P3-Mobile Parallel Peer-to-Peer computing on mobile devices

Daniel Filipe Pereira Moreira da Silva

Mestrado Integrado em Engenharia de Redes e Sistemas Informáticos
Departamento de Ciência de Computadores
2016

Orientador
Fernando Manuel Augusto da Silva, Professor Catedrático
Faculdade de Ciências da Universidade do Porto

Coorientador
Luís Miguel Barros Lopes, Professor Associado
Faculdade de Ciências da Universidade do Porto

Todas as correções determinadas pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, ______/______/_________

Abstract

P3-Mobile is a new parallel computing model based on the crowd sourcing of mobile devices to form a hyper local edge-cloud. The idea behind an edge-cloud is to form a computational/storage cloud comprised solely of a collection of nearby wireless edge devices, with the purpose of pooling these devices data and processing power to support a new class of proximity-aware applications that benefit the owners of these devices. The premise behind these edge clouds is that all of the constituent nodes are edge (and not server-caliber) computers, and that any and all computation is performed completely within the edge cloud, i.e., there is no offloading/tethering of the computation/data to a non-edge, back-end, traditional-cloud infrastructure.

In previous work, a Parallel Peer-to-Peer (P3) model was created towards the Internet and static devices, polling a big sum of resources that as a whole were able to perform highly complex computations. However, when that model is transposed to mobile devices several new problems arise and new techniques must be developed to perform parallel peer-to-peer computations on this very dynamic and volatile environment. P3-Mobile aims to make that transition from physically connected devices to mobile and wireless devices, always having in mind the principles of the original model: high availability, portability and simplicity.

This dissertation describes in detail a computational model for P3 Mobile, focusing on its architecture and implementation. The architecture of P3 Mobile is service based and it includes four main components, a link service to manage communications, a network service to manage group nodes coordination, a storage service to manage files, and a parallel processing service to run parallel programs within the P3 Mobile environment. The algorithms that enable these services to operate, and their implementation, are described in detail. Performance is assessed with an example application to calculate the Mandelbrot set in a distributed fashion using the P3 Mobile system.

To my parents, my brother and Inês

Acknowledgements

I would like to express my deepest gratitude to my supervisors, Prof. Fernando Silva and Prof. Luís Lopes, for all of their support, time and advice, not only during this Master's thesis, but since the first day I entered Faculdade de Ciências da Universidade do Porto.

Also, I would like to thank all the Hyrax team based on Faculdade de Ciências da Universidade do Porto for all of their support during the development of P3-Mobile. This work was supported by a seven month research scholarship from the Hyrax project (CMUP-ERI/FIA/0048/2013).

And last, but not least, to my friends and family that were the moral support during these five years I have spent on this journey.

Thank you.

Contents

List of Tables

List of Figures

Chapter 1

The Case for Mobile Edge Computing

1.1 Motivation

With the increasing popularity of mobile devices with high processing powers, new opportunities arise with these types of devices that most of the time are idle. Having a high amount of these devices in a geographical proximity and with significant processing capabilities when idled, these could be connected as local edge clouds to perform useful tasks. Using mobile devices connected on a peer-to-peer environment to realize an edge cloud (without the assist of any server) still poses unique challenges. Besides the absence of centralized coordination, mobile devices (just like the name suggests) can move from one place to another in a very short period of time so the connection quality becomes an issue. Using only wireless communication means more unreliable connections than traditional wired connections. And although mobile devices are steadily increasing their processing power, they are still not as powerful as standard personal computers.

1.2 Distributed systems

Due to the decrease in hardware and network services costs (as showed in Figure 1.1) and the improvement of network technologies, in the last years distributed systems are becoming increasingly more popular and more pervasive [1]. In fact, distributed systems are getting larger, more complex, handling larger volumes of data, more efficient and with a huge amount of different types of applications.

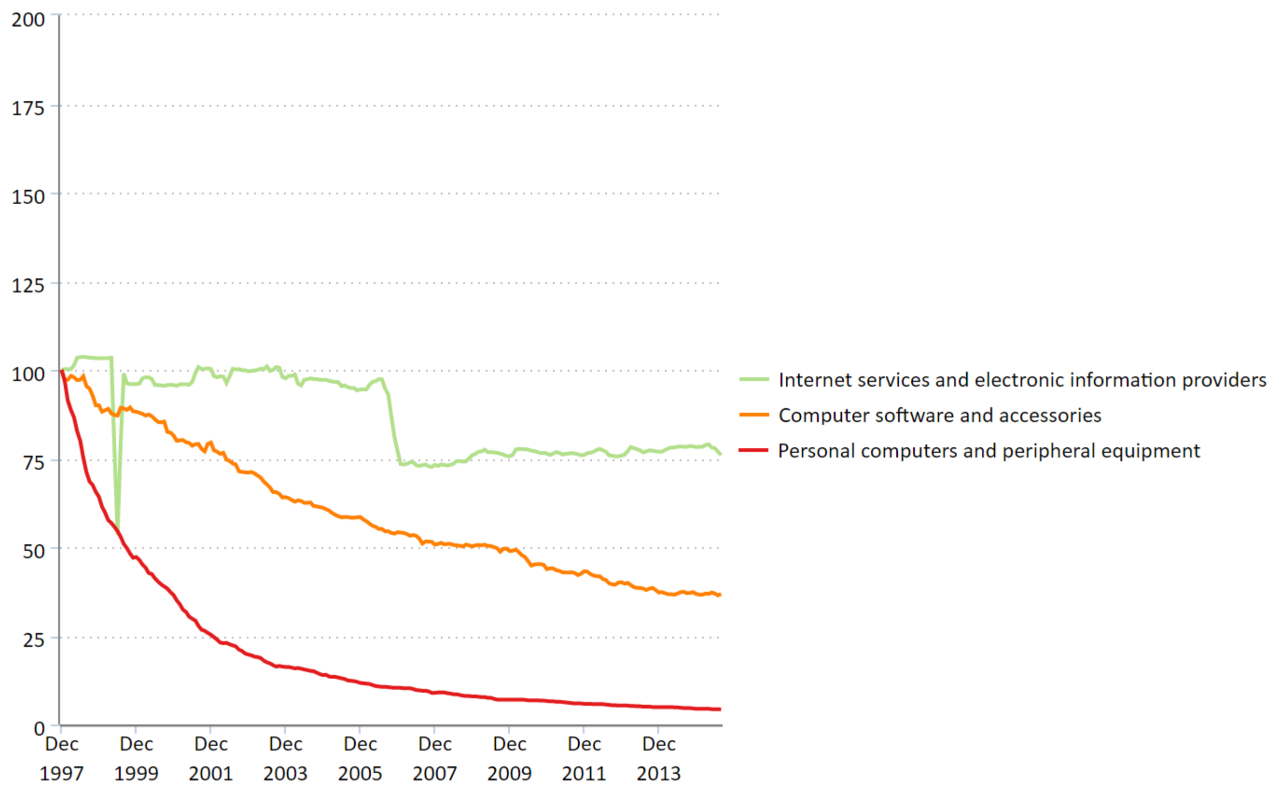


Figure 1.1: Consumer price indexes for computers, software and internet services between 1997 and 2015, according to U.S. Bureau of Labor [1].

1.2.1 Client server model

There are a few basic architecture models of distributed systems. The client-server model is probably the more common model of a distributed system. There are two parts of these systems: the clients (may be devices controlled by a user or another software on another device) request a service or resources to servers, and the servers that have one or more services running constantly and are permanently listening for incoming requests. When a new request is received, it is handled by one or more of the services that are being executed in the server. The server then creates and sends a response to the client. In this process, clients always initiate the communication and servers are constantly listening for requests [3]. The Figure 1.2 shows a basic setup based on this model.

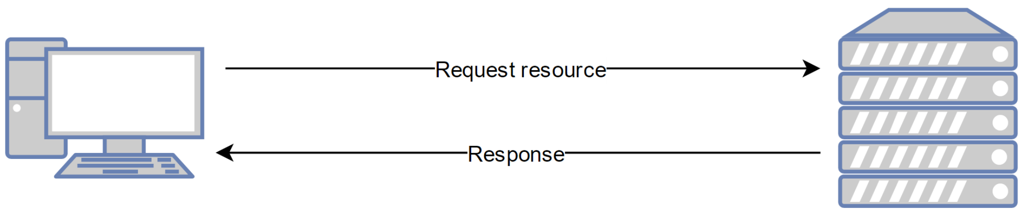


Figure 1.2: Client-server network model.

One problem inherent to this model is scalability. Servers might become a single point of failure. If overloaded by requests, there might be a crash and all services might become unavailable. Until now, to scale systems based on a client-server model, the solution is to add more servers to distribute load between them. It works well, specially when hardware and network services prices are continuously dropping.

A common example of these types of distributed systems are email services. Clients (in a browser or in a client application like Mozilla Thunderbird) make requests to an email server (using IMAP protocol, for example) and the server responds with a list of new email, the spam folder, for example.

1.2.2 N-tier architecture model

Another common model of architecture for distributed systems is a multi-tier architecture (or n-tier). It is a specific type of client-server model where different parts of the applications are split into several pieces called layers. The main idea is to modularize the application to the point that it becomes easier to develop, maintain or even completely change each layer without having to rework the entire application. Having separate tiers makes it possible to adapt each individual tier to for better operation within the whole system. It is possible, for example, to have different layers in distinct geographical locations and still operate as a whole application [4].

The most common type of multi-tier distributed systems are systems with three layers. These systems are composed by a presentation layer, which is responsible to directly interact with the user, a business logic layer, responsible for processing inputs coming from the presentation layer or doing routine tasks on the data, and a data layer where all the application information is stored on some sort of database or file system [4]. Figure 1.3 shows a graphical representation of a three-tier system and the interaction between them.

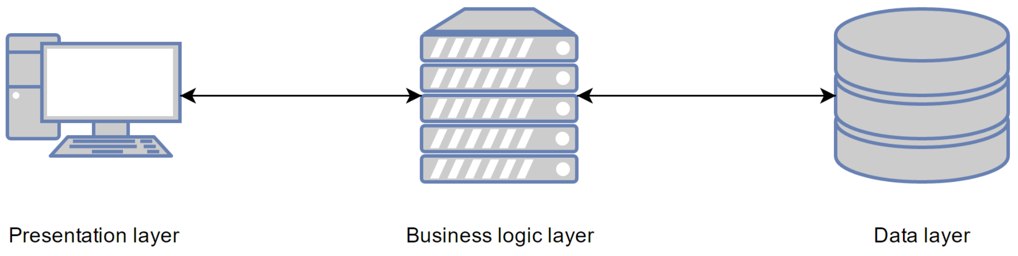


Figure 1.3: Multi-tier client-server architecture model (3-tiers).

It is easier to develop and, if hardware is available, to distribute multiple components through multiple machines providing reliability and efficiency to the application. This is the most popular model for most of Web applications and services available. For example, a user requests a Web page to a server and presents it on a Web browser. This is part of the presentation layer. If the user submits a form, the data will be sent to the server and that data will be handled by the business logic layer. The result of that processing would be stored in a database, part of the data layer.

1.2.3 Peer-to-peer model

Another type of distributed architecture is the peer-to-peer model. This type of system eliminates the need for central point, like a server, in a distributed system. A Peer-to-peer system works by peers (peers are connected devices in a Peer-to-peer system) sharing computational resources with other peers in the network to perform a set of tasks [5, chapter 10].

In contrast to the client-server model, in peer-to-peer networks there is no clear infrastructure, no defined client or server. Every node of a peer-to-peer network at some point in time may send data to other machines, just like a server, and at some other time it may receive information from other nodes. Figure 1.4 illustrates the difference between the two models. In the client-server model (on the right), every client sends requests to the same server (or cluster of servers) and the server produces responses to those clients. The server is a clear central point of the architecture and its functions are well defined. However, in the peer-to-peer model (on the left) every node is equal in functionalities and responsibilities. There is no clear structure between them and there is no central point of connection, nodes connect directly between them.

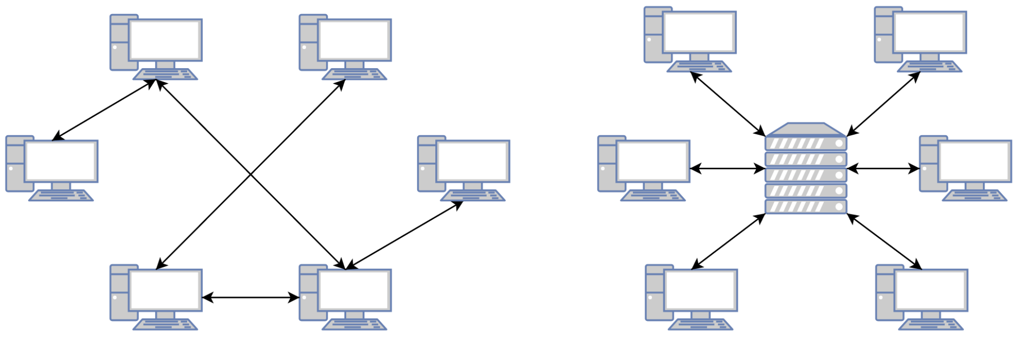


Figure 1.4: Peer-to-peer network model and client-server network model.

These differences in the architecture result in several pros and cons between the peer-to-peer and client-server model. Due to the fact that there is no special infrastructure, the monetary costs of implementing a peer-to-peer network are very low. Also, because there is no central point in this type of networks and information usually is replicated redundantly between different nodes, peer-to-peer networks tend to be more stable due to the failure of some machines should not compromise the functioning of the rest of the network. Furthermore, peer-to-peer networks scale much better than typical client-server architectures. In peer-to-peer, a certain node only knows a few nodes on its neighborhood while in the client-server model, the server (or cluster of servers) needs to reply to all the requests from clients (hence the cost factor, the more servers, the bigger the monetary cost), being a far less scalable solution.

1.2.3.1 Historical overview of peer-to-peer networks

Peer-to-peer networks are not a new concept. The initial concept of Internet was in some ways a peer-to-peer network. Terminals in several institutions shared a link between them, connecting them directly and forming a large decentralized network where everyone received and provided content all the time. Today, the client-server model is more prevalent where content is usually more centralized in servers and clients make requests for accessing or modifying contents.

Peer-to-peer networks started to gain more notoriety by being used in several services of file sharing over the Internet. Probably the most famous example of this type of services is Napster. Released in 1999, Napster was one of the first music sharing services on the Internet. It created a peer-to-peer network between every Napster user and files were shared directly between them. Napster also used a server for querying for content. This hybrid approach allowed for an easier and quicker way of finding content and in which nodes it was stored. However, as Napster became a bigger success and the number of users started to grow, the amount of meta-information on the servers about the content was huge and it started to overload the service (the meta-information server became a bottleneck).

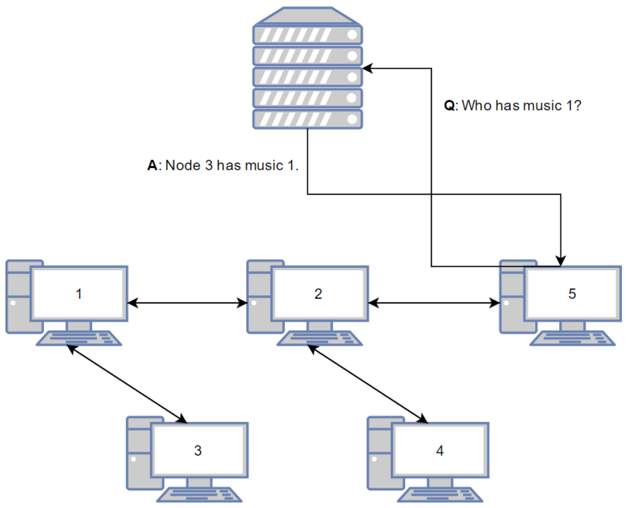


Figure 1.5: Napster hybrid architecture.

From that point, new peer-to-peer file sharing services started to appear. Some of the more famous examples are Gnutella, Kazaa or BitTorrent. Every service has different protocols and implementations, advantages or disadvantages, but the peer-to-peer core ideas are always present: decentralized communication between different machines scattered through the Internet.

1.3 Volunteer computing

Every day new and more powerful hardware components arrive at the market at affordable prices. Regular computers increase performance rapidly (getting closer to the performance of an average server), smartphones have more processing power than the average personal computer of ten years ago [6]. However, most of the time these devices are only using a small portion of the total processing power available. Why should we waste these resources when we could be doing something useful with them?

That is the main idea behind volunteer computing, use the available exceeding resources of a device without getting in the way of the user when the device is not being used. This may be not much on a single machine but, if we gather the resources of a large number of devices the amount of resources available will be enough to handle very complex tasks. This is the principle behind all existing projects of volunteer computing. You start with one or more very complex tasks and you start by dividing these tasks into smaller and easier to solve problems. These small problems are distributed to volunteer machines that solve them and return a result. If the problem is divided into enough parts and the amount of available resources is high, the very complex task is solved in a fraction of the time it would take for a single machine to do it.

There are some notable examples of systems that work based on volunteer computing. Probably the best known volunteer computing project is Folding@home from Stanford University [7]. The objective of Folding@home is to study the process of protein folding, the process of creation of proteins. When a protein is malformed (misfold) several diseases might occur like Alzheimer's, ALS or Parkinson's. It is critical to understand this biological process but it is still relatively unknown how it actually works. One way of studying this process is to simulate it on a computer. However, this process involves crunching an astronomical amount of data and thus running on a single computer is not an option. Computing time in a supercomputer is expensive and scarce. A practical solution to this problem is volunteer computing. Anyone can donate a part of the unused resources on their machine and solve part of this problem. This particular application for the the Folding@home project (each project of volunteer computing usually has their our specific application) is available for an huge amount of devices running Windows, Linux, Mac OS X, Android or even on a PlayStation 3. To this date, the project states that is has around 98 000 machines contributing to the project that output about 102 000 teraflops of processing power [7].

Another good example of volunteer computing is the SETI@home project (SETI - Search for

Extraterrestrial Intelligence) from the Berkeley university. The motivation behind this project is to analyze radio signals coming from space to find evidence of intelligent life outside the planet Earth. Today, the project has almost 200 000 results per hour coming from numerous machines scattered through the Internet helping by analyzing data [8].

1.4 MapReduce

MapReduce is a parallel programming model created by Google for processing and generating large data sets, usually stored in distributed storage systems. It is designed to run on a cluster of machines and its strengths are scalability and fault-tolerance.

MapReduce execution model is easy to understand. The programming model is composed by two main methods: `map()` and `reduce()`. `map()` takes an input and processes it to generate key/value pairs. The input data is distributed to a group of worker nodes that will execute this `map()` on their given input. Then, MapReduce takes all the key/value pairs from the worker nodes and groups all values by key. These grouped values passed to `reduce()`. This method takes a key a set of values as input and produces a smaller set of values based on that input. Just like the `map()` method, `reduce()` is ran on several different worker nodes, each with a different key and values as input. The Figure 1.6 shows a scheme of how MapReduce works [2].

A master server coordinates operations and worker nodes operate the map and reduce methods. The master has a scheduler that assigns tasks to workers. To handle failure on worker nodes, the master server frequently pings worker nodes to check if they are still alive. If no response from a worker nodes is received after a period of time, that worker is considered dead. All the tasks that were assigned to it are invalidated and go back to the scheduler to be reassigned. [2, section 3.3]

One of the key strenghts of MapReduce is the simplicity of the model and how easy it is to create parallel distributed applications just by defining two methods of the model.

Lets see an example, taken from the original MapReduce paper [2, section 2.1], to better

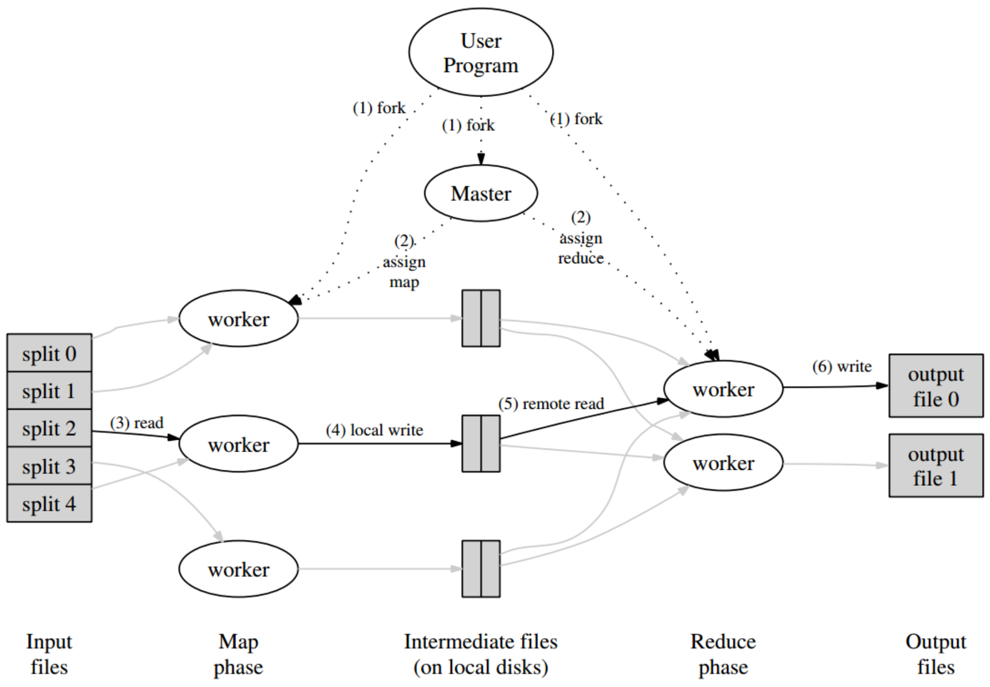


Figure 1.6: MapReduce execution model [2]

.

understand how MapReduce applications works. Imagine that we want a program that counts number of occurrences of each word in a large collection of documents. In the map method, for each work found, a new key/value pair is created with the word and number one. The Algorithm 1 describes this map method.

```
map(key, value):
// key: document name
// value: document contents
for word in value do
    EmitIntermediate(word, 1);;
end
```

Algorithm 1: Map method for counting word occurrences in documents.

Then, the reduce method will sum of the values emitted for a given word. The algorithm 2 describes how the reduce method does that.

```
reduce(key, values):
// key: word
// values: list of counts
result = 0;
for v in values do
    result += v;
end
Emit(result);
```
Algorithm 2: Reduce method for counting word occurrences in documents.

1.5 Mobile Edge Computing

Developed by in a research consortium from Portugal and Carnegie Mellon University, the Hyrax project proposes a novel vision of a hyperlocal edge-cloud, i.e., a computational/storage cloud comprised solely of a collection of nearby wireless edge devices, with the purpose of pooling these devices' data and processing power to support a new class of proximity-aware applications that benefit the owners of these devices. The premise behind these edge clouds is that all of the constituent nodes are edge (and not server-caliber) computers, and that any and all computation is performed completely within the edge cloud, i.e., there is no offloading/tethering of the computation/data to a non-edge, back-end, traditional-cloud infrastructure. Going a step further, there might even be situations where it is impractical/impossible to offload computation to a back-end cloud because of infrastructure unavailability or bandwidth limitations, e.g., in disaster scenarios, in high user-density settings such as stadiums/concerts.

One of the current work in progress of the Hyrax project is the development of a middleware to provide a common API for communication between devices, construction of a logic network and other services like RPC, storage and video streaming. The objective is to abstract the complexity of network operations behind the API so that the developer only needs to setup a few environment options and the middleware handles all the logic behind all of these processes. The Figure 1.7 shows the architecture structure of this middleware.

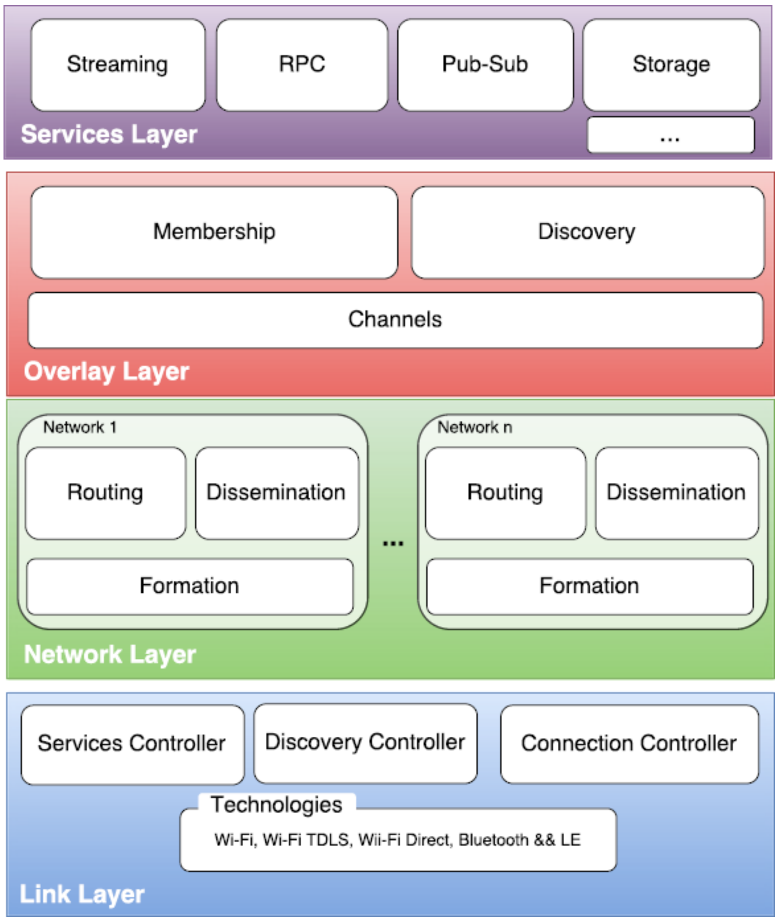


Figure 1.7: Hyrax middleware architecture.

At the bottom of the stack is the link layer, the closest layer to the device operating system. This layer aims to abstract several aspects of the communication between devices like the hardware state, connection to other devices, discovery of devices and services in the environment and visibility of the current device to the local network. To the developer using this middleware, all of this is agnostic to the technology behind these operations. Using Wi-Fi, Wi-Fi TDLS, Wi-fi Direct or Bluetooh, all of the API methods provided to the outside are the same. The layer implementation handles the logic behind each technology.

On top of the link layer, the network layer handles the formation and management of the logic network built on top of the physical network where all the devices are connected. The middleware itself contains an extensible set of different types of network meshes, available out of the box.

The overlay layer allows for the creation of groups on top the network formed in the network layer and the services layer provides implementations of common network services, like streaming or storage.

This organization relates closely to the P3-Mobile architecture. At the base of P3-Mobile there is a similar link service for device to device communication. On top of that, both Hyrax middleware and P3-Mobile have a network service for forming and managing all the operations of the logic network built on top of the physical network. Having a similar design, P3-Mobile is easily ported to use the Hyrax middleware in a near future. Unfortunately, because the Hyrax middleware is in early development, it could not be used yet on the P3-Mobile project. Still, P3-Mobile architecture was designed having in mind a future use of the Hyrax middleware.

1.6 Goals and Contributions

The main goal of this thesis is to create a working prototype of a framework based on the P3 programming model [9] for edge cloud computing using mobile devices that run on Android operating system. To adapt the original P3 programming model to this new environment some issues needed to be solved, such as removing the need for a server for coordinating the network (the P3 portal), improved fault tolerance of parallel tasks or integrate concepts from another research [10] to create a simple distributed storage.

This thesis aims to be a step towards a viable solution of developing distributed applications on edge clouds formed by heterogeneous devices. With an easy to implement interface, developers have control of how the applications run and how they are divided with other devices. The framework provides fault tolerance of tasks and stored files and show promising performance results.

1.7 Thesis outline

This document is organized in five chapters, each one explaining how this work evolved until the actual implementation and its results. The following summary describes each chapter:

Chapter 1: The Case for Mobile Edge Computing. This is current chapter. In this chapter, several concepts of mobile edge computing are reviewed from the most generic definition of distributed systems to concrete mobile edge computing project like Hyrax.

Chapter 2: P3: Parallel Peer-to-Peer. The original P3 Parallel Peer-to-Peer programming model is described. It is analyzed how the programming model works, the network topology that it relies on, the P3 portal and how nodes use it to join the network and how the whole system is organized.

Chapter 3: The P3-Mobile Model. One of the main chapters of this work, in chapter 3 it is described how the P3-Mobile model works and what changes from the original P3 work. It is analyzed how communication between devices works with this system, how the network that the model uses works and how the distributed storage and the distributed parallel programming services work. It also presented the organization of the system in different services and its importance.

Chapter 4: Implementation of the P3-Mobile. Other of the main chapter is the analysis of the actual implementation of P3-Mobile. This chapter reviews all the components of the P3-Mobile and its implementations, it is presented and analyzed a problem used to demonstrate the P3-Mobile functionality, the Mandelbrot set generation problem, and it is analyzed its performance results.

Chapter 5: Conclusions and future work. In the final chapter of this document we synthesize and was analyzed in this thesis and list a few options for future work on top of this thesis.

Chapter 2

P3: Parallel Peer-to-Peer

At the time, motivated by the lack of models and solutions for parallel processing on dynamic and volatile environments, P3 aimed to tackle this need. Also, there was not any kind of model ready for the development of distributed applications on the Internet where resources are volatile and the communication costs are high. In this chapter we describe the original P3 Parallel Peer-to-Peer model created by Licínio Oliveira, Luis Lopes and Fernando Silva [9] [11].

2.1 Motivation

At the time, there were very few models or frameworks for developing high performance applications on dynamic systems, like the Internet. Although some projects already worked on the principle of distributed computing over the network, like the SETI@Home or the Folding@Home projects, their purpose was very strict and specific for a particular type of task [9, p. 57, 58].

Also, there were not a specific model specifically tailored for dynamic and heterogeneous environments (like the Internet). These types of environments present unique challenges. Devices connect and disconnect all the time, without any warning. The network topology changes every second and almost nothing is guaranteed to stay connected and available all the time. Besides that, communication performance is not the same across the network and sometimes it might be of very poor quality. And finally, not every device connected to the Internet is the same.

Today, in the age of the Internet of Things, pretty much anything electronic is in some way connected to the Internet. Not just computers are different from each other, now the Internet is filled by smartphones, smartwatches, tablets, cars and even dishwashers [12].

With all this is mind, a model that tackles all of these challenges is the basis for the development of new environments and frameworks for an easier development of modern distributed applications. This is the main objective of the P3 project.

2.2 P3 network topology

A P3 overlay is a logic network built on top of some physical infrastructure [9]. The objective is to abstract the network details from the programming model.

A smallest unit of a P3 overlay is a node (device). Nodes can be of two types: coordination nodes, responsible for managing the network and services running on it, and computation nodes, that just perform computation tasks.

Nodes are arranged into coordination cells, with one or more nodes taking a coordination role. Coordination nodes are responsible for managing the network and service tasks related to the nodes on their coordination cell and communication between coordination cells. Coordination cells are organized in a tree data structure just like the one represented in the Figure 2.1. In this tree of coordination cells, the coordinator nodes of parent coordination cells know everything about the sub-tree of children coordination cells beneath them. The Figure 2.1 shows a possible P3 overlay tree with a maximum number of nodes per cell of six and number of coordinator nodes per cell of two. Note that when a new node joins the network, the topmost cells are always filled before spawning a new coordination cell below.

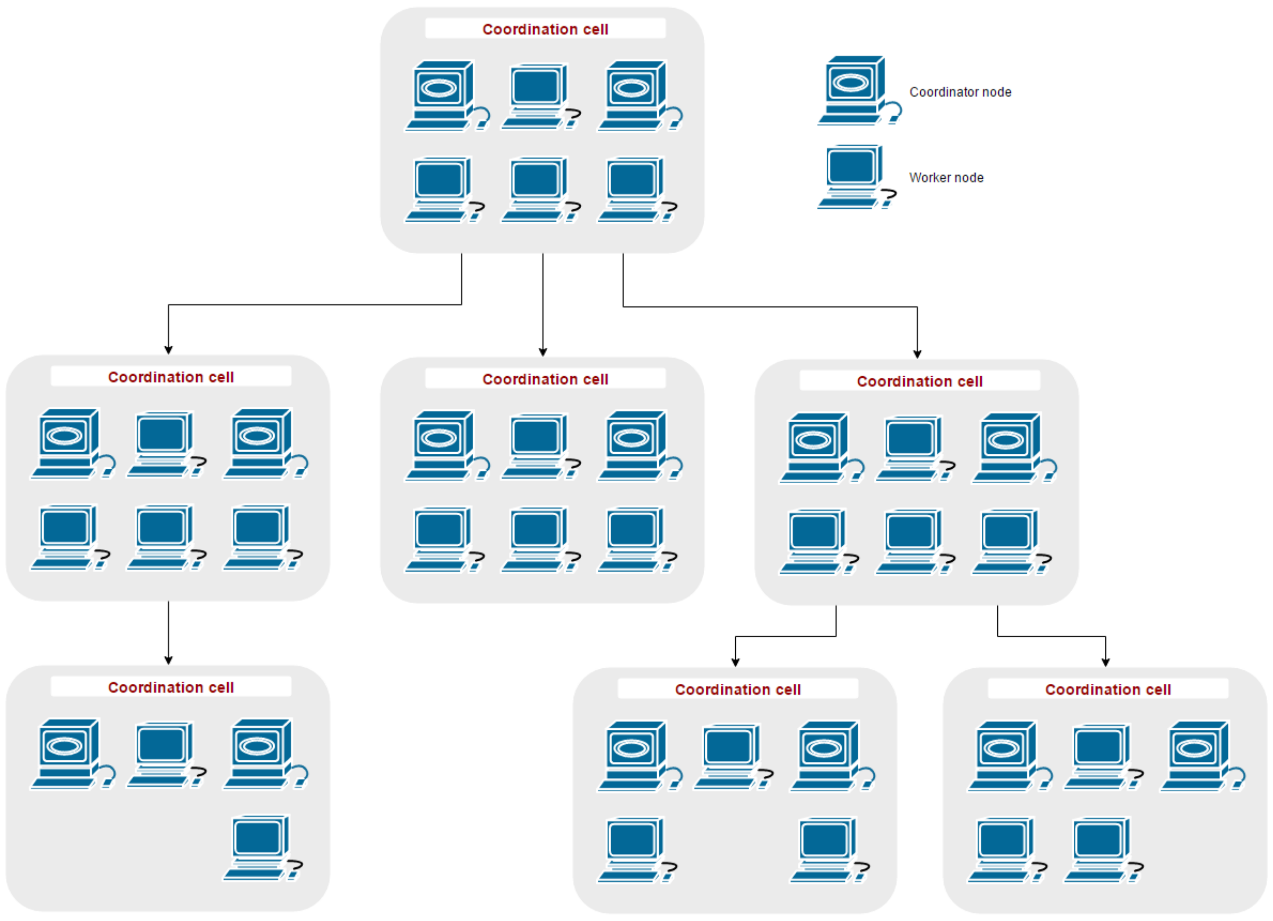


Figure 2.1: P3 network topology.

2.3 P3 portal and joining the network

All the nodes have a unique identifier that is registered in a server called P3 portal [9]. This P3 portal is a publicly available and know server that stores all the identifiers of all the nodes that are connected to the P3 network. When a new device tries to join the P3 network, if it does not already have an unique identifier assigned, the P3 portal gives the contact information of the root coordination cell to the device and this cell creates and assigns a new identifier for this new device [9]. With the unique identifier assigned and stored in the P3 portal, the device can get a list of coordinator nodes from the P3 portal and start a join process in any coordination cell. Every coordination cell has a maximum capacity (value defined by the specific implementation), this if there is no coordination cell with capacity to allocate the new node, a computational node already member of the network is promoted to coordinator node and creates a new coordination cell. With enough space for the new node, it is allocated where there is space available and the joining process ends.

The Figure 2.2 describes how the process just described for a new node to try to join the P3 network.

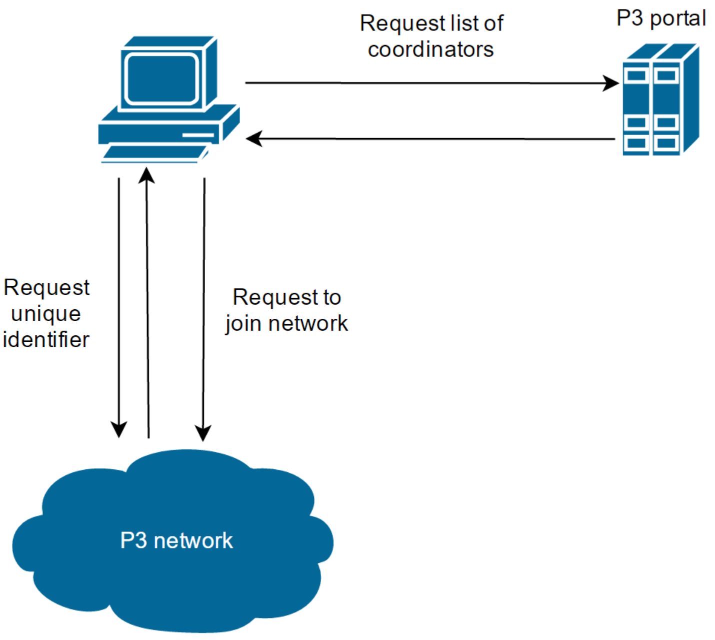


Figure 2.2: P3 portal role in the network join process.

2.4 P3 system architecture

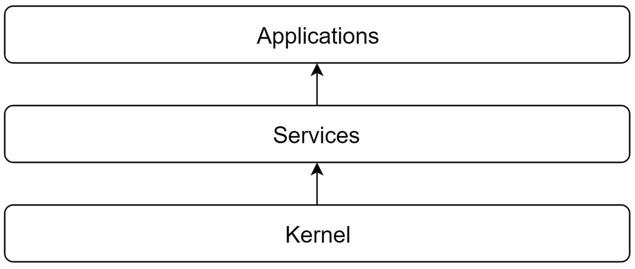


Figure 2.3: P3 system architecture layers.

The P3 system is organized in three main components as shown in Figure 2.3 [9]. The first component is the **kernel** and it is responsible for the core functions: network management and maintenance, communication between nodes, logic network structure management and service initialization. This low level component is the engine that drives the rest of the system.

On top of the kernel layer it is the **services** component. This component works on top of the kernel functionalities to provide a whole new set of higher level functionalities. Each service provides a set of interface methods that applications can use in their implementation. For example, the P3 parallel computing model implementation is one of the services of this component. Using the kernel functionalities (node to node communication or network management, for example), it creates and manages all the logic for the distribution of tasks, fault tolerance of tasks on failed nodes, among others. Also, provides several interface methods for starting a new task and look for work in the network. Another example of a service is a distributed file system that takes advantage of the space available of all the nodes in the network. Such a service like this provides several methods to query and manipulate the file system like store, fetch, list and delete files and folders. The master's thesis written by João Paulo Magalhães describes in greater detail how a service such as this works in a P3 environment [10].

The last component of the P3 architecture is the **applications** component. In this component resides all the applications implementation that takes advantage of the interface methods that the P3 services provides. For example, an application can use the parallel processing service to compute some task and then store the results on a file using the distributed file system. All of that, using the interface methods of the system services.

2.5 P3 programming model

One of the main components of the P3 system is the parallel programming model [9]. The objective of the P3 programming model is to provide the developer with a simple and easy way to create new distributed parallel applications, hopefully effortlessly. To do that, the model must be simple and easy to implement but also be efficient and powerful.

We could classify this model as a *work-sharing* model. Unlike the master-slave paradigm, where a master node sends tasks to slaves and then the slaves returns results to the master, in a *work-sharing* environment worker nodes proactively search for work from other worker nodes and other nodes divide their tasks and share them with others. The key difference here is that in a master-slave paradigm, the master (or the node that is sharing work with other nodes) is usually a centralized infrastructure, and in the case of a work-sharing scenario, the work division responsibility is completely decentralized. In work-sharing, the work division is made on on-going computation of any node of all participating nodes.

In this model, a task can be in a set of states: **initialized**, when the task is ready to start, **running**, when task is being executed, **suspended**, when task execution is stopped for work splitting, and **terminated**, when the task has completed its processing. The Figure 2.4 demonstrates the life cycle of a task, as just described.

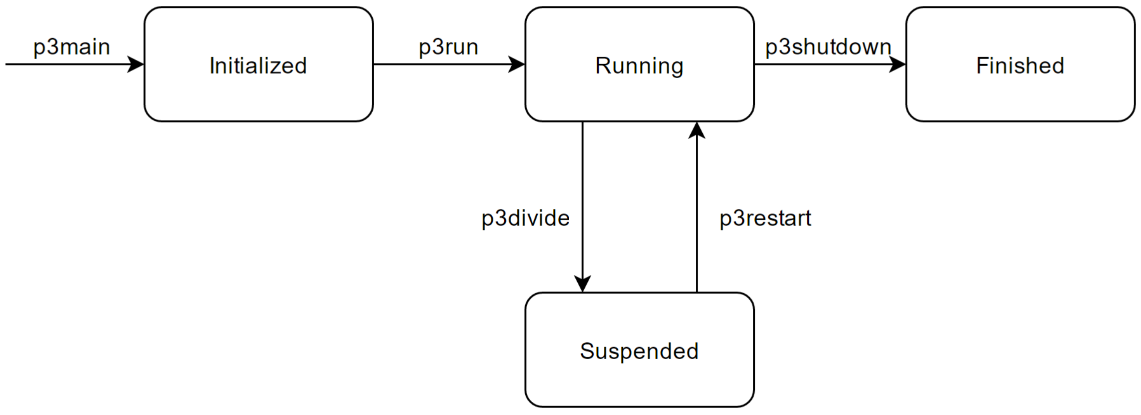


Figure 2.4: P3 programming model task life cycle.

This simple scheme of possible states maps the entire life cycle of a task. To develop a new parallel distributed application with this model, requires that the application include the implementation of the methods for each state and transition, synthesized in the Table 2.1.

In total, the model defines five different interface methods that applications must implement. First, `p3main()` in the initialization method of a task. In here, any value or parameter used by the task processing logic can be initialized. It is only executed once and before any other function of the task. After initialized, the task enters the running state with `p3run()`. This method includes the main processing loop and it is where the task begins processing. From this state, the task can be suspended for work division or terminated when the task is finished. Also, any task that results from work sharing from another task starts in the p3run method. To define how the division of the task must be made, `p3divide()` is called. When another node requests work, the current task is interrupted and the task is split into two by this method.

Method	**Description**
p3main	Initialization of task. Only runs once, before any other method.
p3run	Task processing main loop method.
p3divide	Divides current task into two new tasks, one for the current node, other for the node requesting work.
p3restart	Handle end of division process and restart task processing.
p3shutdown	Task post processing logic.

Table 2.1: P3 programming model methods.

After the division, one of the tasks remains in the current node and another goes to the other node. When the division process is completed, then it is called `p3restart()`. This method gives the developer a chance to perform any adjustments to the task (if needed) after the division process. When the task is completed, the method `p3shutdown()` is called. This provides a way to do any post processing needed on the task.

Having describe the P3 model and corresponding system architecture, we are now ready to investigate whether this model is still a good basis for a mobile environment in which churn plays an important role. This is goal of the next chapter.

Chapter 3

The P3-Mobile Model

Just like the original P3 project tried to create a new tool for quick and easier development of high performance distributed applications on dynamic and heterogeneous environments (like the Internet), P3-Mobile tries to apply the same principles of P3 adapted to mobile devices and edge clouds. P3-Mobile is a new approach of the original P3 model that tackles the specific issues of mobile devices connected through wireless communication and provides a better tool for developing distributed applications on these devices.

3.1 Main differences between P3 and P3-Mobile

Although based on the original P3 project, the P3-Mobile has some key features that differentiate from it:

- P3 relied on a central server (the P3 portal) for keeping track of unique identifiers assigned to a certain IP address. In P3-Mobile, there is no fixed central structure. P3-Mobile is intended to be ran on edge clouds, relying only on the devices themselves to create and maintain the network.
- P3-Mobile networks target only a single application at a time and are formed for that application. The device that starts the computation take the root and becomes the entry

point for the network. This node is responsible for advertising in the network that he is the "host" for that network and other devices can initiate the join process through it. This also means that in the same physical network might be indefinite instances of P3-Mobile applications running without interfering with each other.

- P3-Mobile takes a different approach to the organization of the framework itself from P3.

3.2 Services architecture

The P3-Mobile framework is composed by a collection of services that, in coordination, provide all the functionality necessary to run distributed applications in a crowdsourcing environment. This type of architecture has several advantages.

Each service is self contained and performs some specific task. This way, every service can be developed and tested separately from the others and, more importantly, implemented in different ways, as a separate module. By having different modules of a service but maintaining a fixed set of interface methods, it is possible to have a service that performs a specific task in different environments where the only code that needs to be rewritten is the service module (where the rest of the application remains the same).

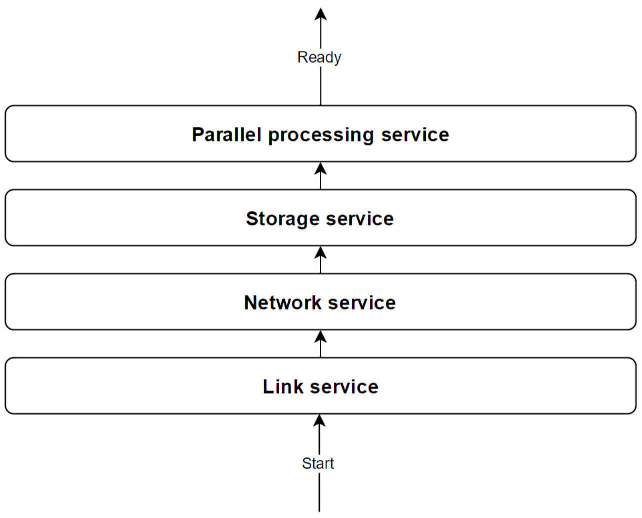


Figure 3.1: P3-Mobile core services list.

Another important feature of having an architecture based on services is that it is easy to set an idea of dependency between services. Some services might depend on functionalities provided by other services, therefore require that the services it depends must be initialized. By dividing all the architecture in several services, it is possible to start one at a time making sure that a new service only initializes after the last service finishes setting up.

Also, by separating all the functionalities of the framework into services, it becomes easier to the developer to extend the framework by developing its own services and attach them wherever desired in the service dependency chain. Aside from attaching new services, the developer also has the option to enable or disable services at will, adapting the framework to the application needs.

P3-Mobile has a few base services that provide all the necessary functionalities for device to device communication, network formation, distributed storage and parallel processing. Each service depends on all the services preceding it (hence the importance of the notion of dependency). The Table 3.1 summarizes the four base services of P3-Mobile that we explain in more detail in the next sections.

Service	**Description**
Link	Handles communication between two nodes (send and receive messages).
Network	Creation and management of the logic network of nodes.
Storage	Management of the distributed storage mounted on top of the network.
Parallel processing	Management of distributed parallel task running on the network nodes.

Table 3.1: P3-Mobile core services.

3.2.1 Link service

The first service of the P3-Mobile services queue is the link service. The objective of this service is to provide communication between two devices in the P3 network.

Although the purpose of this service is simple to describe, the actual implementation might be not that simple. By separating the main functionalities in different services, it is possible to create different implementations of any service that use different technologies. In the link case service this is specially relevant because there are different communication technologies that might be useful in different environments.

3.2.1.1 Possible scenarios and communication technologies

In an environment with an infrastructure with a wireless access point with WiFi technology, the link service implementation can create a connection between devices using the interface connected to the WiFi network provided by that access point.

On a different scenario, where there is no infrastructure available other technologies like WiFi-Direct or Bluetooth might be useful. With WiFi-Direct, devices can create peer-to-peer connections with other devices with WiFi without using a wireless access point. Currently, WiFi-Direct has some limitations on Android devices. In recent tests, WiFi-Direct could only form networks with only five or six devices at a time [13]. Although currently implementations present some limitations in network formation, WiFi-Direct could provide useful for setting communication between devices. Another option for setting up communication between devices could be by using Bluetooth technology. Like WiFi-Direct, using Bluetooth does not require a physical infrastructure to be present (like a wireless access point). WiFi-Direct provides better transfer speeds than Bluetooth (theoretical maximum of 250Mbps for WiFi-Direct and theoretical maximum of 25Mbps) but with a higher energy consumption with WiFi-Direct than Bluetooth 4.0 [14].

In the implementation of P3-Mobile in this thesis it was used a wireless access point with WiFi technology to connect all the devices but with further work the other models could be used as well.

3.2.1.2 Link service operation

Different technologies have necessarily different implementations, but the objective of the link service is to abstract the complexity of setting up communication between devices by providing simple interface methods that the developer can use. The link service provides two main functionalities: send or receive messages and files to and from other devices. The receive functionality works, in general, by having a separate thread actively listening for incoming connections from other devices and data is read from there. Then, this data is broadcasted to the collection of registered listeners of the link service (this could be other services of the framework or any other component of the application). To send a message or a file, the service provides a method where the developer provides the information of the destination device and data to be sent. The service will then create a connection between the devices (using the technology chosen by the developer) and send the data supplied. The developer does not have to know the details about the chosen technology. The only needs to know the method to send messages and files, and register a listener to receive data.

3.2.2 Network service

The network service layer handles all the logic for the formation and the management of the integrity of the network. The P3-Mobile network is organized in a tree of coordination cells and each coordination cell is composed of several nodes (a node is a unique device). The developer of the application can define through an environment variable several parameters of how the network organizes itself: the maximum number of nodes per coordination cell, maximum number of coordinator nodes per coordination cell and can define the maximum number of children coordination cells of a single cell. The Figure 3.2 shows an example of a P3-Mobile network with a maximum number of nodes per cell of five and a maximum number of coordinator nodes per cell of two, in which the root coordination cell has two children.

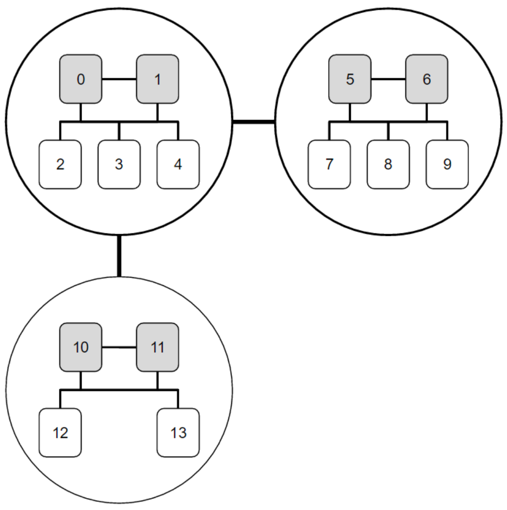


Figure 3.2: Scheme of a P3-Mobile network with 3 coordination cells and 13 devices. The device number 0 is the device that started the computation, highlighted devices are cell coordinators and the rest are workers nodes.

3.2.2.1 Coordinator and worker nodes

In each coordination cell, the nodes can be of two types: coordinator or worker. A coordinator node can also be a worker node, it just has additional responsibilities. Coordinator nodes need to know the details of all the nodes in their coordination cells. These nodes know the identification values of all the nodes and the files or directories of the distributed file system in the current cell and the identification values of the parent coordination cell coordinators. Because these special nodes have an higher knowledge, they are the communication points for discovery of specific nodes in the network tree of coordination cells. This is a similar organization of the DNS service in the internet. Also, coordinator nodes are responsible for keeping track if every node in their coordination cell is alive. When a node fails, the coordinator nodes detects that failure and handles it accordingly. If a worker node fails, that failure is detected and every coordinator node of that cell is notified, and every one of them updates their records. On the other hand, if it is a coordinator node that fails besides all the steps taken when a worker node fails, it also needed to elect a new coordinator node from the worker nodes of the current coordination cell (if there is at least one worker node in the coordination cell) and replicate the data in the coordination nodes to the new elected coordinator node.

3.2.2.2 Fault tolerance in coordination cells

Each coordination cell must have more than one coordinator for two reasons: redundancy of the information about the coordination cell (if one of the coordinator nodes leaves the network or fails for some reason, no information is lost and another coordinator can be elected) and load balancing of the requests from and to the coordination cell. Because of that, the minimum number of coordinator nodes in a coordination cell must be at least two. In an environment variable, the developer can define the maximum number of coordinators per cell, but this value must always be equal or higher than two. The only instance where a coordination cell has only one coordinator node is when a new coordination cell is created has only one member node. In that case, when new nodes join that coordination cell, they become coordinator nodes of that cell until reaching the maximum number o coordinators defined by the environment variable. All the nodes that join that cell after that limit is reached become just normal worker nodes. And because it is assumed that any node can fail at any time, coordinator nodes are not an exception. In normal conditions, a coordination cell has exactly the maximum number of coordinator nodes allowed and in case of failure of one of them, a new coordinator nodes must be elected.

3.2.2.3 Electing a new coordinator

The process of election of a new coordinator node is fairly simple. If there is at least one worker node in the coordination cell, randomly one of them is chosen and becomes a coordinator. When elected, this new node receives all the data that other coordinator nodes contain (like information about coordinator nodes of children and parent coordination cells) and start checking if nodes in their coordination cell are alive. Note that when a node is elected coordinator, the network service broadcasts a message to all of its registered listeners informing that the current node just became a coordinator node. This could be a useful feature for other services that their logic might change if the current device is a coordinator or just a worker node.

3.2.2.4 Joining the network

When a new node wants to join the network and help in the computation, the network service uses the node join protocol. This simple protocol tries to allocate the new node on the best coordination cell possible. It is avoided the formation of very sparse networks by trying to fill all the coordination cells available before creating a new one (creating a new coordination is the last resort).

But how exactly does a node joins the network? First, the node must find and entry point to the network. For this, the root node of the network (the node that creates the network) must advertise in the environment that it is a root node of a P3-Mobile network. One way of doing this is when a new node wants to discover if there is a P3-Mobile network available. For that it could send a broadcast packet to the network and the root node of the network would listen to these packets and respond accordingly with the information that there is a P3-Mobile network available. Other solution (that was used in the implementation of P3-Mobile of this thesis) is to use Android's Network Service Discovery (NSD). The NSD allows to advertise and discover services in a network (which is exactly what we need in this situation) or in a peer-to-peer environment. Using NSD, the node that wants to join the network discovers the root node and sends a request to join the network. When the root node receives this request, first it sees if the current coordination cell has any space available (according to the defined value for the maximum number of nodes per coordination cell described before). If the current coordination cell has available space, the root node allocates the new node in its cell and notifies the node that joining process is finished. Otherwise, the root node checks, first if it has any children coordination cells and if it has, checks if any of its children coordination cell have available space for allocating a new node. If any children coordination cells has room available for the new node, the root node redirects the node that wants to join the network to one of the coordinator nodes (chosen randomly) of the children coordination cell that has room available. Redirecting the negotiation means that the node that wants to join the network will repeat this exact same process but instead of sending the message requesting to join the network to the root node, it will send to coordinator node that was redirected to. In the case that there are no children coordination cells, a new coordination cell will be created and the only member of that cell will be the new node. On the other hand, if there were any children coordination cell but all them were full, it is checked if the current number of children cells is maximum (according to the value described before) and if it is not maximum, a new coordination cell will be created with the new node as the only member. If the value is maximum, one of the children coordination cells is chosen randomly and the negotiation is redirected to one of the coordinators of the chosen cell, also chosen randomly. The Figure 3.3 shows a scheme that describes this algorithm.

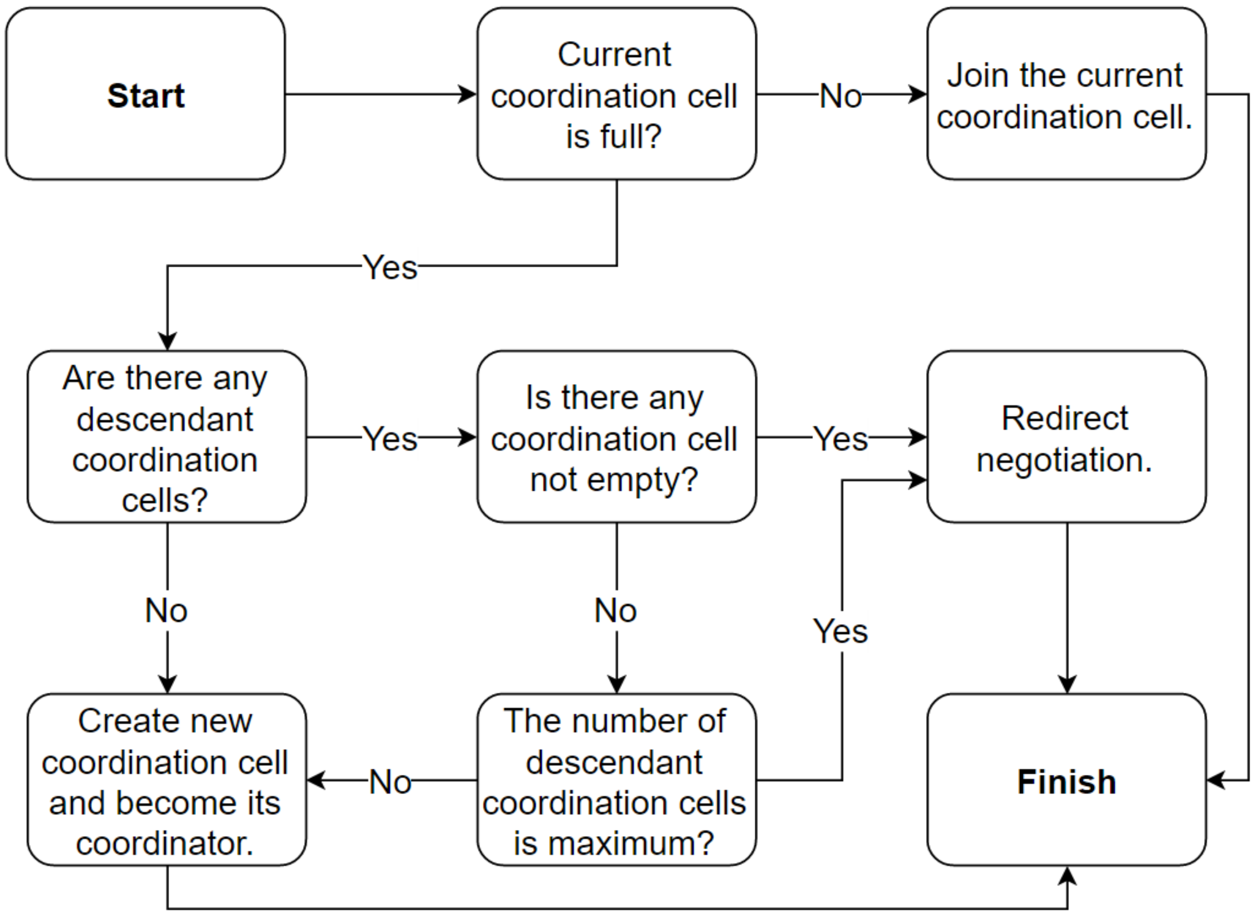

Start
Current coordination cell is full?
No
Join the current coordination cell.
Yes
Are there any descendant coordination cells?
Yes
Is there any coordination cell not empty?
Yes
Redirect negotiation.
No
No
Yes
Create new coordination cell and become its coordinator.
No
The number of descendant coordination cells is maximum?
Finish

Figure 3.3: P3-Mobile network join protocol.

This protocol is relatively simple to understand and implement. It provides a recursive method of joining a new node to the network structured as a tree of coordination cells.

3.2.2.5 Node discovery in the network

To discover a particular node in the network, the service randomly contacts one of the coordinators of the current coordination cell and tries to find the desired node in the current cell. If it is there, that node information is sent to requesting node, otherwise the coordinator node will try to find the desired node in the sub-tree of children coordination cells beneath the current cell by contacting one random coordinator in each cell, repeating the same process that the current coordinator node just went through. If the node is not in the sub-tree of the current coordination cell, the request is moved to the parent coordination cell to search the remaining parts of the network tree. Eventually, if the node is found, the node contact information is sent to the requesting node, otherwise, the requesting node will be informed that the requested node does not exist in the network.

3.2.2.6 Abstraction interface

Just like the link service, the abstraction made with this service is that none of the other services needs to know specific details of how the network is implemented. The network service provides simple methods for membership and discovery. To join the network, the only method to be called by the developer is a simple join method that starts all the process described before. The network service also has a method for discovering a particular node in the network where the only required information is the unique identifier of the node to find.

3.2.3 Storage service

The storage service works on top of the link and network services and its purpose is to implement and provide a method for storing and accessing files distributed throughout the P3 network. From the developer point of view, it is like a disk where its capacity is the sum of the available space of all nodes connected in the network. The developer only needs to save and fetch files, the service must abstract all the necessary implementations to make this possible. This storage service is a simpler version of the distributed storage system created by João Paulo Ferreira de Magalhães [10].

For redundancy, files are stored in more than one device at a time. If one device fails, copies of the files that the device stored are scattered throughout the network and just need to be replicated on other devices to maintain the redundancy. To keep track of files, coordinator nodes of coordination cells of the P3-Mobile network store metadata about the files contained in their cells. The Table 3.2 show the structure of this metadata.

Field	**Description**
Name	Identifier of the file (for example, absolute path).
Size	Number of bytes of the file.
Date created	Date of creation of file.
Locations	Locations where the file is replicated.

Table 3.2: Storage service file metadata structure.

3.2.3.1 Fetching files

Just like the previous services, the storage service abstracts several actions. One of them is fetching a file on the distributed system. When a file is requested, first the service checks if the file is already in the current device. If it is not, the request is sent to a random coordinator node of the current coordination cell. The coordinator node has the metadata for all the files stored in the nodes of the current and children coordination cells. If there is any metadata that corresponds to the requested file, the node that is requesting the file is notified with the information of the list of locations that have the requested file and, with this information, the requesting node can send a direct request for the file to one of the locations received. In the case that the coordinator node did not have any metadata about the requested file, the request is then sent to a random coordinator node of the parent coordination cell and the process is then repeated until the file is found or by reaching the root coordination cell and if then the requested file metadata is not found, an error is returned to the node that originally requested the file informing that the file does not exist. If the file exists and it is successfully retrieved, the node that requested the file also becomes a host for the file and it is added to the locations list in the metadata.

3.2.3.2 Storing and deleting files

The storage service also has methods to store and delete files. To store a file, it is provided to the service and the service abstracts all the necessary logic to add the file to the distributed system. The file remains stored in the node where the store method was called and it is the first location to be added to the locations list of the new file metadata. This metadata must be present in all coordinator nodes of the current and parent coordination cells. For this, the metadata for this file is sent to a random coordinator node of the current coordination cell. The coordinator node that receives this request to add the metadata to its records needs first to ensure redundancy of the file by storing the file in another node of the current cell (different from the node that sent the request to store the file initially) and add the new location the locations list in the metadata. When redundancy is assured, the metadata is broadcasted to all coordinator nodes of current and parent coordination cells.

To delete a file from the distributed file system, the process is pretty straight forward. In every location of a file (nodes that are hosting copies of that file), send a request to delete the actual file. When all of the file replicas in the nodes are deleted, broadcast a message to delete the remaining metadata of the file that remains in coordinator nodes of the current and parent coordination cells.

3.2.4 Parallel processing service

Another core service of the P3-Mobile framework is the parallel processing service. This service is responsible for managing the distributed parallel tasks running on the P3-Mobile network. It also serves as the interface for the developer to launch a parallel task and obtain its results.

This service operates with the logic of the P3 parallel programming model. All tasks must implement some interface methods that the framework will call during its life cycle. The

Figure 3.4 shows a representation of the life cycle of a P3-Mobile distributed parallel task.

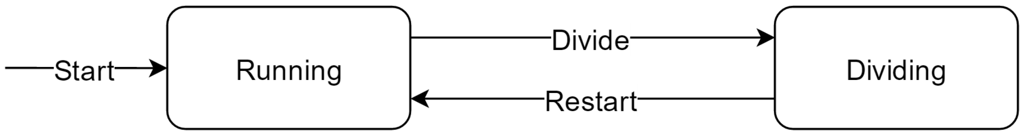


Figure 3.4: P3-Mobile parallel task life cycle diagram.

A task must have a **start** method to initialize its operation. This method is called by the framework when the developer submits the task and the framework is ready to launch it. If the task is being started from scratch (not by being reinitialized from a task originated by a division of a task), it triggers the network formation in the network service. Because of this, the node that starts the computation is always the first node of the network, therefore is the main coordinator node of the root coordination cell. In case of a node just joining the network, it starts by looking for work in his cell.

3.2.4.1 Discovering work in the network

The process of discovering work is relatively simple. In a coordination cell, nodes start by asking for work to one random coordinator node of that coordination cell. If it has work available and the current running task can be divided (the concept of divisibility of a task is explained next), the nodes receives work from this coordinator node. If no work is available from that coordinator node, another coordinator node of that coordination cell will be chosen randomly. Coordinator nodes also look for work in the current coordination cell until there is no more work available. Then, coordinator nodes will try to find more work in the coordinator nodes of the parent coordination cell. This way, the communication with outside the coordination cell is only made by the coordination nodes (just like in the network service). This also enforces that before looking for work outside the coordination cell, all the tasks in the local coordination cell are completed.

3.2.4.2 Task division

At some point in time of execution, a node might request work from another node through the method described before. The P3-Mobile model has a `canBeDivided()` method that decides if the current running task can be stopped for the division process or not. With this method, the developer can allow its task to be stopped only in certain points of its computation or can easily set a minimum granularity of its task, in other words, the developer can define how small tasks can get. This method returns a boolean value representing if the current running task can be divided according to the logic implemented by the developer. If true, the `divide()` is called. This halts the current running task, using this method, the task splits itself into two new smaller tasks. One is sent to the node that requested the task and another remains in the current node to be executed. How exactly a task is divided is up to the developer to define. Different problems have different division strategies (and even for the same problem, different division strategies might be better than others). The only two things that this division method must do is to change the state of the current task object (the task that is being executed in the current node) to the new state of one of the two new smaller tasks, and in this way changes the execution of a task in a node by a smaller version of it, and store a reference to the other smaller task that was created by this process. When the division method finishes running, the parallel processing service will get the reference to the new task, get the task and sent it to the node that requested work (via the link service).

When the task division process is finished, the current node uses the `restart()` method to continue its life cycle by initializing the task that remained in the node after the division process and the whole process might repeat countless times.

When the task is completed, a `shutdown()` method is called. This method gives the developer the opportunity to some post processing work on the task. Table 3.3 shows a summary of these methods.

Method	**Description**
start	Begins the execution of a task.
canBeDivided	Decides if the current running task can be divided or not.
divide	Halts the execution of the current running task and then proceeds to divide the current task into two new tasks.
restart	After the process of task division, one of the resulting task is started and becomes the current running task.
shutdown	Called after the task finishes its main processing loop.

Table 3.3: Parallel processing task main methods.

3.2.4.3 Parallel tasks fault tolerance

As mentioned previously, the P3-Mobile network is volatile and nodes can disappear at any time, without warning. Because of that, the parallel processing service must have that in consideration. Therefore, when a task is divided and sent to another node the service must also save a copy of that task to ensure that it can be re-launched if that node disappears without returning its results. A task from a failed node can be recovered by simply reassigning it to another node when asking for work (instead of halting the current running task and dividing it, send the stored task to the node that requested work). Using this recovery process not just only recovers a failing task but also saves time by avoiding the division process (and the halting process of the running task). The downside of this approach is that when a node fails and the node that had the redundant copy of the task assigned to the failed node also fails. In this case there are two possible scenarios. The first scenario is when the consequences of this multiple failure do not lead to catastrophic failure of the full task because if the failing nodes are in a sub-tree with a parent coordination cell that divided a task with its children. The consequence is that the whole computation of the children coordination cells must be repeated. The second scenario is catastrophic failure of the computation, meaning that the entire task must be repeated. This happens when nodes fail at the root of the tree of coordination cells. To prevent these scenarios it would be necessary to store multiple copies of those task on different devices. This way, when a task is divided, a copy is stored in the node that originated that task and another copy of that task would be sent to a random node (or nodes) in the current coordination cell (in addition of sending the task to the node that originally requested work). In case of successful completion of the task, all copies would be deleted from the nodes, but in case of failure of multiple devices it is possible to recover that task because there multiple copies of it.

Another possible approach is to store a serialized version of the task in the distributed storage as a simple file. When a node fails, the task that was being processed by that node can be recovered by fetching the stored task from the distributed file system, deserialize it and run it.

3.2.5 Other services

One of the objectives of designing the P3-Mobile framework as a collection of services is the possibility of the developer to extend the framework by developing and adding their own services. This adds a lot of flexibility and freedom when designing distributed parallel applications with P3-Mobile framework.

To create a new service the process is easy and simple. The developer only needs to implement a `start()` method so that the framework can boot the service and add the service to the service list (in the most adequate position having in mind the correct dependencies). To be efficient, a service must be added in the first position possible when all of its dependencies are matched.

Examples of some interesting services are:

- **Logging service**: a logging service could be attached at the very bottom of the list of dependencies and catch messages, errors or warnings thrown by other services of the framework or the application itself for processing.
- **Metrics service**: could be used to monitor the P3 network or the environment where the nodes are currently in and optimize network formation or communication routing.

- **Distributed database service**: using the current storage service functionality, it could be possible to develop a new service to create a distributed database in the devices connected to the P3 network.

Chapter 4

Implementation of the P3-Mobile

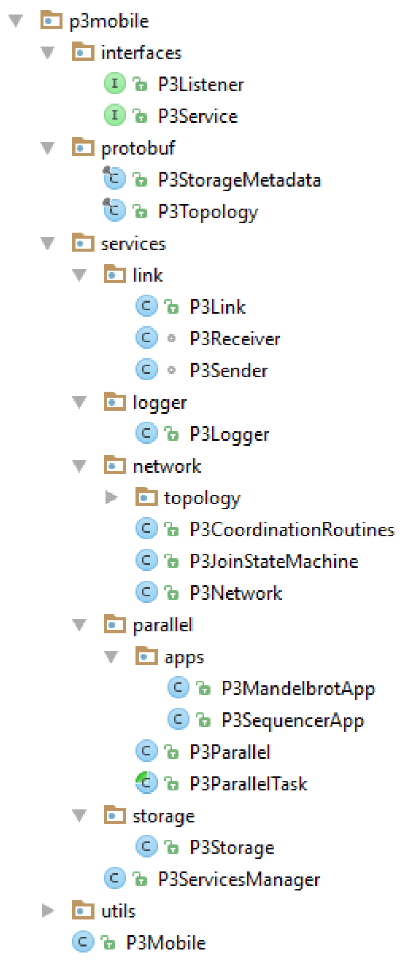
In this chapter we describe the implementation of the P3-Mobile framework on Android. We analyze each component of the framework, describe how they are organized and describe the flow of development with P3-Mobile.

4.1 Overview

The Android SDK and P3-Mobile are written in Java. Most of the components of the P3-Mobile framework rely exclusively on the Java Development Kit 7 (JDK 7). This makes the framework easily portable to a different platform.

The main idea for the P3-Mobile framework is to provide a quick and easy way for developers to start writing code for their distributed apps. The framework is organized in a queue of services. The developer can turn on or off the services that are needed and even add its own services, extending the framework.

The framework services are controlled by some manager classes that provide a simple API for developers, abstracting the details of the implementation of many components (device to device communication, network management, distributed storage and parallel processing). Also, this gives more flexibility to the framework to use different implementations of the interfaces it provides without changing the application itself. There are five implemented services: logging, link, network, storage and parallel processing services. Besides the services, there are two defined interfaces: `P3Listener` for objects that implement this interface can listed for broadcast messages of services, and `P3Service` that contain the mandatory methods that define a service. Finally, there are two files containing the topology and metadata models definitions generated by Google's Protocol Buffers. Figure 4.1 shows the package tree organization for P3-Mobile implementation.



```
▼ p3mobile
    ▼ interfaces
          P3Listener
          P3Service
    ▼ protobuf
          P3StorageMetadata
          P3Topology
    ▼ services
        ▼ link
              P3Link
              P3Receiver
              P3Sender
        ▼ logger
              P3Logger
        ▼ network
            ▶ topology
              P3CoordinationRoutines
              P3JoinStateMachine
              P3Network
        ▼ parallel
            ▼ apps
                  P3MandelbrotApp
                  P3SequencerApp
              P3Parallel
              P3ParallelTask
        ▼ storage
              P3Storage
          P3ServicesManager
    ▶ utils
      P3Mobile
```

Figure 4.1: P3-Mobile package tree.

The root of the package contains one class (P3Mobile), an interface for generic operations on the framework (start client and start server) and four packages. Every package name has a distinctive and informative name that give an immediate idea of what it does.

- **interfaces** - interfaces that other classes in the framework need to implement.
- **protobuf** - Google's protocol buffer based classes that the framework uses to compress information for communication.

- **services** - implementation of the different services that are part of the framework.
- **utils** - useful utilities used throughout the framework.

From these four packages, the most relevant is the services package where vast majority of the logic of the framework resides. The services package contains a single class for the `P3ServicesManager`, responsible for setting up each service and relaying notifications from services to registered listeners, and a separate package for each service. The services packages are all relatively different but all of them contain a general class with the name of the service that implement all the interface methods the service expose to the developer.

4.2 The services bootstrapping

Services are the core of the P3-Mobile framework. The concept that everything is a service is adopted here. From the device to device communication to the parallel processing logic, every functionality of the framework is a different service.

Services are managed by a service manager singleton class (`P3ServicesManager` class located in `p3mobile.services` package) where services are registered and launched, one after the other. The service manager guarantees that the notion of dependency between services is respected. This means that a service may or may not depend on functionalities provided by services launched first. A simple example is the network service that depends of the link service functionalities (send and receive messages to and from other devices) to setup the P3 network overlay.

In Java code, each service registers with the service manager and the service manager stores the reference to each service in an ArrayList (that end up behaving like a simple queue). The ArrayList data structure provides a simple and easy way to store an ordered collection of objects. It is also iterable.

The following code shows how the services are registered in the services manager constructor method and the function callback called after a service finishes its bootstrapping and starts the next service on the ArrayList.

```
private ArrayList<P3Service> services;
protected P3ServicesManager() {
    this.services = new ArrayList<>();
    this.services.add(P3Logger.getInstance());
    this.services.add(P3Link.getInstance());
    this.services.add(P3Network.getInstance());
    this.services.add(P3Parallel.getInstance());
}

/* Call for every successful start of a service, ordered by dependency. */
public void serviceIsReady() {
    if (bootstrapIterator.hasNext()) {
      bootstrapIterator.next().start();
    } else {
      bootstrapIterator = null;
      this.notifyListeners("P3ServicesManager System Ready");
    }
}
```

When every service finishes setting up, a message is broadcasted to every listener registered to the service manager and the application is ready to go. This is useful, for example, if the network service has a considerable delay because it needs to negotiate the node entry in the P3 network.

4.3 Service structure

Each service has only one mandatory class, the class with the same name of the service (rule of thumb for better legibility of code when used) that implements the `P3Service` interface. The

P3Service interface has just one method, `start()`.

The start method is called by the service manager when the service reference is retrieved from the `ArrayList` of registered services. Then, the service class, when everything is ready, calls the `serviceIsReady()` method on the services manager.

4.3.1 The logging service

The first service to be started by the services manager is the logger service. This services is the simplest service present in the P3-Mobile framework but it is really useful. It provides a single interface method for printing a log message to the indicated output. In Android, the preferred output for logging messages is the Log library, while in Java a simple `System.out.println()` might be enough. Just like was mentioned earlier, services might be implemented differently but it is expected that they contain the same interface methods for similar expected results. The following code is the implementation of the `P3Logger` service.

```
public class P3Logger implements P3Service {
    private static P3Logger instance = null;

    protected P3Logger() {}

    /**
     * Creates or gets the singleton instance of the class.
     * @return Singleton object of the P3Logger
     */
    public static P3Logger getInstance() {
        if (instance == null) {
            instance = new P3Logger();
        }
        return instance;
    }
    @Override
    public void start() {
        P3ServicesManager servicesManager = P3ServicesManager.getInstance();
        servicesManager.serviceIsReady();
    }

    /**
     * Log a message to the defined output.
     * @param title Title of the message
     * @param message Body of the message
     */
    public static void log(String title, String message) {
        Log.d(title, message);
    }
}
```

From this code we can see some details that are common to all services in P3-Mobile. First, the `P3Logger` class implements the `P3Service` interface described earlier. Then, we can see that `P3Logger` implements the `Singleton` design pattern [15]. The start method only notifies the service manager that this service is ready. Finally, this service exposes the log method that takes two strings as arguments and calls the `Log.d` function to send a debug message to Android's LogCat.

4.3.2 The link service

The next service to start is the link service. The objective of this service is abstract the details of device to device communication. The following code shows how the link service is started.

```
public void start() {
   listeners = new CopyOnWriteArrayList<>();
   servicesManager = P3ServicesManager.getInstance();
```

```
  P3Receiver receiver = P3Receiver.getInstance();
  P3Listener listener = new P3Listener() {
      @Override
      public void p3notify(String msg, Object... params) {
          switch (msg) {
              case "P3Receiver READY":
                  if (P3Mobile.getInstance().getExecutionMode().equals("server"))
                      setMyServerInfo();
                  servicesManager.serviceIsReady();
                  break;
              case "P3Receiver MESSAGE":
                  notifyListeners("P3Link Message", params[0]);
                  break;
              case "P3Receiver FILE":
                  notifyListeners("P3Link File", params[0]);
                  break;
          }
      }
   };

   receiver.addListener(listener);
   receiverThread = new Thread(receiver);
   receiverThread.start();
}
```

First, the data structure that stores the listeners for the link service is initialized. It was chosen the `CopyOnWriteArrayList` data structure because it is a thread safe version of an `ArrayList` [16]. Next, it gets a reference to the services manager and the receiver class, `P3Receiver`. The receiver class is a `Runnable` class [17] that opens a `ServerSocket` and listens for incoming TCP connections. When a new connection arrives, `P3Receiver` reads the data from it, checks if the data is an object of a file, parses it using Google's Protocol Buffers and dispatch the results to the link service that will broadcast it to its listeners. The Algorithm 3 describes the high level operation of the `P3Receiver` class works.

```
Open server socket on a random port;
Notify link service that the receiver is ready to receive connections;
while true do
    Accept incoming connection;
    Read data and parse header;
    if object then
        Parse object with Google's Protocol Buffers;
        Attach information about sender node;
        Dispatch object to link service;
    else if file then
        Read file name;
        Read file size;
        Read file bytes and store file in local storage;
        Dispatch file name to link service;
    end
end
```

Algorithm 3: P3Receiver algorithm.

Back to the start method of the link service, it creates a listener to attach to the receiver class. This will listen for three types of messages: `"P3Receiver READY"` for when the P3Link service is ready to send and receive messages to and from other devices, `"P3Receiver MESSAGE"` for when data coming through a connection is an object, and `"P3Receiver FILE"` for when data coming through a connection is a file. When an object of a file is received, it is immediately broadcasted to all registered listeners of the link service. This listener is attached to the receiver class and a new thread is created to run the receiver class. Upon receiving the `"P3Receiver READY"` message, the service is booted up and the next service can start.

The `P3Link` also provides two main interface methods for sending data:

- **sendMessage** - public method to send a message with a title and an optional ByteString payload to a certain device on the network.

- **sendFile** - public method to send a file to a certain device on the network.

These methods delegate the delivery of the data to the P3Sender class. The P3Sender receiver the data to send (object or files) and the information about the destination node and opens a TCP socket with that node. The data is sent to the destination where an instance of P3Receives is receiving the sent data as described before.

4.3.3 Network service

The P3Network is one of the most important services on the entire framework. It is responsible for creating and managing all operations of the P3 network that most of the other services rely on.

4.3.3.1 Topology data structures

In this service, besides the implementation of network management and its procedures, there are also definitions for the elements of the topology of the network, nodes and coordination cells. The node data structure holds the information about a particular node. This data structure stores the information about: unique identifier, IP address and port. The coordination cell data structure holds the information about a coordination cell. The information in this data structure is: unique identifier, a map of coordinator nodes of that cell as values and the nodes unique identifiers as the keys, a map of member nodes of that cell as values and the nodes unique identifiers as the keys, a map of the descendant coordination cells as the values and the children cells unique identifiers as the keys, and a map of the coordinator nodes of the parent cell as values and the nodes unique identifiers as keys.

4.3.3.2 Network bootstrapping

If the application is running on the root node (the node that starts a computation), the P3Network service starts all the necessary data structures of the network (the P3Node for himself and the P3Cell for his coordination cell). After setting up of the service, this node is ready to receive requests to join the network and allocate them where possible. If the framework is running on another device that it is not the root node, the first thing that the P3Network service does is to look for the root node and request to join the network. If the root node manages to allocate the new node in his coordination cell, it receives all its network information, otherwise the negotiation is redirected to another node (responsible for another cell) or it is instructed to create a new coordination cell with certain parameters. The following code shows the start method of the network service:

```
public void start() {
    if (P3Mobile.getInstance().getExecutionMode().equals("server")) {
        myDevice = new P3Node();
        myDevice.setRoot(true);
        myDevice.setId("0:0");
        myDevice.setAddress(P3Globals.SERVER_IP);
        myDevice.setPort(P3Globals.SERVER_PORT);
        cell = new P3Cell("0");
        cell.addCoordinator(myDevice.getProtobuf());
        cell.addMember(myDevice.getProtobuf());
        myDevice.startCoordinationRoutines();
        // Start coordination routines
        P3Link.getInstance().addListener(new P3NetworkMessageProcessor());
        setServiceReady();
    } else {
        P3Link.getInstance().addListener(new P3NetworkMessageProcessor());
        join();
    }
}
```

The start method has different logic for the root node (labeled as "server" here) and non-root nodes. If the current node is a root node, it starts by initializing the node data structure with its unique identifier (root node has always the id "0:0") and with the IP address and port associated with the `ServerSocket` in the receiver class of the link service. It also initializes the data structure for the root coordination cell and adds itself as a member and a coordinator of that cell. Then, it starts the coordination routines. The implementation of these coordination routines are responsibility of some coordinator. In this case, the coordinator nodes periodically send short messages to the nodes in their coordination cells, to the coordinator of the parent coordination cell and to the coordinators of children coordination cells to ensure they are alive and still connected to the network. Coordination routines create a separate thread just for this task. The Algorithm 4 shows how `P3CoordinationRoutines` checks other nodes for their vital signs.

```
cell = current coordination cell;
cellMembers = cell.getMembers();
for node in cellMembers do
    Send PING message.;
end
for node in cell.getParentCell().getCoordinators() do
    Send PING message.;
end
for childrenCell in cell.getChildrenCells() do
    for node in childrenCell.getCoordinators() do
        Send PING message.;
    end
end
Wait 5 seconds;
After three consecutive PING messages without response, mark node as dead.;
```

Algorithm 4: Algorithm for node vital signs.

4.3.3.3 Joining the network

In the case the application running this start method is not the root node, the only thing that is done is start the process of joining the network. Just like we have seen in the last chapter, P3-Mobile has a quite simple network joining algorithm. The Algorithm 5 describes how a node makes a join request to another node already in the network.

```
ip, port = Get IP address and port of root node through Network Service Discovery;
target = Node(ip, port) while true do
    link.sendMessage("JOIN", target);
    response = Receive response from the root node;
    if response.status == "JOIN OK" then
        node = response.node;
        if node.isCoordinator() then
            Start coordination routines;
        end
        return;
    else if response.status == "CREATE CELL" then
        node = response.node;
        cell = new Cell();
        cell.addMember(node);
        cell.addCoordinator(node);
        cell.addParentCell(target.cell);
        Start coordination routines;
        return;
    else if response.status == "JOIN REDIRECT" then
        target = response.newtarget;
    end
end
```

Algorithm 5: Algorithm for requesting to join the P3-Mobile network.

On the other side, the node that receives the JOIN request implements the Algorithm 6.

```
cell = current coordination cell;
if cell.size < MAX NODES PER CELL then
    Add node to cell;
    link.sendMessage("JOIN OK", node);
else
    if cell.descendantCells > 0 then
        if cell.descendantCells any has size < MAX NODES PER CELL then
            target = Random coordinator from cell with available space;
            link.sendMessage("JOIN REDIRECT", node, target);
        else
            if cell.descendatCells < MAX CHILDREN CELL then
                link.sendMessage("CREATE CELL", node);
            else
                target = Random coordinator from random coordination cell;
                link.sendMessage("JOIN REDIRECT", node, target);
            end
        end
    else
        link.sendMessage("CREATE CELL", node);
    end
end
```

Algorithm 6: Algorithm for handling network join requests in P3-Mobile.

Besides the network setup, the network service also provides a method for finding a specific node in the network. To do that it sends a request to a coordinator node that implements the Algorithm 7.

```
cell = current coordination cell;
id = identifier of node being searched;
if cell.members.has(id) || cell.childrenCells.members.has(id) then
    return cell.members.get(id) || cell.childrenCells.members.get(id);
end
Send request to random cell.parentCell.getCoordinators();
if response.status == "NOT FOUND" then
    return error;
else
    return response.node;
end
```

Algorithm 7: Algorithm for discovering a node in the network.

The idea here is very simple. Coordinator nodes know every node in their coordination cell and in children coordination cells and can quickly find if the specified node is in its coordination cell or in the sub-tree beneath. Otherwise, it moves this search to a coordinator node of parent cell. If the request reaches the root coordination cell and the specified node is not found, that node does not exist in the network and an error is returned.

4.3.4 The distributed storage service

The next service on the services queue is the distributed storage service. This service is responsible for managing the distributed storage running in the P3 network. The objective of this service is to abstract the details of fetching, storing and deleting files from the distributed file system.

Each coordinator node stores metadata information (described in the last chapter) about the files that are stored in the devices on his and child coordination cells in a HashMap data structure, wrapped by a Google's Protocol Buffers generated class. An HashMap creates an efficient dictionary where keys are the absolute paths for the file and the value is the metadata object.

Because coordinator nodes have the information about the files that are stored in their coordination cell and in children coordination cells, these nodes become responsible for handling requests regarding the distributed storage.

When a node wants a certain file, it sends a request to a random coordinator node of its coordination cell. In the coordination node, the Algorithm 8 finds if the file stored in the distributed storage and returns to the requesting node the metadata data structure that contains all the details about the file, including the list of locations where it is stored.

```
coordinator node receives a "FIND FILE" message;
filename = Name of the file to be retrieved;
cell = Current coordination cell;
if metadata.hasKey(filename) then
    Add requesting node to the locations list of the file's metadata;
    return metadata.get(filename);
else
    if cell.getParentCell() != NULL then
        target = random cell.getParentCell().getCoordinators();
        link.sendMessage("FIND FILE", target, filename);
    else
        return "FILE NOT FOUND";
    end
end
```

Algorithm 8: Algorithm for finding a file in the distributed storage.

Besides fetching files, the service also provides a method to store a file in the distributed storage. The developer only needs to supply the file and the service will store it somewhere in the network. As described in the last chapter, when storing a file redundant copies of it must be created and stored on different nodes. This way, in case of failure of a node, the files stored in it are not lost because there are redundant copies on other nodes. The downside of this is that storing one file will occupy more space in the global space available because several copies are being stored as well. The Algorithm 9 describes how files are stored in the distributed storage.

```
In the node requesting to store the file:
Create metadata structure with information about the file;
if Current node has space available then
    Store file locally;
    Add current node to the locations field of the file metadata;
Send file and metadata to a random coordinator node of the current cell;
In the coordinator node receiving the request:;
Receive metadata.;
Receive file.;
n = number of file replicas;
Choose n nodes from the current coordination cell;
Add the chosen nodes to the locations field in the file metadata;
Send files to the n chosen nodes;
```

Algorithm 9: Algorithm for storing a file in the distributed storage.

Finally, it is also possible to delete a file in the distributed storage through a simple method call to the service. The process is also relatively simple, first find a metadata information about the find, get all the locations and then send a message to delete the file from each node. The Algorithm 10 describes how a file is deleted from the distributed storage.

```
In the node requesting to delete the file:
Send request to a random coordinator node of the current coordination cell;
In the coordinator node receiving the request:;
metadata = storage.fetch(filename);
for node in metadata.locations do
    link.sendMessage('DELETE FILE', node, filename);
end
```

Algorithm 10: Algorithm for deleting a file in the distributed storage.

4.3.5 Distributed parallel computation service

Another important service of the framework is the P3Parallel service, the service responsible for supporting the P3 parallel programming model on the devices connected to the P3 network. This service manages all the aspects related to the work sharing logic.

The P3Parallel main class contains a method "executeTask" that the developer can use to start the distributed computation on all the devices that are connected to the network. While the task is executing, new nodes might arrive to the network, nodes might leave the network unexpectedly or tasks on other nodes might be concluded and ask for more work. The service receives these messages and can pause the execution of the current task, divide it, restart it and send the new divided task to the nodes asking for work. The service also stores all the tasks that are shared with other nodes and that are still not concluded. In this way, it is possible to recover any task lost due to disconnects or errors on the other nodes in the work sharing process. Algorithm 11 shows how a node looks for work in the network.

```
In the node looking for work:
if !tasksPool.isEmpty() then
    for task in tasksPool do
        if task.node is not alive then
            executeTask(task);
            return;
        end
    end
end
target = get random coordinator node of the current or parent coordination cells;
link.sendMessage("DIVIDE", target);
```

Algorithm 11: Algorithm for looking for work.

Before asking for more work in the neighborhood, the node first checks if its tasks pool is not empty and if any of the nodes assigned to those tasks are still connected to the network. If they are not, the current node starts executing that task and that task is now recovered. The

Algorithm 12 describes how the work sharing process is done when a request for diving work is received.

```
**In the node receiving the request to share work:**
if currentTask != NULL AND currentTask.canBeDivided() then
    currentTask.stop();
    newTask = currentTask.divide();
    tasksPool.add(newTask);
    link.sendMessage("DIVIDE OK", target, newTask);
    currentTask.restart();
else
    link.sendMessage("DIVIDE ERROR", target);
end
```

Algorithm 12: Algorithm for sharing work with another node, when requested.

First it is checked if there is a running task and if that task can be divided. If it can, the current task is halted momentarily, call the divide method of that task, the new task is added to the tasks pool and it is sent to the node that requested work in the first place.

Also worth mentioning is that inside the `P3Parallel` service package there is the abstract class `P3ParallelTask` that every task that will be executed by the framework must extend. The `P3ParallelTask` abstract class contains several methods that control the execution of the task and that allow the `P3Parallel` service to interact with task (stop, start and divide task at the current state). The following code excerpt shows the more relevant methods of the `P3ParallelTask` abstract class.

```java
public abstract class P3ParallelTask implements Runnable, Serializable {
    protected P3ParallelTask newTask;
    protected AtomicBoolean stopThread = new AtomicBoolean(false);

    /**
     * Signals task that needs to interrupt it’s execution cycle.
     */
    public void stop() { ...}
    /**
     * @return Boolean value representing the status of this task
     */
    public boolean isStopped() { ... }

    /**
     * Can used to determine the maximum granularity of the task to be divided.
     * @return Boolean value representing if the task can be stopped to be divided.
     */
    public boolean canBeDivided() { ... }

    /**
     * @return P3ParallelTask produced by a division of the current task
     */
    public P3ParallelTask getNewTask() { ... }



    /**
     * @return ByteString representation of the new task generated by the p3divide
         function
     */
    public ByteString getNewTaskByteString() { ... }

    public abstract void p3divide();
    public abstract void p3restart();
    public abstract void p3shutdown();
}
```

4.4 Example: Mandelbrot set generation

4.4.1 Problem definition

To test the framework, it was developed a simple example application that generates an image of the Mandelbrot set. The Mandelbrot set is a set of complex numbers initially investigated by the mathematicians Pierre Fatou and Gaston Julia and it is named after the mathematician Benoit Mandelbrot who was the first to create a visualization of this set. The Mandelbrot is the set of complex numbers c of the function 4.1 that, when iterated through z, remains bounded in absolute value, in other words, the values of c in which the equation 4.1 does not tend to infinity [18, p. 56] [19].

$$\begin{aligned} p_c(0) &= 0 \\ p_c(n) &= p_c(n-1)^2 + c \\ c \in M \iff & \limsup_{n\to\infty} |p_c(n+1)| \leq 2 \end{aligned} \tag{4.1}$$

For each complex number c of a complex plane, the sequence progresses as described in the equation 4.2.

$$\begin{aligned} p_c(0) &= 0 \\ p_c(1) &= p_c(0)^2 + c \\ &= x + iy \\ p_c(2) &= p_c(1)^2 + c \\ &= (x+iy)^2 + x + yi \\ &= x^2 + y^2 + x + (2xy + y)i \\ p_c(3) &= ... \end{aligned} \tag{4.2}$$

In this sequence, with $p_c(n+1) = p_c(n) + c$ and $c = a + bi$, if we write the sequence in order of x and y we get the equations 4.3.

$$\begin{aligned} p_c(n+1) &= x_{n+1} + y_{n+1}i \\ x_{n+1} &= x_n^2 - y_n^2 + a \\ y_{n+1} &= 2x_n y_n + b \end{aligned} \tag{4.3}$$

Progressing this sequence, with different values of c, it either tends to infinity or it does not [20]. This means that c is part of the Mandelbrot set if this progression does not tend to infinity, $c \in M \iff \limsup_{n\to\infty} |p_c(n+1)| \leq 2$.

With this approach it is possible to create a visualization of the Mandelbrot set by iterating each point of a complex plane with $x, y \in [-2, 2]$ and if the point is part of the Mandelbrot set, do not paint it, otherwise give it a color. The color given to a point that is not part of the Mandelbrot set could be representative of the number of iterations that the sequence took to violate the condition $c \in M \iff \limsup_{n\to\infty} |p_c(n+1)| \leq 2$, giving a notion of how fast the progression tends to infinity. The Figure 4.2 show a visualization of the Mandelbrot set according to this definition.

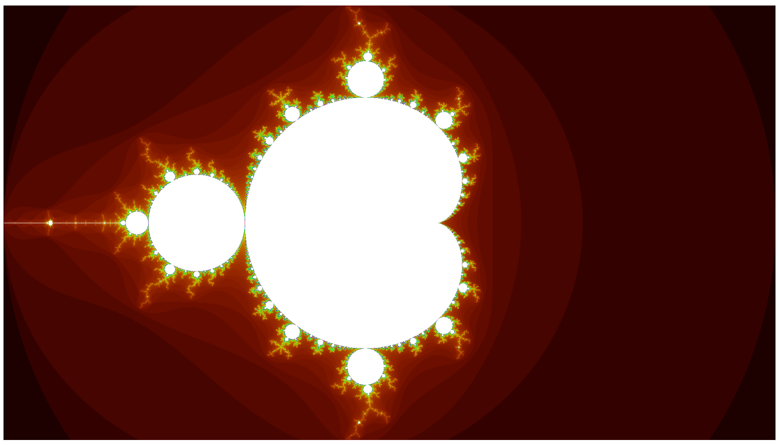


Figure 4.2: Mandelbrot set visualization.

4.4.2 Generating a Mandelbrot set with a P3-Mobile task

4.4.2.1 Run method - pixel color calculation

Using the definition given before, it becomes possible to compute and create a Mandelbrot set visualization. For each pixel of the image of the representation of the Mandelbrot set, compute the progression $p_c(z) = z^2 + c$ until it breaks the condition $c \in M \iff \limsup_{n\to\infty} |p_c(n+1)| \leq 2$ or until a maximum number of iterations is reached. If the progression is stopped because it broke the $c \in M \iff \limsup_{n\to\infty} |p_c(n+1)| \leq 2$ condition, do not paint pixel (or paint it black with full transparency), otherwise give it a color according to the steps made in the progression. The result is an image like Figure 4.2 [20].

Because this computation involves iterating through all the pixels of the image to calculate a color, a simple way of doing it is by iterating from left to right, one row of one pixel wide at a time. This also means that this application is very easy to divide in chunks (in this case by rows) that can be easily distributed to other devices. In the end, every chunk generated by each device is merged into one final image.

The run method of the Mandelbrot task implements these calculations. The following code shows the implementation of the `P3MandlebrotApp` task. It starts by creating a bitmap with the width and height of the image to be created and setting the colors for the pixels of points that are part of Mandelbrot set (black with full transparency) and for the points that are not part of the Mandelbrot set. It creates an array with the size of the maximum number of iterations where each position of the array is a color for the number of iterations made. The following code shows this setup.

```java
Bitmap image = Bitmap.createBitmap(width, height, Bitmap.Config.ARGB_8888);
int black = 0x000000;
int[] colors = new int[max];
for (int i = 0; i < max; i++) {
    float[] hsv = {i/256f, 1, i/(i+8f)};
    colors[i] = Color.HSVToColor(hsv);
}
```

With the bitmap and all colors ready, the pixels processing loop begins. One pixel wide row at a time, it iterates through every pixel and calculates the absolute value of the point (x, y) in each iteration. We want to find out if the value of the absolute value of the point (x, y) is less than 2 (according to $c \in M \iff \limsup_{n\to\infty} |p_c(n+1)| \leq 2$). To calculate the absolute value of a complex point it is calculated the square root of the sum of the squares of x and y, $\sqrt{x^2+y^2}$. In this case we want to know if $\sqrt{x^2+y^2} < 2$. We can simplify this calculation by squaring both sides of the inequation, getting $x^2 + y^2 < 4$ [21]. When the sequence is stopped, if it reached the maximum iterations it means that the sequence do not tend to infinity, therefore it is part of the Mandelbrot set and it is painted black. If the sequence is interrupted before reaching the maximum number of iterations, it means it broke the condition $c \in M \iff \limsup_{n\to\infty} |p_c(n+1)| \leq 2$ and it is not part of the Mandelbrot set and it is given a color from the colors array setup before. The following code shows the implementation of this calculation loop.

```
while (currentRow < endRow) {
    for (int col = 0; col < width; col++) {
        double c_real = (col - width / 2) * 4.0 / width;
        double c_imaginary = (currentRow - height / 2) * 4.0 / width;
        double x = 0;
        double y = 0;
        int iterations = 0;

        double n1, n2;
        while ((n1 = x * x) + (n2 = y * y) < 4 && iterations < max) {
            double x_new = n1 - n2 + c_real;
            y = 2 * x * y + c_imaginary;
            x = x_new;
            iterations++;
        }
        if (iterations < max) image.setPixel(col, currentRow, colors[iterations]);
        else image.setPixel(col, currentRow, black);
    }
    currentRow++;
    while (this.stopThread.get()) {}
}
```

When all the rows are processed, the bitmap is stored to file, sent to the device that attributed the task to this device and the shutdown method is called to notify the parallel processing service to look for new work to do.

The Figure 4.3 show the different chunks processed by ten different devices to create a single final image.

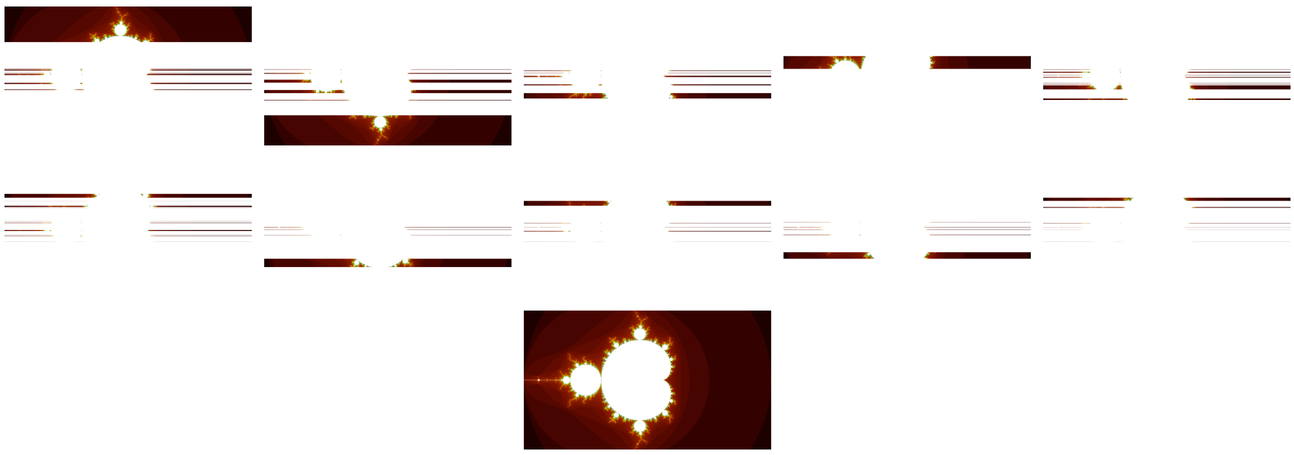


Figure 4.3: Mandlebrot set fractal generation with ten devices.

4.4.2.2 Divide method - from one task to two smaller ones

As described before, the division process involves two methods from the task: the `canBeDivided()` method to assert if the current running task can be divided or not, and the divide method that does the actual division of the task. The `canBeDivided()` method defines the minimum granularity for these problems, in other words, how small the chunks of the task can get. For this problem it was chosen a minimum granularity of six. This means that chunks will have at least six rows. The divide method simply finds the mid row of the current running task, creates a new one with bottom half of rows and updates the current task to finish its processing at the mid row point where the division was made. The following code shows how these two functions are implemented.

```
@Override
public boolean canBeDivided() {
    if (currentRow >= endRow) return false;
    if (endRow - currentRow <= 6) return false;
    return true;
}


@Override
public void p3divide() {
    int offset = (int) Math.floor((endRow + currentRow) / 2);
    this.newTask = new P3MandelbrotApp(width, height, max, offset, endRow);
    endRow = offset;
}
```

4.5 Testing and performance analysis

4.5.1 Test setup and methodology

The Mandelbrot set fractal generation problem provided an interesting example application for evaluating the framework. It is a fairly computational intensive task and gives a graphical way of seeing how the problem is being solved by multiple devices.

For this analysis, we used eight Nexus 9 devices running Android 5.1.1 Lollipop. Each device is equipped with a dual-core 2.3 GHz processor, 2 GB of RAM, 16 GB of internal flash storage, Wi-Fi 802.11 a/b/g/n/ac, dual-band, Wi-Fi Direct, Bluetooth 4.1, GPS, NFC, accelerometer, gyroscope and a non-removable 6700 mAh battery.

The main objective of these test is to evaluate the performance of the framework with a variable number of devices. To do this, the Mandelbrot set fractal generation application is ran with different number of devices and it is tracked the time between the start of the task until the final image with the full Mandelbrot set fractal is completed. For each configuration of devices, the same test is performed three times and the final score of that particular configuration of devices is the average time of the three tests performed. The devices configurations tested are networks of one, two, four, six and eight. The test with only one device serves as a benchmark for the Mandelbrot set generation application running sequentially.

4.5.2 Minimum granularity analysis

First of all, it was studied how the minimum granularity allowed for the problem affects the performance, in other words, how little the tasks can get to be divided with other nodes of the network. In this particular problem, the fractal is calculated row by row, from the top to bottom. Therefore, the minimum granularity is measured in the minimum number of rows left to process.

To test this minimum granularity, it was tested with a configuration of six devices different values. The Figure 4.4 show this variation of the minimum granularity for this problem in this configuration.

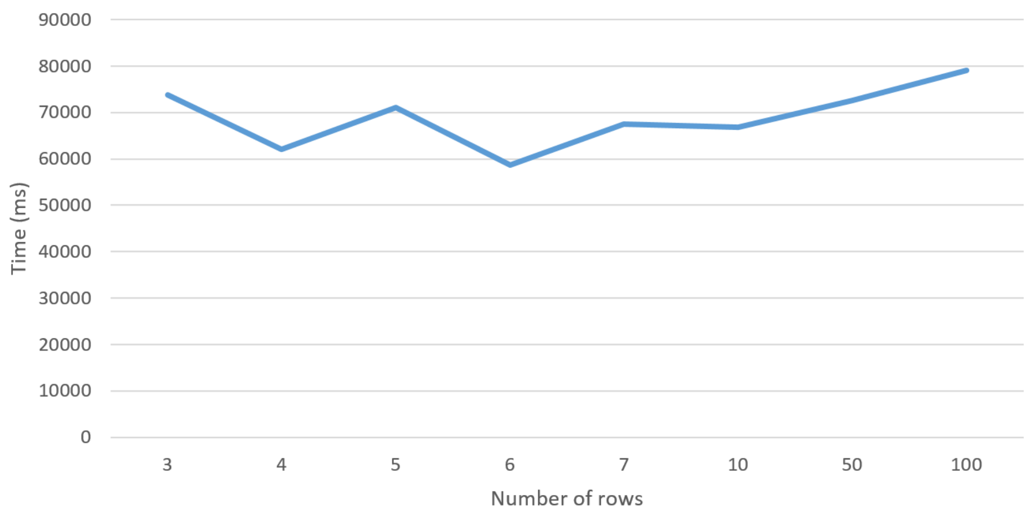


Figure 4.4: Minimum granularity allowed variation in the Mandelbrot set fractal generation application with six devices.

The x-axis is the value of minimum granularity and the y-axis is the time, in milliseconds, that took to complete the task. We can easily see that between one and fifty rows the times are fairly close to each other, with a more distinct increase in time beyond the value of fifty rows. The lowest value of running time observed was for a value of minimum granularity of six rows. As a result of this analysis, the performance analysis test was performed with a value of minimum granularity of six rows.

4.5.3 Running time analysis of Mandelbrot set generation

Table 4.1 shows the running times of all the performed tests in the conditions described in Subsection 4.5.1 and the Figure 4.5 shows the graphical representation of those results.

	Devices				
Attempts	**1**	**2**	**4**	**6**	**8**
1	255394	117456	68656	76842	75815
2	260005	117639	80280	68194	70259
3	278531	117475	88937	76030	76070
Average	264643.3	117523.3	79291	73688.67	74048

Table 4.1: Mandelbrot task app results.

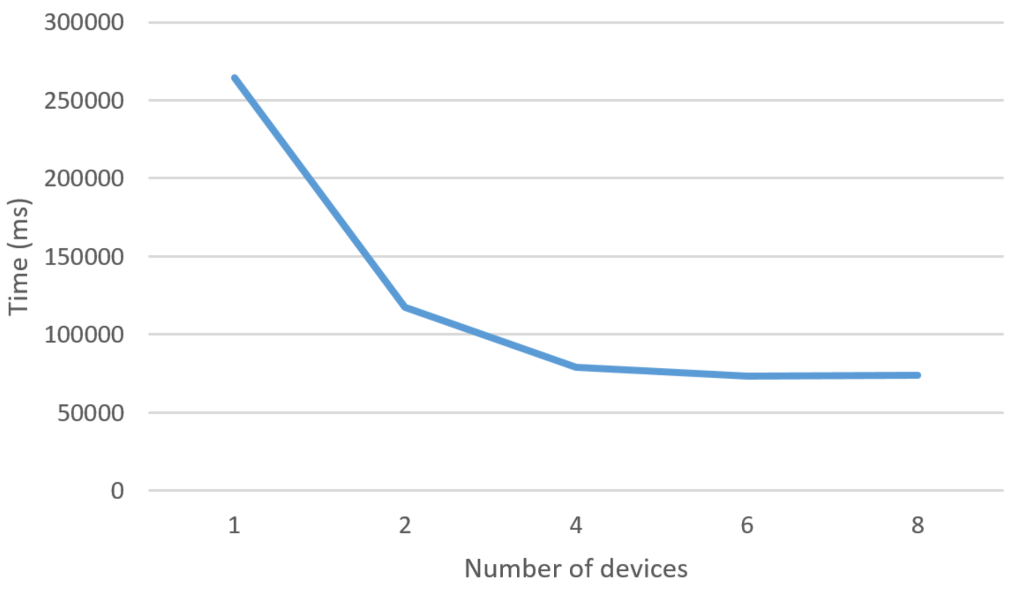


Figure 4.5: Mandelbrot task results chart.

At a first glance, the chart in Figure 4.5 indicates that the application has very good time improvements from one to two devices and from two to four devices, and with less improvements with six and eight devices. This indicates a logarithmic speed up for this particular problem. For deeper analysis of these results, some metrics were calculated.

4.5.3.1 Performance metrics

To enhance our result analysis, we calculated two different metrics: speedup and efficiency [22]. The objective of calculating these metrics is to have a better understanding of how problem is being solved in consideration of the hardware used.

The **speedup** metric measures the ratio between a sequential execution time (in this case the execution with one device) and the parallel execution time (more than one device). The equation 4.4 shows how the speedup $S(n)$ is calculated, where $T(1)$ is the sequential execution time for one device and $T(n)$ is the parallel execution time for n devices.

$$S(n) = \frac{T(1)}{T(n)} \tag{4.4}$$

The **efficiency** metric measures how efficient the application is at using the resources available. It is a ratio between the speedup value and the amount of resources available (in this case, the number of devices). The equation 4.5 shows how the efficiency $E(n)$ is calculated, where $S(n)$ is the speedup value for n devices and n is the number of devices.

$$E(n) = \frac{S(n)}{n} \tag{4.5}$$

Using the measured times (Table 4.1), Table 4.2 show the results for the metrics described before.

	Devices				
Metric	**1**	**2**	**4**	**6**	**8**
Speedup	1	2.251836515	3.337621336	3.591370903	3.573943028
Efficiency	1	1.125918257	0.834405334	0.598561817	0.446742878

Table 4.2: Performance metrics for the Mandelbrot app results.

As expected, the speedup increases greatly with two and four devices and for six and eight devices it stabilizes. The speedup and efficiency metrics helps to demonstrate that by dividing the task by two devices, it has a superlinear speedup. Superlinear speedup is defined by $\frac{T(1)}{T(n)} \geq n$, or in a simpler way, $S(n) \geq n$. In this case, the speedup is a superlinear speedup when the its value is greater or equals to the number of devices used. Therefore, when that happens the efficiency is also greater than one. Although these values indicate a superlinear speedup, we cannot confirm a superlinear speedup with just six test runs. To confirm this result, further testing with different sizes of the problem and with different configuration of devices would be needed. Beyond that, with four, six and eight devices the efficiency decreases steadily. With this data, for this particular problem and this particular size, we can say that a reasonably efficient setup for solving this problem is to use a network with a size of between two and six devices. Beyond that, although the problem is being solved slightly faster, the resources used to do that are being used less efficiently and could be used in a different task.

Chapter 5

Conclusion and future work

5.1 Summary

Throughout this thesis we worked towards a final objective of designing and building a working prototype of a framework for building parallel distributed applications on mobile devices in edge clouds. That was the main goal of this investigation work. And after all of the previous analysis, it is safe to say that this main goal was achieved successfully. Building on previous works of the P3 - Parallel Peer-to-Peer programming model by Licínio Oliveira, Luis Lopes and Fernando Silva [11] [9] and the P3 storage system by João Paulo Magalhães [10], with the necessary changes it was possible to create a new framework dedicated to the development of parallel distributed application on devices grouped in edge clouds, the P3-Mobile.

To achieve the main goal of building a working prototype of the P3-Mobile framework, the work was divided in a subset of smaller objectives that when put together build towards the main objective. We can summarize these smaller objectives and summarize this thesis in the following points:

Identify the problems of working with mobile devices in edge clouds: The original P3 programming model was designed towards its use in the Internet. But since the date of publishing of that thesis, the environment has changed and mobile devices and the Internet of Things begins to grow exponentially. With that paradigm change, new problems arise. In this case, the main issue was that mobile devices use wireless communications that are more unreliable and unstable than wired connections (like Ethernet, for example) and, as the name indicates, mobile devices tend to move from one place to the other. Also, by grouping mobile devices in edge clouds, the entire environment is self-contained by the devices themselves, which means that no additional infrastructure, like servers, could be used to support operations.

Design a network model without a P3 portal: Directly tied to the fact that in the edge cloud we could not have additional infrastructure to work as server (like the P3 portal in the original P3 model), a new solution to remove the need of the use of the P3 portal needed to be created. This was one of the main challenges of this work. By moving the responsibilities of the P3 portal to the root node of the network tree in P3-Mobile and using the Network Service Discovery (NSD) technology, we created a simple and working solution for this problem.

Design an architecture compatible with the Hyrax middleware: This thesis was integrated in the Hyrax project that in a separate work is creating a middleware solution as described in section 1.5 of this thesis. Our goal was to design a flexible architecture that, when the middleware is ready for usage, P3-Mobile could be easily ported to it. This was achieved by staying close to the development process of the Hyrax middleware which lead to the services architecture described in the section 3.2. Each service can easily be rewritten to use the Hyrax middleware functionality in the future.

Implement a simple distributed storage: Based on the P3 storage system by João Paulo Magalhães [10] we implemented a simple version of this system to support the parallel processing functionality. Files are stored in different devices on the network, with copies stored across the network for redundancy in case of node failure. It provides a simple functionality that aims to be proof of concept for distributed storage with mobile devices in edge clouds.

Implement the parallel distributed model: Based on the original P3 programming model by Licínio Oliveira, Luis Lopes and Fernando Silva [11] [9], we implemented a service that manages parallel applications running on the network. This service, besides running the actual task, it accepts requests from other nodes in the network for division and sharing of the work. It is also able to recover tasks from failed devices, as described in the section 4.3.5.

Fault tolerance: As described before, because we are working with mobile devices that move from one place to the other and use wireless connections, nodes of the P3-Mobile network will eventually disappear. We analyzed in the section 4.3.3 how coordinator nodes actively monitor their coordination cell looking for dead nodes. In the section 4.3.5 we described how a parallel task running on a dead node can be recovered storing the task redundantly in other nodes, and in the section 4.3.4 we described how files of the distributed storage are stored in multiple nodes so that the file redundancy allows to recover files that were stored on failed devices.

Scalability: In a distributed system, scalability is always a concern. P3-Mobile can scale well due to the nature of its network. By organizing the network in coordination cells (and with coordination cells organized in a tree), where only coordinator nodes have extra knowledge about the cell neighborhood, the system can grow to large numbers of devices. In the case of parallel processing tasks, the scalability is also directly tied to the size of the task where problems of different sizes can benefit or not from larger and larger networks.

Performance: Although performance was not the main concern of this work, it plays an important role in every distributed system. As described in the section 4.5, very good results were achieved in terms of performance with the given task. Speedups reached more than the double in a particular setting, giving promising performance indicators.

5.2 Future Work

This could be just the starting point of a new framework specialized in distributed applications for edge clouds. The objective of this thesis was to create a working prototype of a framework like this but there is a lot of room for improvement.

Integrate with the Hyrax middleware: When it is ready, the Hyrax middleware will provide an excellent abstraction for several communications related aspects. Porting P3-Mobile to use the Hyrax middleware will give it a better link service that can use a greater array of technologies (Wi-Fi, Wi-Fi Direct, Bluetooth). The Hyrax middleware also provides different types of logic networks with proper formation, routing and discovery and a channel based overlay with membership and discovery. P3-Mobile was designed already having the Hyrax middleware in consideration, so both systems are compatible with each other.

Better load balancing: The current implementation of the P3-Mobile uses, for load balancing, a very simple approach but it could be that is not the most efficient. Currently, load balancing relies on when doing something in the network, go through a random coordinator node of the current coordination cell. By choosing a random coordinator node, in the long term, it is expected that all coordinator nodes of a cell have handled a similar amount of requests. Because this is a random process, it is not guaranteed that the requests are equally distributed through all the coordinator nodes. This approach also does not take in consideration if an unusual amount of requests gets assigned to a coordinator node while other might be free. This requires a deeper study with different types of problems and different types of strategies for load balancing.

Different structure of logic network: In this thesis the network implemented was based on the network of the original P3 work, a network composed by coordination cells organized in a tree. Perhaps other network structures might be more efficient than the one used on this thesis. This is another aspect that can be further studied in more detail.

Better fault tolerance: As described before, P3-Mobile has in consideration that node might fail without warning and it is ready to handle a few missing nodes. Stored files or running tasks can be, in the vast majority of cases, recovered because files and tasks have always redundant copies in different nodes. Is is possible that in some extreme failure cases, recovery might not be possible. This is also another aspect of P3-Mobile that can be studied in further detail, looking for different and more efficient solutions to cover more failure scenarios.

Better performance: Although the results obtained in the tests done are very satisfactory, it is still possible to fine tune the system to increase performance. In future work, different types of problems could be studied to try to comprehend how problems are being solved and how to make the parallel processing faster with them.

5.3 Final considerations

In conclusion, it is clear that the main objective of this thesis was successfully achieved, functional prototype of a framework for developing distributed applications in mobile devices in an edge cloud.

The framework successfully abstracts complex details like network management or the parallel processing work sharing logic, leaving the developer free to create distributed applications in an easy and simple way.

In the practical tests, the framework demonstrated that has reasonably good performance on a problem that, when solved on a single device, takes a great amount of time. The code for the task is easy to read and understand and the developer is in control of how its task is solved and divided.

Although with a lot more future work needed to make it suitable for real world applications, P3-Mobile is a step towards the future of mobile computation of devices connected in edge clouds.

Bibliography

[1] U. B. of Labor Statistics, "Long-term price trends for computers, tvs, and related items on the internet." http://www.bls.gov/opub/ted/2015/long-term-price-trends-for-computers-tvs-and-related-items.htm, 2015. [accessed 2016-06-14].

[2] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[3] L. Svobodova, *Client/Server Model of Distributed Processing*, pp. 485–498. Berlin, Heidelberg: Springer Berlin Heidelberg, 1985.

[4] M. D. Network, "Tiered distribution." https://msdn.microsoft.com/en-us/library/ff647195.aspx. [accessed 2016-06-14].

[5] G. F. Coulouris, J. Dollimore, and T. Kindberg, *Distributed systems: concepts and design.* pearson education, 2005.

[6] E. Exchange, "Processing power compared." http://pages.experts-exchange.com/processing-power-compared/. [accessed 2016-06-15].

[7] V. Pande and S. University, "Folding@home." https://folding.stanford.edu/. [accessed 2016-06-15].

[8] U. of California, "Seti@home." http://setiathome.berkeley.edu/sah_status.html. [accessed 2016-06-15].

[9] L. S. de Oliveira, "P3: Parallel peer-to-peer," informatics, Faculdade de Ciências da Universidade do Porto, June 2003.

[10] J. P. F. de Magalhães, “Um sistema de ficheiros distribuído para uma arquitectura peer-to-peer,” informatics, Faculdade de Ciências da Universidade do Porto, April 2004.

[11] L. Oliveira, L. Lopes, and F. Silva, “p3: Parallel peer to peer an internet parallel programming environment,” in *International Conference on Research in Networking*, pp. 274–288, Springer, 2002.

[12] G. Appliances, “Connected dishwashers.” `http://www.geappliances.com/ge/connected-appliances/dishwashers.htm`. [accessed 2016-06-15].

[13] J. Rodrigues, J. Silva, R. Martins, L. Lopes, U. Drolia, P. Narasimhan, and F. Silva, *Benchmarking Wireless Protocols for Feasibility in Supporting Crowdsourced Mobile Computing*, pp. 96–108. Cham: Springer International Publishing, 2016.

[14] B. Clark, “The differences between bluetooth 4.0 and wi-fi direct you need to know.” `http://www.makeuseof.com/tag/the-differences-between-bluetooth-4-0-and-wi-fi-direct-you-need-to-know/`. [accessed 2016-06-18].

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[16] Oracle, “Class copyonwritearraylist.” `https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/CopyOnWriteArrayList.html/`. [accessed 2016-06-18].

[17] Oracle, “Interface runnable.” `https://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html/`. [accessed 2016-06-18].

[18] H.-O. Peitgen and P. H. Richter, *The beauty of fractals: images of complex dynamical systems*. Springer Science & Business Media, 2013.

[19] W. MathWorld, “Mandelbrot set.” `http://mathworld.wolfram.com/MandelbrotSet.html`. [accessed 2016-06-23].

[20] R. L. Devaney, “The fractal geometry of the mandelbrot set.” `http://math.bu.edu/eap/DYSYS/FRACGEOM/FRACGEOM.html`. [accessed 2016-06-23].

[21] wikiHow, “How to plot the mandelbrot set by hand.” `http://www.wikihow.com/Plot-the-Mandelbrot-Set-By-Hand`. [accessed 2016-06-23].

[22] F. Silva and R. Rocha, “Parallel and distributed programming: Performance metrics.” `http://www.dcc.fc.up.pt/~fds/aulas/PPD/1112/metrics_en.pdf`. [accessed 2016-06-18].