloud computing & networking*. ACM. 2013, pp. 19–26.

[22] S. Pal. "Extending Mobile Cloud Platforms Using Opportunistic Networks: Survey, Classification and Open Issues". In: *J. UCS* 21.12 (2015), pp. 1594–1634.

[23] D. Remédios, A. Teófilo, H. Paulino, and J. Lourenço. "Mobile Device-to-Device Distributed Computing Using Data Sets". In: *Proceedings of the 12th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services, MobiQuitous 2015, Coimbra, Portugal, July 22-24, 2015*. Ed. by P. Zhang, J. S. Silva, N. Lane, F. Boavida, and A. Rodrigues. ICST / ACM, 2015, pp. 297–298. ISBN: 978-1-63190-072-3. DOI: 10.4108/eai.22-7-2015.2260273.

[24] P. Sanches, A. Teófilo, F. Cerqueira, J. A. Silva, and H. Paulino. "Computação Distribuída em Redes Formadas por Dispositivos Móveis". In: *INFORUM 2017 - Atas do Nono Simpósio de Informática*. Ed. by I. 2017. INFORUM. INFORUM, Oct. 2017, pp. 185–195. ISBN: 978-972-789-522-9. URL: http://inforum.org.pt/INForum2017/docs/atas-do-inforum2017.

[25] C. Shi, V. Lakafosis, M. H. Ammar, and E. W. Zegura. "Serendipity: enabling remote computing among intermittently connected mobile devices". In: *The Thirteenth ACM International Symposium on Mobile Ad Hoc Networking and Computing, MobiHoc '12, Hilton Head, SC, USA, June 11-14, 2012*. ACM, 2012, pp. 145–154. ISBN: 978-1-4503-1281-3. DOI: 10.1145/2248371.2248394.

[26] D. F.P. M. da Silva. "P3-Mobile Parallel Peer-to-Peer computing on mobile devices". MA thesis. Faculdade de Ciências da Universidade do Porto Departamento de Ciência de Computadores, 2016.

[27] J. A. Silva, H. Paulino, J. M. Lourenço, J. Leitão, and N. Preguiça. "I Know What You Did Last Summer: Time-Aware Publish/Subscribe for Networks of Mobile Devices". In: *CoRR* abs/1801.00297 (2017). URL: https://arxiv.org/abs/1801.00297.

[28] I. Stojmenovic and S. Wen. "The Fog Computing Paradigm: Scenarios and Security Issues". In: *Proceedings of the 2014 Federated Conference on Computer Science and Information Systems, Warsaw, Poland, September 7-10, 2014*. Ed. by M. Ganzha, L. A. Maciaszek, and M. Paprzycki. 2014, pp. 1–8. DOI: 10.15439/2014F503.

[29] N. Sumi, A. Baba, and V. G. Moshnyaga. "Effect of computation offload on performance and energy consumption of mobile face recognition". In: *2014 IEEE Workshop on Signal Processing Systems, SiPS 2014, Belfast, United Kingdom, October 20-22, 2014*. IEEE, 2014, pp. 19–25. ISBN: 978-14799-6588-5. DOI: 10.1109/SiPS.2014.6986056.

[30] C. L. V. Teo. "Hyrax: Crowdsourcing Mobile Devices to Develop Proximity-Based Mobile Clouds". MA thesis. School of Computer Science Computer Science Department Carnegie Mellon University Pittsburgh, PA 15213, 2012.

[31] The Apache Software Foundation. *Apache Hadoop*. URL: http://hadoop.apache.org/.

[32] The Apache Software Foundation. *Apache Kafka A distributed streaming platform*. URL: https://kafka.apache.org/.

[33] The Apache Software Foundation. *Apache Spark*. URL: http://spark.apache.org/.

[34] The Apache Software Foundation. *Apache Storm*. URL: http://storm.apache.org/.

[35] The Apache Software Foundation. *HDFS Users Guide*. URL: https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsUserGuide.html.

[36] The Apache Software Foundation. *Introduction to Apache Flink*. URL: https://flink.apache.org/introduction.html#flink-and-other-frameworks.

[37] The Apache Software Foundation. *MapReduce Tutorial*. URL: https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html.

[38] R.-G. Urma. *Processing Data with Java SE 8 Streams, Part 1*. http://www.oracle.com/technetwork/articles/java/ma14-java-se-8-streams-2177646.html. Accessed: 2017-06-29.

[39] T. Verbelen, P. Simoens, F. De Turck, and B. Dhoedt. "Cloudlets: Bringing the Cloud to the Mobile User". In: *Proceedings of the Third ACM Workshop on Mobile Cloud Computing and Services*. MCS '12. Low Wood Bay, Lake District, UK: ACM, 2012, pp. 29–36. DOI: 10.1145/2307849.2307858.

[40] L. Yang, J. Cao, S. Tang, T. Li, and A. T. S. Chan. "A Framework for Partitioning and Execution of Data Stream Applications in Mobile Cloud Computing". In: *2012 IEEE Fifth International Conference on Cloud Computing, Honolulu, HI, USA, June 24-29, 2012*. Ed. by R. Chang. IEEE Computer Society, 2012, pp. 794–802. ISBN: 978-1-4673-2892-0. DOI: 10.1109/CLOUD.2012.97.

[41] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing". In: *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*. Ed. by S. D. Gribble and D. Katabi. USENIX Association, 2012, pp. 15–28.

[42] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. "Discretized streams: fault-tolerant streaming computation at scale". In: *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*. Ed. by M. Kaminsky and M. Dahlin. ACM, 2013, pp. 423–438. ISBN: 978-1-4503-2388-8. DOI: 10.1145/2517349.2522737.