Alma Mater Studiorum · University of Bologna

MSc in Engineering and Computer Science

INTEGRATING MOBILE DEVICES INTO A SCALABLE GRID COMPUTING ARCHITECTURE DESIGNED TO PERFORM DISTRIBUTED COMPUTATIONS

Thesis in

CONCURRENT AND DISTRIBUTED PROGRAMMING

Supervised by:
Prof. ALESSANDRO RICCI

Presented by:
ALESSANDRO TALMI

Third Graduation Session
Academic Year 2021-2022

*Do the hard work, **especially** when you don't feel like it.*

Contents

List of Figures

Acknowledgements

I would like to acknowledge and give my warmest thanks to my supervisior **Professor Alessandro Ricci** for guiding and supporting me through the stages of writing this project.

The completion of this work would not be possible without my girlfriend, **Elisa Tronetti**; thanks for your continuous support during this time, your love is what kept me going.

A special thanks also goes to:

- **Luca Rossi**, **Angelo Filaseta**, **Tommaso Bailetti** and, once again, Elisa Tronetti, for helping me to perform the prototype's real-world experiments
- **Elisa Mencucci**, for helping me to keep my English grammar in check while writing this thesis
- **Professor Gylfi Þór Guðmundsson**, for igniting this project idea in me through his teachings in his excellent course in Reykjavík University

Last but not least, I would like to thank my family and my friends for the support during this long, difficult, but rewarding journey.

I'm really grateful to all of you from the bottom of my heart, thanks for making me the person I am today.

Abstract

The idea of Grid Computing originated in the nineties and found its concrete applications in contexts like the SETI@home project where a lot of computers (offered by volunteers) cooperated, performing distributed computations, inside the Grid environment analyzing radio signals trying to find extraterrestrial life.

The Grid was composed of traditional personal computers but, with the emergence of the first mobile devices like Personal Digital Assistants (PDAs), researchers started theorizing the inclusion of mobile devices into Grid Computing; although impressive theoretical work was done, the idea was discarded due to the limitations (mainly technological) of mobile devices available at the time. Decades have passed, and now mobile devices are extremely more performant and numerous than before, leaving a great amount of resources available on mobile devices, such as smartphones and tablets, untapped.

Here we propose a solution for performing distributed computations over a Grid Computing environment that utilizes both desktop and mobile devices, exploiting the resources from day-to-day mobile users that alternatively would end up unused.

The work starts with an introduction on what Grid Computing is, the evolution of mobile devices, the idea of integrating such devices into the Grid and how to convince device owners to participate in the Grid. Then, the tone becomes more technical, starting with an explanation on how Grid Computing actually works, followed by the technical challenges of integrating mobile devices into the Grid. Next, the model, which constitutes the solution offered by this study, is explained, followed by a chapter regarding the realization of a prototype that proves the feasibility of distributed computations over a Grid composed by both mobile and desktop devices. To conclude future developments and ideas to improve this project are presented.

Chapter 1

Introduction

This chapter provides a high-level introduction attempting to offer an easy-to-read definition of the topics that are faced in this work as well as the end goal to reach; here it will be discussed what Grid Computing is, the evolution of mobile devices, the idea of integrating such devices into the Grid and, last but not least, how to convince device owners to participate in the Grid.

1.1 What Grid Computing is

Grid Computing is a type of Distributed Computing where the resources of many computers, connected by a network, are used in combination to reach a certain common goal.

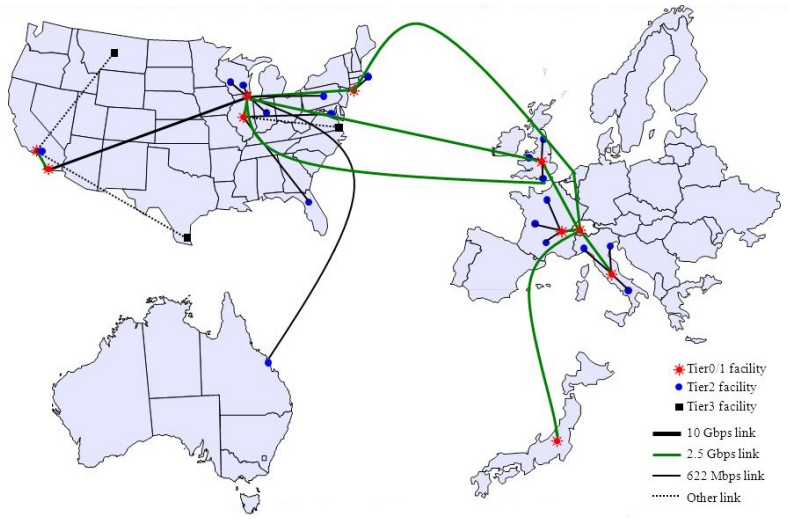


Figure 1.1: Example of a Grid - iVDGL, the Globus project [1]

Compared to traditional **Distributed Computing** such as Cluster Computing, **Grid Computing focuses on large-scale resources** with **geographically dispersed nodes** that also tend to be **more heterogeneous regarding their hardware and software**; this is also due to the fact that while a Cluster is only composed of computers entirely dedicated to perform the tasks requested inside the Cluster (thus coming from an investment and a subsequential setup of the Cluster), **a Grid is composed by machines offered mostly on voluntary basis** by regular day-to-day users scattered around the planet.

Another important aspect that differentiates Grid Computing from Cluster Computing is how the last one tends to be more focused on a particular task while **the Grid is designed to be a general-purpose tool**.

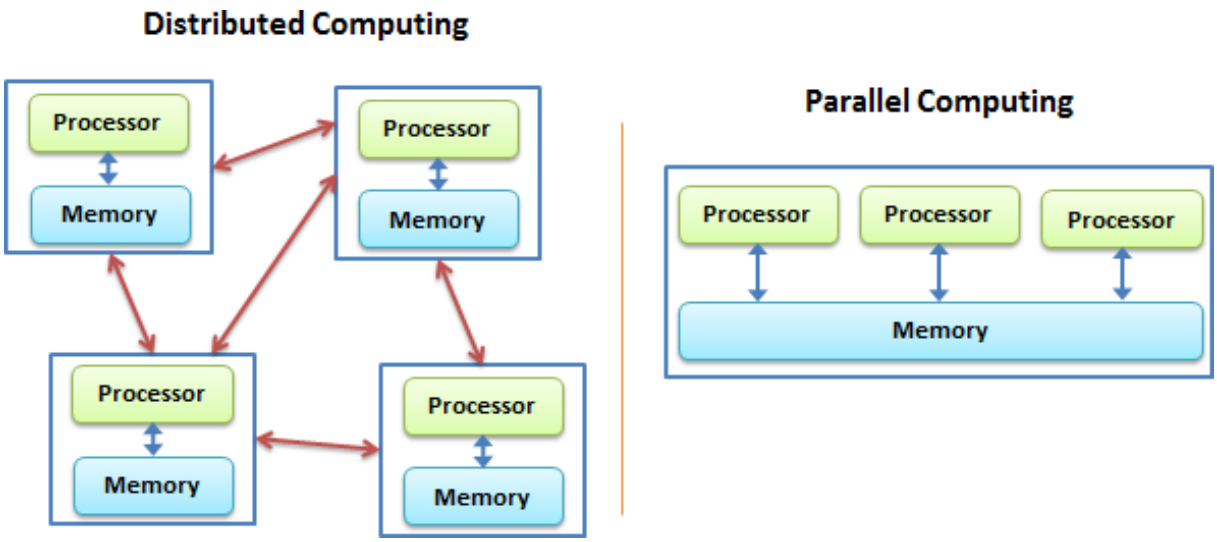


Figure 1.2: High-level comparison of Distributed Computing and Parallel Computing [2]

For certain applications, Grid Computing can be seen as a special type of Parallel Computing that distributes computation among the nodes connected to the Grid utilizing their resources; this approach is in contrast with the traditional notion of a supercomputer, which has many processors connected by a local high-speed bus. From this difference in approach comes the strength of performing parallel computations in a distributed Grid environment: resources from the machines connected to the Grid can be quickly gathered to perform a task and, after completion, they can be dismantled just as quickly, removing the necessity of maintaining a highly expensive supercomputer; while this is, in a certain way, also true for Cluster Computing, **the strength of Grid Computing lies in its ability to reach much greater levels of scalability with relative ease**.

1.1.1 The "Grid problem"

The "Grid problem" is defined as "***flexible, secure and coordinated resource sharing among dynamic collections of individuals, institutions and resources***" [3] While being a powerful tool, Grid computing brings with it some **inherited challenges** that are a byproduct of the flexibility that the grid aims to offer; while designing a Grid system, **diverse usage models have to be taken into consideration**, ranging from single user to multiuser and from performance sensitive to cost sensitive and hence embracing issues of quality of service, scheduling, co-allocation, resource discovery, security and accounting. Quoting a passage from the foundational article "*The anatomy of the Grid: enabling scalable virtual organizations*" by Ian Foster, Carl Kesselman and Steven Tuecke [3]:

> "*The sharing that we are concerned with is not primarily file exchange but rather direct access to computers, software, data and other resources, as is required by a range of collaborative problem-solving and resource-brokering strategies emerging in industry, science and engineering. The sharing is, necessarily, highly controlled, with resources providers and consumers defining clearly and carefully just what is shared, who is allowed to share and the conditions under which sharing occurs.*"

1.1.2 History and applications using the Grid

Before being referred to as "Grid Computing", the idea of connecting geographically distant computers in order to share resources was known as "Utility Computing"; the term is based on the idea of seeing computing as a public utility, analogous to the phone system. This field was then renamed to "Grid Computing" in the early 1990s, based on the metaphor of making computational power as easy to access as an electric power grid. Thus, the nomenclature of this technology shifted its focus, representing first the goal of providing public service that could be used by anyone, then an easy-to-scale enabling technology for gathering a vast number of computational resources. In 2007, the term "Cloud Computing", which is conceptually similar to the definition of Grid Computing, came into popularity. Nowadays, Grid Computing is often associated with the delivery of Cloud Computing systems.

Volunteer computing (having people that voluntarily offer their machines to contribute to a Grid) became relevant first in 1997 with **distributed.net**, a project that aims to solve mathematical complex and resource-consuming problems using Grid computing; then, in 1999, the **SETI@home** project used voluntarily offered computers performing distributed computing over a Grid in order to Search for Extraterrestrial Intelligence (hence the acronym "SETI") analyzing radio signals. Another important example of Grid Computing application can be found in the **LHC Computing Grid**; this project, created by the CERN organization, is used to support the Large Hadron Collider. This Grid is capable of analyzing the approximately 27 TB of data generated by the collider every day.

1.2 Mobile devices and their evolution

Today, the mobile devices' panorama is dominated by smartphones and tablets. Before the introduction of such devices, the market was very different, with the predecessors of the smartphones: PDAs.

1.2.1 From PDAs to smartphones

A **Personal digital assistant** (PDAs) is a mobile device that acts as a personal information manager. These type of devices were the **first attempt at providing the capabilities of a computer in a relatively small mobile device** (hence they were also known as handheld PC). A PDA device typically features:

- **A display** (and possibly physical buttons, depending on if the specific model uses a touch display);
- **Audio capabilities** (with the possibility of using it as a portable media player);
- **Telephony** (acting as a mobile phone);
- **Internet connectivity** (only via Wi-Fi);
- **A web browser**;
- **Bluetooth connectivity**.

In 1984 Psion released the first PDA: the Organiser I; the term PDA though came to existence in 1992, once Apple released the Apple Newton. PDAs started to include telephony capabilities in 1994 with the IBM Simon; this device is particularly important, since it can be considered the first smartphone.

Figure 1.3: Example of PDA device - Compaq iPAQ 3650

Although PDAs had a considerable share of the mobile market (but still dominated by traditional mobile phones), **in the mid-2000s PDAs started to be replaced more and more with modern smartphones** until they completely replaced them. Today the term "personal digital assistant" has found a new meaning in the definition of virtual assistants based on speech synthesis (ex: Amazon's Alexa).

Contemporary mobile devices' market is not remotely comparable to the market of the PDAs era, reaching an **exponentially greater number of smartphones and tablets units sold**. Despite this, some experts think that the market is saturated, and it is reaching its peak [4]. One important aspect of today's market is the fact that **the number of smartphones currently sold is far greater than the number of PCs sold** (that is actually gradually declining) as shown in *figure 1.4*.

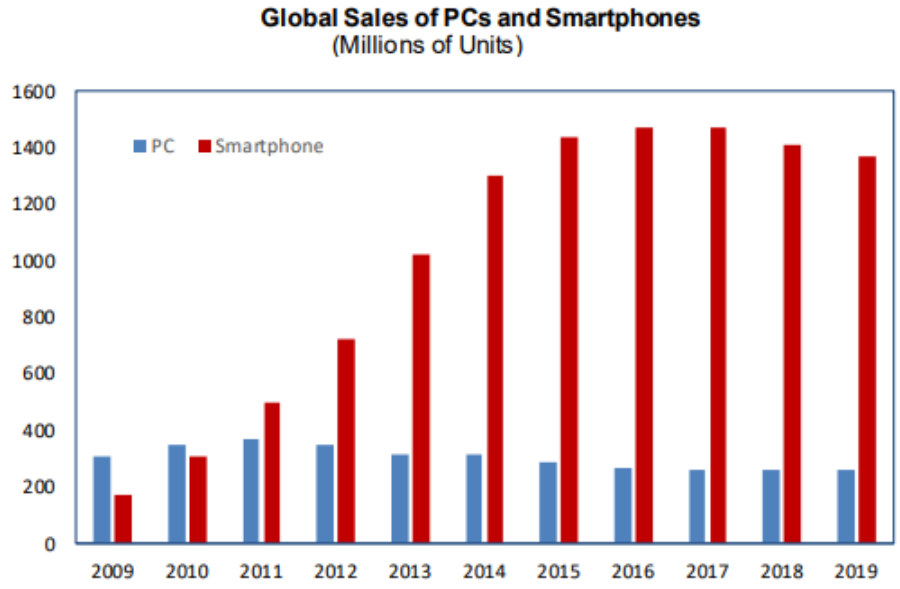


Figure 1.4: Global Sales of PCs and Smartphones from 2009 to 2019 [4]

1.2.2 Technological progress

Hardware has had a general improvement in the computational world and mobile devices' technological capabilities are no exception; as can be seen in *figure 1.5*, PDAs had very poor performances able to only mildly satisfy the limited use cases of such devices.

System	Entry year	CPU	Storage	Connectivity
Casio Cassiopeia E-125	2000	150 Mhz NEC VR4122	32 MB RAM, Compact Flash TII	56K modem via CF
Compaq iPAQ 3650	2000	206 Mhz Intel StrongARM	32 MB RAM, Compact Flash TII expansion	56K modem via CF
HP Jornada 548	2000	133 Mhz Hitachi SH3	16 MB RAM, Compact Flash TI	56K modem via CF
Compaq iPAQ 3975	2002	400 Mhz Intel X-Scale	64 MB RAM, Secure Digital Card	Built-in Bluetooth

Figure 1.5: Example of technological capabilities of 2000s PDAs [5]

To better understand the **technological gap between devices from the year 2000 and today's smartphones**, figure 1.6 provides a comparison between these three examples:

- Compaq iPAQ 3650 (*figure 1.3*), the best PDA from *figure 1.5* among the ones that could also be connected to the internet;
- Power Mac G4 - M5184 (EMC 1864), the best possible configuration in the Power Mac G4 line from the year 2000;
- Xiaomi Redmi Note 9 Pro, a 2020 mid-range (as far as value for money) smartphone.

System	Year	CPU	RAM	Storage	Gigaflops
Compaq iPAQ 3650	2000	Intel StrongARM SA-1110: 1x Core 206 MHz	32 MB	16 MB	n.d.
Power Mac G4 - M5183 (EMC 1864)	2000	PowerPC G4 (7400) dual: 2x processors with 1x Core 500 MHz each	256 MB	40 GB	20
Xiaomi Redmi Note 9 Pro	2020	Qualcomm Snapdragon 720G: 2x Cores 2.3 GHz + 6x Cores 1.8 GHz	6 GB	128 GB	435

Figure 1.6: Comparing a 2020 mid-range smartphone to devices from the year 2000

Despite information about the number of Gigaflops (billion floating point operations per second) for the Compaq iPAQ 3650 is not available anywhere, it can be confidently said that its value is far less than the 20 Gigaflops that the Power Mac G4's CPU was capable of; considering that the Redmi Note 9 Pro is capable of executing a number of floating point operations per second that is 21.75 times higher than the Power Mac G4, a modern smartphone outmatches a PDA by an enormous factor. Available RAM and Storage have also made a significant step forward, with the chosen smartphone having 187.5 times more RAM than the PDA and 8000 times more memory available for the storage. The gap has also to take into consideration other technologies of modern smartphones, such as Wi-Fi and 4G connections as a standard (with newer models moving towards 5G connections) and a multitude of sensors, as well as high resolution cameras. It is apparent that **modern smartphones are on a vastly different technological level compared to old PDAs, significantly outmatching also computers that coexisted with them**.

1.3 Enormous untapped potential: integrating mobile devices into the Grid

Now, with a general understanding of what Grid Computing is and how mobile devices evolved over the years, the **main topic of this work** can be discussed: **integrating** (using volunteer computing) **mobile devices into Grid Computing alongside desktop computers exploiting an enormous quantity of computational resources that lay unused in the pockets of billions of users**.

1.3.1 Idea: strength in numbers

As seen in *figure 1.4*, the market has an enormous number of smartphones sold, thus resulting in an abundance of available devices that billions of users use every day (tablets

have also to be taken into consideration).

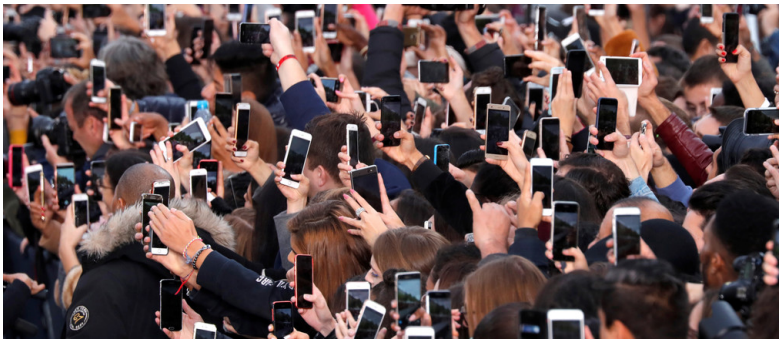


Figure 1.7: Smartphones are now used daily on a regular basis

Despite smartphones and tablets are not on the same level (in terms of hardware capabilities) with current computers, **it is possible to exploit the availability of a vast number of devices as a leverage to compensate the performances of such mobile devices**. This concept becomes **increasingly relevant** also **considering the tendency of users to progressively use less and less desktop computers in favor of their mobile devices**.

1.3.2 Previous works and how limitations evolved

A number of **previous works discussed the idea and all of them agreed on the potential of exploiting mobile devices resources**; quoting directly from the 2003 article "*Should one incorporate Mobile-ware in Parallel and Distributed Computation?*" by Mustafa Sanver, Sathya Prya Durairaju and Ajay Gupta [6]:

> "*At first glance, an individual mobile device may not have sufficient capacity and computational power for the integration. However, if we can harness the aggregated mobile power instead of individual power and consider exponential-rise of mobile units marketed and significantly evolving mobile technology, then one may conclude that it is worth the effort using this vast, pervasive, and untouched computational source for parallel computation.*"

While recognizing the potential behind this concept, previous works pointed out two important **arguments against integrating mobile devices into Grid Computing**:

- **Challenges inherently linked with adding mobile devices to the Grid**, that this work will discuss in detail later in *section 3.1*;

- **Technical limitations**, that will now be discussed in order to show how (mostly) they no longer apply, and thus this idea can be realized.

1.3.2.1 Mobile devices competitive performances

The first limitation that was pointed at was the **poor performances of mobile devices**; while this was **certainly true for PDAs**, as discussed in *section 1.2.2*, **today's smartphones and tablets do offer resources that are vastly more powerful than what was available in the past**.

Wireless connectivity is also much stabler, faster and wildly available, especially considering the possibility of limiting participation in the Grid only while the device is connected to a Wi-Fi network.

T**he mobility factor** of such devices is also **not a limiting issue** since one could argue that a volunteer that moves its device in space (thus possibly interrupting the connection to the Wi-Fi network) can be **assimilated to a desktop volunteer that turns off its computer or is suddenly isolated from the network**, reducing the mobility to a challenge inherently tied to Grid Computing.

1.3.2.2 Mobile users common behavior

Users current common behaviors **solve two limitations addressed in the past**:

- **Mobile devices need to be always on to contribute to a Grid**
  While this was certainly a problem in the past, *in today's normal usage of mobile devices* (and with the enhancement of the capabilities of batteries), **it is normal to maintain a mobile device on 24/7**, alternating cycles of running only on battery power and cycles of recharging said batteries.

- **Mobile devices do not possess enough resources to both satisfy its use cases and, simultaneously, contribute to a Grid.**
  As explored before, resources in devices are wildly more available today, compared to the past. It could be argued that while capabilities are increased, also resource usage to run computational tasks has increased; while this is true, it must be addressed how those **devices are unused for relatively long periods of time** (especially in the case of tablets). There is also the fact that **the typical mobile user uses its**

device for relatively light-weighted computational tasks such as browsing the internet and interacting with social media applications, leaving still a great portion of resources unused.

1.3.2.3 Economically sustainable Internet connections

In the 2000s, internet access was progressively rising, but **Internet service providers still offered their services to consumers with a connection according to a consumption plan**; that means that the more data a user consumed, the more expensive the connection fee becomes every month. The problem with the usage of such consumption plan was that **it made it expensive for end users to voluntarily offer their devices to participate in a Grid**.

Such limitation does not exist anymore since the most common way of accessing the Internet today is by a subscription model. It could be argued that mobile users still get a limited amount of traffic data available each month, therefore they could be reluctant to invest it in the participation in a Grid but, as mentioned before, the participation in the Grid could be limited to only while the mobile device is connected to a Wi-Fi network (that commonly has unlimited access to the Internet).

1.3.2.4 Standardized mobile market

Another important limitation pointed out in the past was poor software interoperability and security of mobile devices.

At the time, the poor software interoperability was the result of how heterogeneous the mobile market was, especially regarding the operative systems that ran on such devices:

- Palm OS;
- Microsoft Windows Mobile;
- EPOC;
- Linux;
- Newton;
- QNX.

Having a multitude of OSes resulted in making the process of integrating the mobile devices into the Grid much more difficult, requiring to manage multiple implementations compatible with a specific OS. While the OSes panorama was also relatively wide in the first years of the smartphones and tablets era, **over time the market slowly converged into having primarily two main operative systems: Android and iOS**.

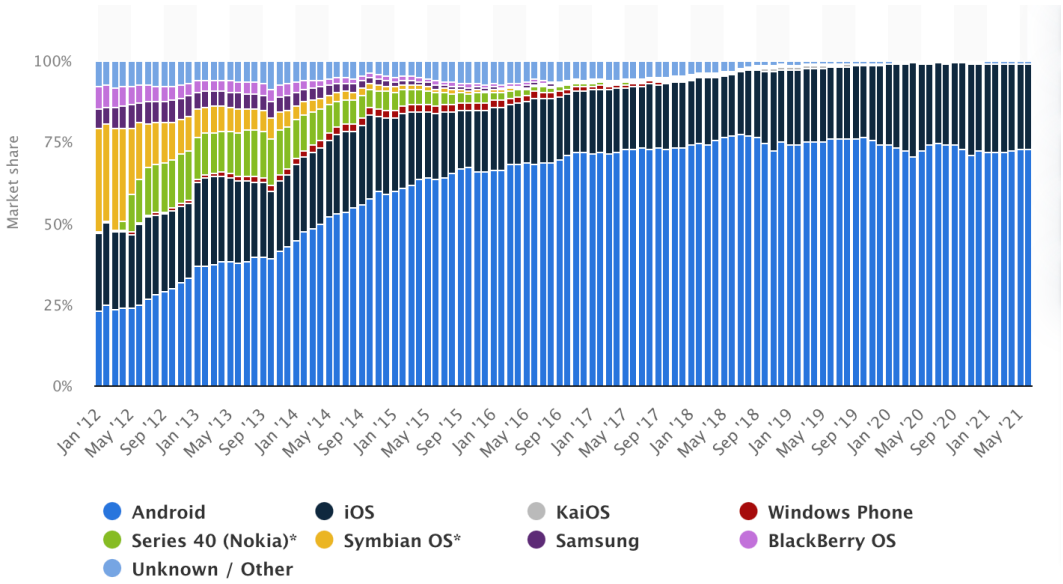


Figure 1.8: Mobile OSs Market Share from January 2012 to June 2021 [7]

It must also be mentioned that today there are frameworks that allow to have a singular codebase for creating native applications for both Android and iOS, making the maintenance process even simpler.

On the security side, having only two real competitors that constantly update their OSes makes the current situation more favorable than the 2000s (while of course additional security measures have to be used).

1.3.2.5 Not only mobile: devices transparency principle

As mentioned at the beginning of *section 1.3*, the vision of Grid Computing that this work wants to discuss does not only include mobile devices, but continues to also use desktop computers to expand the capabilities that can be achieved.

The **coexistence of both mobile devices and computers in the Grid** at the same time makes necessary to introduce a **device transparency principle**: a node that participates in the Grid is just **characterized by the resources it has to offer, independently of the**

nature of the device. If this principle is respected, while performing tasks inside the Grid, the complexity of having different types of devices disappears, making the process transparent to someone that requests a service from the Grid.

1.4 How to convince device owners

Since this work assumes the voluntary contribution to the Grid by users, said people have to be convinced to participate in the project; alternatively, as much as the project allows to reach a great level of scalability, without any physical machines, no tasks can be performed, making it useless. This section discusses ways and prerequisites to incentivize users to participate in the dynamically allocated Grid.

1.4.1 Zero-effort configuration

In order to **reduce as much as possible the barrier of the active effort** needed to participate in the Grid, users should be presented with a configuration as easy as possible, especially considering that the vast majority of users do not want to be bothered with technical details. The goal is possibly to just **require an initial guided installation** (using easy to access methods such as applications stores and executables) **and then never require direct input from the contributor**.

While the **possibility of partially customizing the setup** must be offered, it should be **as minimal and easy as possible**, offering possibilities like enabling/disabling participation while using mobile data, while the device is alimented using only the battery and possibly a scheduling, depending on the time of the day.

1.4.2 Unnoticeable impact: smart resource management

It is important that, while contributing to the Grid, **the end user does not experience slowdowns** resulting in a worsening of the user experience **during the normal activities** that the user performs; in order to achieve this, the application running on the device must implement a **smart resource management system** that adjusts resource usage depending on the current utilization of them by the user. If this mechanism is not well implemented, there could be a risk of the user stopping its contribution because of the discomfort created.

1.4.3 Security and privacy

One of the most important factors is **granting security and privacy inside the Grid**. This is important for multiple reasons:

- **The end user wants to feel safe** participating in the project, being sure that their devices and personal data do not get compromised;
- **The entities that request the services offered by the Grid also want to feel safe** that their sensitive data (including the results of the computations performed inside the Grid) will not be accessible to non-authorized third parties;
- **The stakeholders of the project do not want to be involved in legal actions** deriving from a breach in security.

This requires a particular effort on security in every component of the systems and in the communications that are performed over the network.

1.4.4 Fair share business model

Contributors to the Grid must be **economically rewarded in order to be incentivized to offer their devices** to the project. Hence, a **fair share business model** will be useful to this purpose: **users receive a fraction of the earnings** obtained by the payments of the requestors of Grid services; **the reward is proportional to the contribution made by the device** that participated to offer the requested service.

This business model allows the owners of the Grid to make a profit (while needing only to sustain the costs of the machines used to connect the machines of the requestors and the nodes of the Grid) while also rewarding contributors of the Grid for the usage of their computational resources (while not needing an active effort other than the initial setup).

Chapter 2

How Grid Computing works

This chapter elaborates on Grid Computing details. The first section provides lower-level details on the architecture that governs the functioning of Grid Computing. The second and last section conclude this chapter presenting the intrinsic challenges of Grid Computing; these challenges must be faced even in a Grid composed of only desktop computers.

2.1 Layered Grid Architecture

Following the work of Ian Foster, Carl Kesselman and Stevel Tuecke [3], the Grid Architecture identifies fundamental system components, and indicates how these components interact with one another. This **Grid Architecture is first and foremost a protocol architecture**, since the **interoperability among any potential participant is the central issue** and thus requires the definition of common protocols. Protocols govern the *interaction* among components in the Grid and not the *implementation*, maintaining local control.

> "*Why are protocols critical to interoperability? A protocol definition specifies how distributed system elements interact with one another to achieve a specified behavior and the structure of the information exchanged during this interaction. This focus on externals (interactions) rather than internals (software, resource characteristics) has important pragmatic benefits.*" *[3]*

In order to provide abstractions to interact with the Grid and develop applications that use it, application programming interfaces (**APIs**) and software development kits

(**SDKs**) must also be provided; these are built on top of the protocols. Together, APIs, SDKs and the protocols in the architecture constitute a **middleware**.

The architecture is organized in a **layer structure** (*figure 2.1*), where components within each layer share common characteristics but can build on capabilities and behaviors provided by any lower level. **The number of protocols must be contained**, focusing on Resource and Connectivity protocols.

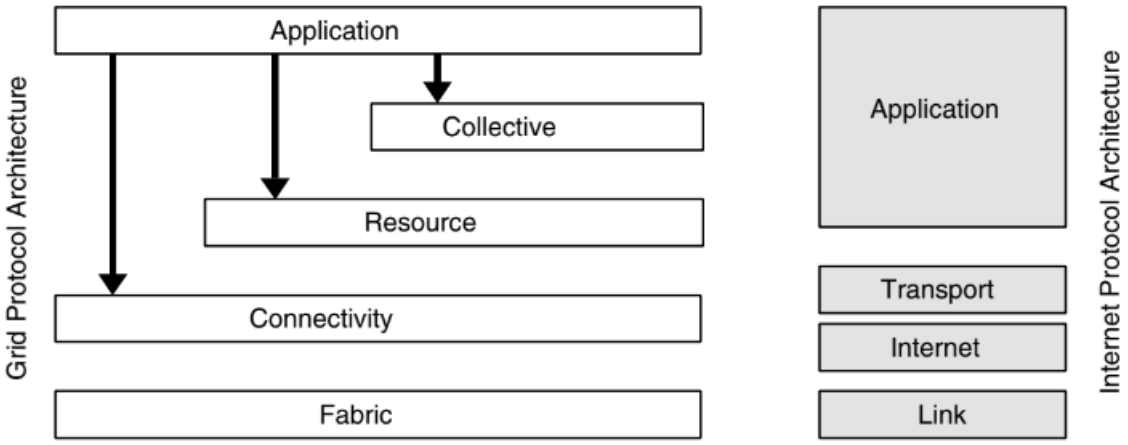


Figure 2.1: Layered architecture relationship to Internet Protocol (IP) architecture. [3]

2.1.1 Fabric layer

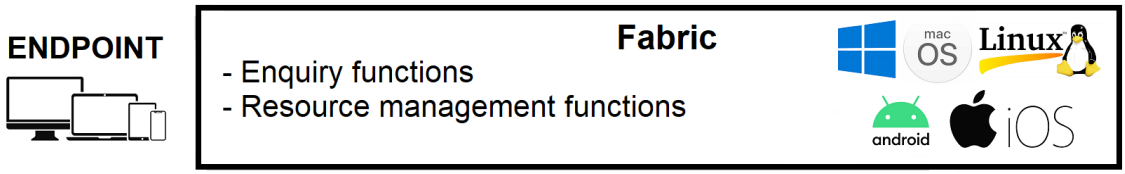


Figure 2.2: Layered Grid Architecture - Fabric layer

This is the layer that **implements the local, resource-specific operations that occur on specific resources** (whether physical or logical) **as a result of sharing operations at higher levels**. The external access to such resources is mediated by protocols defined in the Grid, while **the internal workings of this layer depend on the specific implementation that runs on a machine**; this organization results in a relatively tight interdependence between the operations defined in this layer and the operations defined in higher layers.

Operations in this layer should be as simple and minimal as possible in order to easily extend compatibility with as many devices as possible; said operations must involve:

- **Enquiry functions**

  Allow to discover the Node inside the Grid, as well as describe its resources and status. Resources description provides info about software and hardware capabilities, while status handling offers mechanisms to enquire current load and queue state.

- **Resource management mechanisms**

  Allow performing operations using the resources offered by the Node, such as access to storage, computation and network.

2.1.2 Connectivity layer

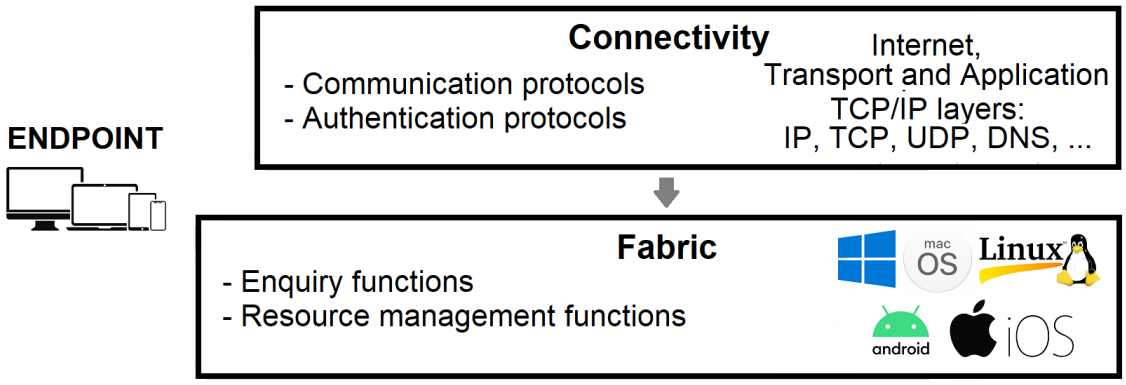


Figure 2.3: Layered Grid Architecture - Connectivity layer

This layer **defines core communication and authentication protocols that are used for network transactions inside the Grid** in order to enable the exchange of data among fabric layer resources.

- **Communication protocols**

  This protocols handle transport, routing and naming. Here, technologies are mapped to the TCP/IP stack, in particular to the Internet, Transport and Application layers; communication is built on already well-established protocols like IP, TCP, UDP, DNS, etc...

- **Authentication protocols**

  Mechanisms regarding authentication must possess the following characteristics:

  – *Single sign-on*: users authenticate just once, and then they have access to a multitude of resources;

  – *Delegation*: a user should allow executing authorized instructions on their machine;

  – *Integration with various local security solutions*: the layer should include security solutions provided by the local machine;

  – *Enable users' control over authorization*: the user must be able to change authorizations regarding the resources that their machine offers.

2.1.3 Resource layer

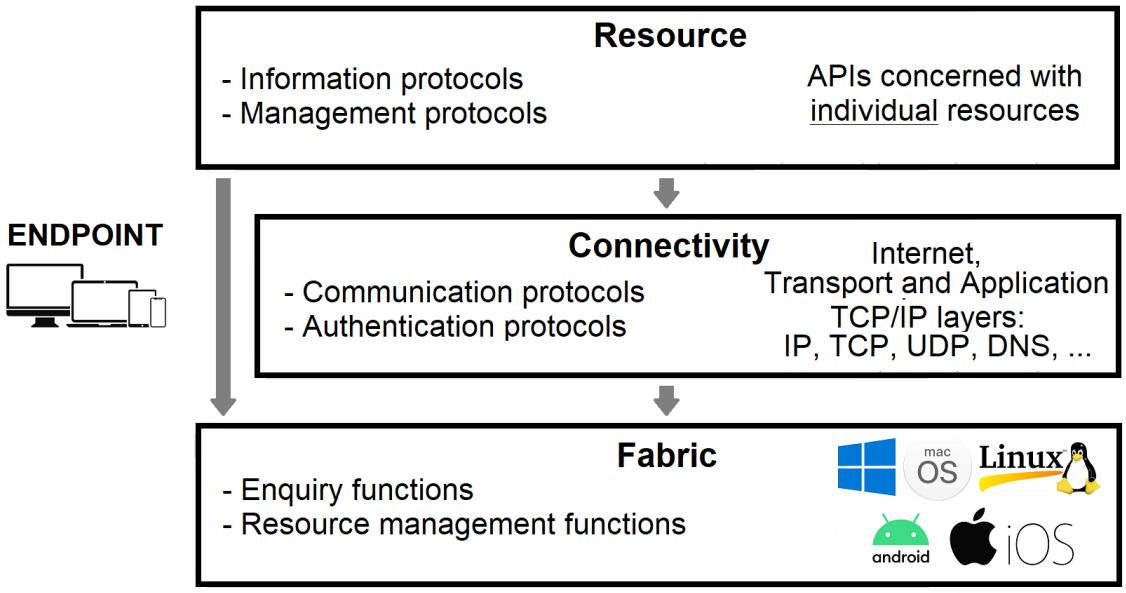


Figure 2.4: Layered Grid Architecture - Resource layer

In the Resource layer, the primary focus is **the secure negotiation, initiation, monitoring, control, accounting and payment of *individual* resources**; this means that **this layer is not concerned with the global state the Grid is in** (that will be handled in the Collective layer), but it just deals with communicating with a single machine. Here, protocols should be limited to a small set that captures fundamental mechanisms of sharing across different resource types while, at the same time, not overly constraining higher protocols defined in the Collective and Application layers.

Implementations of the Resource layer **rely upon functions defined at fabric layer in combination with previous layer's communication and authentication protocols**, resulting in creating **APIs** used to utilize this layer's capabilities. Two protocol classes are defined at this level:

- **Information protocols**

  Provide information about the structure and state of a resource (ex: configuration, current load, usage policy, etc...).

- **Management protocols**

  Deal with the negotiation process performed in order to access a shared resource (resource requirements, operations to be performed and applying usage policies).

2.1.4 Collective layer

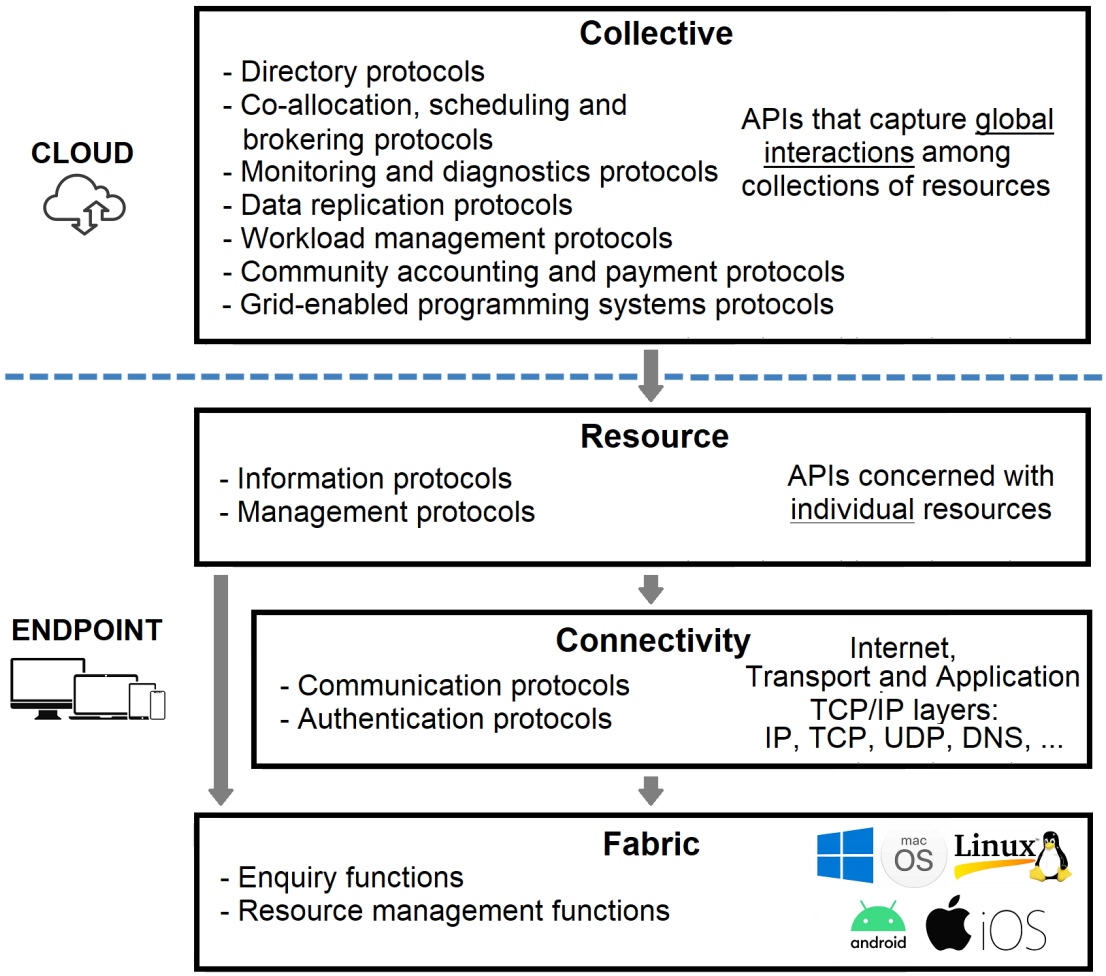


Figure 2.5: Layered Grid Architecture - Collective layer

Here the main focus shifts from the management of a single specific resource to **dealing with questions that are global in nature, capturing interactions across collections of resources**. Previous layers dominated the realm of a single machine (with its resources) acting as an endpoint, but this layer changes context, becoming a **cloud-based domain**.

If the limited number of protocols at the Resource layer are well-designed, **this layer can offer a wide variety of sharing behaviors** that build upon those protocols. Such sharing behaviors depend on what capabilities the Grid wants to offer. Those behaviors, that are then accessed by using APIs, *can* include:

- **Directory protocols**
  Used to discover and query resources by name and/or attributes such as availability, type and workload.

- **Co-allocation, scheduling and brokering protocols**
  With such protocols, resources are assigned and allocated for a specific purpose in order to execute a scheduled task.

- **Monitoring and diagnostics protocols**
  Through these protocols resources are monitored for failure, overload, external attacks, etc...

- **Data replication protocols**
  In order to maximize performances, these protocols manage data replication to increase reliability while reducing response time.

- **Workload management protocols**
  Used to solve situations where, through information obtained using discovery protocols, an excessive workload is detected.

- **Community accounting and payment protocols**
  Such protocols collect information about Grid usage; with such information payments involving resources usage can occur.

- **Grid-enabled programming systems protocols**
  Lastly, these protocols offer familiar grid-enabled programming models and paradigms to be used to perform computations inside the Grid.

2.1.5 Application layer

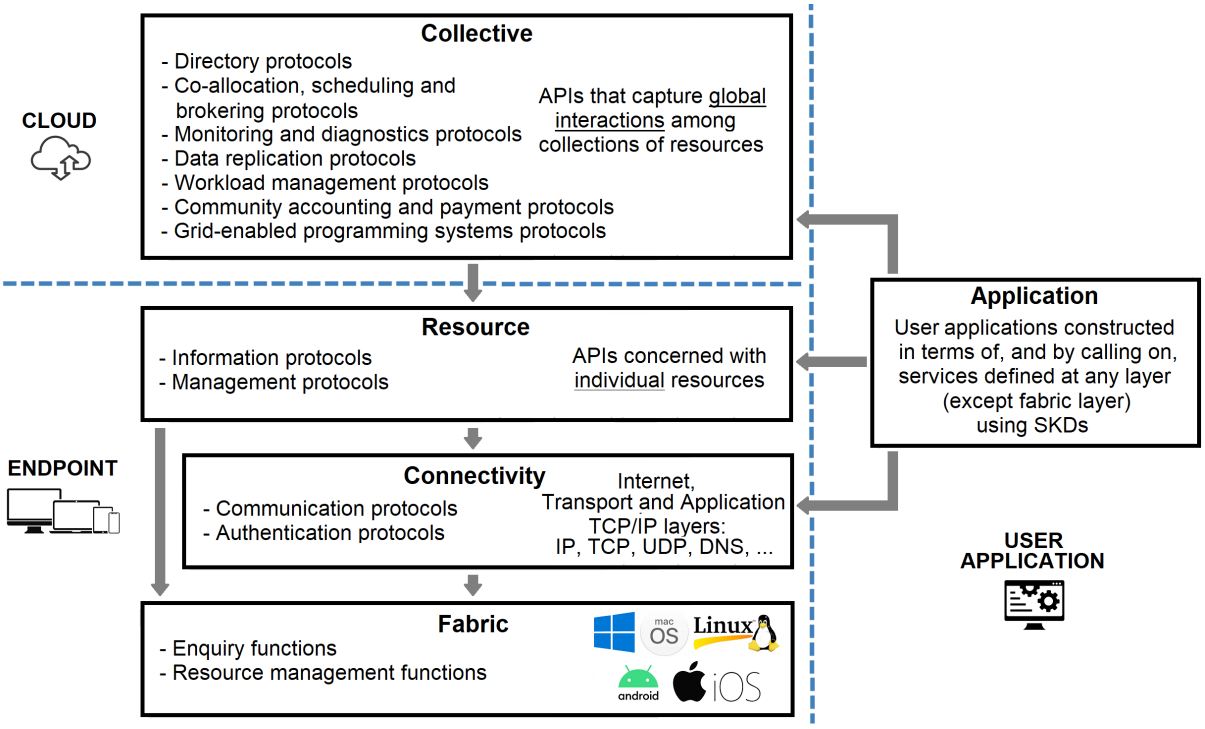


Figure 2.6: Layered Grid Architecture - Application layer

The final layer constitutes **all the applications that use services offered by the Grid**; the **desired behavior for the user application is constructed in terms of, and by calling on, services defined at any layer** (except fabric layer), accessed via an SDK provided by the implementor of the Grid; through a combination of services defined in the other layers, software development and, possibly, collaborating with third party libraries, more complex behaviors can be built in order to satisfy specific use cases.

2.2 Intrinsic challenges of Grid Computing

While designing a Grid system, intrinsic challenges have to be taken into consideration; that means that **these challenges exist in Grid Computing independently of the inclusion of mobile devices or operating with just desktop computers**. This section presents a list of the main challenges that a Grid system must face.

2.2.1 Interoperability

As mentioned in *section 2.1*, interoperability among any potential participant is the central issue when designing a Grid Architecture, requiring a meticulous focus on designing reliable and versatile protocols for interaction.

Unless the system is designed with the limitation of utilizing a set of identical machines, which is not desirable, machines connected to a Grid operate with **different hardware**; this differentiation requires a **layer of abstraction** that separates the functioning of the Grid from the interaction with a single physical node that will use a specific implementation of the standardized Fabric layer.

Anyway, not only do machines offer different hardware, but **they also differentiate each other in general by the resource they offer**. In the context of distributed execution of computational tasks, for example, nodes offer **different programming languages** that they are capable of executing. This becomes a problem for interoperability since a requestor might want to execute some task with an implementation in a specific programming language to respect a performance requirement or just because that language offers useful libraries for that particular task. While an implementation of a Fabric layer using a language that is capable of running on almost any device (such as JavaScript) is certainly possible, this creates a great limitation in the Grid's use cases related to distributed computing while at the same time constraining the architecture to a specific technology, which is bad while designing any software.

Obviously, an important interoperability issue comes from the fact that **nodes can utilize any operating system**, implying different ways of installing the software necessary to run the node's tasks and, most importantly, a differentiation with the interaction with the file system and the security mechanisms specific for that particular OS.

Another resource where nodes differ is the network capabilities that they can offer. Even though this is not a differentiation, in a narrow sense, that comes from the machine in itself but from the environment where it operates, this still becomes an interoperability problem inside a Grid system; for this reason the architecture has to be designed taking into consideration that different nodes will offer **different levels of reliability and speed for their network connection**, requiring mechanisms for handling errors, disconnections and distributing work accordingly to the network's speed of the node.

2.2.2 Security

Security is an essential prerequisite in every system and the Grid makes no exception. First of all, in order to access to the Grid, it is necessary to have an authentication mechanism (as discussed in section *2.1.2*); said mechanism must **uniquely identify every user/entity alongside their unique machines inside the Grid**. While being fundamental in order to grant security, this mechanism is also required by the resources discovery and selection problem (*section 2.2.4*).

Having a unique identity given by the authentication is also required for another important security problem: respecting resources **access policies** defined by the nodes providers. Said policies define what resources are offered by a machine, as well as who is authorized to use them. Mechanisms that regulate the enforcement of access policies must also take into consideration the dynamic nature of said policies, since they can be changed by a user at any time.

Unique identification becomes even more important when it is applied to requestors of the services of the Grid. In order to grant the security of volunteers that offer their machines, **when a user or an entity request a service, it must be trustable (verified) and accountable**. While it is important that also contributors are verified and accountable, this is a lesser concern considering that they can only act as passive entities.

Regarding **privacy**, once authentication is granted, it is also important that communications among entities inside the Grid are encrypted. The encryption is put in place in order to avoid attacks from malicious entities that could intercept packets.

Finally, countermeasures have to be taken in order to **avoid the execution of malicious code**; despite the accountability of uniquely identified users helps to reduce this possibility, it is fundamental to implement security measures for avoiding the access to everything that is outside the defined access policies.

2.2.3 Scalability, geographical distribution and load balancing

The main strength of the Grid is its ability to scale, becoming a central principle that guides the design of the system. In such a system, it is possible to identify two main contexts where it is necessary to design **horizontal scalability**:

- *The nodes' context*, i.e. the ability for the system to work with a varying amount of machines offered by volunteers;

- *The cloud's context*, i.e. the capability of the cloud architecture to scale in order to support efficiently the nodes' context.

It is clear the cloud's context is the foundation of the system, therefore here it is necessary to design a **load balancing mechanism for the cloud services**, increasing/decreasing the number of machines running such services, depending on the number of nodes that are participating in the Grid in a certain moment.

Moreover, this load balancing has to implement a **geographically-aware behavior**, since the nodes are scattered around the planet without a fixated location; this results in the necessity of creating new replicas of the cloud services' servers in a geographical location that takes into consideration not only the costs of running such server, but it also **optimizes the proximity with as many nodes as possible in order to reduce latency**. Once a server is instantiated, it is necessary to let the new instance take charge of the handling of part of the nodes connected to the overloaded service. Vice versa, if there is not an excessive amount of traffic, an instance might be removed and its nodes given to another previously instantiated server. Since the load balancing introduces a problem of reachability for both the cloud services and the nodes (the dynamically instantiated services with a geographically-aware behavior lack a fixed address), it is also necessary to introduce **discovery services** that, on the contrary, have a static address and act as intermediaries, providing the address of the required service instances.

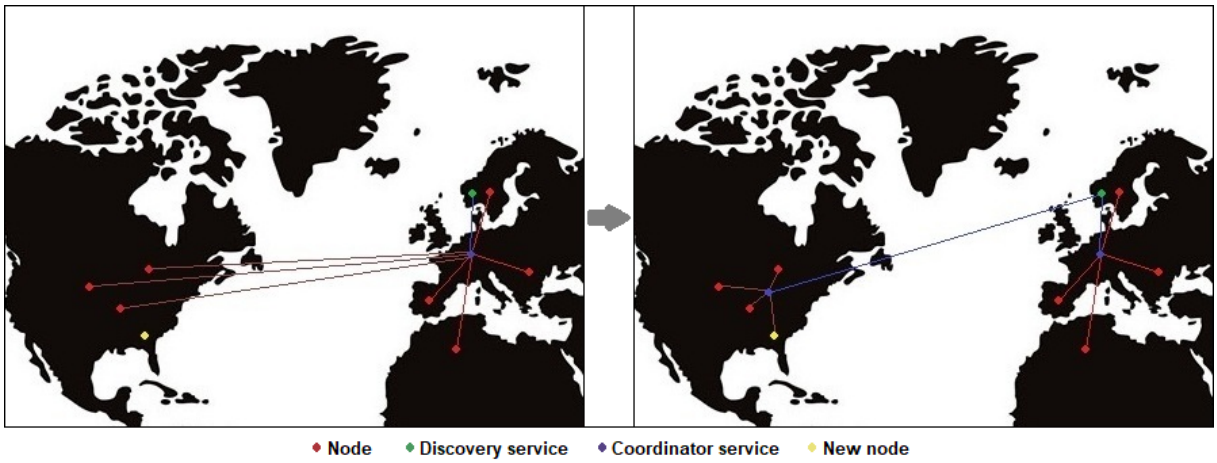


Figure 2.7: Example of geographically-aware load balancing

On a final note, this dynamic instantiation in the nodes' context has to be considered while designing the authentication mechanism (discussed in *section 2.2.2*), requiring for it to work even if something changes with the servers' status.

2.2.4 Resources discovery and selection

Not only does the Collective layer (*section 2.5*) handles global interactions among collections of resources, but, first and foremost, it has the responsibility of managing the discovery of resources and the selection of the correct ones to perform a task.

Cloud services devolved to this duty build a **map of all the nodes connected to the Grid, keeping track of also which resources they can offer**. With this knowledge, said services can **connect a requestor to the desired resources**, eventually **choosing the best fit among multiple possible resources of a certain type** that satisfy the condition for a certain task.

The map of the nodes connected to the Grid has to be designed to be stored in multiple server instances in order to respect the scalability discussed in the previous section.

2.2.5 Scheduling, fault tolerance and Quality of Service (QoS)

An operation that needs to be performed inside the Grid needs a **scheduling mechanism** in order to handle the execution of said operations. Since the operations are executed in a distributed environment prone to errors, **replication of data** needs to be performed among multiple nodes; this is done to increase **fault tolerance** since any node can fail at any moment. Increasing the replication factor to strengthen the tolerance to errors necessarily means a worsening of the performances inside the distributed system and vice versa. The correct replication factor highly depends on the tasks that the Grid needs to perform and how performances are valued compared to availability.

All these factors (along with what discussed in the previous sections) influence the quality of service that the Grid can offer. Every design decision for the Grid has to be made with the intent of **maximizing the QoS**, increasing performances as much as possible while, at the same time, offering good fault tolerance, scalability, and security.

Chapter 3

Integrating mobile devices into the Grid

After having discussed how Grid computing works and the intrinsic challenges of such technology, now it is time to move towards the specific objective of this work: integrating mobile devices into Grid Computing. First, the challenges specifically derived from the inclusion of such devices in the system will be discussed. Then, the focus will shift to a discussion of some architectures proposed by previous works. This will be useful since the solution that this work proposes takes some ideas from those architectures.

3.1 Challenges derived from including mobile devices

While the intrinsic challenges described in *section 2.2* remain still valid, the inclusion of mobile devices in the Grid arises new issues that have to be taken into consideration while designing the system. Some challenges have been resolved by the evolution of technology and the new behaviors adopted by mobile users (*section 1.3.2*) but other ones still need to be dealt with.

3.1.1 Network instability

Compared to a computer, **a mobile device is afflicted by network instability more often**. This issue is linked to the **mobility factor** [5] that comes with such devices, creating a **more dynamic and less reliable environment connection-wise**.

Aside from Wi-Fi connection (that still can suffer from instability but is generally stabler), the **main source of instability** comes from **mobile data connection**. Enormous progress in mobile data connection technologies has been made, but still the reliability

suffers, especially in use cases where having a stable connection is required. Recently, **5G technology is slowly emerging as the new technological leap**. But, as other works have observed [8], 5G is still in its infancy and, while offering **great performances under optimal condition**, it is affected by **greater instability compared to 4G**.

While discussing properties about mobile data technology, it is important to first understand the concept of handoff:

- **Horizontal handoff**: occurs when the connection switches from one repeater to another. In the case of 5G this happens more often, since the technology requires a greater number of repeaters in order to work because of the different wave lengths used.

- **Vertical handoff**: occurs when the networking technology changes. This is still particularly a problem for 5G connections since, being less stable, they often require a switch to 4G ones.

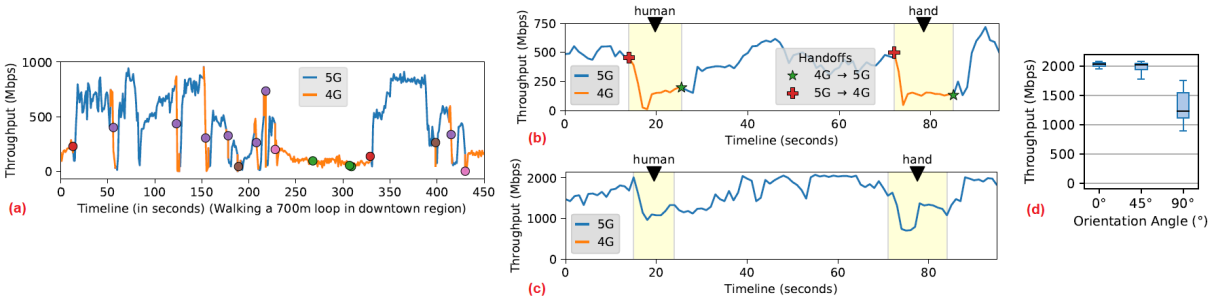


Figure 3.1: 5G's throughput under different circumstances [8]

Here are a few more relevant factors affecting the quality and performance of a mobile network:

- **Traveling speed**

  Moving through space causes fluctuation in the stability of the connection, with usually better performances while standing still, a slight worsening while moving at walking pace and, finally, a progressive drop in performance with higher speeds. *Figure 3.1 (a)* shows how a 5G network throughput varies over time by simply walking for 700 meters; in this graph, dots represent horizontal and/or vertical handoffs, showing how unstable mobile connections (especially 5G ones) are.

- **Obstacles**

  Having obstacles between the device and the connection source significantly alters performances. In simple terms, 4G utilizes signal waves in a more sparse way; on the other hand, 5G utilizes waves that are highly focused, requiring a trajectory to reach the device, whether being on LoS (line-of-sight, i.e. there are no obstacles between the device and the repeater) or on NLoS (non-line-of-sight, i.e. the repeater finds an indirect trajectory that bounces the signal using objects). *Figure 3.1 (b)* shows how throughput varies in response of physical obstacles (a human body and a hand) when there are no efficient trajectories to reach the device; on the other hand, *figure 3.1 (c)* shows the same phenomenon when there are many optimal trajectories for the signal waves.

- **Placement relative to the connection source**

  Even the way that the device is placed in space relatively to the connection source alters performances; generally speaking, this is due to the placement of the mobile device's antenna but, in the 5G case, it also depends on the orientation angle relatively to the source (*figure 3.1 (d)*).

- **Weather**

  Also, atmospheric conditions slightly alter the efficiency of mobile connections, reaching up to 30% reduction in throughput speed during rainfall on a 5G mobile network [8].

Since it will become the standard technology in the following years, a lot of examples using 5G have been used but, even if with different severity, the same instability issues also apply to the other mobile data connection technologies. Three approaches (in ascending order of complexity) can be used to deal with the mobile devices network instability:

- **Allowing contributing to the Grid only while using a Wi-Fi connection**.

- **Limiting contribution while on mobile data connection only to a few specific use cases** that do not require stability.

- **Implementing smart context-aware mechanisms** that take the decision to allow or not mobile data contribution based on a combination of technology type, throughput,

movement in space (using accelerometers) and possibly current weather condition (using GPS).

Considering that, currently, **mobile connectivity data plans still have limited (although increasing) traffic available each month**, not all users may want to dedicate it to the Grid contribution; hence, **the second approach is probably the best one**, since it expands use cases and capabilities while not wasting effort on a slightly useful feature. Finally, active contribution to the Grid using mobile data increases battery consumption, which will now be discussed in the following section.

3.1.2 Battery consumption

Having devices that run on battery and are not connected to a reliable power source can be a source of problems for the Grid. **Sudden disconnections or failures have to be monitored anyway** since devices connected to a power source may be subjected to a sudden interruption of energy from the energy source or, more in general, it can incur in any kind of error. **The problem with mobile devices is more about the increased possibility of unplanned interruptions due to energy exhaustion of the battery**. There are **five main factors** that **affect power usage** in mobile devices [9]:

1. **Hardware** (CPU, GPU, memory, storage, display, sensors);
2. **Signaling and networking modules** (cellular network, Bluetooth, hotspot, Wi-Fi, GPS, FM radio);
3. **Software** (operating system, background applications)
4. **Usage patterns** (calling, internet browsing, social networks usage, gaming, music playback, video playback, running heavy applications);
5. **Other factors** (inter-device communication, heating, aging and faulty battery).

How long the battery of a mobile device lasts depends on a combination of such factors, ranging from days to (in extreme cases) only hours.

There are a set of **possible countermeasures** that limit the participation to the Grid only to devices that respect certain prerequisites that aim to mitigate this problem:

- **External power source enforcement**
  The first obvious countermeasure is to only allow devices to contribute to the Grid while connected to a reliable power source. While certainly effective and totally feasible (since commonly mobile devices tend to be charged every day), this can limit the percentage of contributors active in a certain moment;

- **Battery health status check**
  Another method is checking for the device's battery health, allowing contribution to the Grid while disconnected from an external power source only to devices that have healthy battery units.

- **Battery percentage minimum level**
  The final countermeasure is allowing battery-powered contribution only when the device's battery is above a certain battery percentage.

The **best solution** to mitigate this problem is **a combination of the three countermeasures**, allowing healthy devices to contribute while above a certain battery percentage and limiting devices with an aged battery to only contribute while connected to a power source.

On a final note, what was discussed in this section can be mostly applied also to laptops, even though people usually tend to use them while connected to a power source.

3.1.3 Mobile-specific security

As the number of smartphone increases, the number of computers used by people are gradually declining (*figure 1.4*); as a direct consequence of this phenomenon, **malicious attack efforts are now being redirected to exploiting mobile devices' vulnerabilities**. *Figure 3.2* shows how **Android has more vulnerabilities compared to iOS**. This can be explained for the following two reasons:

- **Android devices are more diffused**
  Attackers tend to attack these devices more, since the pool of individual units is larger (as shown in *image 1.8*) and thus, obtaining a profit is more likely.

- **iOS has more strict security measures** [10]
  One of the byproducts of designing Android to be able to run a vast number of

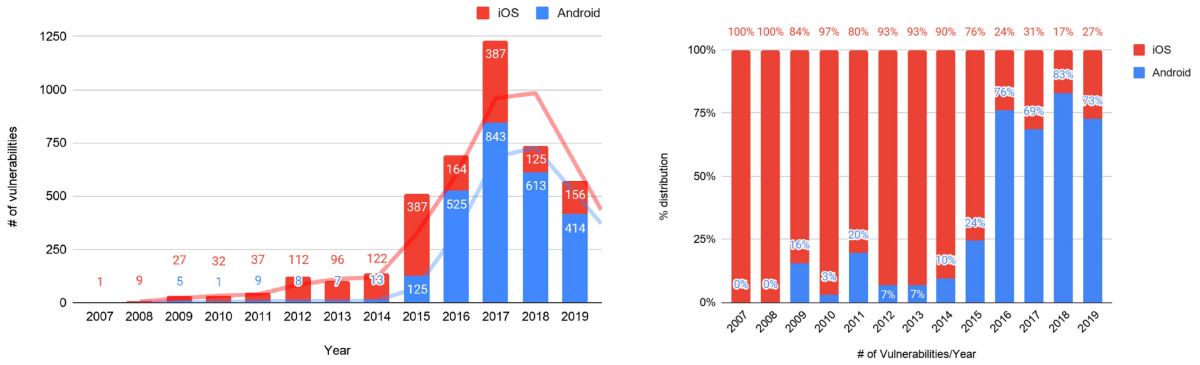


Figure 3.2: Number (left) and distribution (right) of known vulnerabilities on iOS and Android from 2007 to 2019 [10]

different hardware is the fact that compromises have to be made in order to grant compatibility. On the other hand, iOS runs on only a limited number of models, making it easier to design more effective security mechanisms.

Through the **Linux Kernel**, Android resources are managed and protected. Furthermore, Android employs an **Application Sandbox** security mechanism, meaning that each application is run with a unique ID and the **processes are executed in spaces separated from each other and from the Kernel**, therefore preventing an application from interfering with the functioning of another. Despite this, Android employs an **authorization-based mechanism for accessing resources**; while **a process that has not the correct authorizations cannot access determinate resources**, the greatest weakness of this mechanism is the fact that **authorizations are granted by the user** who is not usually really aware of the implications of such choices. Application, consequently, can launch **various types of attacks depending on the type of authorizations** that they obtained. Another weakness of Android systems is the fact that **Application provenance is not guaranteed**; the Play Store's digital certificate can be easily obtained from malicious parties since Google only requires a fee paid by a credit card to get it (and payments might be done with a stolen credit card). Applications can also be installed from third parties, making it easier to introduce malicious software that is completely unchecked.

On the other hand, **iOS** utilizes a similar Sandbox system, but **authorizations are completely managed by the OS**, removing the possibility of human errors. From the Application provenance perspective, iOS' applications are only distributed by the App Store; developers have to register, pass a **vetting process** that evaluates the Application

which, in case no harm is detected, is **digitally signed** and only then published on the store. This process makes more difficult the introduction of harmful applications.

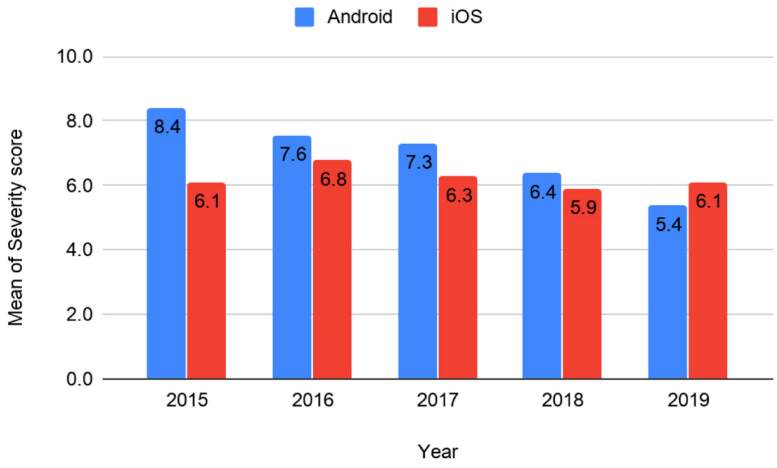


Figure 3.3: iOS and Android vulnerabilities severity score from 2015 to 2019 [10]

Even though iOS has the advantage when it comes to sheer number of security breaches, **with time the severity of Android's breaches has decreased, becoming less severe than the iOS' ones** (*figure 3.3*).

This data highlights how it is particularly important to focus on the security of the Grid's Fabric Layer (*section 2.2*) while dealing with mobile devices because, in the current panorama, they are the greatest source of security vulnerabilities, especially considering that the information inside them has great value for attackers.

3.1.4 Compatibility issues

Developing software for mobile devices tends to be more prone to compatibility issues compared to desktop environments, **due to the frequent changes that mobiles OSes undergo** in order to add new features, increase performances and enhance security. It is possible to distinguish between **two different types of compatibility** when it comes to software in relation to the version of the supporting SDK:

- **Forward compatibility**, i.e. developing software that will run even on future OS distributions.
- **Backward compatibility**, i.e. developing software that runs also on older OS distributions.

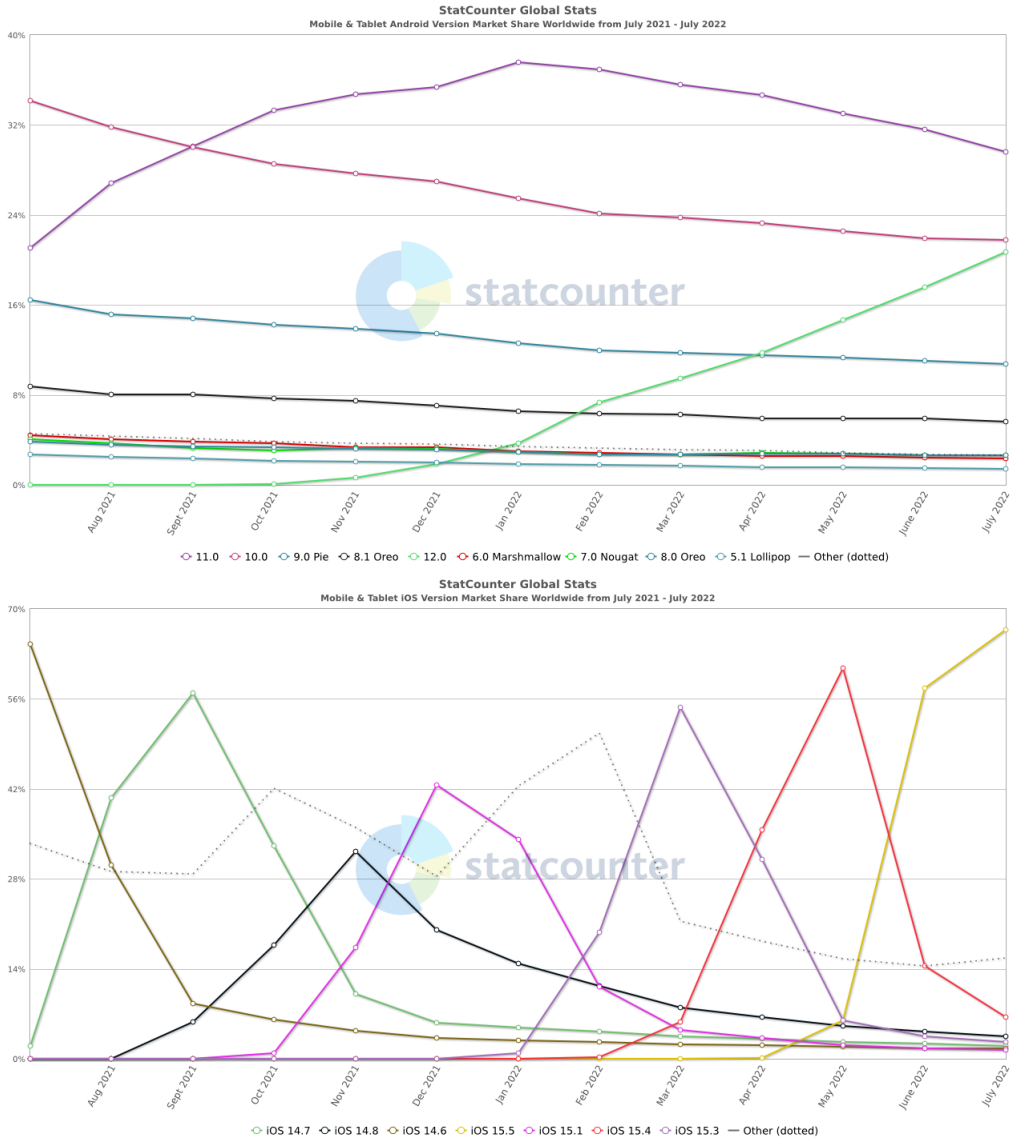


Figure 3.4: Statcounter's statistics for OSes distribution between Android (top) and iOS (bottom) devices from July 2021 to July 2022

Google's mobile OS uses an **API level system** in order to **check for the compatibility** of applications **across different versions of the OS** running on devices. As the OS rapidly evolves, so **the APIs offered in the SDK frequently change**, **introducing compatibility problems**. Android documentation states that changes in API do not threaten forward compatibility, but this turns out not to really be the case:

> *"Even if Google claims that Android apps do not suffer from forward compatibility issues, mainly because removed APIs are still kept in the framework side as hidden APIs, forward compatibility induced APIs are still encouraged to be replaced because of security and performance concerns. Furthermore, since*

> *hidden APIs are also subject to remove or change,* ***forward compatibility is*** *also* ***not fully guaranteed in practice*** *anyway." [11]*

Backward compatibility, on the other hand, **tends not to be a great concern**, resulting only in incompatibility issues due to developers setting an improperly low **minSdkVersion** flag (in order to target as many devices as possible) without actually testing if the application works on older Android's releases.

> *"Accessing backward compatibility induced APIs, without proper protection, will simply result in app crashes, giving poor experience to end-user." [11]*

Similarly, **APIs in iOS' SDK have compatibility issues between different versions of the OS**; Since changes are generally less drastic, it is **easier to test** for compatibility issues (given also the limited number of different Apple devices), but, on the other hand, iOS development suffers from a great limitation: **development of native apps can only be done on Apple's line of computers**.

Whether the development for Android and iOS devices uses native technologies or cross-platform technologies (such as React Native), **there is really no solution to this compatibility issue other than extensive testing** for every targeted version of both OSes, making the **development on such platforms more time-consuming and complex** both at the beginning and in the maintenance life cycle.

3.2 Previous works

In the past, a number of previous works has been made about this topic. While the participation of mobile devices to the Grid was theorized, it was never concretized (especially due to technical limitations). This section presents three previous works that provide interesting concepts and ideas that influence the solution offered by this work.

3.2.1 Proxy-Based Cluster Architecture (2002)

In early 2000s, *Thomas Phan*, *Lloyd Huang* and *Chris Dulan* proposed a solution to integrate devices with low capabilities (such as PDAa) into Grid computing. The work is highly influenced by the technical difficulties resulting from the technological panorama at

the time, trying to **isolate the new devices in a sub-environment that can be integrated into the Grid without affecting its inner workings**.

3.2.1.1 Architecture

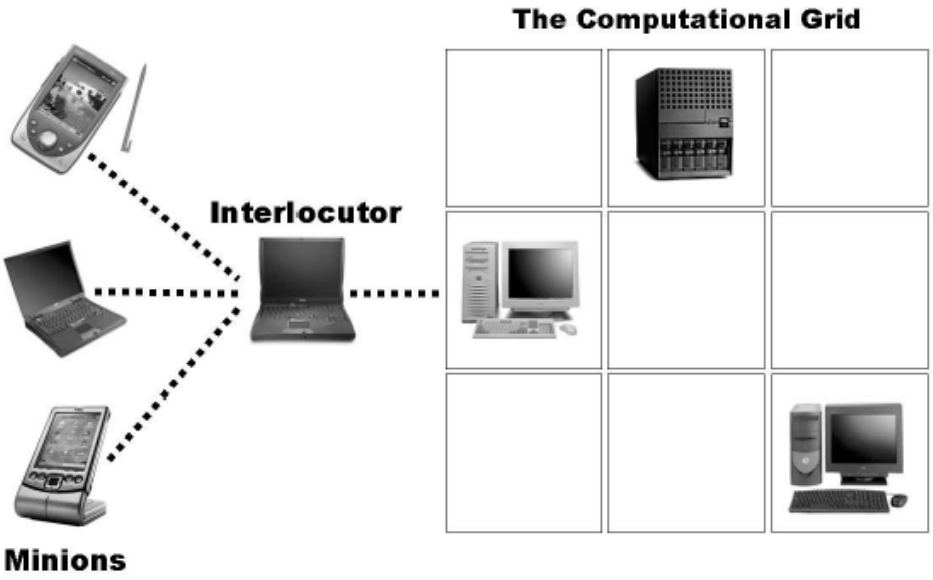


Figure 3.5: Proxy-Based Cluster Architecture [5]

This architecture revolves around the idea of having a **cluster** constituted by a number of **BASELINE** (Barely Adequate Systems Leveraging Internet NEtworking) **devices** that act as ***Minions***. Such BASELINE devices coexist (in the **same wireless network**) with a **proxy device** powerful enough to coordinate them. In this architecture, this proxy device is known as ***Interlocutor***; this machine runs a software that acts as a middleware with an existing Grid system.

> *"When a resource request arrives at the Interlocutor from a resource consumer, the request is handled by the Interlocutor. For simplicity, we proceed in this example with the assumption that the request is for CPU time to process incoming data. The interlocutor must decompose the request accordingly among its minions; [...]. After the problem has been distributed to the minions, the interlocutor waits for results and sends them back to the requester. The Interlocutor has the option of aggregating the data before responding with the result in order to amortize the cost of per-message communication overhead." [5]*

3.2.1.2 Interesting concepts and ideas

One of the strengths of such architecture is **delegation**; Minions rely entirely on the Interlocutor, making only the active decision (based on the user's will) to either participate or not to the Grid. Having such division in responsibilities **simplifies the software running on BASELINE devices** (which was the one mostly affected by compatibility issues given the number of OSes for PDAs) while moving such complexity on the stabler and more standardized Interlocutor. Another advantage gained from this architecture is the **reduction of latency**. Since the devices operate under the same wireless network, a requestor that is located anywhere on the planet has only to execute a long request to the Interlocutor, while coordination in the cluster is almost instantaneous given the short distances. The local coordination allows **managing resources more easily** since the responsibility of determining whether certain resources are available and/or adequate becomes distributed among all Interlocutors. **Device discovery is also greatly simplified** since the Grid services do not actually need to discover every single BASELINE device but only Interlocutors. Furthermore, **the Interlocutor hides devices heterogeneity from the Grid**, allowing also the possibility to apply mechanisms to compensate for BASELINE devices performances.

While using a local Interlocutor seems very effective, every benefit gained from the use of it comes at the cost of a great limitation: **when it comes to practically scaling the Grid, having the devices operating under the same local network makes it extremely more difficult for volunteers to offer their devices**. It is still possible to use some sort of variation of an Interlocutor, but this obviously comes at the cost of losing some benefits such as the reduction of latency while operating in the same network.

3.2.2 Mobile-to-Grid Middleware (2005)

While the previous work focused on mobile devices contributing to the Grid, this one (by *Umar Kalim*, *Hassan Jameel*, *Ali Sajjad* and *Sungyoung Lee*) provides a solution for **enabling the possibility for mobile devices to utilize the Grid's services**. The **Middleware gateway** represents the main element in this architecture, allowing mobile devices (running a client) to access Grid's services through it; using this expedient, limitations on computational power needed on requestor's devices are eliminated, making

possible to **trigger complex operations even from a device that is not adequate in performances**.

3.2.2.1 Architecture

A machine devolved to act as a Middleware gateway runs three different modules:

- **Middleware service**

  The main software module that handles access to Grid's resources by the clients executing requested operations.

- **Ontology server**

  Used to gain access to information that define different possible client devices.

- **UDDI registry**

  The discovery of Middleware gateways by mobile devices is performing by using the UDDI registry where every new instance of the gateways is registered. Hence, a machine acting as a Middleware gateway needs this module to interact with such registry.

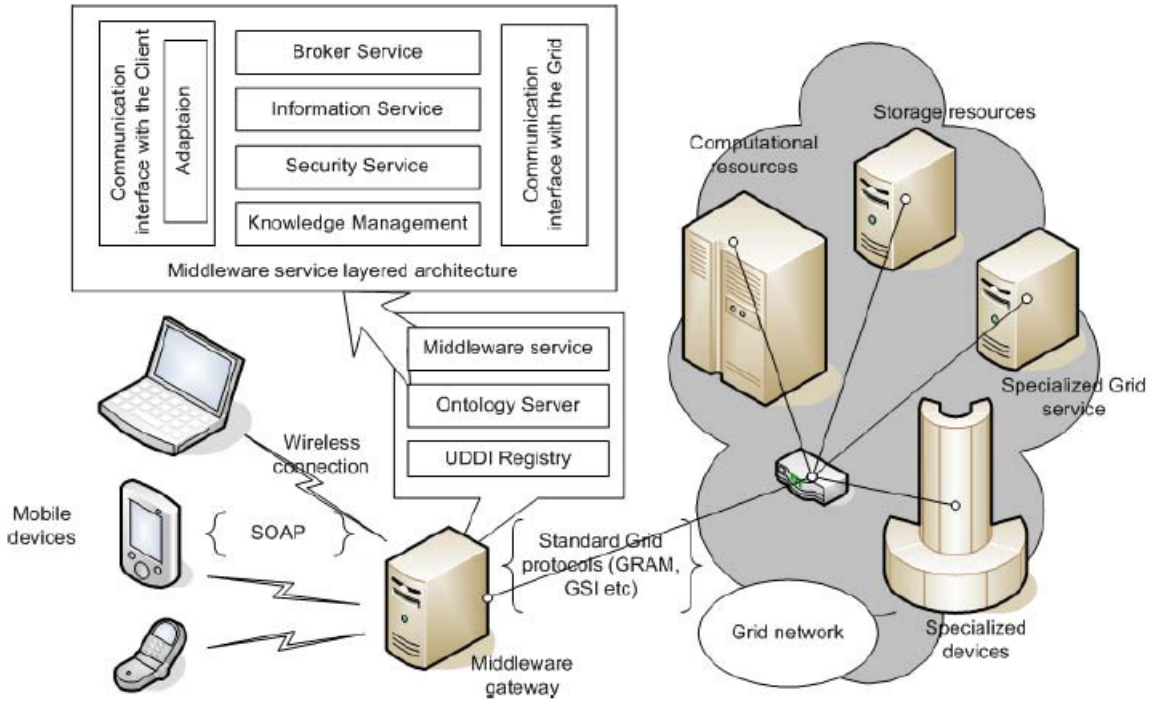


Figure 3.6: Mobile-to-Grid Middleware [12]

> "*Firstly the client application discovers and connects with the Middleware. Then the service, after authentication, submits its device specification along with the job request. The middleware then locates the relevant Grid service and after authorization forwards the request. The client may then request some status information (regarding the job or the service). If the client wishes to disconnect (and collect the result later), the Middleware would facilitate a soft state registration and to which would later help in the reintegration. After disconnection all the requests are served locally (with the cached information). Requests that result in updates at the Middleware service are logged for execution at reconnection. Upon reconnection pending instructions are executed and information updates at the client end are made to maintain consistency.*" *[12]*

Being the most complex module among the three, the Middleware service is built with a service layered architecture:

- **Broker service**
  This service takes job requests from the Client devices and, after verifying that adequate Grid's services are available (using the Information service), it downloads the code to be executed (from the device or from another source) and starts to perform the job interacting with the Grid. Results are stored in the Knowledge management layer.

- **Knowledge management**
  This layer manages relevant information, both on the client side and on the Grid side. Here results are also scaled down and/or formatted before being returned to the Client device.

- **Security service**
  Here there are all the security measures used to create a safe and authenticated connection with both the mobile Client and the Grid.

- **Information service**
  Through this service, the Middleware is able to determinate which services and resources are available on the Grid.

- **Communication interface with the Grid**
  This layer handles the implementation of communication protocols established by the Grid.

- **Communication interface with the Client**
  Similarly, this layer has the responsibility to perform communication with the Client running on a device; given the different types of possible clients running on different devices, an Adaption module is added to implement device-specific communication (using information gained by the Ontology server)

3.2.2.2 Interesting concepts and ideas

The main advantage of this idea is the **versatility obtained by increasing the number of devices capable of using the Grid's services**. Development on the requestor side is greatly simplified, removing heterogeneity since the actual complexity is moved to the Middleware gateway. To extend compatibility of devices, it is just necessary to implement the communication modules on the new client and add a new Adaption module in the Middleware service.

While this is surely a great advantage, **it comes at a great price: the necessity to provide machines that act as a Middleware gateway**. Considering that the computation is mainly done in such machines (with the collaboration of the peers inside the Grid), this results in an intense usage, **increasing costs for the Grid owners**. As the paper also states, there is also a **problem with scalability**. If a client submits a task and disconnects from the gateway (whether voluntarily of involuntarily), there is no guarantee that it will later reconnect to the same Middleware gateway, thus requiring mechanisms to deal with this issue.

3.2.3 Autonomous Mobile Middleware (2006)

Similarly to the previous work, *Fabio Navarro, Alexandre Schulter, Fernando Koch, Marcos Assunção and Carlos B. Westphall* propose a solution to **enable mobile devices to utilize the Grid's services**. Even though the idea is very similar, the **Middleware** changes physical location, becoming **integrated directly into the mobile devices**.

3.2.3.1 Architecture

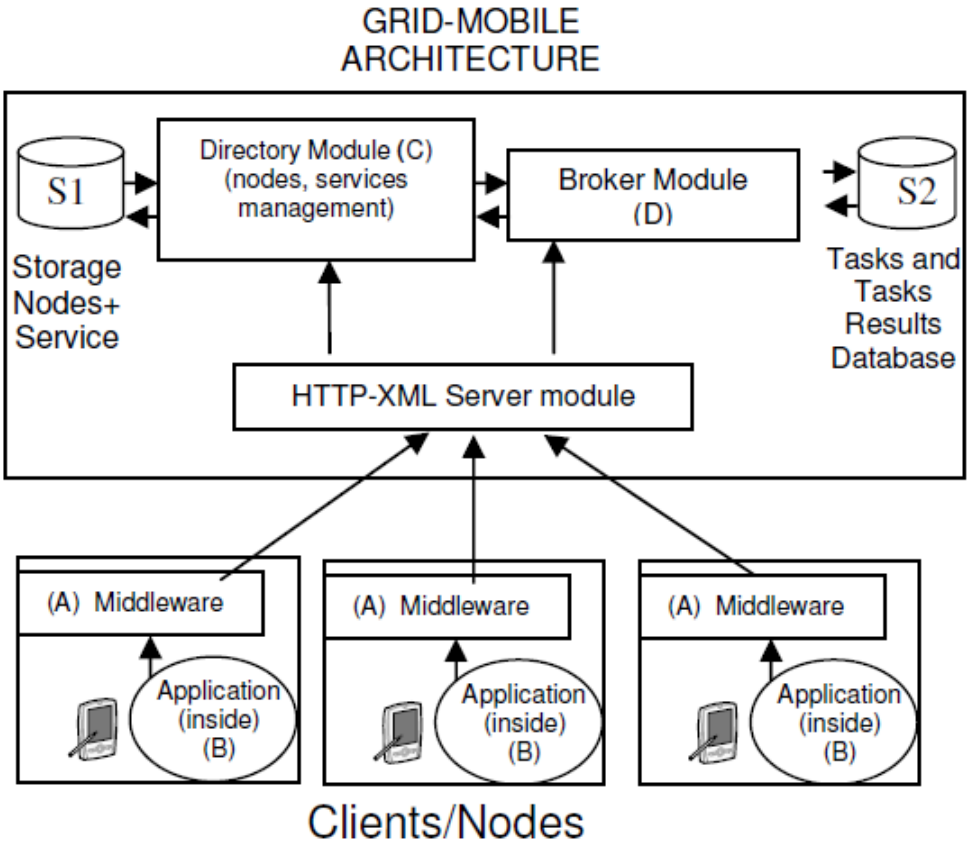


Figure 3.7: Autonomous Mobile Middleware [13]

Figure 3.7 shows the architecture proposed by this work. The **Middleware** (A) acts as a bridge that links **applications** (B) running on mobile devices and the Grid's services. The **Broker Module** service (D) manages resources, schedules jobs and monitors their execution **eventually saving results in a dedicated database** S2. Through the use of the LDAP protocol, the **Directory Module** service (C) manages information about nodes that participate into the Grid, keeping said information also in a dedicated database S1. Communication between a Middleware and the Grid's services is filtered by an **HTTP-XML Server module for nodes data representation**.

3.2.3.2 Interesting concepts and ideas

This solution inverts the benefits and drawbacks of the *section 3.2.2*'s solution: **at the cost of moving back the complexity** from a dedicated external machine to the mobile device, **scalability is facilitated** while, at the same time, **costs to maintain the infrastructure running are decreased** for the Grid's owners.

Considering that the ability of easily scaling is the main strength of a Grid architecture, this is probably a better approach compared to the previous solution, especially because a correct application of the layered Grid architecture (*section 2.1*) reduces complexity on the device side through the existence of the fabric layer.

While pursuing this solution it is important not to forget one of the reasons why the previous work moved the Middleware out of the device: the computational load that such software component requires to run. **It is extremely important that a Middleware running on the mobile-side is as light-weight as possible in order not to overconsume resources and battery**.

This work also highlights the **necessity of using a standard protocol to represent devices** inside the Grid. While HTTP-XML is a good solution, more modern standards can be applied such as using the JSON format.

Chapter 4

Transparent scheduling model over heterogeneous devices

After all the necessary context introduced by the previous chapters, here the solution proposed by this work will start to be discussed. In particular, the first section focuses on discussing the characteristics of the solution while the second and last chapter present an analysis where the Domain, the Use Cases and the Requirements are addressed, preparing the ground for the next chapter where an Architecture implementing this solution will be shown.

4.1 Grid computing evolution: this project's solution

Coming from the overview on how Grid Computing works (*chapter 2*), followed by the challenges and previous ideas on how to integrate mobile devices into the Grid (*chapter 3*), it is now time to expand this project's idea, already introduced in *chapter 1*.

The solution proposed by this work takes the name of "***Transparent scheduling model over heterogeneous devices***"; the model aims to **maintain** the **core characteristics of a Grid computing system** while, at the same time, **redesign some aspects** in order to enhance and evolve its capabilities **to better accommodate new necessities arising from the current technological panorama**.

In particular, this Grid computing system (following the Cloud Computing modern approach) aims to **offer services**, ranging from simple to complex ones, **to everyone that might need access to a large pool of computational resources**.

An example of computationally demanding distributed service is the execution of a **MapReduce computation**, which will be the service **realized for this project's prototype**.

Info: See *appendix A* for an overview on the MapReduce paradigm.

This section, then, presents the **project's aspects that differ from a traditional Grid computing approach** and the reasons **why it might represent a viable alternative to current Cloud computing services**.

4.1.1 Transparent heterogeneous devices: more contribution

The **core difference**, compared to a traditional Grid approach, comes from the **inclusion of mobile devices into the contributing devices base**. This project's aspect was discussed previously (*section 1.3* and *chapter 3*) and **results in a broader contribution to the Grid system coming from people that tend to primarily use their mobile devices** (the vast majority nowadays).

One core characteristic emerges from the discussion of a previous work: **the contributing mobile devices need to autonomously interact with the Grid system**. This autonomy principle stands in direct contrast with the approach proposed by the work discussed in *section 3.2.1*, but it is mandatory in a system where simplicity is key; moreover, **the limitation of an Interlocutor device is no longer justified with the performances of current mobile devices** given by the technological progress of the recent years. It is important to specify that, while mobile devices have now better performances, **technological limitations** (compared to contemporary computers) **still need to be taken into account while designing the solution**. The characteristics discussed in this section influence the solution's name:

- **Transparent scheduling**

  Traditional Grid systems operate with the assumption of the presence of a single type of contributing devices: computers; **since this assumption does not hold here, a layer of abstraction needs to be put into action. A device is seen as a collection of resources that it can offer**, which will be the only relevant aspect when scheduling tasks that said devices will need to execute to collectively

complete a service offered by the Grid; hence, **the devices' heterogeneity becomes effectively transparent to the system**.

- **Heterogeneous devices**
  This aspect of the solution's name is **influenced not only by the fact that heterogeneous devices are supported for contribution, but also by the fact that different types of devices can also access the Grid's services** (this will be discussed in *section 4.1.4*).

4.1.2 Low infrastructural costs: Volunteer computing philosophy

When it comes to **Cloud computing**, **one critical problem is the amount of resources** (hardware and, consequently, economical) **needed to maintain up and running the infrastructure required** to perform the services that the Cloud computing platform wants to offer; **Grid computing combined with Volunteer computing** (*section 1.1.2*) **vastly reduces this problem**.

A typical Cloud computing platform operates, in simple terms, by a multitude of hosted machines (running the required software) communicating among each other; the maintenance costs of said machines is completely sustained by the Cloud computing platform owner. **With Volunteer computing**, on the other hand, **the machines that compose the Grid architecture are divided in two categories**:

- **Maintenance costs sustained by the Grid owner**
  Here reside the machines running the core coordination mechanisms of the Grid system. The instances of said machines can be dynamically increased or decreased depending on the necessities dictated by the current workload, resulting in paying the strict necessary to make the system work at any given time.

- **Maintenance costs sustained by the Volunteers**
  This category comprehends the actual machines (Computers, Smartphones and Tablets) that cooperate in order to execute the services provided by the Grid. Since the devices are owned and used by the Volunteers, the vast majority of the costs associated with said devices are demanded to them. Said costs are represented mainly by the device maintenance, the electricity needed to make them work, and the Internet traffic generated; but, as discussed in *sections 1.3.2.2 and 1.3.2.3*, these

costs will be sustained by the Volunteer anyway (especially in the case of mobile devices), regardless contributing to the Grid or not. Furthermore, the Volunteer receives a compensation for its contribution (*section 1.4.4*), making it actually convenient for them to contribute.

On a final note, this approach also has a positive effect on the environment: the vastly diffused Cloud model results in a great consumption of electricity to run the dedicated machines composing the infrastructure; on the other hand, **demanding the execution of software into Volunteer machines** (that would be turned on anyway) **reduces the environmental impact derived by the electricity consumption**.

4.1.3 Removing the complexity: broader contributing user base

Although **Volunteer computing** has been a concept for many years, **the volunteering was typically done by people with relative confidence in using technology**, in particular (necessarily) computers.

It is not uncommon that the typical computer user, for example, does not have the knowledge (or the confidence) to find and launch a Windows installer or, even worse, install a program from an integrated manager on any Linux distribution. But, as technology increased in diffusion, **in recent years a broader audience has become familiar with the concept of installing Apps through the use of a store** (mainly the App Store and the Play Store), **making it easier to reach new users**. Using such established means of distribution makes it **realistic to also reach potential Volunteers that mainly use mobile devices** and may not have the technological skills to contribute using a computer. Furthermore, the "store" approach on mobile devices has recently generated similar stores on the desktop OSes thus creating a **new simplified distribution medium** (even if still not vastly used today).

But, even if the first barrier on reaching users is removed, it is important to keep in mind the **simplicity principle** behind the idea, ensuring that the Volunteer has to perform a **setup as easy and guided as possible** in order to **configure its device once and then never have to do that again**.

4.1.4 Toward ubiquitous computing: Grid services for all devices

The "***heterogeneous***" **keyword** in the solution's name **refers not only to the possibility of using a multitude of different devices in order to contribute to the Grid, but it also includes the feature of accessing Grid's services from computers as well as smartphones and tablets**.

The previous work described in *section 3.2.2* introduced this idea with the use of a middleware machine; a subsequent work (*section 3.2.3*) improved this idea by removing the necessity of such middleware. This approach mirrors the one previously described for devices' contribution to the Grid and better suits this solution, obtaining broader devices' compatibility from both ends of the execution of a service (requestor machine side and executor machines side). **As a primary consequence, a multitude of applications running on various devices can build complex behaviors with the support of the Grid's resources**.

The possibility of executing computationally heavy tasks even from low-performance devices is a **step toward the ubiquitous computing idea** for the future, having **great versatility and computing power accessed by day-to-day objects**.

4.1.5 Cheaper access to Cloud services: Grid services for everyone

As a result of having broad compatibility for devices' contribution (*section 4.1.1*) **and low infrastructural costs** (*section 4.1.2*), **the cost of accessing to the Grid services** (that can be seen as Cloud computing services from the customer point of view) **becomes cheaper** than utilizing other platforms that run their infrastructures hosting their private machines.

Realistically, performances of services ran on Grid computing systems will be slightly worse compared to services that host their dedicated machines; this is mainly because of the potential faults arising from the less stable connection of the devices. **Nonetheless, this is a highly viable and much cheaper alternative in domains where performances are not a highly strict requirement** and a little delay can be accepted, opening the possibility of using such services also to entities that do not have a lot of financial resources.

Expanding the considerations seen up until now, **the Grid owners get value and a revenue from the Customers that**, at the same time, **save money accessing cheaper services**. **On the Volunteer side, there is a compensation** for doing absolutely nothing more than easily configuring a device once **and the environment is less impacted** while still performing advanced computations. In conclusion, **everyone wins**.

4.1.6 Anticipating market trends: current devices' panorama

As discussed in *section 1.2.1*, **the current market trend shows how computer sales are decreasing** and, as experts say [4], this trends is not expected to reverse. In light of these considerations, **as the number of computers decreases while other devices increase** (mainly mobile), **it becomes increasingly important to think computational solutions around devices that will actually be used**.

The solution proposed, although already relevant, progressively increases its significance with time with the gradual but inevitable global shift of device types composition.

4.2 Solution analysis

After a high level discussion about the characteristics of the solution proposed by this work, now it is time to make an additional step in the road that leads to the concretization of a working system that realizes such idea.

Here, the Domain will be explored, clarifying concepts and nomenclatures in order to uniquely identify the building blocks needed to move forward. Then, use cases will be discussed, explaining what the system needs to do, seen from the point of view of the actors interacting with it. Finally, the requirements will be presented, aiming to provide a clear set of criteria to satisfy for a system that wants to use the Transparent scheduling model over heterogeneous devices.

4.2.1 Domain: Ubiquitous Language

Before continuing, **it is important to specify the nomenclatures and relative definitions on what composes the domain in which the projects operate**; in order to do so, the **Ubiquitous Language** (UL) tool, coming from the **Domain Driven Design** (DDD)

discipline [14], will be used. From this point and forward, a term listed here will be uniquely used to identify its mapped concept:

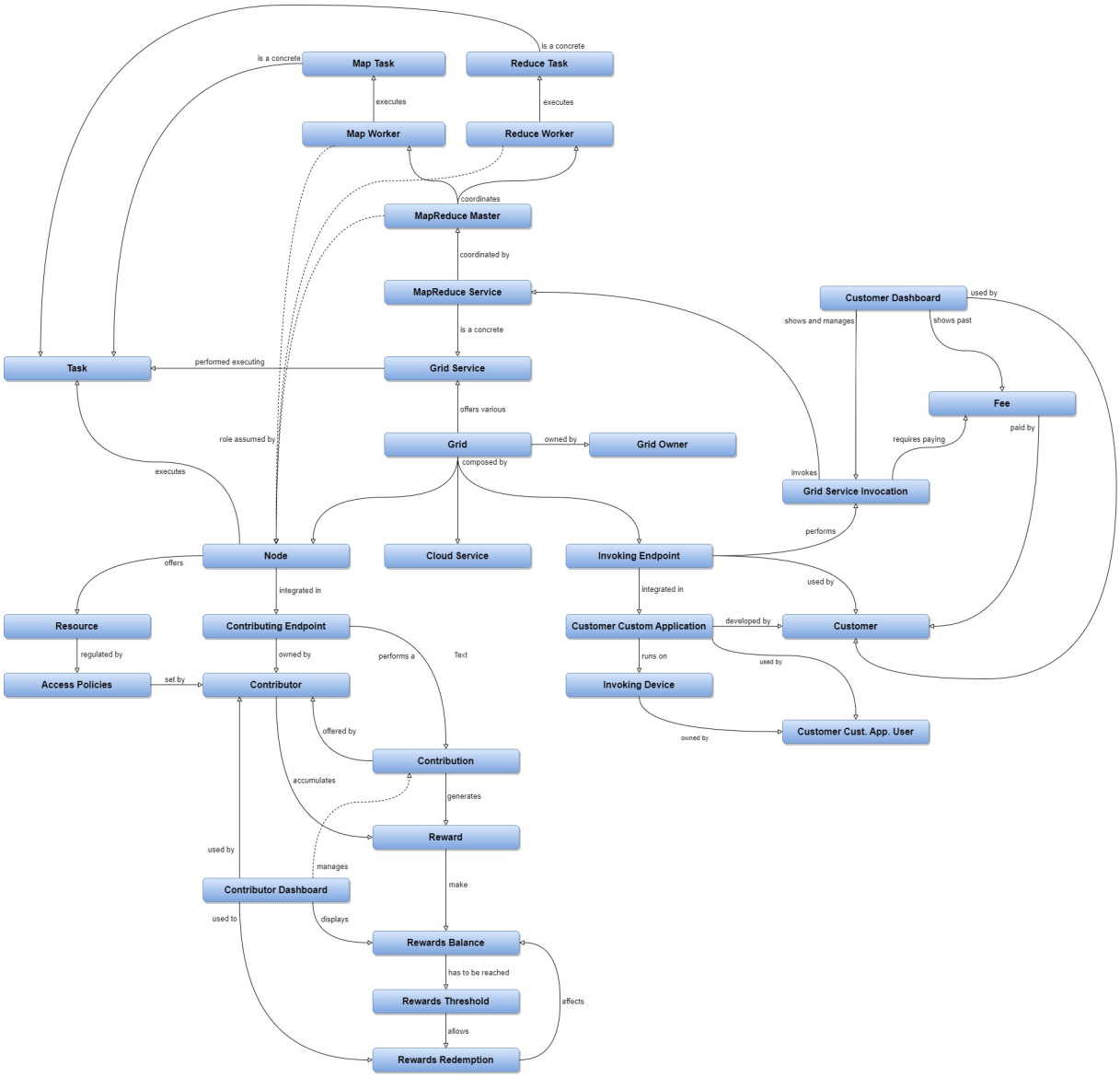


Figure 4.1: Ubiquitous Language

- **Contributor**

  A person (up until this point identified as the Volunteer in Volunteer Computing) which offers one or more *Contributing Endpoints* in order to perform a *Contribution* finalized to the completion of *Grid Services*. Through its *Contribution*, it gains *Rewards* that are accumulated in its *Reward Balance*; once a certain *Reward Threshold* is reached, a Contributor can perform a *Rewards Redemption*. The Contributor can manage its data through a *Contributor Dashboard*.

- **Contribution**

  The automated act performed by a *Contributing Endpoint* (offered by a *Contributor*) in order to complete *Grid Services* and, consequently, gain *Rewards*.

- **Customer**

  An entity (whether a single person, a business, etc...) that wants to perform a *Grid Service Invocation* through an *Invoking Endpoint* integrated into a *Customer Custom Application*. This entity can manage its data and past *Service Invocations* and *Fees* utilizing a *Customer Dashboard*.

- **Grid Service Invocation**

  The act, requested by a *Customer* through the use of an *Invoking Endpoint*, of requesting a certain amount of *Resources* (paying, proportionally, a *Fee*) in order to perform a *Grid Service*.

- **Grid Owner**

  Entity that owns the *Grid* system and is therefore responsible for its maintenance, providing all the medium to access to the *Node* and *Invoking Endpoint* software, as well as managing the *Cloud Services*.

- **Grid**

  The system as a whole (owned by the *Grid Owners*), based on the Grid computing principles. It offers various *Grid Services* and, architecturally, it is composed by a multitude of *Cloud Services*, a dynamically ever-changing number of *Nodes* (provided by *Contributors*) and an also variable number of *Invoking Endpoints* (owned by *Customers*) used to invoke its Grid Services. The parts composing the Grid are organized following the Layered Grid Architecture (*section 2.1*).

- **Grid Service**

  A service (computation, storage, etc...) offered by the *Grid* which is requested by a *Customer* through an *Invoking Endpoint*, paying the corresponding *Fee* (which can vary depending on the amount of *Resources* requested). In order to be performed, the *Cloud services* coordinate the *Contribution* by the *Nodes* (which perform *Tasks*) offered by *Contributors* (who gain *Rewards* for such Contributions). One particular concretization of a Grid Service is the *MapReduce service*.

- **Task**

  Abstract unit of *Contribution* performed by a *Node* in order to obtain the result of a *Grid Service Invocation*. Depending on the *Grid Service*, different Tasks exist.

- **Cloud Service**

  A backend server, maintained by the *Grid Owners*, operating in the Cloud. Its duties depend on its role, but the distinct traits of a Cloud Service include being owned and maintained by the *Grid Owners* and the ability to coordinate the *Contribution* of *Nodes* participating in the *Grid* and the access to *Grid Services* invoked by *Invoking Endpoints*. In relation to the Layered Grid Architecture (*section 2.1*), Cloud Services all together embody the Collective Layer (*section 2.5*) as shown in *figure 4.2*.

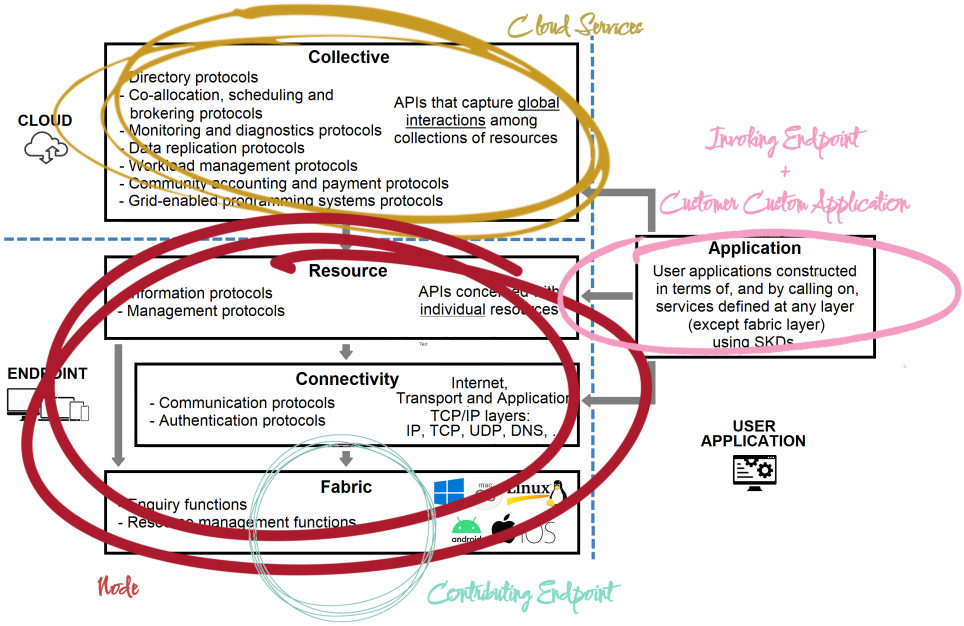


Figure 4.2: UL relationship with Layered Grid Architecture

- **Node**

  A Node is an abstraction representing a software entity providing the mechanisms used by a *Contributing Endpoint* (owned by a *Contributor*) to connect to the *Grid* (through the *Cloud Services*) in order to perform a *Contribution* finalized to the completion of *Grid Services* offering *Resources* and executing *Tasks*. Referring to the Layered Grid Architecture (*section 2.1*), a Node can be logically mapped to the concrete realization of the Resource and Connectivity layers (*section 2.1.3* and *2.1.2*), as well as the definition of the Fabric layer interfaces (*section 2.2*) which will be concretely implemented in the *Contributing Endpoint* (*figure 4.2*).

- **Contributing Endpoint**

  An abstraction representing a client software running on a device (Computer, Smartphone or Tablet), owned by a *Contributor*, which is able to perform a *Contribution* finalized to completing *Grid Services*; such *Contribution* is performed by the software (running on the Contributing Endpoint) that is built interfacing with the *Node* software abstraction. Not all Contributing Endpoints are able to perform every *Grid Service* that can exist; such ability is determined by the *Resources* that the specific Contributing Endpoint can offer. A Contributing Endpoint can be logically mapped (*figure 4.2*) to the concrete implementation of the Fabric Layer (*section 2.2*) of the Layered Grid Architecture (*section 2.1*).

- **Resource**

  An abstraction representing what a *Contributing Endpoint* can offer (through the *Node* software) in order to offer a *Contribution* to the completion of a *Grid Service*. A resource can belong to one of the following types:

  - Hardware (processor, memory, disk, etc...)
  - Software (OS, supported programming languages, compatible *Tasks*, etc...)
  - Network (connection type, download speed, upload speed, etc...)

- **Invoking Endpoint**

  An abstraction representing a software entity providing the mechanisms used by an *Invoking Device* (owned by a *Customer*) in order to perform a *Grid Service Invocation*. This abstraction is used in a *Customer Custom Application*, integrating the *Grid Services* into such application. Continuing the mapping (*figure 4.2*) to the fundamental Layered Grid Architecture (*section 2.1*), an Invoking Endpoint (together with the *Customer Custom Application*), embodies the Application Layer (*section 2.6*).

- **Customer Custom Application**

  Software application, developed by the *Customer*, that uses the *Invoking Endpoint* software abstraction in order to integrate *Grid Services* into such custom application ran on an *Invoking Device*.

- **Invoking Device**

  A device that runs the *Customer Custom Application*. Such device can be owned by the *Customer* itself or by people using the *Customer Custom Application* that the *Customer* is commercializing. In regard to the *Grid*, the focal point here is that it is a device that is performing a *Grid Service Invocation* through the *Customer Custom Application* that is utilizing the *Invoking Endpoint* software.

- **Customer Custom Application User**

  End user of the *Customer Custom Application*. The *Customer* and the *Customer Custom Application User* can coincide, but it highly depends on the specific *Customer Custom Application* developed by the *Customer*.

- **Contributor Dashboard**

  A software entity through which a *Contributor* can manage its *Contributing Endpoints* and check its *Rewards Balance* (eventually performing a *Rewards Redemption*).

- **Customer Dashboard**

  A software entity through which a *Customer* can manage its present and past *Grid Service Invocations*, check the relative *Fees* as well as registering a payment method.

- **MapReduce Service**

  A particular incarnation of a *Grid Service* that performs a MapReduce computation over the *Grid* utilizing a *Node* as *MapReduce Master* and several nodes as *Map Workers* and *Reduce Workers*.

- **MapReduce Master**

  Role taken by a *Node* in the execution of a *MapReduce Service*. Its roles follow the ones described in *appendix A*.

- **Map Worker**

  A *Node*, coordinated by the *MapReduce Master*, that executes *Map Tasks* and sends the results to the correct *Reduce Worker*.

- **Map Task**

  A concrete *Task* performed by a *Map Worker*. A Map function provided by the *Customer* (obtained through the *MapReduce Master* coordination) is executed on data of which the location is specified also by the *Consumer*.

- **Reduce Worker**

  A *Node*, coordinated by the *MapReduce Master*, that executes *Reduce Tasks* on data received by *Map Workers*.

- **Reduce Task**

  A concrete *Task* performed by a *Reduce Worker*. A Reduce function provided by the *Customer* (obtained through the *MapReduce Master* coordination) is executed on data received from *Map Workers*.

- **Fee**

  A monetary amount that the *Customer* needs to pay in order to perform a *Grid Service Invocation*. The amount is influenced by the type of service requested and the quantity of resources employed to complete such service.

- **Reward**

  A monetary compensation for the *Contributor*, calculated based on the *Fee* paid by the *Customer* (minus the *Grid Owner* profit) proportionally to the *Contribution* performed by a *Contributing Device* in a *Grid Service Invocation*. Every new Reward contributes to the *Reward Balance*.

- **Rewards Balance**

  The sum of the *Rewards* obtained by the *Contributor* for the *Contribution* of the *Contributing Endpoints* owned by it. This balance can be lowered when the *Contributor* performs a *Rewards Redemption*.

- **Rewards Redemption**

  An action performable, by the *Contributor* (only if the *Rewards Balance* is currently higher or equal to the *Rewards Threshold*), that lowers the value of the *Rewards Balance* transfering that monetary value by using a payment method.

- **Rewards Threshold**

  The value that needs to be reached in order to allow the *Contributor* the possibility to perform a *Rewards Redemption*.

- **Access Policies**

  Contribution Endpoint-specific Access policies defined by the *Contributor*, specifying which *Resources* are offered to the *Grid* for *Contribution*.

4.2.2 Use Cases

This section presents the **Use Cases diagrams**, **graphically explaining what features need to be available from the point of view of the two actors interacting with the Grid**: the Contributor and the Customer.

An explanation of single use cases here is omitted since the previous section already explained the non-trivial ones indirectly; furthermore, in the next chapter, *section 5.2* will show how said use cases are realized using Grid entities collaborating among each other, making redundant a detailed explanation here.

- **Contributor**

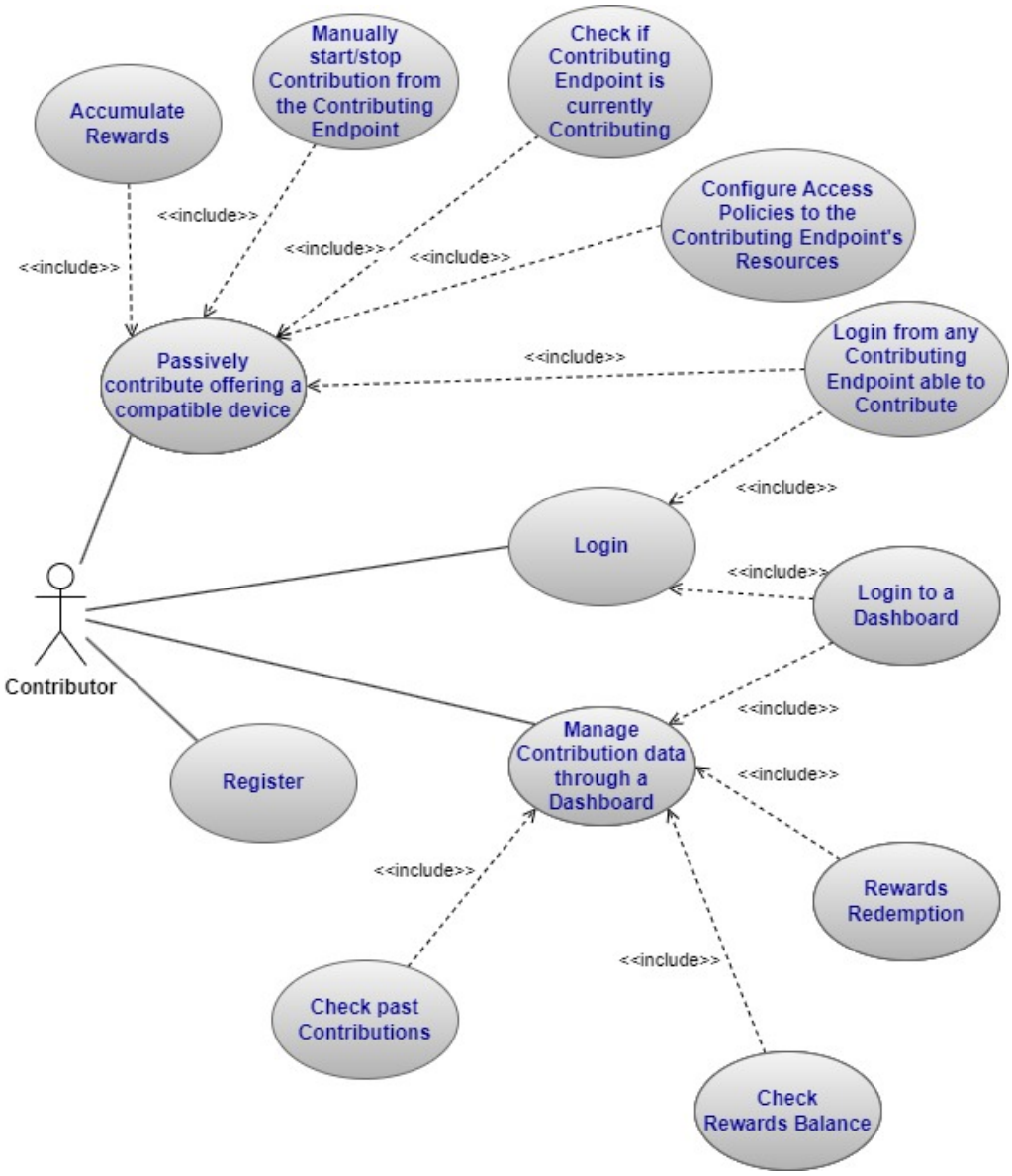


Figure 4.3: Use Cases diagram - Contributor

- **Customer**

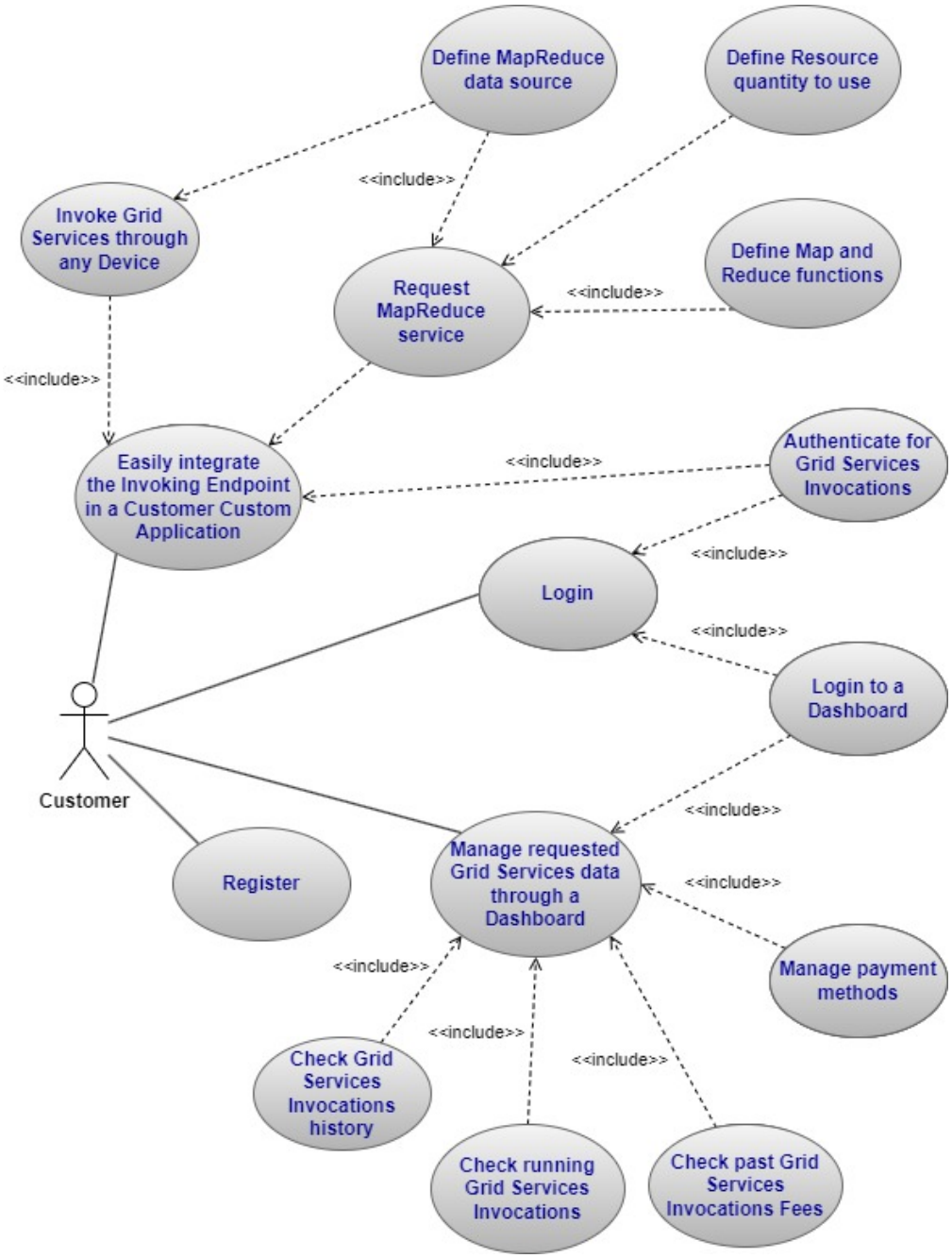


Figure 4.4: Use cases diagram - Customer

4.2.3 Requirements: MoSCoW Prioritization

Now that it is clear what the system needs to do from the perspective of the two major actors utilizing it (Contributor and Customer), it is possible to create a **list of requirements** that such system needs to satisfy; to do so, the **MoSCoW method** will be used.

The MoSCoW method for **requirements categorization** presents **four different** categories to put requirements into, **ordered by priority**. The categories, in descending order, are the following:

- **Must have**
- **Should have**
- **Could have**
- **Won't have**

Other than priority, this methodology focuses also on the concept of release, aiming to use an **evolutionary approach**; this means that **for a first production release it is not necessary to satisfy all requirements but just the one with higher priority** (Must have), leaving the satisfaction of the remaining requirements to subsequent versions (always trying to first satisfy remaining requirements with the highest priority).

Requirements in this list are also divided in **three areas**: **Contributor, Customer and Grid Owner**; **the last area** in particular (which was not included in the use cases) **is introduced to assign tasks that are more technical in nature but are necessary to the success of the project**, both in the short term (first working release) and in the long term (easily introduce new features) given the particular technical nature of this Domain.

4.2.3.1 Must have

All the requirements that are necessary for the successful completion of a first working production release.

Contributor area

- **Registration and identification**
  - The Contributor must be able to register to the Grid system in order to uniquely identify Contributions performed by its Contributing Endpoints.
  - Once authenticated to a Contributing Endpoint, the Contributor must not be requested to authenticate again unless it is absolutely necessary.

- **Contribution**
  - It must be possible to Contribute to the Grid using a Contributing Endpoint running on any device among Computers, Smartphones and Tablets running the most popular respective Operative Systems.
  - Configuration for the Contributor must be easy, guided and fast (5 minutes maximum).
  - Every device that the Contributor offers must be configured just once and requires no additional efforts unless the Contributor chooses to change configurations like Access Policies.
- **Rewards**
  - Every successful Contribution to the execution of a Grid Service must be registered, increasing the Rewards Balance of the Contributor.
  - The Contributor must be able to see its current Rewards Balance.
  - The Contributor must be able to see past Contributions in which its Contributing Endpoints contributed to the execution of a certain Grid Service.
  - Once a certain Rewards Threshold is reached, the Customer must be able to perform a Rewards Redemption.
- **Security**
  - Execution environment-specific security measures must be implemented to protect the Contributor.
- **User experience**
  - The Contributing Endpoint running on mobile devices must be aware of battery status in order not to drain battery in low-battery situations.
  - The Contributing Endpoint running on mobile devices must be aware of the type of connection used in order not to drain data from the Consumer mobile plan (unless the Consumer allows this possibility).

Customer area

- **Registration and identification**
  - The Customer must be able to register to access Grid Services.
  - The Customer must go through a verification process in order to be reliable and accountable before being able to use Grid Services.

- **Payments**
  - The Customer must be able to register its payment method once and use it every time it performs a Grid Service Invocation.
  - The Customer must pay individual Grid Service Invocations.

- **Framework utilization**
  - The Customer must be able to integrate the Invoking Endpoint in its own Customer Custom Application in order to utilize the Grid Services.
  - The Customer must be able to perform Grid Service Invocations from any adequate device (Computer, Tablet or Smartphone) using the integrated Invoking Endpoint.
  - The Customer must be able to easily increase/decrease the amount of Resources tapped into, removing the underlying complexity.

- **MapReduce service utilization**
  - The Customer must be able to implement its own Map and Reduce functions in a MapReduce Service invocation.
  - The MapReduce parameters, such as number of Map and Reduce workers, must be configurable by the Customer.
  - The Customer must be able to define the data source for the MapReduce execution (whether from the Invoking Device or an external source).

Grid owner area

- **Scalability**
  - The Grid system must be able to easily scale horizontally in a fully automated way.
  - Cloud Services must be as loosely coupled as possible, resulting in a modular composition of technologically independent services that operate exchanging messages over defined channels.
  - It must be possible to create on the go (depending on needs dictated by the current traffic) multiple instances of the Cloud Services, employing a mechanism to discover and contact such new instances.
- **Efficiency**
  - Employed resources (i.e. dedicated machines) used to run the Cloud Services must be adequate to the load that the system has to currently handle, dynamically adjusting to cut expenses.
  - The Grid must be geographically-aware and manage Nodes utilization taking into account physical distance in order to reduce latency, improving performances.
  - When possible, Grid Services must be implemented in a way that reduces traffic to the Grid's systems, moving such traffic to the Nodes.
  - Every execution of a Grid Service must use Nodes that have sufficient Resources to complete the Tasks adequately.
- **Compatibility**
  - Invoking Endpoint software and Node software must be implemented using a technology that can be easily integrated in all major Operative Systems that run on the target devices.
- **MapReduce**
  - The MapReduce execution must grant an adequate level of reliability and fault tolerance.

  - The Map Tasks and Reduce Tasks on Node side must be designed in a way that allows to integrate support for new programming languages for the execution of Customer-defined Map and Reduce functions.

- **Expandability and maintainability**
  - The solution must be designed in a way that allows to easily expand the capabilities of the Grid, allowing the creation of newer Grid Services that can be executed using the Nodes.
  - It must be possible to parallelize implementation of features and maintenance, structuring the solution using a microservices approach allowing multiple teams to work on multiple sub-portions of the project.

- **Security**
  - Communications between the components of the Grid must be cyphered and use other security measures.
  - There must be an Access Policies system.
  - Operations inside the Grid must be logged in order to have a fully comprehensive view of every movement that happens inside the system.

4.2.3.2 Should have

Requirements that are important but not necessary for a production release and can be postponed to subsequent releases.

Contributor area

- It should be possible to see the status of every Contributing Endpoint owned by the Contributor.
- It should be possible, from a Contributing Endpoint, to see the local current status.
- The Contributor must be able to customize Access Policies to its Contributing Endpoints.
- It should be possible to manually enable/disable Contribution of a Contributing Endpoint.

- It should be possible to schedule a trigger (time and/or events) to start/stop the Contribution.
- The Contributing Endpoint should limit usage of resources if the user is actively using the device in order not to have a negative impact on user experience (avoiding incentives for the user to uninstall the application).

Customer area

- There should be a tool for testing the Map and Reduce functions.
- It should be possible to specify the minimum Resources of the Contributing Endpoints involved in the execution of a Grid Service.

Grid owner area

- There should be device-specific optimizations.

4.2.3.3 Could have

Requirements that are nice to have but have a much smaller impact when left out of the release.

Customer area

- There could be a standardized MapReduce web tool that enables less specialized organizations (from a technological point of view) to perform operations constructed by a facilitated GUI.

4.2.3.4 Won't have

Requirements that have not been recognized as a priority for the release timeframe.

Contributor area

- Development of a browser-based Contribution limited to certain types of Grid Services.

Grid owner area

- Development of additional Grid Services other than MapReduce.

Chapter 5

Design

Here the design of a software architecture concretizing the model described in the previous chapter will be discussed. First, the architecture itself will be discussed; following that, the use cases described in *section 4.2.2* will be explained in terms of how the entities composing the architecture communicate among each other in order to satisfy such use cases. Concluding this chapter, a formalization of the desired MapReduce Service's behavior is presented.

5.1 Architecture

This section describes the software architecture proposed by this project in order to realize the Transparent scheduling model over heterogeneous devices and, consequently, satisfy the requirements and use cases previously described.

Figure 5.1 provides a **complete view of the Architecture**; being a complex project based on the interaction among multiple distributed entities, the schema will be divided into **three areas** (easier to understand) that will be discussed in the following sections:

- **Cloud Services area** (*section 5.1.1*)
- **Contributor area** (*section 5.1.2*)
- **Customer area** (*section 5.1.3*)

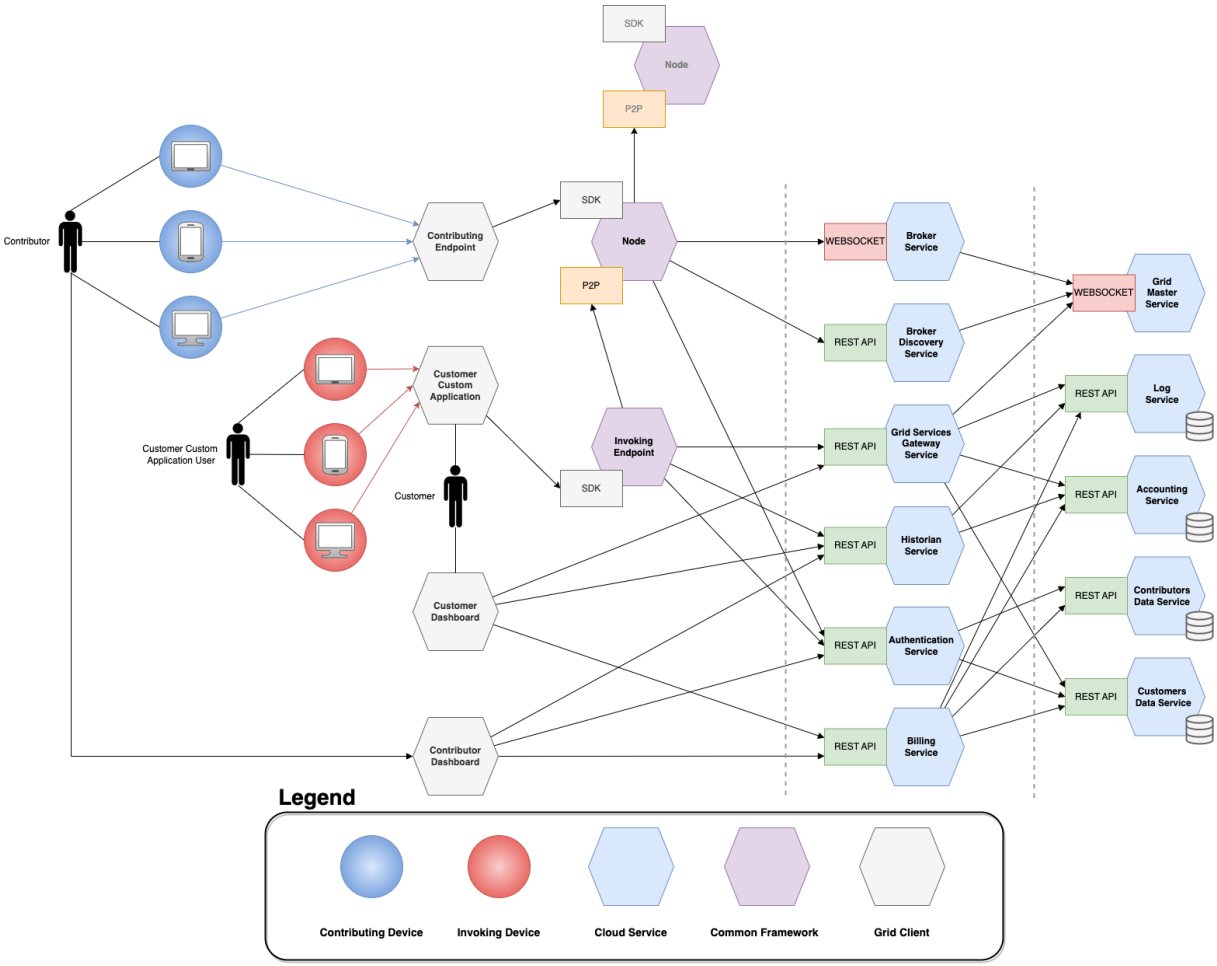


Figure 5.1: Complete view of the Architecture

5.1.1 Cloud Services Area

The **Cloud Services** follow a **hexagonal architecture** composed by a multitude of **microservices** (*Figure 5.2*). Each Microservice is **accessed through communication interfaces** (whether **Rest APIs and/or Web Sockets**, depending on the particular microservice needs) **and belongs to one of the following two layers**:

- **Business logic**

  Microservices that **expose core business logic for Grid functionalities realization and data managing**. Entities that reside here are **protected**, meaning that they are isolated from the outside and **their functionalities can only be accessed through the entities placed in the Adapters layer**.

- **Adapters**

  Microservices that **expose functionalities accessed by the entities residing in**

the Contributor and Customer area; such functionalities are **realized combining the services** offered by entities residing **in the Business logic layer**.

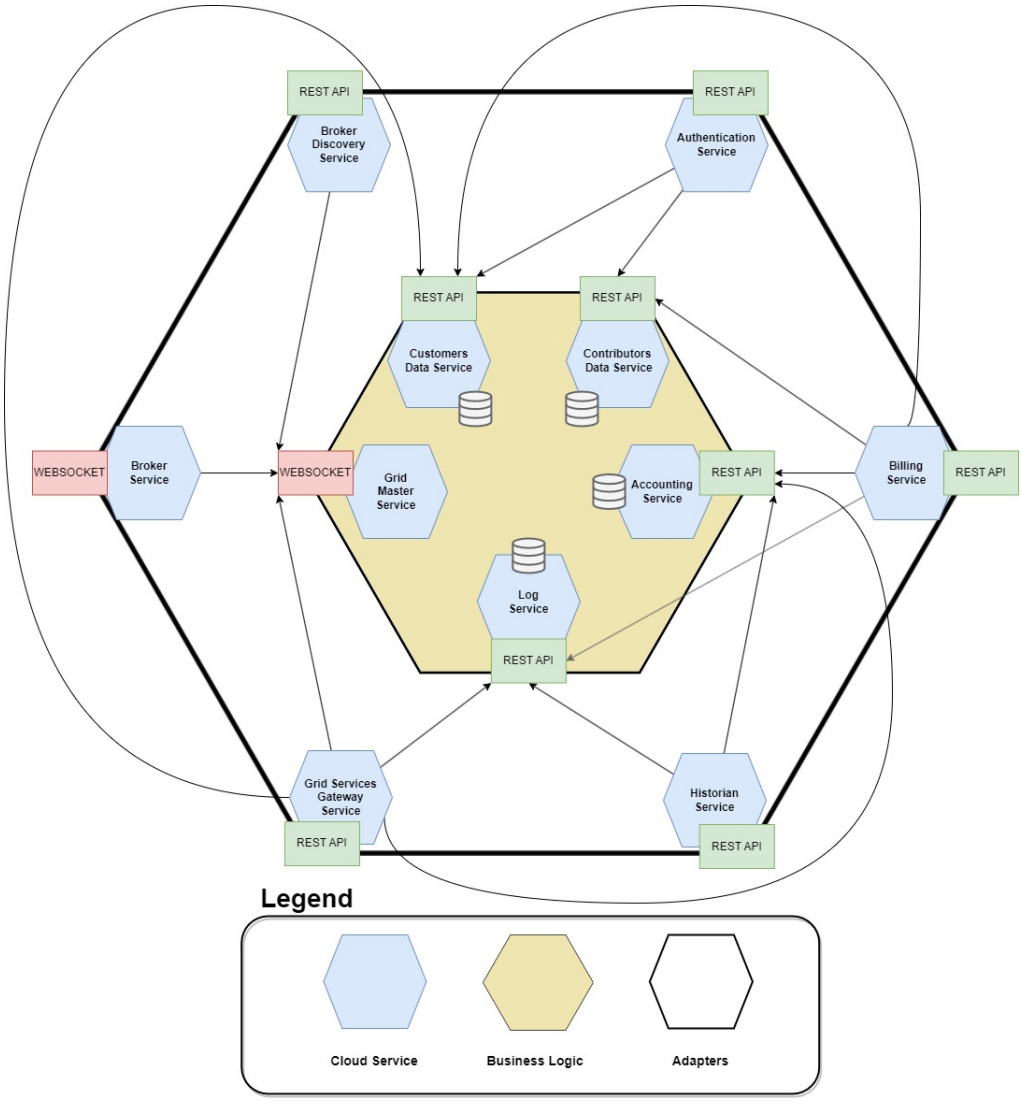


Figure 5.2: Architecture: Cloud Services

5.1.1.1 Business Logic

- **Grid Master Service**

  The **main coordinator in the Grid** system. It **dynamically creates/removes Broker Service instances** in order to sustain and balance the traffic generated by Nodes and Invoking Endpoints connected to the current instances of Broker Service; such instances need to be contacted by Nodes but, being dynamically instantiated, they do not possess a static address. As a consequence of that, **the Grid Master Service** (that knows such addresses, being the creator of the instances) **is connected to the**

Broker Discovery System (which will provide a Broker Service instance address to a connecting Node).

Figure 5.3 shows a **high level view of the connection between Grid Master, Brokers and Nodes** while also providing an **example of load balancing**. When the new Node [N11] wants to connect to the Grid in order to Contribute, given that no Broker instance is able to handle the Node connection, the Grid Master instantiates [B2] to which [N11] will connect and part of the load handled by [B1] (just [N9] in this example) will be redirected to.

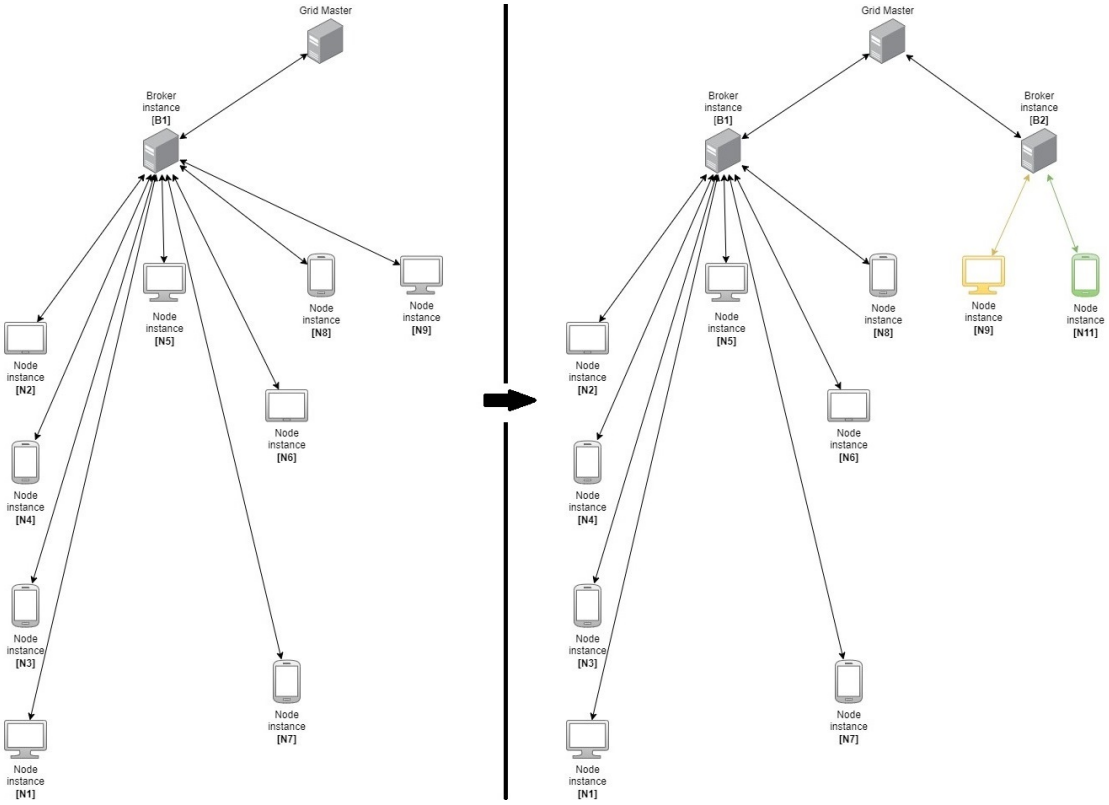


Figure 5.3: Grid Master Service load balancing

Lastly, the Grid Master Service is **also connected to the Grid Services Gateway Service**; through this last Cloud Service, the Invoking Endpoints request the execution of Grid Services. Thus, the Grid Master (collaborating with the Broker Service instances) will **provide the Resources needed to execute the requested Grid Service**.

To conclude, the importance of this Cloud Service's is vital to the functioning and scalability of the Grid, requiring to expose **communication interfaces for the discovery of Brokers, Resources obtainment and Grid coordination**.

- **Customer Data Service**

  REST server that provides **APIs used to read and write data used to uniquely identify a Customer inside the system**. It **does not contain data about payments or logs** about Grid events that involve the Customer since those are handled by the Accounting Service and the Log Service respectively.

  Customers are, numerically speaking, considerably less compared to Contributors; this results in **less frequent invocations** of this Cloud Service's APIs and a **far smaller volume of data to persist**. Regarding the CAP theorem, it is then important to **focus on a database technology that can grant Consistency and Partition Tolerance** sacrificing Availability (e.g. MongoDB, BigTable, etc...).

- **Contributor Data Service**

  REST server that exposes **APIs used to read and write data used to uniquely identify a Contributor and its devices inside the system**. The same rules used in the Customer Data Service, regarding the handling of logs and payments data, also hold here; when it comes to the CAP theorem application, on the contrary, **this Cloud Service requires AP database technologies** (e.g. DynamoDB, Cassandra, etc...).

- **Accounting Service**

  REST server exposing APIs to **perform and record the history of monetary transactions, involving both Customers and Contributors** (i.e. Fees payments and Rewards Redemptions) inside the Grid; **Consistency and Partition Tolerance** here are key requirements.

- **Log Service**

  REST server providing APIs used to **read and write unmodifiable logs about Contributions and Grid Services Invocations**; this Cloud Service is particularly important to both monitor what is happening inside the Grid and also correctly identify which Node has performed a Contribution and how much of said Contribution it has done. **Access speed is the most relevant factor here, making it acceptable to have eventual consistency but not delays**; then, **AP database technologies are required here**. In particular, said AP database technology should use an RDF model (also known as Triplestore) which is particularly suited for log data.

5.1.1.2 Adapters

- **Broker Service**

  Cloud Service that **acts as middleware between the Grid Master and the Nodes** (*figure 5.3*). There are **as many instances as needed to sustain the load of the connections to the Nodes**; instances are **created and removed by the Grid Master** taking into account the **geographical location** of said Nodes in order to reduce latency for the Broker-Node connection. The Broker **executes the coordination commands given by the Grid Master** while **also managing the Nodes connected** through its communication channels exposed by its Web Socket.

  Let us take, for example, a Grid Service Invocation: the Grid Master contacts a Broker Service instance that is geographically convenient in relationship to the location of the Invoking Endpoint; the Grid Master will delegate to the selected Broker Service instance the responsibility of gathering adequate Resources for the execution of the particular Grid Service requested. The Broker will then spread the request of said Resources to its connected Nodes and gather the responses of the ones that are adequate and available. The Broker will group the info needed to contact such nodes and forward it to the Grid Master that will then be responsible to forward in turn to the requestor.

- **Broker Discovery Service**

  This REST server has just one simple responsibility: **it provides to a Node the address of a Broker Service in order to make possible a connection between them**. This discovery mechanism is necessary since, as already stated, the Broker Service instances are dynamically created, and thus they do not possess a static address; on the contrary **the Broker Discovery Service will necessarily need a static address known by the Nodes**. The traffic directed to this Cloud Service and its computational load tend to be minimal and require only one instance but, as time goes on and the contributing user base increases, new static instances can be added easily.

- **Grid Services Gateway Service**

  In order for Invoking Endpoints to access Resources, they need to interact with this REST server; **it exposes APIs for a standardized and parameterized access**

to Resources through the Grid Service abstraction, meaning that a Customer expresses its request in terms of what Grid Service it wants to execute, not in terms of single Resources.

This Cloud Service is **connected via Web Socket to the Grid Master Service in order to gather the necessary Resources** but, before that, the Grid Services Gateway Service **needs to contact the Accounting Service** in order to execute the Fee payment. Lastly, the Cloud Service is **also connected to the Log Service** in order to register the Grid Service invocation and the consequent usage of Resources happened during the computation.

Similarly to the Broker Discovery Service, the number of static instances can easily vary in the project's lifecycle.

- **Authentication Service**

  Before being able to communicate with any other Cloud Service belonging to the Adapters layer, **any entity needs to authenticate to the Grid through this Cloud Service**. Given that **the two actors that require authentication are the Customer and the Contributor**, the Authentication Service utilizes both the **Customer Data Service and the Contributor Data Service** in its functioning. As a consequence of the authentication, the entity that wants to interact with the Grid will receive a **token that uniquely identifies it inside the Grid system**.

- **Billing Service**

  Cloud Service exposing APIs used **to make monetary transactions for Rewards Redemptions and access monetary balances**. It acts as an intermediary, protecting the actual payment process effectuated by the Accounting Service.

- **Historian Service**

  The Historian Service exposes APIs that allow the **read-only access to event already happened in the Grid System as well as the registration of new events** such as monetary transactions, Contribution, Grid Services Invocations, etc...

5.1.2 Contributor area

The **Contributor area**, as shown in *figure 5.4*, comprehends **three entities**:

- **Node**
- **Contributing Endpoint**
- **Contributor Dashboard**

The relevant Cloud Services are shown in order to explain the interactions that Node and the Contributor Dashboard have with them; in particular, the Grid Master Service is shown to emphasize the Grid Master - Broker - Node connection seen in *figure 5.3*.

Moreover, the Invoking Endpoint (which belongs to the Customer area) and the additional Node instance are included to show the connections that a Node can have with other entities that do not belong to the Cloud Services.

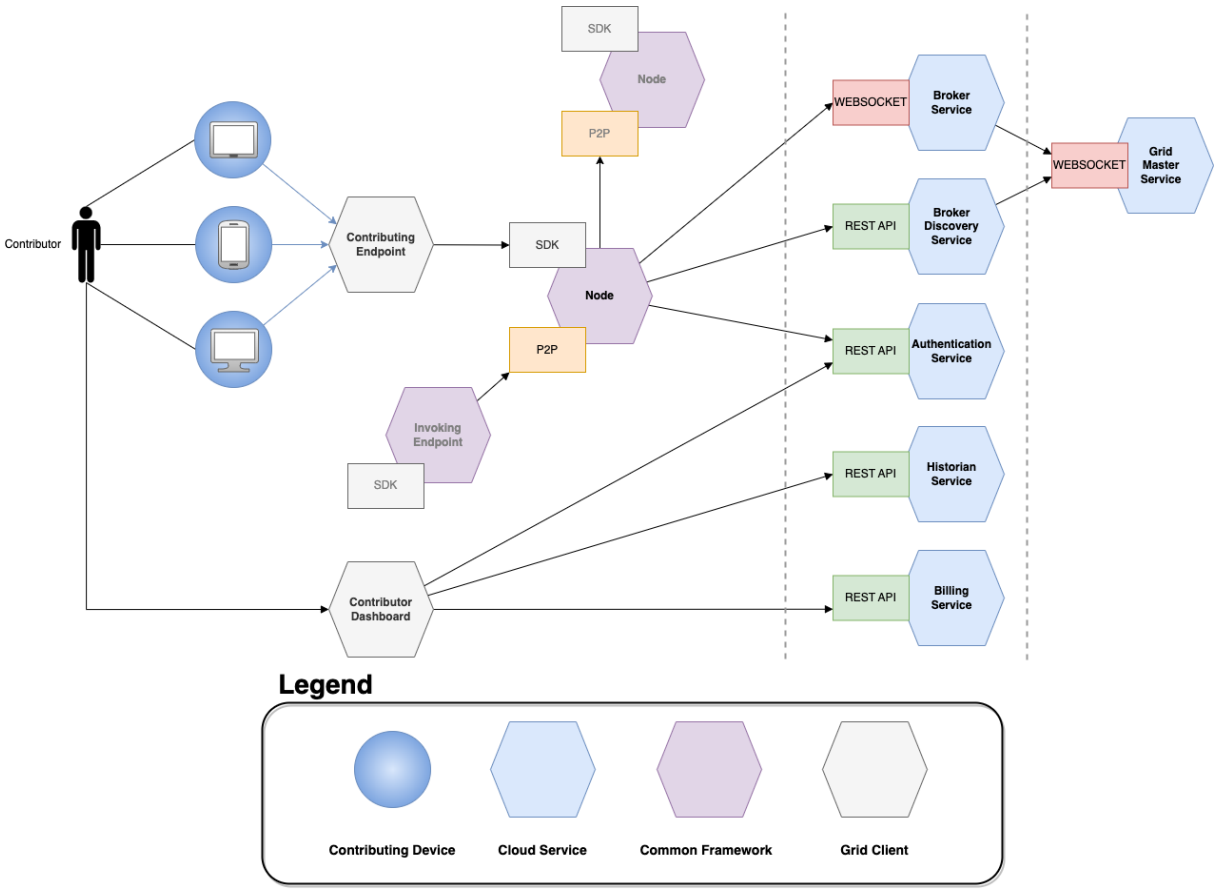


Figure 5.4: Architecture: Contributor area

5.1.2.1 Node

This is the **primary entity when it comes to contribution**. Continuing the definition provided in the Ubiquitous Language (*section 4.2.1*), this software abstraction is identifiable as a **set of core Contribution functionalities and logic which needs to be integrated** (through the offered SDK) **to an actual software application** (Contributing Endpoint) capable of **interacting with the specific resources of the device that is running on**; thus, a Node exists in order to get a **fast integration with as many devices as possible to maximize the compatibility for Contribution**, removing the need to reimplement everything for every new Contributing Endpoint.

The model of this solution aims to grant the access to Grid Services also to low-spec devices; thus, it is important to **delegate as much work as possible to the Nodes**. In order to complete a Grid Service, the Node can **Contribute in one of the following ways** (depending on the particular Service):

- **Direct communication with the Invoking Endpoint**
  A Node is connected, through a P2P connection, directly to the Invoking Endpoint. Let us take, for example, a hypothetical Grid Service that consists of delegating a computation to a single Node (i.e. non-distributed computation); the Node (Slave) and the Invoking Endpoint (Master) will exchange messages in order for the Invoking Endpoint to provide to the Node the necessary data and coordinating the Node that will actually perform the resource-demanding computation. In this communication mode a Node acts only passively. Another example, relevant for this thesis work, is the connection between an Invoking Endpoint and a Node acting as MapReduce Master (*figure 5.16*).

- **Communication with another Node**
  A Node is connected, through a P2P connection, to another Node; there are indeed circumstances that, for the completion of a particular Grid Service, require direct communication among Nodes. While in the other communication type the Node had only a passive role, here one of the two nodes needs to act as Master while the other(s) as Slave(s). Continuing the MapReduce example, a Node acting as MapReduce Master is connected to multiple nodes acting either as Map Workers or Reduce Workers (*figure 5.16*).

Once again, a Grid Service is performed by Nodes that Contribute, communicating with each other, performing Tasks in order to reach the end goal; the Task concept is vital here since, **while every Node will have the same communication interfaces**, **different Contributing Endpoints will implement Tasks based on the device's capabilities**. That means that, when concretized through the Contributing Endpoint, **not all Nodes will be able to execute every possible Task**, but only the ones that are compatible with the Resources possessed (e.g.: an iPhone running iOS will not be able to perform a MapReduce computation that uses Map and/or Reduce functions written in Java).

With the goal of reducing complexity, compatibility wise (*sections 1.3.2.4 and 3.1.4*), **the number of different Node incarnations** (that differ for the technologies used to develop it) **should be very low**; ideally, just one implementation, integrable with a great number of Contributing Endpoints, should exist to maximize the integration speed benefits.

5.1.2.2 Contributing Endpoint

A Contributing Endpoint is a **concrete application that will be used by the Contributor**. Through the Node's SDK, **it implements device-specific access to Resources and, consequently, the Tasks that this particular Contributing Endpoint implementation will be able to support**.

In particular, it is important that a concrete implementation incorporates the **security measures available for that particular execution environment** (collaborating with the general security mechanisms of the Grid). **There can be different Contributing Endpoint implementations**, with every new one expanding the number of devices able to Contribute.

5.1.2.3 Contributor Dashboard

A Dashboard, **used by the Contributor, to perform operations linked to its Contribution** (Rewards Redemption, checking past Contributions, etc...); it should serve as a **unified way to access to information regarding all its different Contributing Endpoints**.

This is certainly an easier and more traditional client application, requiring only to contact Cloud Services to represent through a GUI the information obtained, as well as performing some operations specified in the Contributor's use case diagram (*figure 4.3*).

5.1.3 Customer area

Figure 5.5 shows the **three entities** belonging to this area:

- **Invoking Endpoint**
- **Customer Custom Application**
- **Customer Dashboard**

As for the Contributor area, the relevant Cloud Services and the Node instance are shown to emphasize how these entities collaborate with each other.

It is also important to highlight the fact that both the Customer Custom Application User and the Customer itself are present in the figure since the actor using the Customer Custom Application do not necessarily coincide with the Customer; who the Customer Custom Application User actually is highly depends on the application itself.

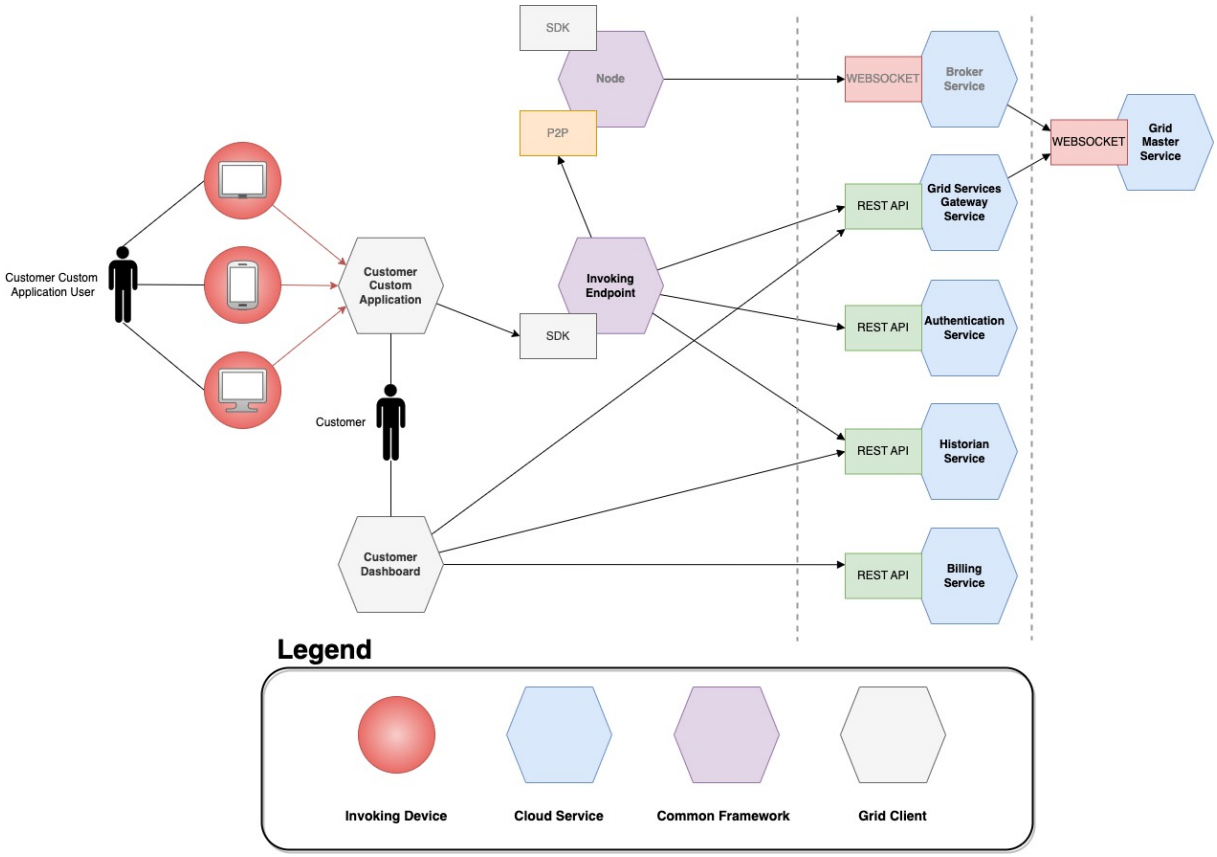


Figure 5.5: Architecture: Customer area

5.1.3.1 Invoking Endpoint

The Invoking Endpoint is the **main software entity used by the Customer to perform Grid Services Invocations**. It is a **library**, developed by the Grid Owner, that can be **integrated as a dependency and utilized into any software solution through the use of the exposed SDK**; in particular, an application utilizing an Invoking Endpoint is identified as Customer Custom Application.

Many Invoking Endpoint implementations can exist, depending on the target execution environment and which Grid Services that specific Invoking Endpoint wants to grant access to (a Customer may want to use only a certain type of Grid Service); **the requirement for any Invoking Endpoint is**, in order to connect to a Node in a Grid Service Invocation, **to conform to the communication interfaces belonging to the Grid Services which is interested to use**.

An Invoking Endpoint (after being authenticated through the Authentication Service) contacts the Grid Services Gateway Service in order to request a Service Invocation and, as a consequence of that, creates a connection to the provided Node(s). In order to grant the possibility of Grid Services invocation from low-spec devices, **the Invoking Endpoint needs just to perform a lightweight coordination of the Resources obtained while the computationally demanding work is executed by the Nodes connected**; if the coordination is too complex and demanding to realistically be executed from any device, the Invoking Endpoint just connects to a Node which actually performs the coordination: an example of such occurrence is the MapReduce Service where the MapReduce Master is the one performing the coordination of Map Workers and Reduce Workers while the Invoking Endpoint is only connected to the MapReduce Master.

5.1.3.2 Customer Custom Application

An application utilizing the Invoking Endpoint. It is **developed by the Customer** and **utilizes Grid Services** (offered by the Invoking Endpoint) in order to build complex behaviors that are dependent on the application's domain.

Grid Services Invocation happens **through the invocation of the functions exposed by the Invoking Endpoint's SDK**, masking the underlying complexity requiring for the Customer to specify some parameters (like, eventually, the quantity of Resources that it wants to use).

5.1.3.3 Customer Dashboard

Similarly to the Contributor's one, this Dashboard is a **centralized means of accessing everything connected to the Contributor's account** (past Grid Service Invocations, payment methods, etc...) **as well as performing actions specified by the Customer's Use cases** (*figure 4.4*).

5.2 Use cases satisfaction: entities interactions

This section will show how the entities introduced in the Architecture just presented actually cooperate with each other in order to satisfy the use cases previously discussed in *section 4.2.2*.

5.2.1 Contributor

- **Register**

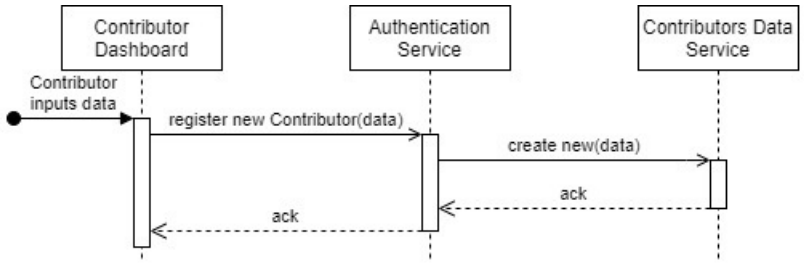


Figure 5.6: Contributor registration

A Contributor registers to the Grid system creating an account using the Contributor Dashboard; once it has inputted the necessary data, the Dashboard will contact the Authentication with the dedicated API that, in turn, will contact the Contributors Data Service, actually completing the registration process.

- **Login**

  When a Contributor performs a Login, both in the Dashboard case and in the Contributing Endpoint, it gets a token that will be used for every subsequent operation for both authenticating the communication with other entities and, at the same time, providing some useful data that will be used by said entities.

Since the token is a prerequisite for every further interaction it will be omitted for the sake of simplicity in the subsequent single use case satisfaction discussions.

– ***Login to a Dashboard***

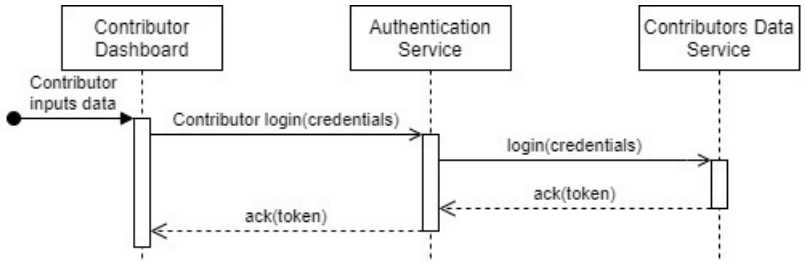


Figure 5.7: Contributor Dashboard login

The login process performed by a Dashboard is very similar to the registration one, diverging only in the invocation of a different specific API.

– ***Login from any Contributing Endpoint able to Contribute***

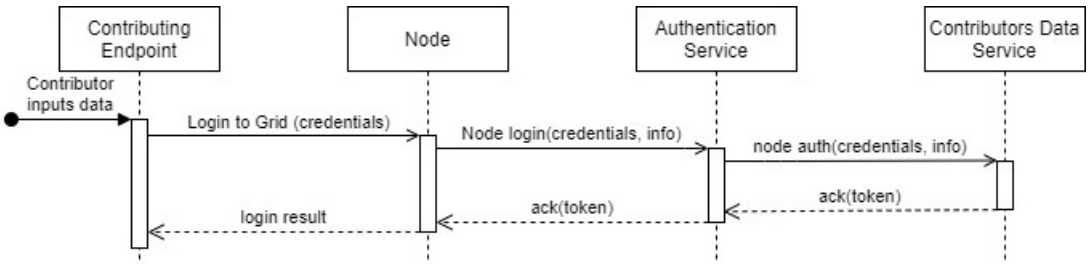


Figure 5.8: Node login

Here, the Contributor Application is the one actually triggering the Login in its integrated Node; then, the Node's login differs from the Dashboard's one calling another dedicated API that also requires to provide some additional info that will uniquely identify the device.

The Customer only needs to make the active effort of registering its account; every Contributing Endpoint will be automatically added to its account if the credentials are correct and no matching device for its account is found. One important thing to notice is that the token does not exit the Node's scope.

- **Passively Contribute offering a compatible device**

  When it comes to Contribution, the first step that actually needs to be performed is for a Node to connect to the Grid.

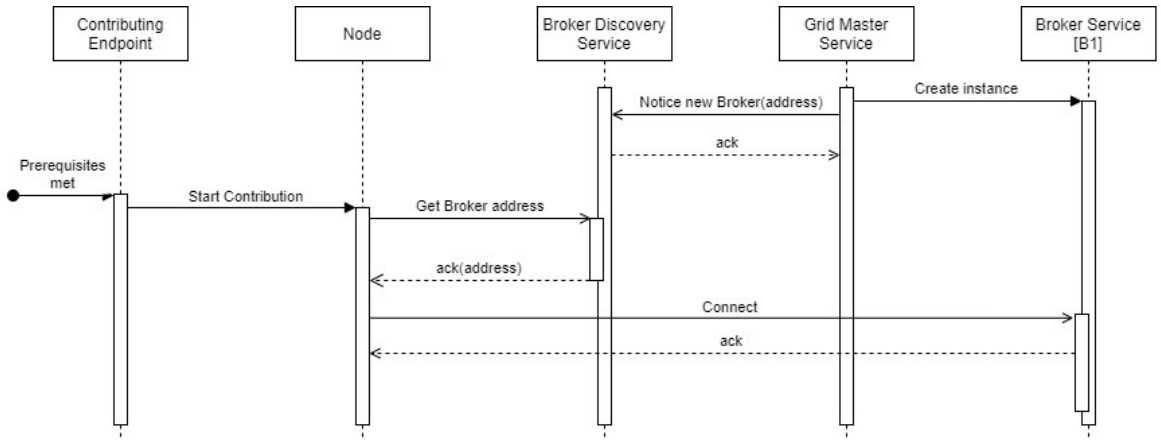


Figure 5.9: Connection to the Grid

First, independently of the Node, the Grid Master Service creates an instance of a Broker Service (there is always at least one instance of such Cloud Service) and, upon creating a connection with it, saves its address; said address is also sent to the Broker Discovery Service that, consequently, will always locally know the addresses of the Broker Service's instances.

The actual connection to the Grid is triggered automatically by the Contributing Endpoint once the prerequisites are met (an available Internet connection is present, the device's battery is above a certain threshold, etc...). The Node will proceed to contact the Broker Discovery Service that (without needing to contact the Grid Master Service for every new Node request) then provides the address of the more geographically convenient Broker Service instance; with the information obtained, the Node completes the connection process contacting [B1].

Now that the Node is connected to the Grid, the actual Contribution can happen. In order to have a clear understanding of the full picture, a Grid Service Invocation needs to be explained alongside the Node's Contribution.

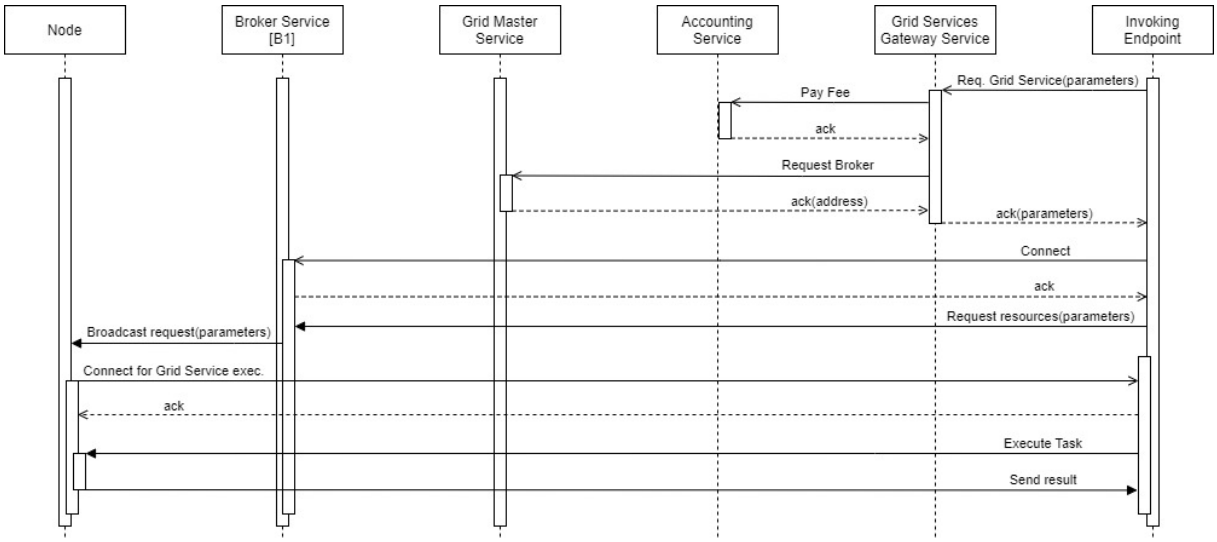


Figure 5.10: Node Contribution

The process starts with the Invoking Endpoint requesting a Grid Service to the Grid Services Gateway Service that, will first retrieve the Customer's payment method data (omitted in *figure 5.10* for simplicity) and then contact the Accounting Service to pay the Fee required for the Grid Service Invocation; then, the Grid Services Gateway Service will contact the Grid Master Service in order to obtain the address of a Broker Service instance.

The Invoking Endpoint, now having B1's address, contacts it and establishes a connection that, once performed, allows the Invoking Endpoint to request Resources. B1 broadcasts the request to all its Nodes, specifying the requirements needed for the Grid Service requested. The Nodes that are currently available, possess compatible Access Policies and do satisfy the requirements, take charge of the request and contact Invoking Endpoint and, if the Endpoint accepts it, a connection is created. Finally, the Invoking Endpoint is able to send Tasks that the Node will perform.

The process describes an Invoking Endpoint to Node connection; when the connection needs to be established between Nodes, the process is very similar. First the Invoking Endpoint to Node connection is created; then, another Resource request will ask for a Contribution to the Nodes but the exchanged messages (containing the Master Node address) will specify that a Node to Node connection is required. Depending on the particular Grid Service then, the Master Node itself will provide the Tasks to its Slave Nodes.

– ***Accumulate Rewards***

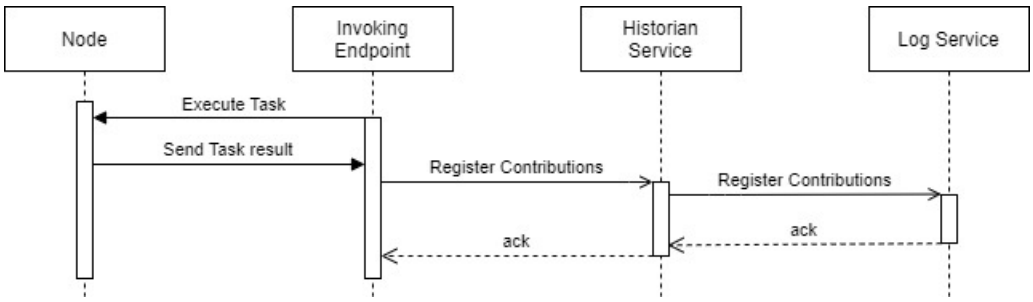


Figure 5.11: Contributor's Rewards accumulation

Upon a Node's Contribution, the Invoking Endpoint groups Contribution info and sends them to the Historian Service that will then proceed to store them through the Log Service.

– ***Manually start/stop Contribution from the Contributing Endpoint***

A Contributor can manually stop its Contributing Endpoint from performing further Contributions; this results in that particular Contributing Endpoint interrupting the connection (established in *figure 5.9*) with a Broker Service instance. This action is performed by the Contributor's input in the running Contributing Endpoint that will simply invoke a Node's function instructing it to stop the connection and, consequently, never connecting again to the Grid until the Contributor allows it again.

– ***Configure Access Policies to the Contributing Endpoint's Resources***

When a Node receives a broadcasted Contribution request by the Broker Instance which is connected to, in order to decide whether to take charge of the request or not, a series of conditions are evaluated; one of such conditions is the compatibility with the local Access Policies configured for that particular Contributing Endpoint. A Contributor, interacting with the Contributing Endpoint, can manually change the Access Policies that, from that moment, will be checked by the Node in its decision process.

– ***Check if Contributing Endpoints are currently Contributing***

The Node exposes functions that tell the current status regarding the Grid connection and if a Contribution to a Task is currently being performed; such information is displayed in the Contributing Endpoint's GUI.

- **Manage Contribution data through a Dashboard**
  - ***Check past Contributions***

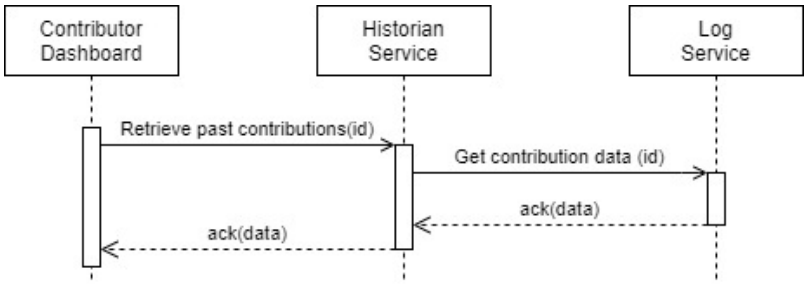


Figure 5.12: Contributor's past Contributions check

The Contributor Dashboard contacts the Historian Service, asking for the past contributions linked to the Contributor's Contributing Endpoints; the Historian Service contacts the Log service where the data is stored (see *figure 5.11*). Finally, the information is retrieved and then shown in the Contributor Dashboard.

  - ***Check Rewards Balance***

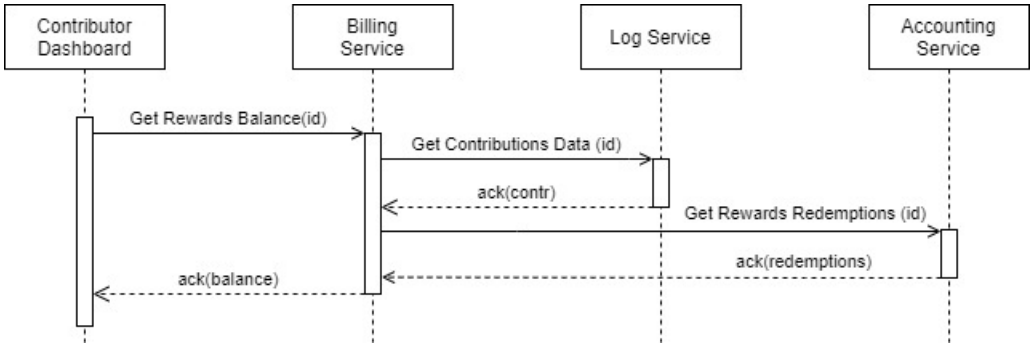


Figure 5.13: Contributor's Rewards Balance check

The Contributor Data retrieves, through a communication with the Billing Service, its current Rewards Balance. In order to calculate the current balance, the Billing service contacts both the Log Service (to obtain the Contribution data) and the Accounting Service (to obtain past Rewards Redemptions). The Rewards Balance is calculated by the Billing Service by summing the monetary value of the Contributions and subtracting from it the total Rewards Redemption.

– ***Rewards Redemption***

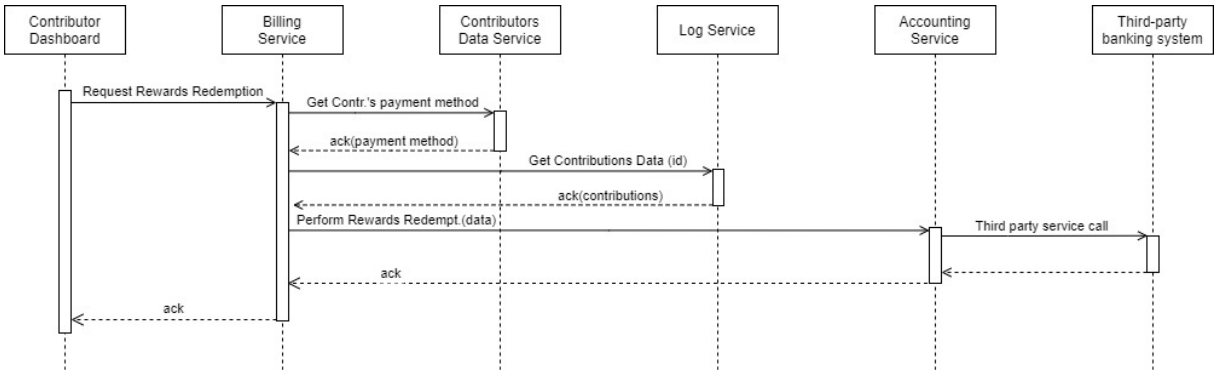


Figure 5.14: Contributor's Rewards Redemption

The Contributor Dashboard asks the Billing Service the execution of a Rewards Redemption. First the Contributors Data Service is contacted in order to retrieve the Contributor's payment method; then, the Log Service is contacted in order to retrieve the past Contributions data. Finally, the Accounting Service is interpellated, asking for the actual operation. The Accounting Service (that can access the past Rewards Redemptions on its own), performs the calculus of the Current Balance and, if the Balance is equal or higher than Rewards Threshold and, at the same time, compatible with the requested sum to redeem, the operation is finalized contacting a Third-party Banking System.

5.2.2 Customer

Various Customer's use cases are satisfied using processes that are very similar to the ones described in the Contributor's section; as a consequence, less dedicated diagrams will be used unless the process differs considerably from anything presented before.

- **Register**

  The overall process is very similar to the one shown in *figure 5.6*, requiring only to interchange the Contributor Dashboard with the Customer Dashboard and the Contributor Data Service with the Customer Data Service, respectively; while also the dedicated APIs invoked are necessarily different, the registration process follows the same exact logical flow.

- **Login**

  - ***Login to a Dashboard***

    Starting from *figure 5.7*, the use case is satisfied applying the exact same entities interchanges described in the previous use case.

  - ***Authenticate for Grid Services Invocations***

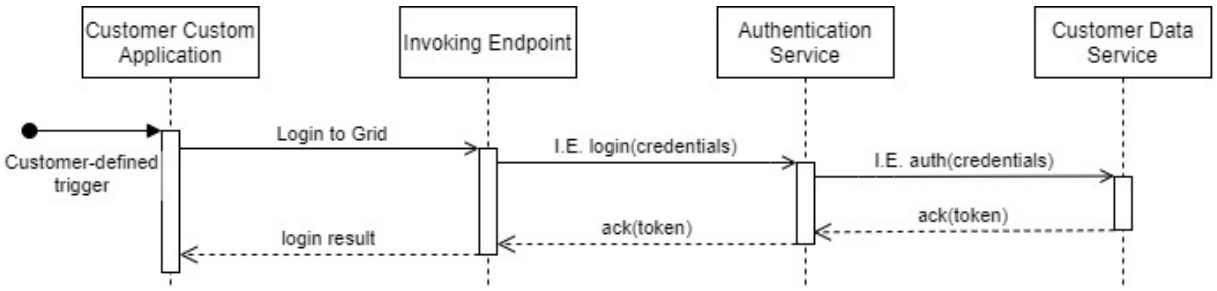


Figure 5.15: Invoking Endpoint login

Very similar process to the one described in *figure 5.8*; given the interchanged entities and the different APIs involved, the key difference here resides in the fact that the Customer Custom Application will implement a custom behavior that will trigger the Invoking Endpoint's login.

- **Easily integrate the Invoking Endpoint in a Customer Custom Application**

  - ***Invoke Grid Services through any Device***

    As previously explained, in certain Grid Services, the Invoking Endpoint demands the coordination of the obtained Resources to another Node (e.g.: MapReduce Service); this design choice allows to extend Grid Services invocation to also low-spec devices. As far as the technological requirements are concerned, this use case requires the existence of at least one compatible Invoking Endpoint implementation for every major platform. The Grid Services Invocation process was previously described in *figure 5.10*.

  - ***Request MapReduce service***

    An Invoking Endpoint implementation will need to expose methods (through the SDK) that enable the Customer Custom Application to perform a Grid Services Invocation.

While the specific design of the MapReduce Service will be explored in *section 5.3*, *figure 5.16* shows a high level view of interactions in the execution of said Grid Service (continuing the example seen in *figure 5.3*).

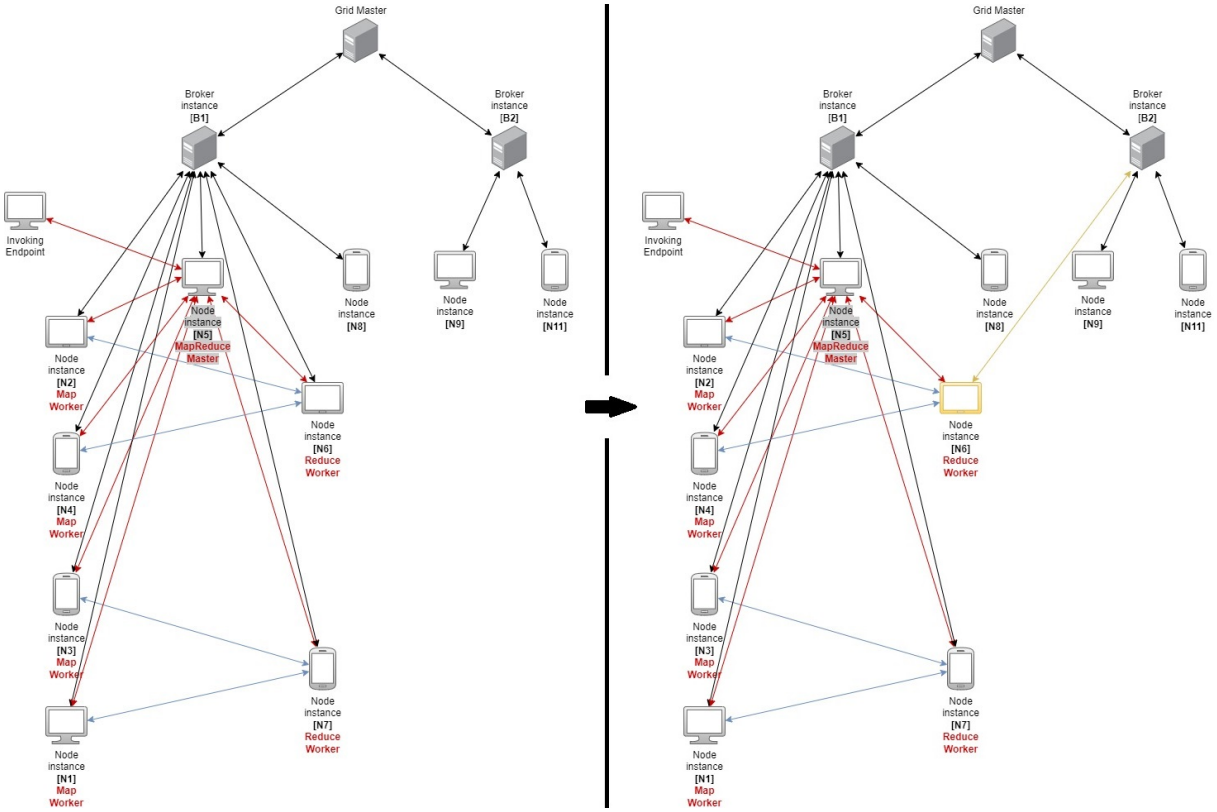


Figure 5.16: High level view - MapReduce Service and load balancing

The Invoking Endpoint connects with a Node acting as the MapReduce Master which, in turn, will act as coordinator for the execution of the MapReduce Service. Being the Contributing Nodes directly connected among each other, the Grid's load balancing (on the right) does not affect the execution.

* *Define MapReduce data source*

  This use case requires that the SDK exposed by the Invoking Endpoint allows to define the source of the data that needs to be analyzed, meaning that the data can come from the Invoking Endpoint directly or from an external source. This is done in order to actually be able to invoke the MapReduce Service from anywhere, since not all devices are able to contain large quantities of data.

* *Define Resource quantity to use*

  A Customer needs to specify, through the SDK, the quantity of Resources

that it wants to reserve for the MapReduce Service execution; while having more resources certainly makes the execution faster, the Fee necessarily increases.

* *Define Map and Reduce functions*
  Lastly, the Customer necessarily needs to define the Map and Reduce functions that will be utilized during the MapReduce Service execution; those are defined and provided through the usage of the Invoking Endpoint's SDK.

- **Manage requested Grid Services data through a Dashboard**

  - ***Check Grid Services Invocations history***
    Very similar to *figure 5.12*, using the Customer Dashboard and dedicated APIs instead.

  - ***Check running Grid Services Invocations***

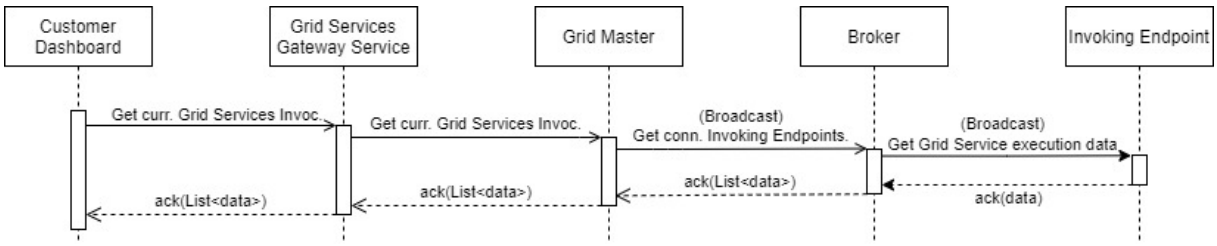


Figure 5.17: Running Grid Services Invocations check

The Customer Dashboard contacts the Grid Services Gateway Service in order to obtain info about the running Grid Services Invocations; then, the Grid Master is contacted which, in turn, will contact every Broker Service instance. Finally, every single Broker will broadcast the message to its connected Invoking Endpoints, asking for only the ones related to the Customer to respond. The responding Invoking Endpoints (if any), will return the data containing info about the current execution. The data will be progressively grouped and, in the end, returned as a final list to the Dashboard.

– ***Check past Grid Services Invocations Fees***
Assimilable to the process seen in *figure 5.12*, with the Contributor Dashboard acting as the invoking entity.

– ***Manage payment method***

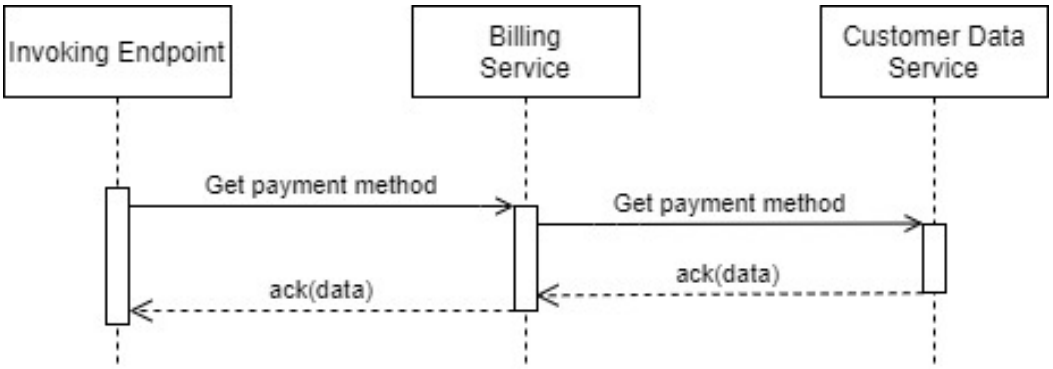


Figure 5.18: Contributor payment method retrieval

The management (retrieval, insertion, delete and update) of the Contributor's payment method is performed through the Contributor Dashboard collaborating with the Billing Service that, in turn, contacts the Customer Data Service, where the payment data is actually stored.

5.3 MapReduce Service

This section will explore the **logical design of the MapReduce operation performed as a Grid Service** (see *appendix A* for details on MapReduce). The process is modeled using three **Petri Nets**, each one **providing the point of view of one among the possible entities types involved**: Map Worker, Reduce Worker and MapReduce Master.

Every Petri Net that will be presented **assumes that the Resources' retrieval process** (seen in *figure 5.10*) **has already been performed, allowing to focus on the actual MapReduce process itself**. Furthermore, in every Petri Net snapshot provided, transitions that can be transitioned (in that particular configuration of the places) are highlighted in red to facilitate its understanding.

5.3.1 Map Worker

The process starts with the Map Worker **establishing a connection with the MapReduce Master** (*figure 5.19*).

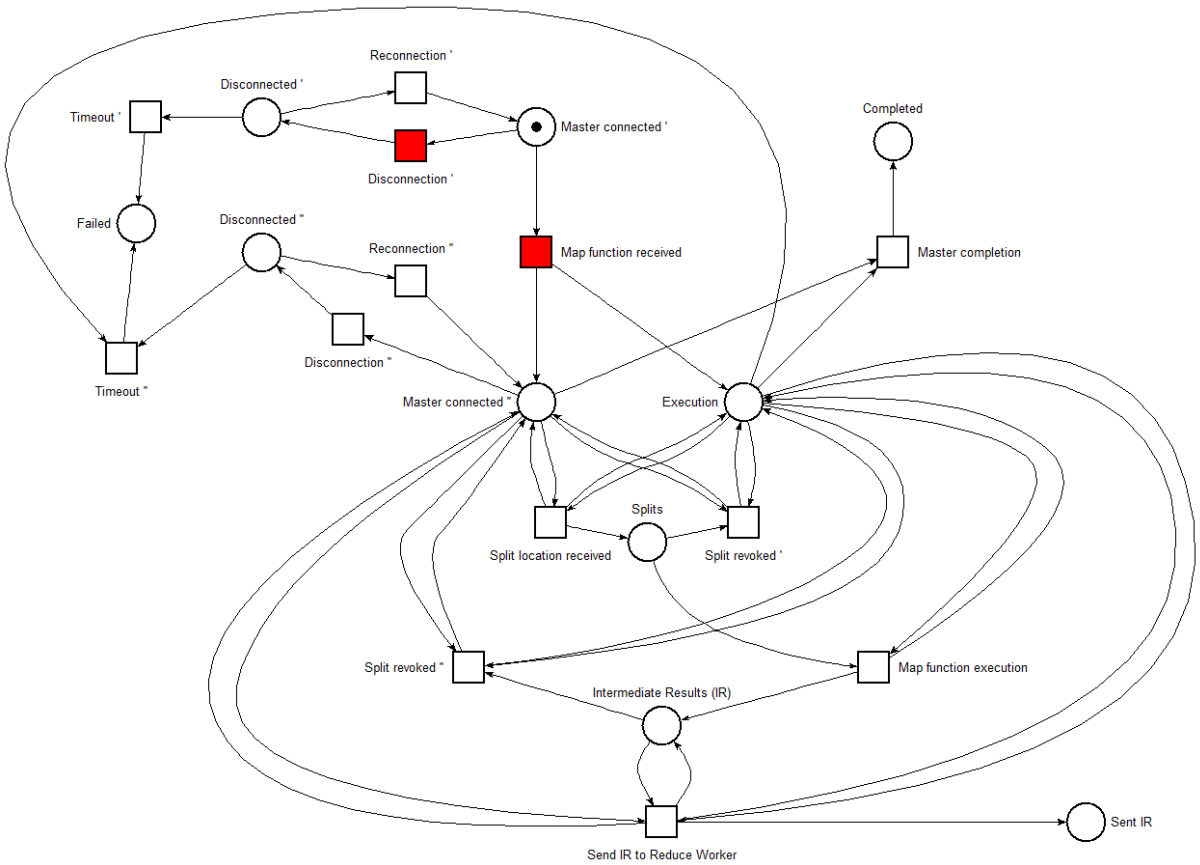


Figure 5.19: Map Worker - Start

Ignoring, for the moment, the *Disconnection'* transition, **the Map Worker receives the Map function from the MapReduce Master**, obtaining a token for the *Execution* place and one for the *Master Connected"* place. The **distinction between the concepts of connection and execution** is useful here since **it is not necessary to be connected to the Grid Master in order to execute the map function**; through this distinction, Resources can be used efficiently even if a temporary disconnection occurs.

Figure 5.20 shows how, **once the map function is received, the MapReduce Master starts to send the location where the data splits can be retrieved**; the Map Worker retrieves the specified data and **applies the map function to it producing the intermediate results (IR) as output**.

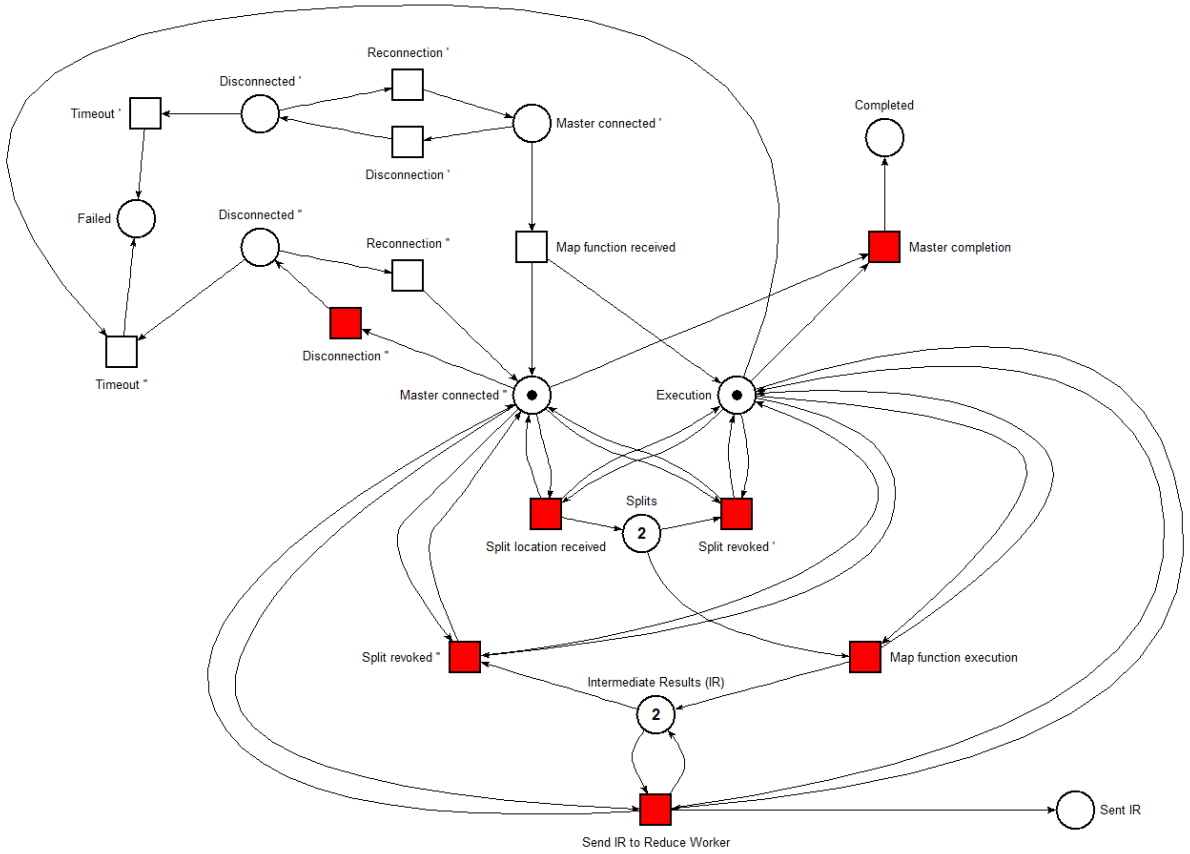


Figure 5.20: Map Worker - Mapping

As can be seen from the operations executed, **the Map Worker follows the instructions received by the MapReduce Master**. While mapping, **a data split can be assigned to multiple Map Workers**, in order to increase fault tolerance; as a direct consequence, **it is possible that a received split location is mapped by another Map Worker**. **Once the IR is already obtained on a particular split, the Map Worker sends a split revoked(1) instruction to every Map Worker assigned to that split**. The two different split revoked transitions possess a conceptual difference:

- ***Split revoked(1)***

  Here, **the Map Worker that receives this instruction cannot possibly have applied the map function to the split**, since the split revoked instruction is received only after the IR (produced by the mapping of that particular split) is reduced by a Reduce Worker. **Practically speaking, the Map Worker drops the split, removing it from memory or avoiding downloading it entirely**.

- ***Split revoked(2)***

  In this situation, **the Map Worker already performed the map operation on the**

considered split. Whether the IR that will be used for the reduce operation is actually the one locally produced or not is irrelevant; the direct consequence of receiving a split revoked instruction here is to **remove from memory the IR**. This is because **the MapReduce Master can ask multiple times to the Map Worker to send the IR to a Reduce Worker** (*figure 5.21*), **requiring to maintain in memory** (or on disk) **the IR until the split revoked instruction is received**.

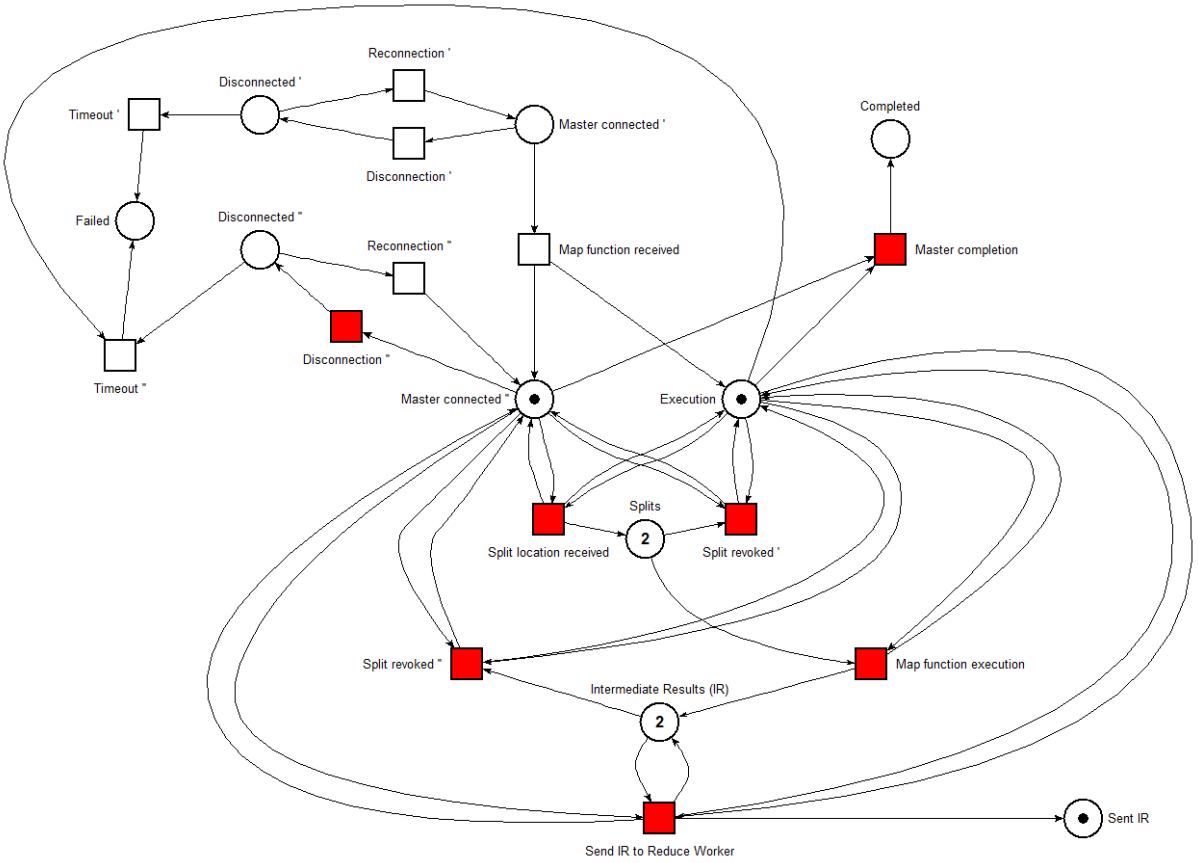


Figure 5.21: Map Worker - Intermediate Result sent

At any given time, **the Map Worker can lose the connection with the MapReduce Master** (*Disconnection'* and *Disconnection"* transitions); **before receiving the Map function, the process is necessarily locked until the connection is established once again or a timeout is reached**, leading to a local **failure**. On the contrary, **if the token on the execution place is present, the Map Worker is still able to apply the map function on a split that is locally available even though it is in a disconnected state**; on the other hand, **a timeout leading to a failure would also remove the execution token**, effectively stopping the entire process.

When the MapReduce operation is completed, **the Master will send a completion instruction** that will result in the removal of the connection and execution tokens, **correctly concluding the execution** (*figure 5.22*).

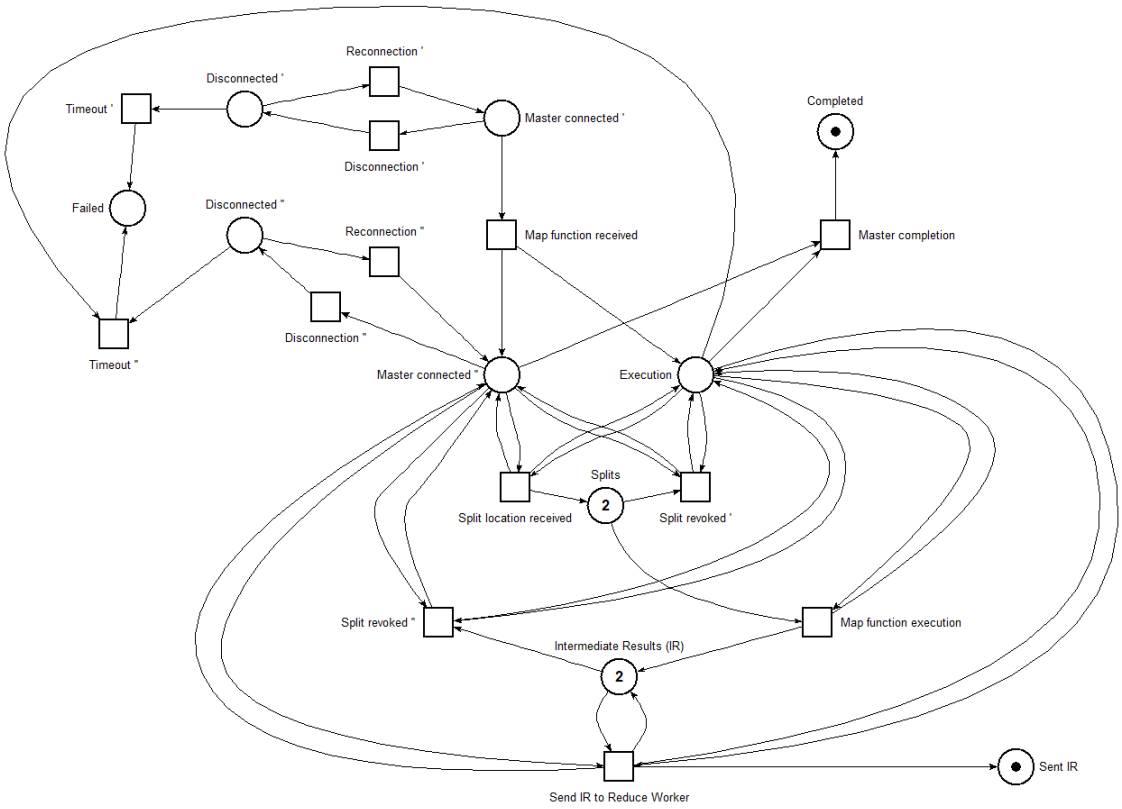


Figure 5.22: Map Worker - Completion

5.3.2 Reduce Worker

As the Map Worker's process, **a connection to the MapReduce Master is established and a reduce function is received, gaining a connection token and an execution token** (used for the same Resources' utilization optimization principle); **as before, a disconnection can occur and, if a timeout is reached, the process ends with a failure**.

The MapReduce Master sends the data about which IRs constitute a region (*figure 5.23* shows an example of a region composed by 10 IRs); **the Reduce Worker also receives**, progressively, **the IRs from the Map Workers that have completed the execution of a map function** and are instructed to send said data to that particular Reduce Worker. **Once all the IRs belonging to a region are received**, the Reduce Worker performs the **reduce function** on said data, producing a **result that is momentarily stored locally**.

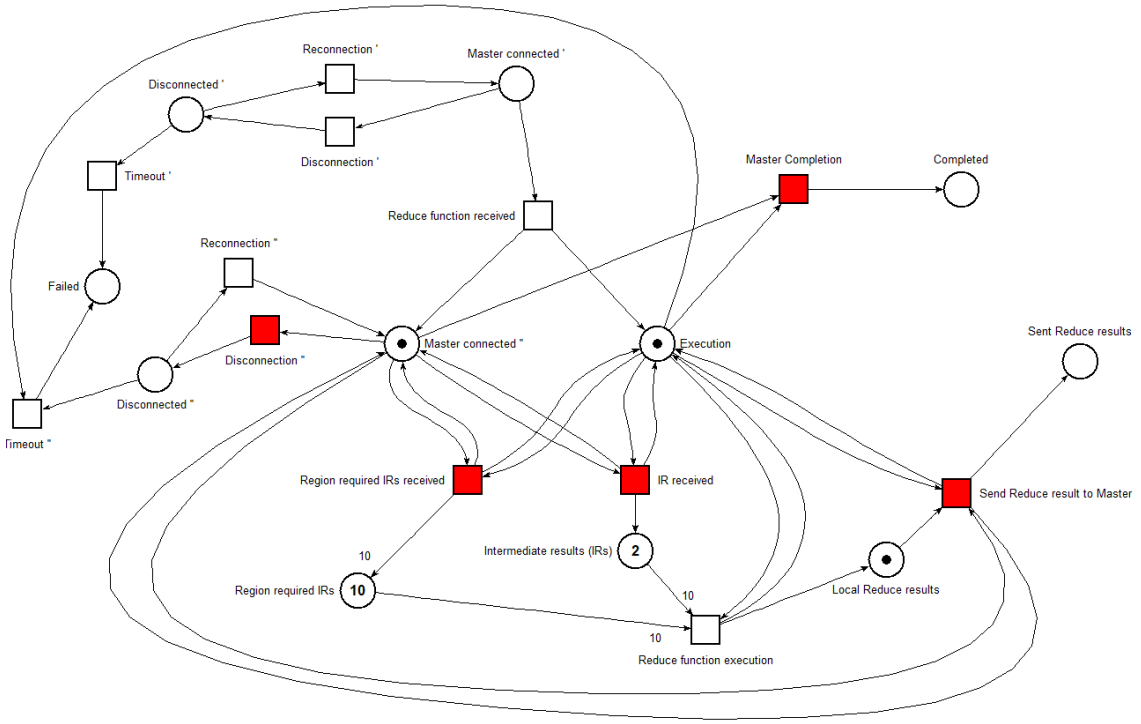


Figure 5.23: Reduce Worker - Local result

A result correctly sent to the Master is deleted locally. **A Reduce Worker handles multiple regions until the MapReduce process is completed**; once the Master sends to the Worker a completion instruction the execution ends correctly (*figure 5.24*).

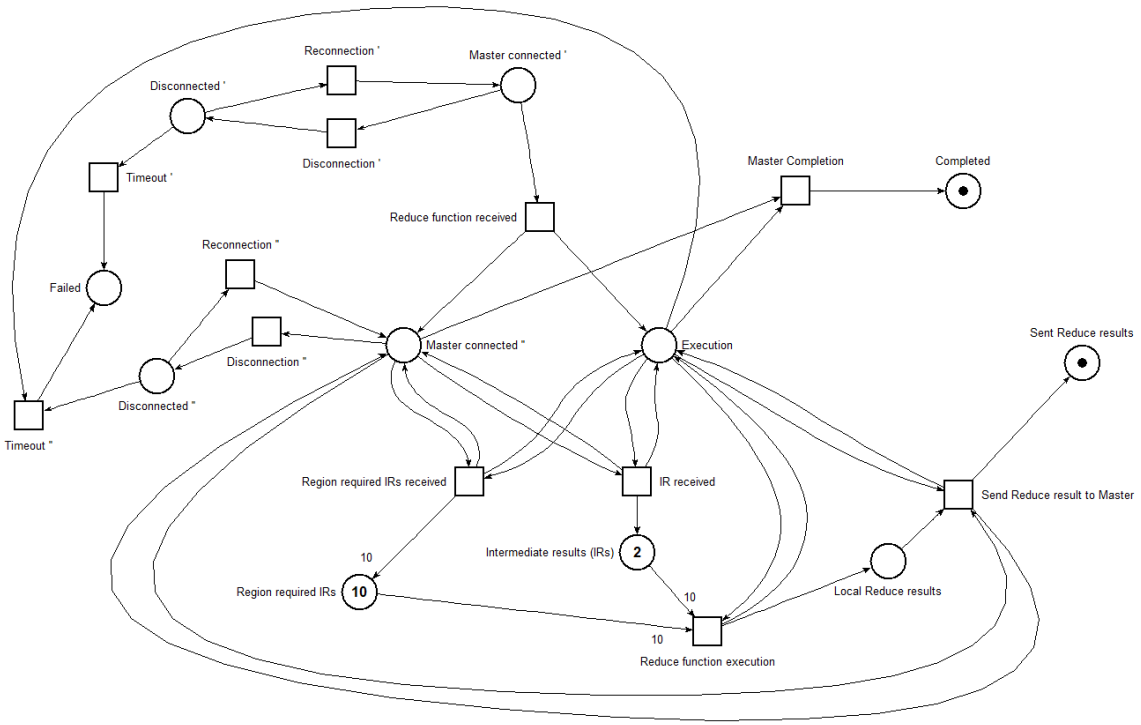


Figure 5.24: Reduce Worker - Completion

5.3.3 MapReduce Master

The Petri Net describing the process performed by the MapReduce Master **focuses only on a single region**, meaning that **this operation needs to be performed (concurrently) for each data region**.

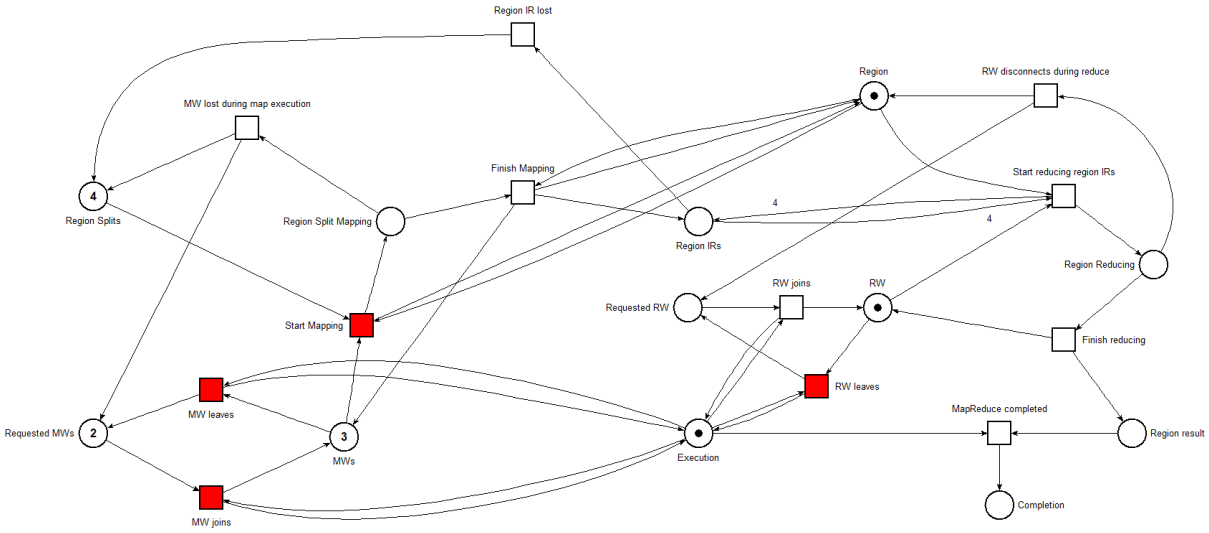


Figure 5.25: MapReduce Master - Recruitment

For this particular example, the region is constituted by 4 splits and the Customer required a quantity of Resources that resulted in the Contribution of 5 Map Workers; **a region is handled by a single Reduce Worker**, so there is only one token representing it.

Initially, **the Map and Reduce Workers are required but not connected** (*Requested MWs* and *Requested RW* places); they can **transition to a connected state** but, if something goes wrong, they **can also disconnect**. *Figure 5.25* shows a situation where 3 Map Workers and the Reduce Worker have joined. **Workers can continue to join and leave until the MapReduce computation for the region is completed**, removing the execution token.

Having at least one MW available, the mapping process can begin (*figure 5.26*). A **split** belonging to the region is **assigned to an available MW which will perform the Map operation on it** (it is assumed that the Worker possesses the map function, abstracted for simplicity). If the mapping happens successfully, **the region's IRs are accumulated**; **in case that a Map Worker is lost** during the map execution, **the split needs to be computed by another Worke**r and is **placed back in the *Regions splits* place** (and the another token in the requested MWs is also placed). It can also happen that **a region's IR is lost due to the disconnection of the Map Worker that is still**

keeping the result in memory; this occurrence also **adds again said split in the *Region splits*** that still needs to be mapped (since its IR is unreachable).

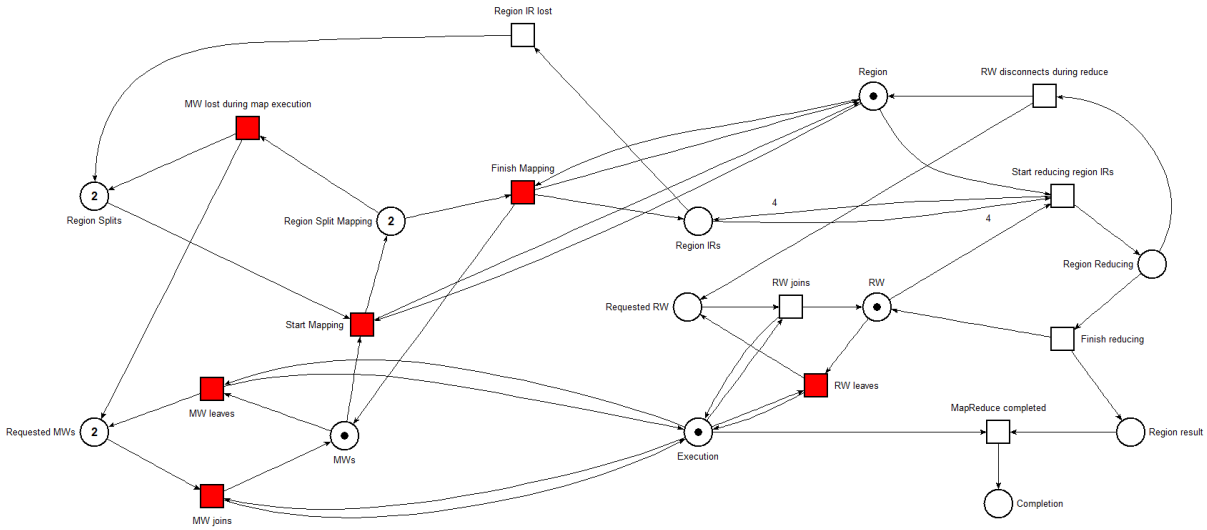


Figure 5.26: MapReduce Master - Mapping process

Once **all the region's split are mapped** to intermediate results (*figure 5.27*), **the reduce process can start** (assuming that a RW is connected).

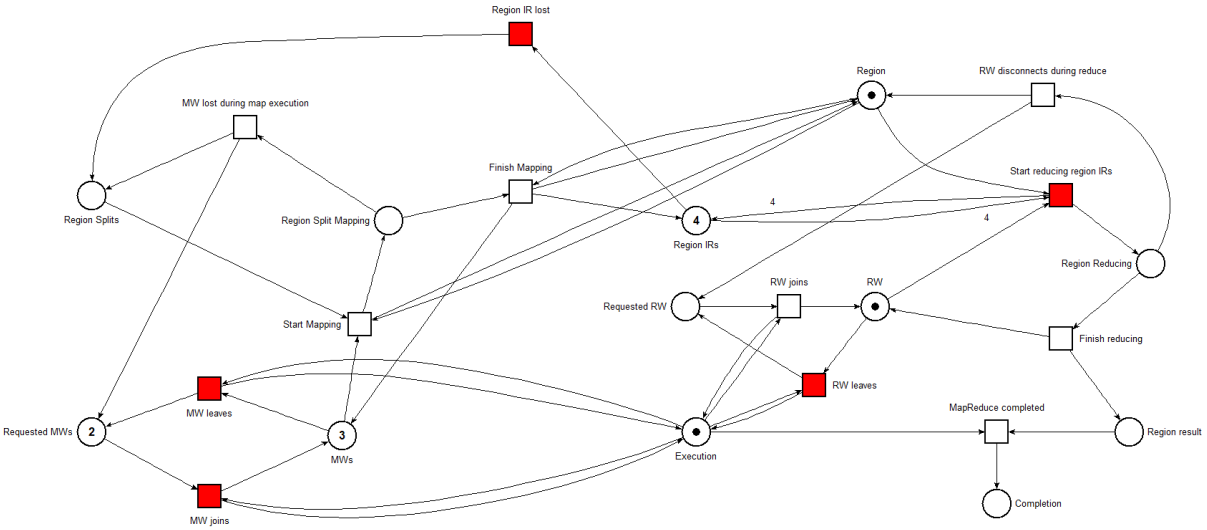


Figure 5.27: MapReduce Master - Region Mapping completed

While the reduce function is being applied, the region token is removed due to the fact that no more splits for this region need to be mapped (*figure 5.28*). **A failure can also happen here, since the Reduce Worker can disconnect during the reduce function execution; in this case, the region token is restored and a new Reduce Worker is required**.

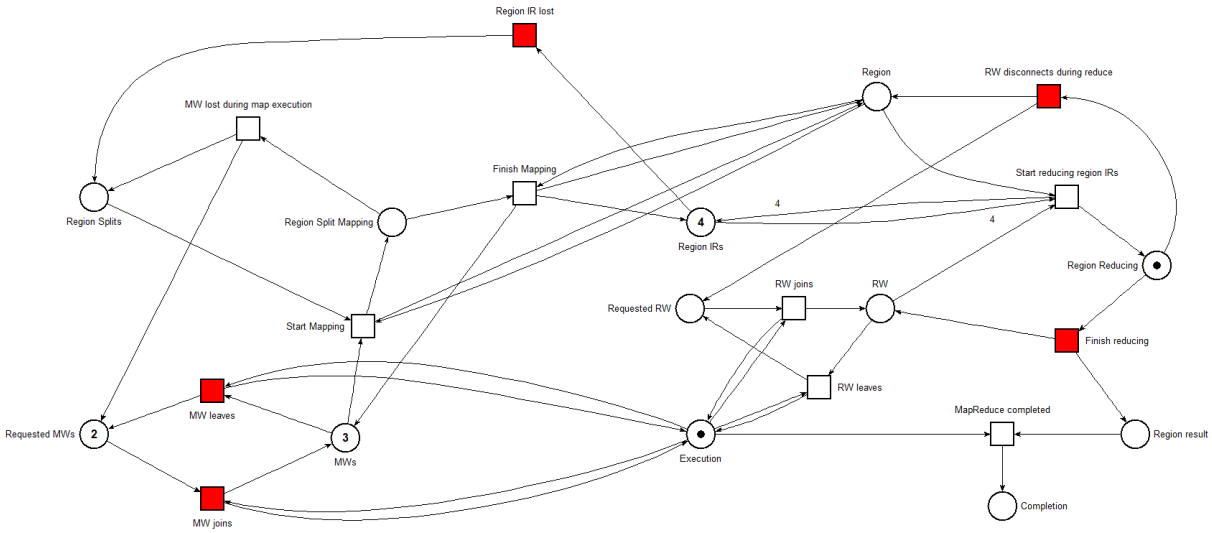


Figure 5.28: MapReduce Master - Reducing process

If the reduce function is completed successfully (as shown in *figure 5.29*), **the region's result is obtained**. As stated before, the overall MapReduce execution will end once every region is computed, providing to the Customer all the regional results.

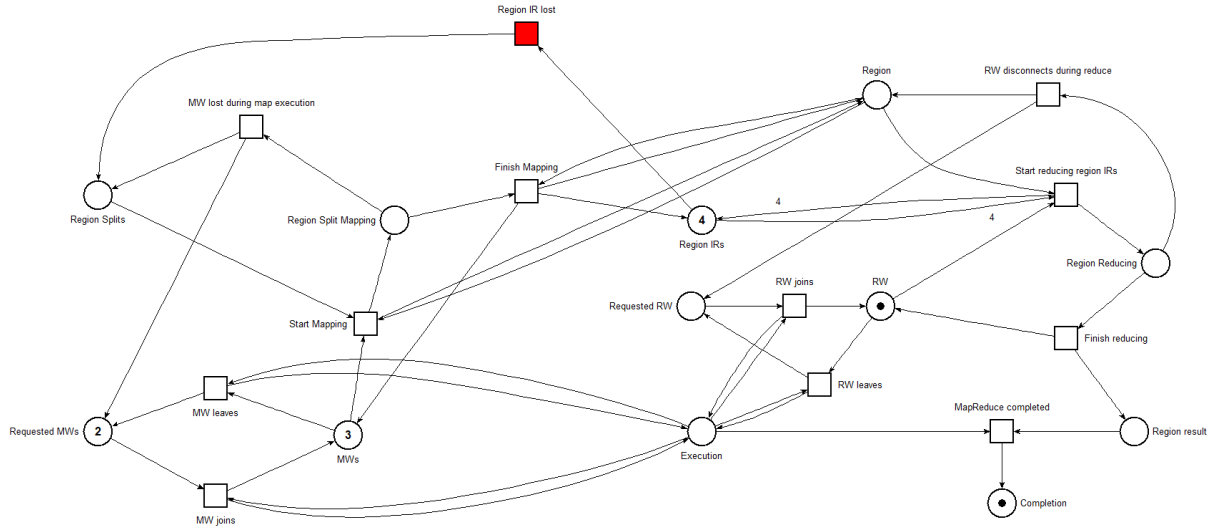


Figure 5.29: MapReduce Master - Region completed

Chapter 6

Interconnected: a working prototype

This chapter discusses the creation of a working prototype that realizes a subset of core functionalities of the system. Firstly, the goals and context for the prototype are discussed. Then, the simplified architecture for the prototype is presented, providing also details on the realization of the single entities that compose such architecture. The third section will then contain a brief explanation of the setup related to DevOps methodologies. After that, the Coordination section will talk about the messages exchanged between the entities in order to realize the desired functionalities. Concluding this chapter, the simplified Proto-MapReduce implementation realized will be discussed and, after that, data regarding real-world experiments will be presented.

6.1 Goals and context

After having engineered a complete system that satisfies the use cases defined in *section 4.2.2*, it is now time to create a **prototype** that brings a portion of it to reality. The prototype takes the name of the "***Interconnected project***".

Before explaining the work behind this prototype, **a few premises** have to be clarified in order to understand some choices taken during the development.

Firstly, the **focus of this work** is to **discuss the topic of integrating mobile devices in a Grid system** and, most importantly, **engineering a solution that defines a blueprint for actually bringing the idea to reality**. **Due to a lack of resources** (economical, time, equipment, team members, etc...), **the prototype does not try to realize the whole system** defined in the previous chapter, **but only a subset of core functionalities** with

the goal to **demonstrate that the core idea is also technologically feasible**.

As a direct consequence of that, **the prototype will not realize aspects that**, while certainly important in a real product, **do possess already well-established technologies** (authentication, data storage, server instances replication, payment systems, etc...), **focusing only on the innovative aspects** of the project.

6.2 Simplified architecture

Figure 6.1 shows a complete view of the entities composing the simplified architecture for the Interconnected project (that can be compared to the full architecture shown in *figure 5.1*).

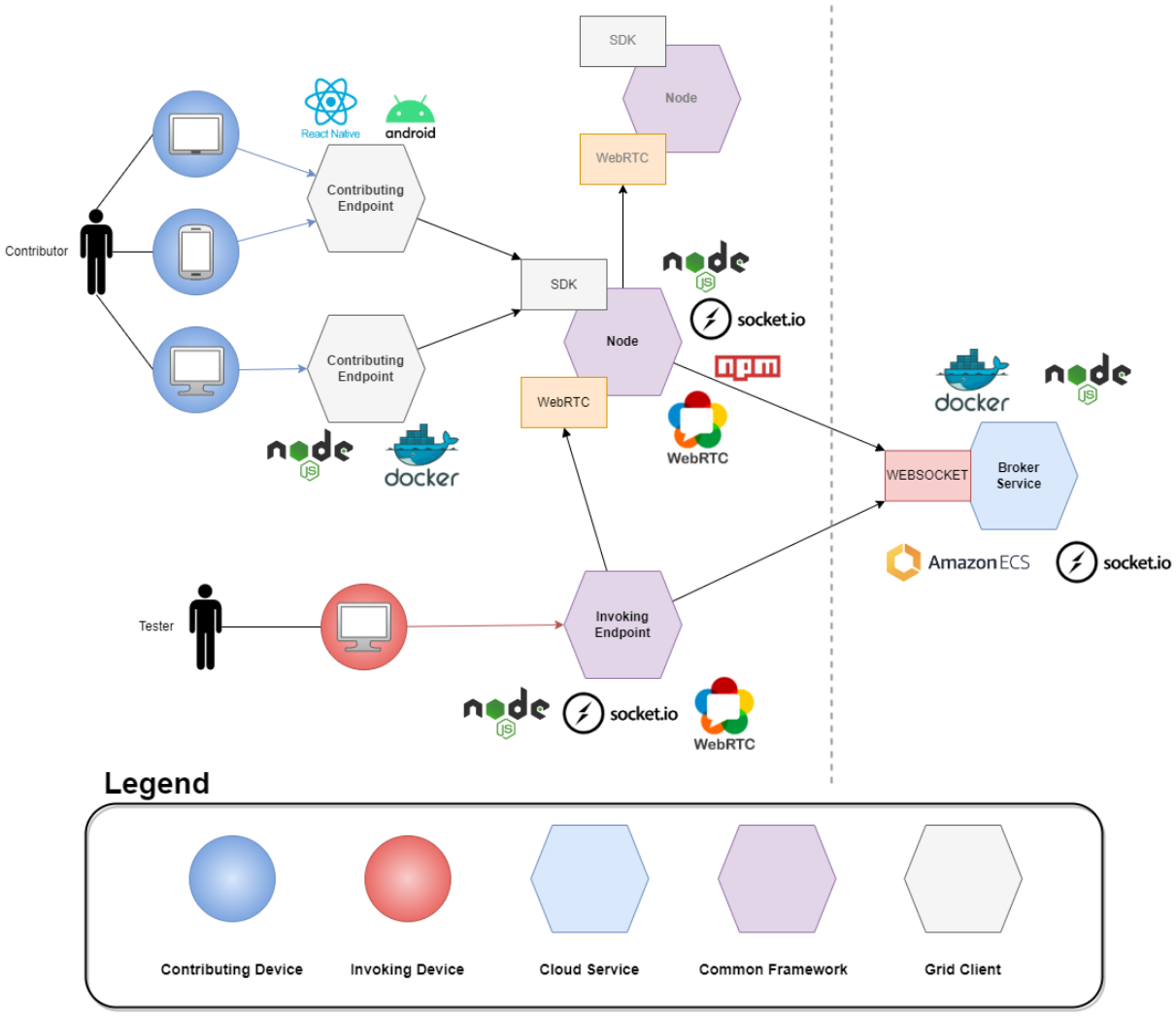


Figure 6.1: Complete view of the simplified prototype's architecture

All the entities composing the architecture **are implemented using technologies based on Node.js** which **offers useful, popular and well-maintained frameworks** as well as an **easy-to-use concurrency model based on an event loop**.

6.2.1 Broker Service

The **only Cloud Service** present in this simplified architecture is the **Broker Service**; since **only one instance** (with a static address) of such Cloud Service is required to sustain the workload of the few devices connected, **a dynamic handling and discovery of multiple instances is thus not required**. As a direct consequence of that, the complementary entities that handled the dynamic system for scalability (Grid Master Service, Broker Discovery Service and Grid Services Gateway Service) are not needed and thus **Nodes and Invoking Endpoints communicate directly with the Broker Service without first undergoing the discovery processes** described in *section 5.2*.

The prototype's Broker Service is implemented by using the **Typescript** language and utilizes **Socket.io** in order to create a web server that is reachable via WebSocket protocol; through the use of this technology, **Invoking Endpoints are able to deposit requests for the recruitment of Nodes that will be used to perform computations**. The messages involved in the recruiting process and the general Grid connection will be discussed in *section 6.4*.

In order to have a running instance of the Broker Service that also has a static address reachable from anywhere, a **Docker image** is created which, in turn, is executed in a container through the use of **Amazon ECS** (Elastic Container Service). More details about the deployment of this entity will be discussed later in *section 6.3*.

6.2.2 Interconnected Node

The **Node** entity, which in the Interconnected project takes the name of "***Interconnected Node***", **contains all the logic regarding the contribution of a device**, and it is **distributed through NPM as a Node.js dependency** that is then **integrated in the concrete clients targeting various devices**.

Interconnected Node is also developed by using the **Typescript** language and, being this a client in the WebSocket connection to the Broker Service, it utilizes the client-side implementation of the **Socket.io** framework.

The **P2P connectivity** requirements are concretized through the use of the **WebRTC** (Real-Time Communication for the Web) protocol; such communication standard it is **used to send audio and video data among peers, as well as any kind of structured non-media data**. **In order for the peers to establish the connection, first the signaling process needs to be completed**: the Peers, through a third intermediary (the Broker Service), exchange some information required for the connection to happen; the Peer that initializes the connection creates an **SDP** (Signaling Description Protocol) object, denominated "**offer**". When the other Peer receives such data, it also creates its SDP object denominated "**answer**". Once each Peer possesses both generated SDP data, they finalize the connection exchanging some **ICE** (Interactive Connectivity Establishment) **candidates**; such ICE candidates need to be specified when realizing a WebRTC connection since they are the actual servers that allow the P2P connection. There are **two types of servers involved**:

- **STUN** (Session Traversal Utilities for NAT)
  Used by Peers that reside behind the same NAT; through this server the IP info of each Peer are retrieved and a direct P2P connection can be established.

- **TURN** (Traversal Using Relays around NAT)
  Used by Peers that reside in different NATs; through this server the limitations of a NAT are overcome, creating an indirect P2P connection that uses the TURN server to forward the messages among the two Peers.

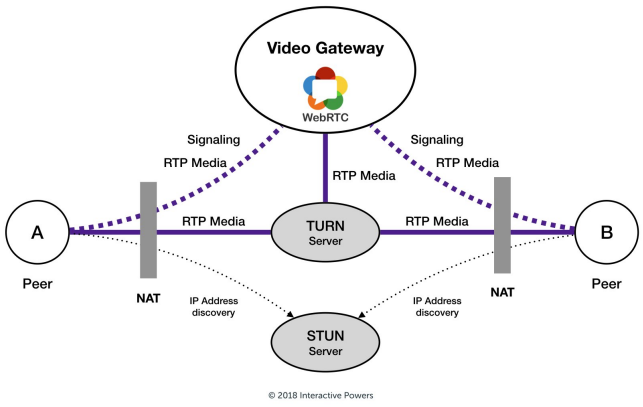


Figure 6.2: WebRTC - STUN and TURN servers [15]

Free STUN and TURN servers (used by this prototype) are available to the general public thanks to the **Open Relay** initiative.

There are both a **Desktop implementation** and a **React Native implementation** of Node.js modules that allow to perform P2P connections utilizing the WebRTC protocol but, despite functioning in the same way and exposing the same interfaces with identical methods, **the Desktop implementation does not work on React Native and vice versa**. In order to circumvent this problem, **the object handling the P2P connection is wrapped in an interface that needs to be concretized in the specific Desktop and Mobile implementations** (*figure 6.3*); this also allows to **add some domain-specific logic**, assigning to the Peer that initializes the connection the "**Master**" role and the "**Slave**" role to the other Peer.

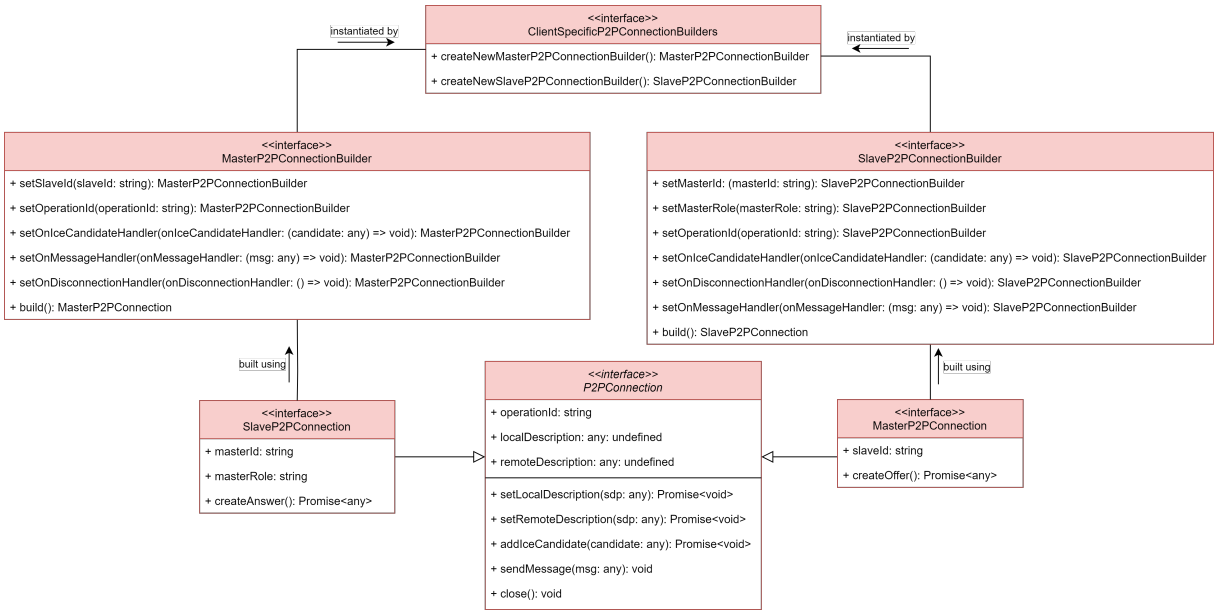


Figure 6.3: WebRTC P2P Wrappers and builders

The construction of the client-specific objects is standardized through the utilization of the Builder pattern, allowing the internal logic of the Interconnected Node to create instances of such objects without actually possessing the concrete implementation. **The P2P Builders** (along with a unique device identifier) **will then be passed at construction time to the InterconnectedNode Facade** (*6.4*) **which exposes simple methods to interact with the core logic**, hiding its complexity and requiring the specific client only to deal with presentational aspects and device-specific responsibilities.

When it comes to the **actual computational contribution** that the Interconnected Node needs to perform, **two key abstractions are present**: **Job** and **Task**; a Job is an activity that a Slave Peer executes under the guidance of its Master that, after instructing the start of said Job, sends Tasks to execute in that specific Job.

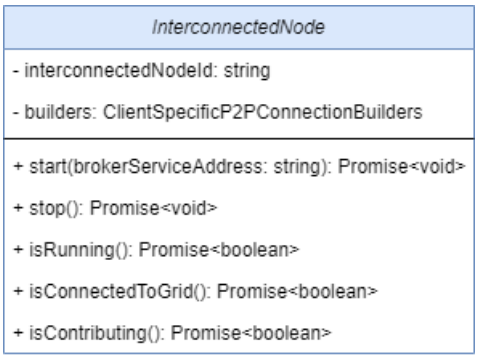


Figure 6.4: Interconnected Node Facade

This abstraction results in the definition of common interfaces for Job and Task that are then concretized realizing new functionalities that will then be used to execute Grid Services (MapReduceMasterJob, MapReduceMapWorkerJob, etc...); through this mechanism, expanding the support for new Grid Services only requires to define new concrete Jobs and Tasks, realizing their logic and their handlers for the messages exchanged among Nodes.

6.2.3 Interconnected Mobile Client

In this prototype, the **mobile incarnation of the Contributing Endpoint** takes the name of "**Interconnected Mobile Client**". The main technology used for realizing this client is **React Native** (which is also **Node.js based**); with this framework, it is possible to **build mobile applications targeting both Android and iOS devices with just one code base** while also **taking advantage of most of the modules available on NPM**.

Thanks to this Node.js compatibility, **the mobile client is able to use the Interconnected Node to connect to the Grid and perform contributions, only requiring to implement previously mentioned client-specific interfaces and to provide an ID for the device** (given the limited number of devices in the prototype setting, a **UUID v4** is generated and used as the ID).

Although React Native also allows targeting iOS devices, **this prototype is tested only on Android devices**; this limitation is caused by the previously mentioned lack of resources (in particular the unavailability of Apple devices to test it on) but, in theory, the application should also function on iPhones and iPads with little to no changes to the code base.

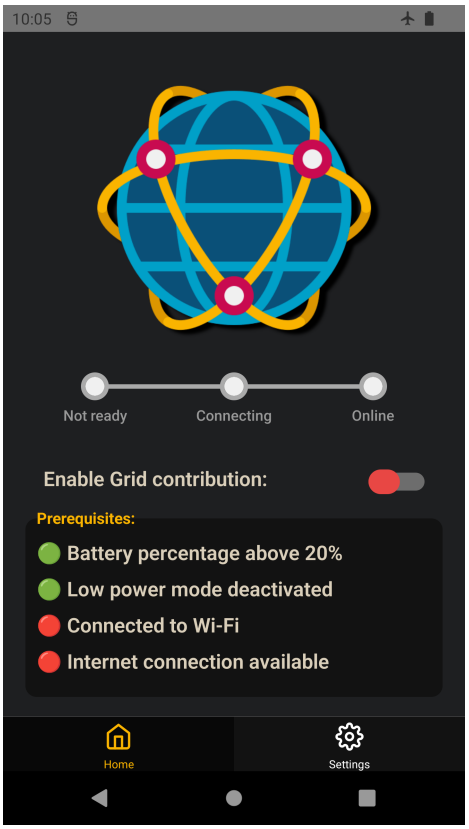
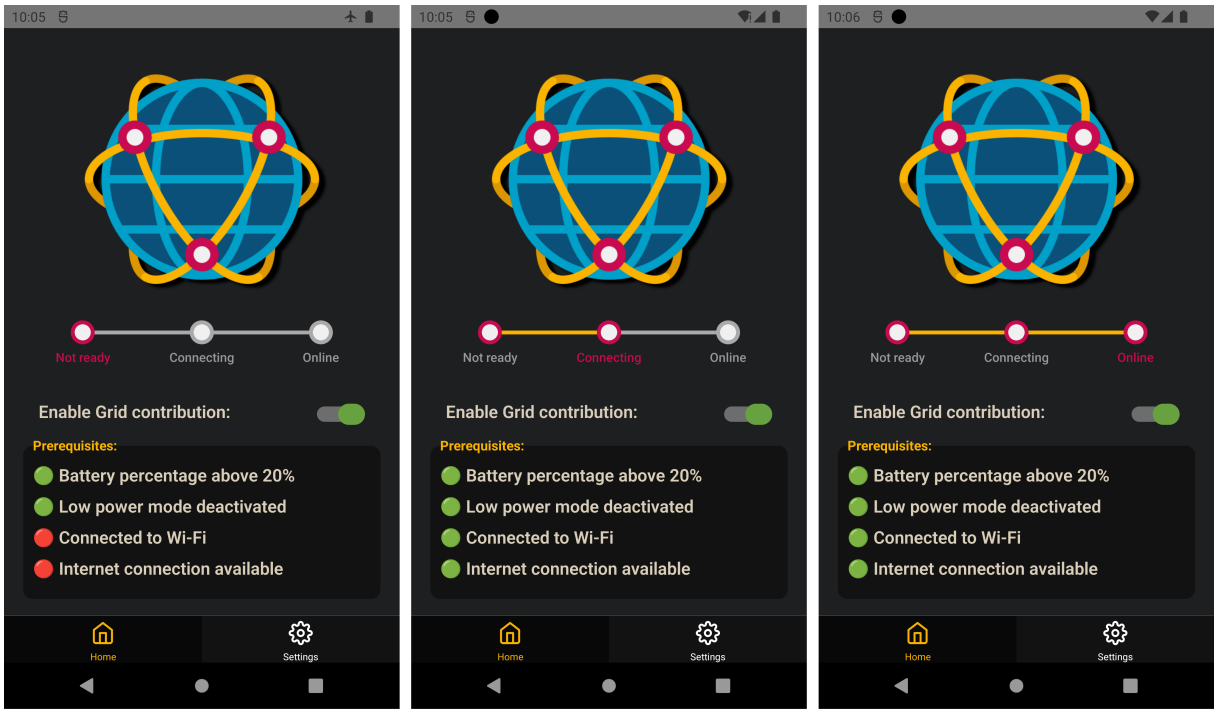


Figure 6.5: Interconnected Mobile Client

Figure 6.5 shows the **GUI** of the application: **by simply activating the "Enable Grid contribution" switch** (*figure 6.7(a)*), **the application starts a background process that continues to operate even after the application is closed**; Android forces the developer to notify the user of the presence of a **background activity** by spawning a **permanent notification** (which can be seen in *figure 6.6*) that will automatically be removed when the background activity is stopped by disabling the switch.

This background process is responsible for checking a series of conditions (that are listed in the "**Prerequisites**" section of the GUI) which are necessary for Grid Contribution; **when all the conditions are met, the Interconnected Node is started through the use of the previously mentioned Facade** exposed by the module. **In case even one of the prerequisites is not satisfied anymore, the Interconnected Node is stopped.**

Figure 6.6: Interconnected Mobile Client - Grid contribution enabled notification



(a) (b) (c)

Figure 6.7: Interconnected Mobile Client - Status

Despite being a background process, **the user can check the application's status through the GUI's indicators**: after the activation of the background process, the application remains in the "**Not ready**" state until all the Prerequisites are met (becoming green); after that, there is a transition to the "**Connecting**" state (*figure 6.7(b)*) and, once the Grid Connection is established, the "**Online**" state is reached (*figure 6.7(c)*).

When a contribution to a Grid Service is currently being performed, the user is notified by a notification that, once the contribution is over, is then updated showing that the process is complete (*figure 6.8*).

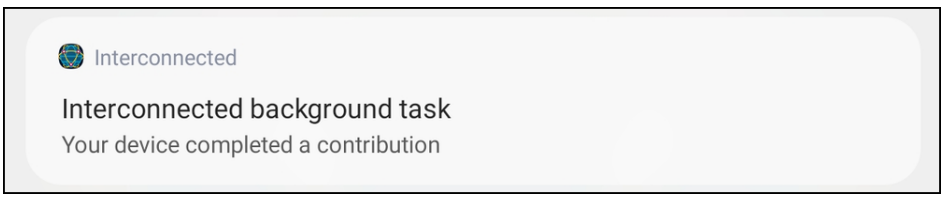


Figure 6.8: Interconnected Mobile Client - Grid contribution completed notification

6.2.4 Interconnected Desktop Client

The **Desktop incarnation of the Contributing Endpoint** takes the name of "***Interconnected Desktop Client***"; like the Mobile counterpart, it is developed using the **Typescript** language and by **relying on the Interconnected Node module dependency**. For this prototype version **no GUI was developed** and, then, a simple **Docker image** (which can run on any computer) is provided.

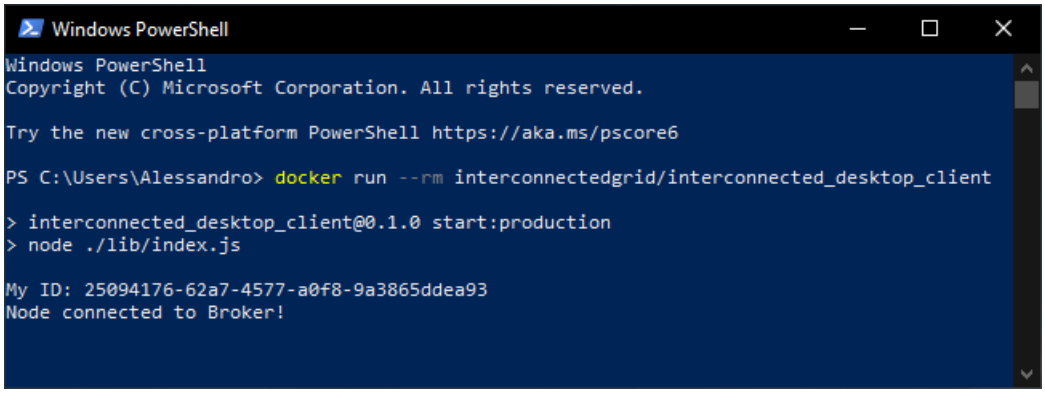


Figure 6.9: Interconnected Desktop Client - Docker image run on a container

In the Interconnected Mobile Client, this prototype's **Node ID** is generated by using a simple **UUID v4** (which can be seen in *figure 6.9*).

6.2.5 Invoking Endpoint Prototype

The last entity that composes this simplified architecture is the ***Invoking Endpoint Prototype***; unlike its counterpart in the complete architecture, which envisions it as a dependency (much like the Interconnected Node) that can be used by a Customer Custom application, this prototype version is a **standalone program that can interact with the Grid launching a MapReduce computation**; more details on the subject will be discussed in *section 6.6*.

This entity is developed, like the others, relaying on the **Node.js framework** but, instead, by using plain **JavaScript** and not the Typescript superset. In order to be able to communicate with the Broker Service and the Interconnected Nodes, it also utilizes the **Socket.io** module and the **WebRTC module**, respectively.

6.3 DevOps

The project utilizes a series of services to facilitate the production cycle, starting from version control combined with automation that simplifies the release process.

GitHub was chosen as the primary hub for the code base. In particular the "**GitHub organizations**" feature of such service allows creating organizations where people can cooperate and, most importantly, it groups together repositories belonging to that specific organization, separating them from personal accounts.

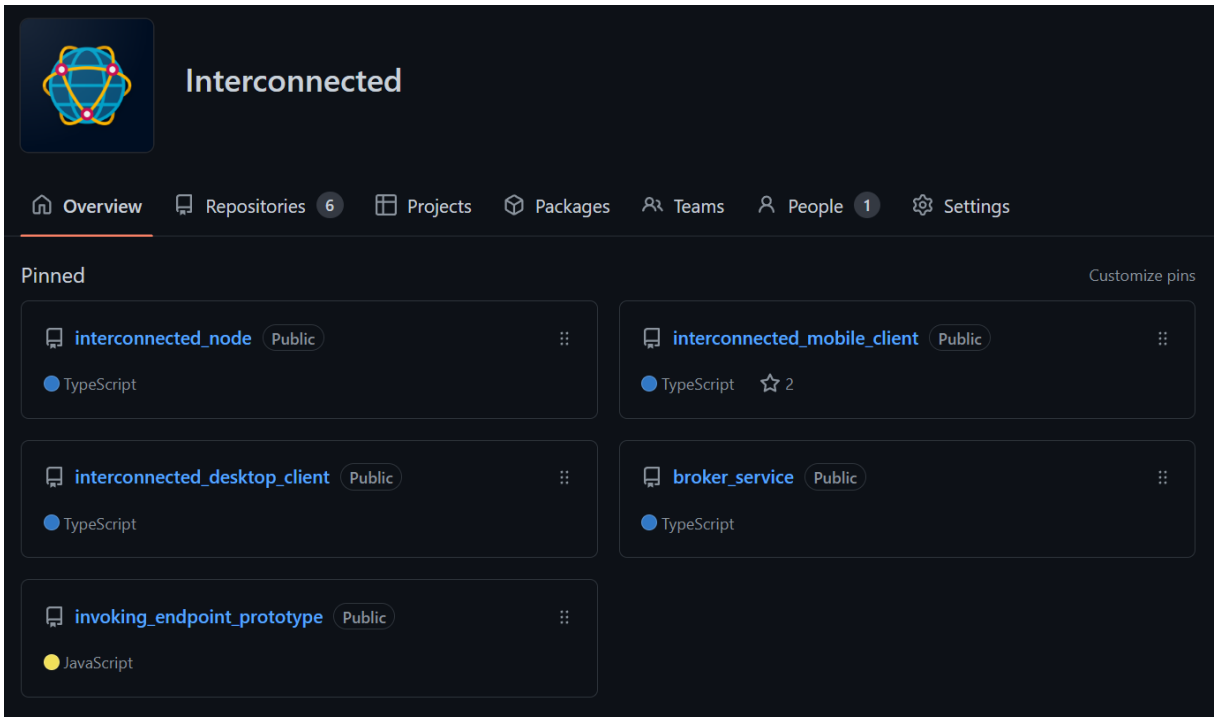


Figure 6.10: Interconnected - GitHub Organization

Furthermore, through the definition of specific **GitHub actions**, a lot of repetitive tasks can be automated; in this project case, actions collaborate with external services in order to obtain **CD** (Continuous Delivery) **automation**.

The first external service used is **NPM**; **when a new release of the Interconnected Node is created, a GitHub action uses the repository's code base in order to publish the module on NPM** (*figure 6.11*), making it available for the Mobile and Desktop Contributing Endpoints.

The Broker Service and the Interconnected Desktop Client both utilize Docker for the execution on target machines; such **Docker images are automatically created and published on Docker Hub whenever a new release is created in the respective**

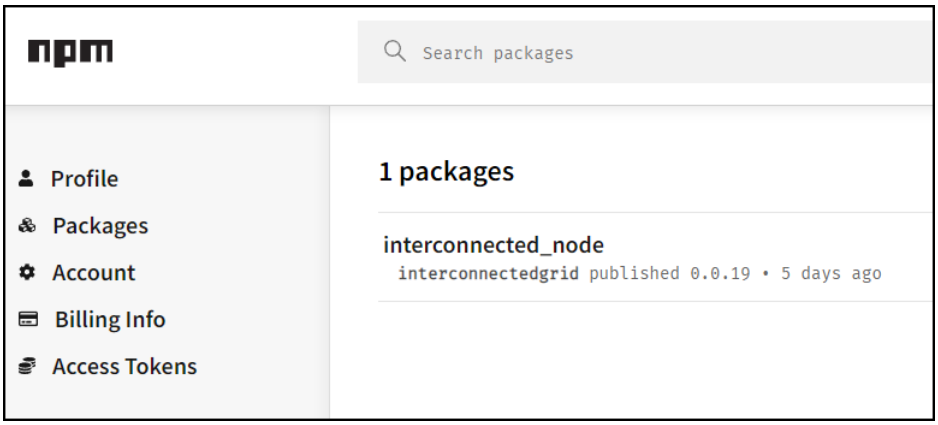


Figure 6.11: Interconnected - NPM

repositories. Thanks to this, a machine that wants to run a container with that specific image will not need to access the code base and run, compile and run the Node.js environment, but it will only need a working installation of the Docker client; knowing the image name, this will automatically be downloaded from Docker Hub to the target machine (*figure 6.9*).

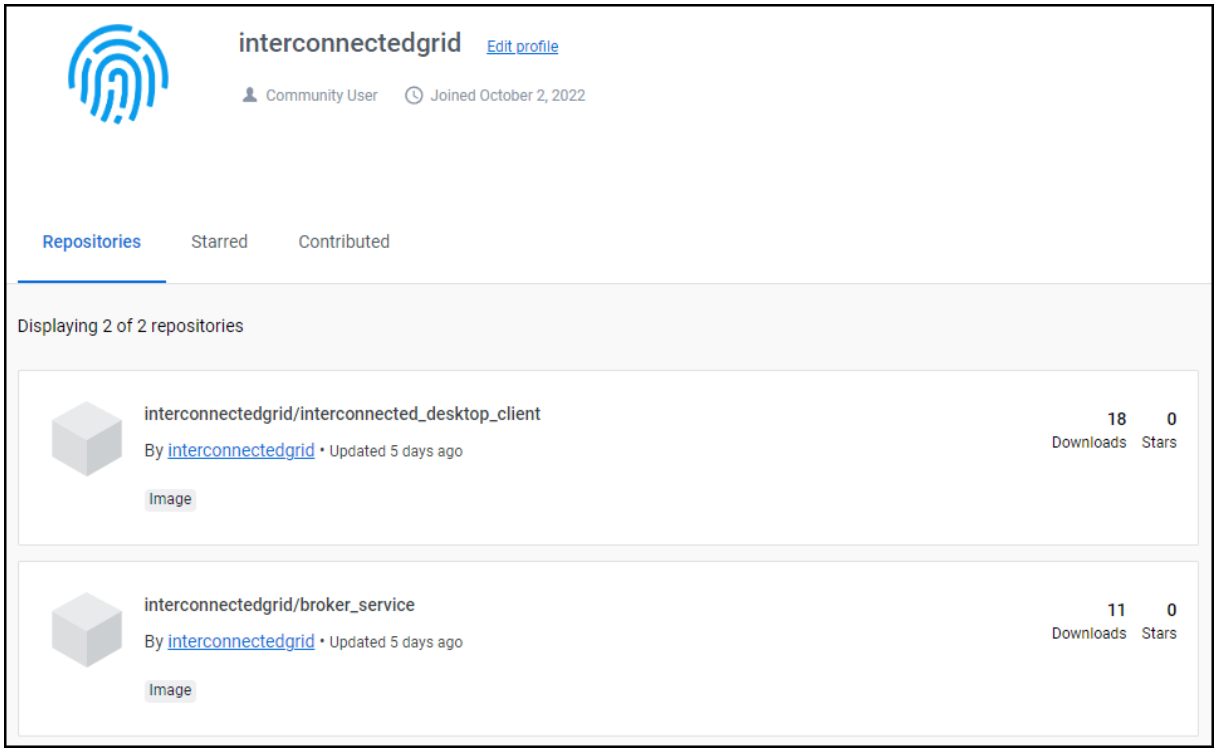


Figure 6.12: Interconnected - Docker Hub

In particular, when it comes to the execution of the Broker Service's image on a container, Amazon ECS offers a simple hosting platform that is able to pull images from Docker Hub by simply defining a task and running it. This way, **every time the Broker**

Service is started, Amazon ECS' defined task will retrieve the latest published image and will run it on a dedicated container (*figure 6.13*).

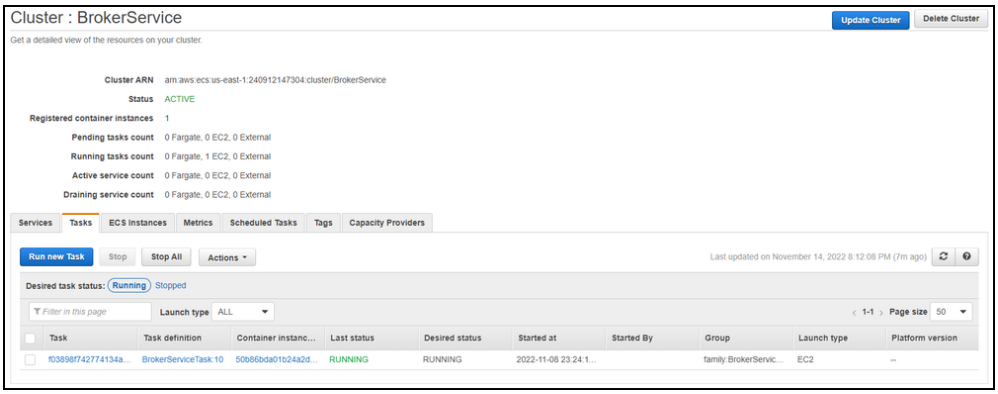


Figure 6.13: Interconnected - Amazon ECS

Finally, despite being run on an emulator during development, the **Interconnected Mobile Client** needs to be delivered in production; **each time a new release is created on its GitHub repository, an action will build an APK and publish it as an attached file to said release**. This way, a ready-to-use installer is automatically made available for installation on real Android devices.

Figure 6.14: Interconnected - Android APK automated release

6.4 Coordination

This section analyses the **messages exchanged among the entities composing the prototype architecture**. There are **three phases** in this coordination process:

1. **Grid connection**
2. **Recruitment**
3. **P2P messaging**

6.4.1 Grid connection

This phase **connects a Node** (whether used by the Mobile or the Desktop client) **or an Invoking Endpoint Prototype to the Broker Service using a Socket connection** through the Socket.io framework. **The Broker Service registers the connections and uses them in the Recruitment phase**.

6.4.2 Recruitment

The recruitment phase **connects a requestor device** (that becomes the Master in the P2Pconnection) **to an actual device that will perform a Contribution** (becoming the Slave). **While a Slave is necessarily a Node, a Master can either be an Invoking Endpoint or a Node**, allowing to obtain either an **Invoking Endpoint to Node connection** or a **Node to Node connection**.

This phase is **executed exchanging messages among the soon-to-be Peers using the previously established Socket connections, but ends with a direct P2P connection** between the Master and the Slave through the WebRTC framework.

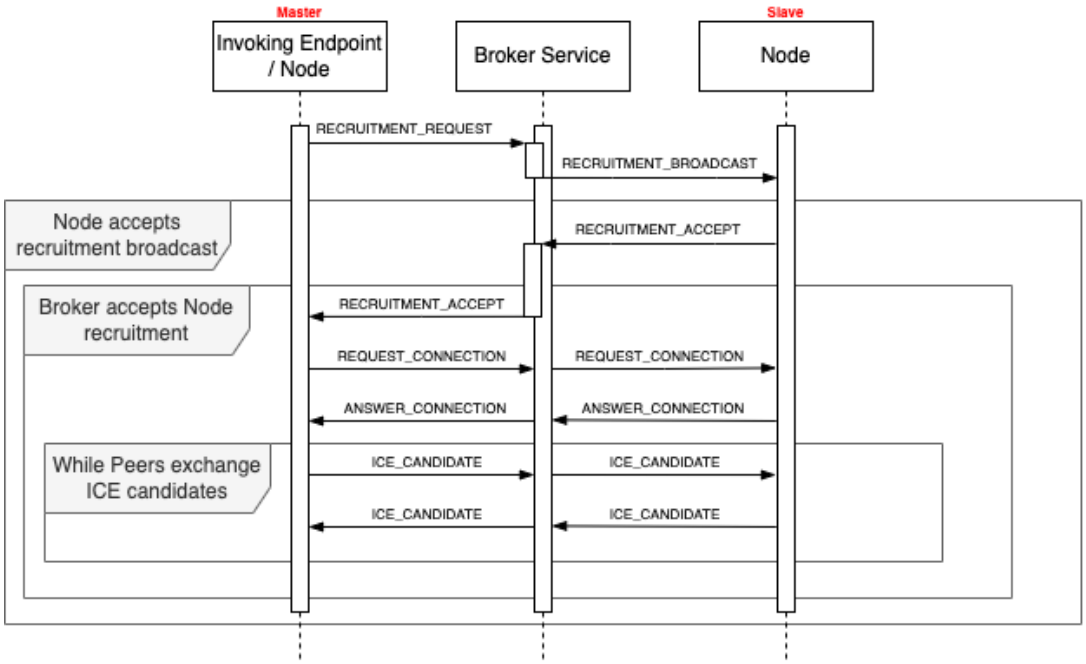


Figure 6.15: Recruitment - Sequence Diagram

Figure 6.15 shows the actual messages exchanged in this phase: the soon-to-be Master sends a ***RECRUITMENT_REQUEST*** to the Broker Service, **containing details regarding**

which kind of resources are needed; the Broker Service, upon receiving this request, creates a **pending Recruitment Request** and broadcasts a ***RECRUITMENT_BROADCAST*** message to all the connected Nodes.

The Node checks locally if it is compatible with the received request and, if all the conditions are true, it sends a ***RECRUITMENT_ACCEPT*** message **to the Broker Service** which, upon receiving it, checks if the request is still unsatisfied, **eventually forwarding the message to the entity that first emitted the request**.

From this point on, **the P2P connection is initialized exchanging the WebRTC's required information** (previously discussed in *section 6.2.2*): the **Master** sends a ***REQUEST_CONNECTION*** message, containing its **SPD offer** and, in response, the **Slave** sends its **SDP answer** contained in a ***ANSWER_CONNECTION*** message. Once the two Peers possess each other's SDP data structures, they **start exchanging ICE candidates until they reach an agreement, finally opening the P2P connection, completing the Recruitment**.

Although the broadcast is executed once, the messages contained in the "Broker accepts Node recruitment" scope need to be exchanged every time a new P2P connection is established among two Peers.

6.4.3 P2P messaging

Once the P2P connection is established, *figure 6.16* shows the **messages that the Master and the Slave exchange in their interaction**.

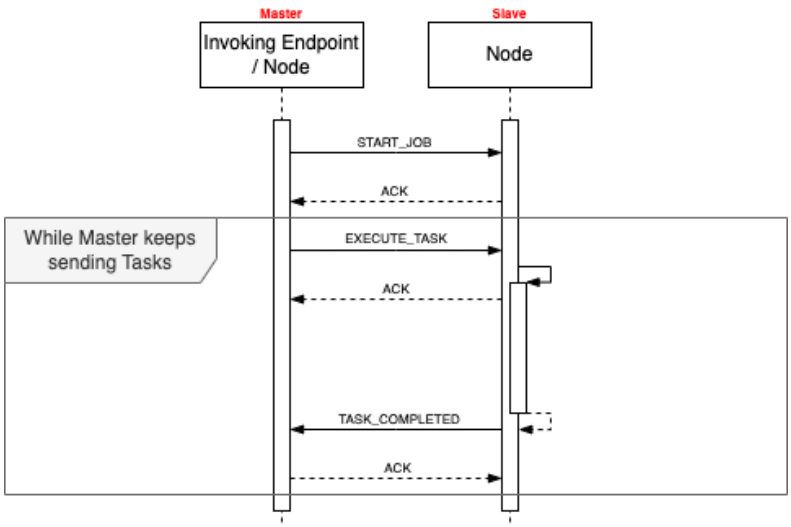


Figure 6.16: P2P messaging - Sequence Diagram

The Master sends a *START_JOB* message to the Slave, possibly including data (which need to be sent only once) that will later be useful when executing the specific Tasks for that Job. Once the Slave has started said Job, t**he Master sends a EXECUTE_TASK message**, **containing further data that**, combined with the Job's payload, **is used to perform a computation**. Upon receiving the EXECUTE_TASK message the **Slave** sends an ACK to the Master and **enqueues the Task**, sending a **TASK_COMPLETED** message (containing the result) to the Master **only when the task is actually completed**.

This asynchronous organization allows the Master to send multiple Tasks to enqueue without waiting that the Slave has already completed its Tasks, speeding up the process. The Tasks submission and collection of results continues until all the Master's Tasks are completed and the P2P connection is terminated, removing the Job from the Slave.

Using this Job/Task generalization, it is possible to construct various kinds of Grid Services, starting from the MapReduce one.

6.5 Proto-MapReduce

The MapReduce implementation used in this prototype is a simplified version of what discussed in *appendix A*; said simplifications, while making it **easier to implement**, have the **side effect of negatively influencing performances but**, given the available time limitations, they are **still acceptable in a prototypical context where the focus is to demonstrate the feasibility of mobile devices Contribution**.

The first difference comes from how the data are handled. The MapReduce paradigms handles a number of splits M and applies progressively the Map function to said splits; the intermediate results are grouped by key (using a partitioning function) in order to obtain R (with R<M, typically) partitions which, upon allying the Reduce function, will produce R final results. This **simplified version**, on the contrary, **takes the M splits** and, after applying the Map function, **maintains the original grouping, producing M intermediate results**; said results are then **computed applying the Reduce function** to each of them, **producing the M final results**. As a consequence, **the number of input data regions is equal to the regions in the final results**.

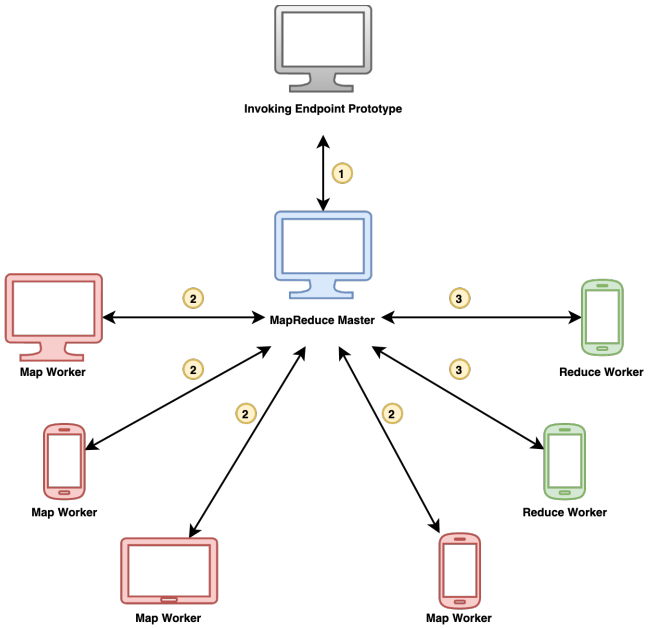


Figure 6.17: Proto-MapReduce Topology

The second important distinction comes from the topology of the connections among the entities involved in the MapReduce operation. Normally, a Map Worker that has completed a run of the Map function would send, under the coordination of the MapReduce Master, its intermediate results directly to the right Reduce worker; **in this simplified version a Map Worker that has completed a Map operation sends its results to the MapReduce Master which, after gathering all the Map results for that particular region, will forward the data to an assigned Reduce Worker**. The final topology for the MapReduce Service performed in this prototype is shown in *figure 6.17* and is **obtained performing these three steps**:

1. **MapReduce Master recruitment**

   The Invoking Endpoint Prototype performs a recruitment (seen in *figure 6.15*), connecting to a Node which receives a MapReduce Master Job (containing all info about resources requested with the Map and Reduce functions definitions as well). This marks the beginning of the next phase but, in the meantime, the Invoking Endpoint starts sending Tasks to the MapReduce Master (*figure 6.16*), each containing a data region to compute.

2. **Map Workers recruitment**

   The MapReduce Master performs a recruitment, requesting the Map Workers speci-

fied in its MapReduce Master Job, and it sends a Map Worker Job (containing the Map function) upon every new connection; once all the requested Map Workers are recruited, the next phase can begin, while, at the same time, also starting to send Tasks containing the data to which apply the Map function.

3. **Reduce Workers recruitment**
   Here the MapReduce Master performs the final recruitment, requesting the Reduce Workers specified in its MapReduce Master Job, sending a Reduce Worker Job (containing the Reduce function) to every new Node recruited. After this recruitment is completed, the MapReduce Master is allowed to send the progressively collected intermediate results to the Reduce Workers.

Once all the data region are computed and the results are collected by the Invoking Endpoint, the P2P connections are closed, completing the execution.

As can be easily deduced, the Invoking Endpoint Prototype acts as a Master in its P2P connection to the MapReduce Master, while Map and Reduce Workers only act as Slaves in their connection to the same entity, **making a Node** (the MapReduce Master in this case) **able to perform both the Master and the Slave roles at the same time**, with the consequence of showing **that the Map Worker to Reduce Worker connection is perfectly feasible in a future implementation** of these Jobs.

Thanks to this topology, every computationally heavy operation is delegated to the Grid's Nodes and, thus, the Invocation can be performed even from a low-spec device, increasing the versatility of the Grid.

On a final note, while performing a recruitment, various parameters can be specified about the resources that a device needs to possess; while the Map or Reduce Worker role can be taken by any device, **the MapReduce Master role is reserved to Desktop devices**. This choice is made **trying to obtain more stability** for the MapReduce process.

6.6 Real-world experiments

This section describes the **experiments performed using the prototype in a real-world scenario** performing a distributed MapReduce computation on distributed heterogeneous devices.

6.6.1 Computation

The operation chosen for the experiment is a **simple classification based on the distance from centroids, each representing its corresponding class**:

- **Red centroid**: (200,900)
- **Green centroid** (700,100)
- **Blue centroid** (1300,700)

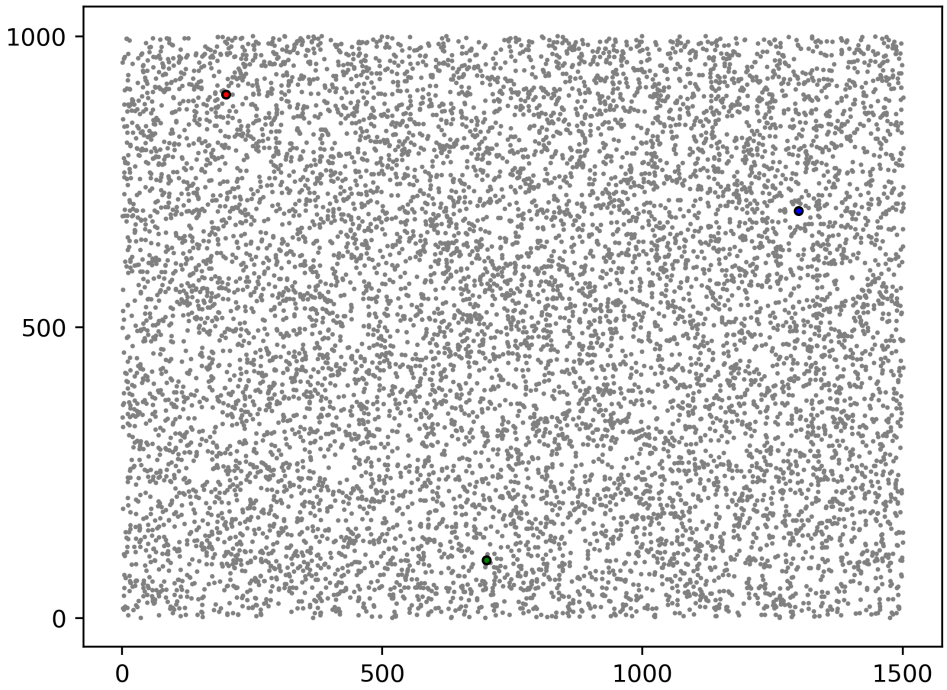


Figure 6.18: Computation - Starting point

Given a Cartesian plane (width: [0,1500], height: [0, 1000]) **and a set of 2D points** contained in it, **the map function takes as input one of said points and calculates the euclidean distance from each one of the centroids**; the centroid with **minimum distance among the three is then chosen, obtaining a key-value output** composed by the chosen class as the key and an array containing the computed point as the value (the array becomes relevant in the reduce function). It is important to note that, in comparing the distance among two points, the square root characterizing the euclidean distance is not needed and, therefore, it is not calculated in the map function.

```
const mapFunction = (p) => {
    const x = p[0];
    const y = p[1];
    const red = Math.pow(x - 200, 2) + Math.pow(y - 900, 2);
    const green = Math.pow(x - 700, 2) + Math.pow(y - 100, 2);
    const blue = Math.pow(x - 1300, 2) + Math.pow(y - 700, 2);
    switch(Math.min(red, green, blue)){
        case red: return ["red", [p]];
        case green: return ["green", [p]];
        case blue: return ["blue", [p]];
    }
}
```

Listing 6.1: Map function

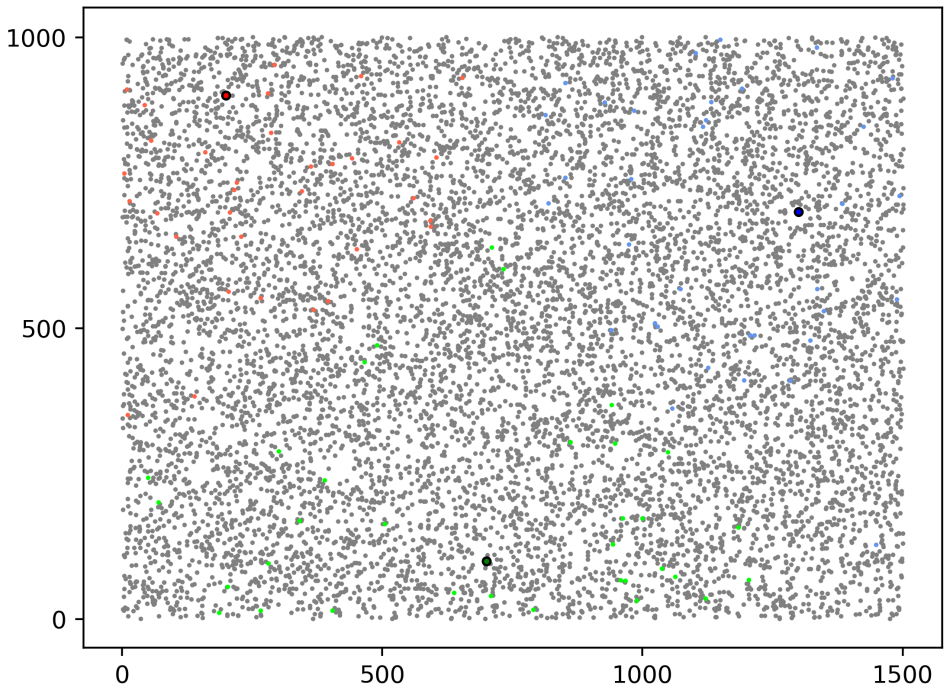


Figure 6.19: Computation - Region computation

Every data region is computed by the Map Workers and, **after every point in a particular region is classified, the output** (visualized in *figure 6.19*) **can be computed in the reduce function which simply reunites the intermediate results with the same key** (hence belonging to the same class) in a single array.

```
const reduceFunction = (p1, p2) => {
    p1.push(p2[0]);
    return p1;
}
```

Listing 6.2: Reduce function

As can be seen in *figure 6.20*, after every region is mapped and then reduced, **each point is assigned to one of the three classes**.

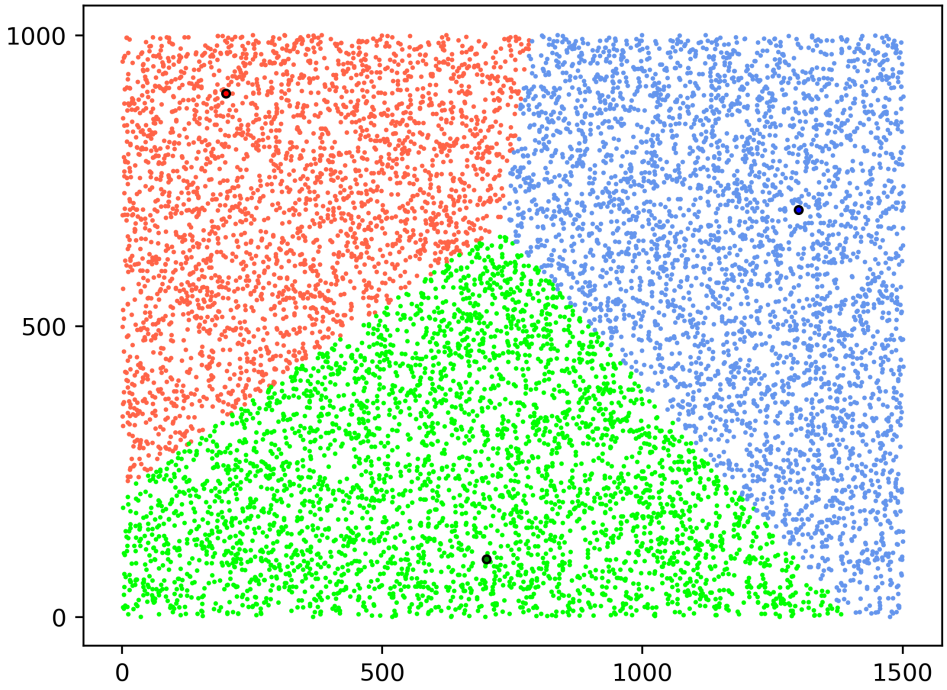


Figure 6.20: Computation - Final result

Five experiments were performed:

- **1000 values** (10 regions, 100 points for each region)
- **10000 values** (100 regions, 100 points for each region)(shown in *figure 6.20*)
- **100000 values** (100 regions, 1000 points for each region)
- **1000000 values** (1000 regions, 1000 points for each region)
- **5000000 values** (2000 regions, 2500 points for reach region)

6.6.2 Setup

The experiment was performed in a controlled environment, meaning that only these devices were connected to the Grid (*figure 6.21*):

- **A**: 1 smartphone
- **B**: 1 smartphone
- **C**: 1 computer
- **D**: 2 smartphones
- **E**: 1 tablet and 1 computer

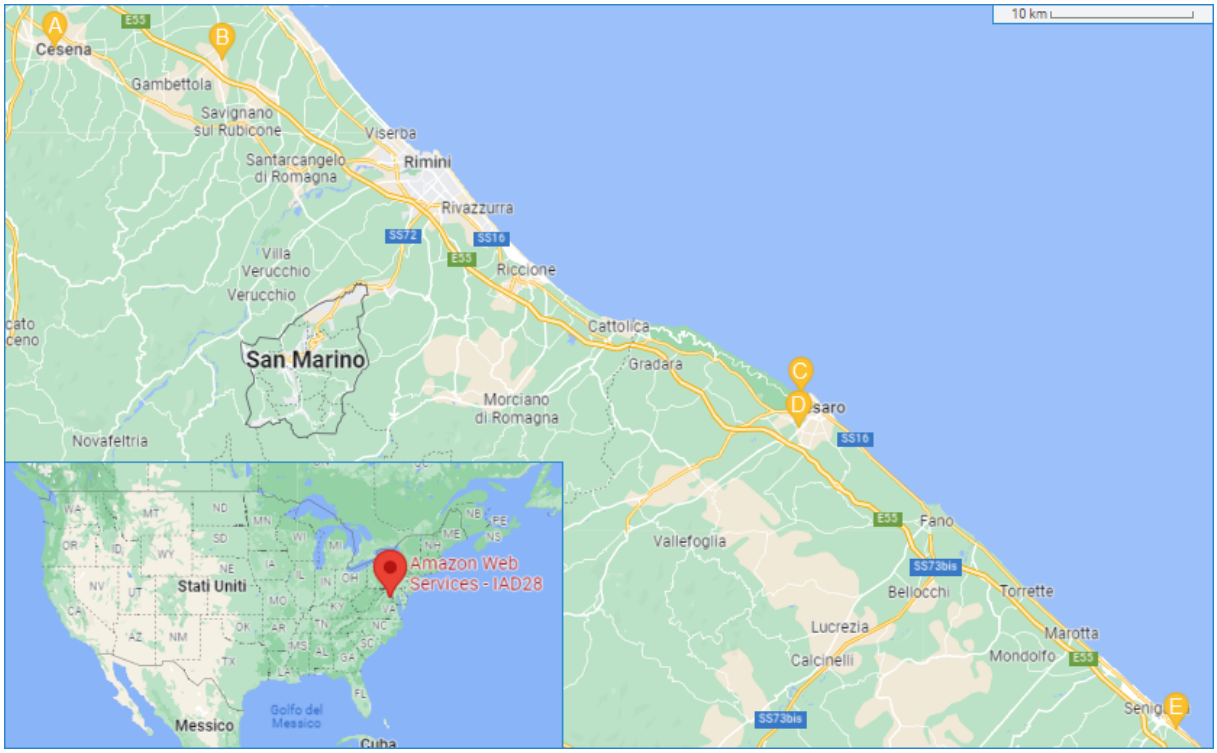


Figure 6.21: Devices setup

The setup of Contributing Endpoints was thus composed by **2 Interconnected Desktop Clients** and **5 Interconnected Mobile Clients**, placed in a **100 km range** in central Italy, which were forcibly chosen isolating them in a dedicated American server in order to perform multiple experiments with the same setup.

The Invoking Endpoint Prototype instance was placed in the E location (although it was not executed in the same computer which ran the Interconnected Desktop Client);

said Invoking Endpoint requested the following resources for executing the MapReduce computation:

- **4 Map Workers**
- **2 Reduce Workers**

Including the implicit MapReduce Master (which will be taken by one of the two computers), **this adds up to the 7 devices specified earlier**, which were used in each of the five experiments.

6.6.3 Results

Figure 6.22 shows the **results for the five experiments performed**, focusing on the **total time** and the **average time taken by each value** (both **measured in milliseconds**). Once again, these results were obtained using a very small pool of devices and the simplified nature of the MapReduce algorithm in this prototype significantly slows down the whole process (primarily because the intermediate results are first sent back to the MapReduce Master that then forwards them to the Reduce Worker, instead using a direct connection among Map Worker and Reduce Worker).

Values	Time (ms)	Avg time (ms) for each value
1000	5693	5,693
10000	8494	0,8494
100000	19613	0,19613
1000000	129419	0,129419
5000000	495169	0,0990338

Figure 6.22: Experiment results

The **first significant observation** can be made l**ooking at the first two experiments**: the **average time for each value drastically drops** (~6.7 times faster); this can be **explained by considering that the total time also includes the recruitment phase where no computation is executed**. In other terms, **the number of values used in the first experiment is so small that their computation time becomes irrelevant**, meaning that the recruitment phase is basically the only factor that influences the average time. **The more values are computed (assuming the same number of devices are used), the less impactful the recruitment time becomes**.

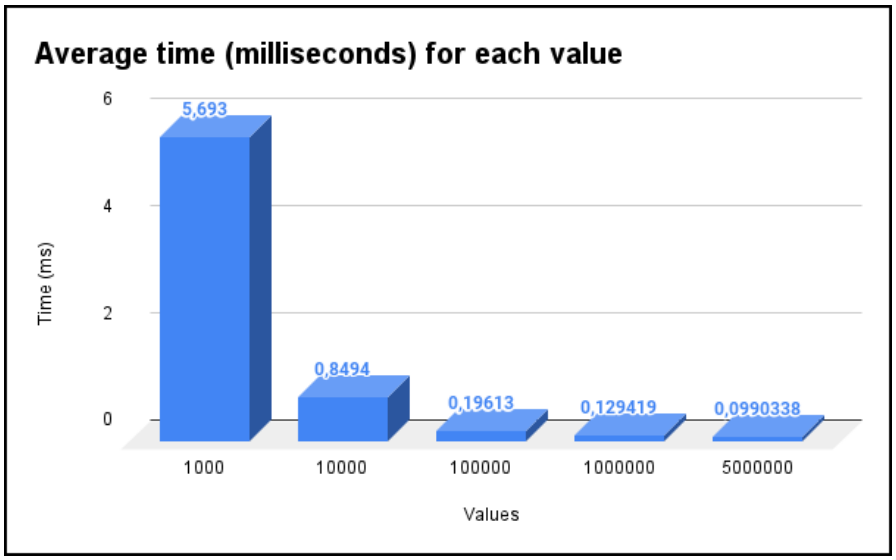


Figure 6.23: Average time (milliseconds) for each value

Finally, *figure 6.23* focuses on **comparing the average time for each value**; it becomes apparent that, despite the not optimized algorithms used in this prototype, **the more values are computed, the greater the advantage becomes, showing that a distributed computation participated also by mobile devices is not only feasible, but it can also provide value to the Customer**.

Chapter 7

Future development and ideas

This final brief chapter will touch on the future development and ideas for the project, first addressing how to expand the prototype and, then, proposing some ideas for future Grid Services other than MapReduce.

7.1 Expanding the prototype

The prototype seen in the previous chapter is just a fraction of the architecture theorized in this thesis work; this means that, with an **incremental approach**, further development for the actual software entities has to be made, starting from:

- **iOS tested compatibility**
- **An easy-to-use GUI for the Interconnected Desktop Client, coupled with a guided installation process and automated start**
- **MapReduce's algorithm implementation completion**
- **Making the Invoking Endpoint an actual dependency usable by other projects**

Once these steps are completed, reaching a stability and maturity in the project development, **the cloud side must be expanded** with the final goal of obtaining the engineered cloud structure, **starting from the creation of the dynamic system centered around the Grid Master Service**.

After reaching a full working architecture, the next step in the evolution of the project comes from the implementation of new Grid Services.

7.2 Implementing new Grid Services

Given the high focus on expandability that has driven this project, **the possibilities for expanding the Grid's capabilities are endless**. This section lists **some ideas** for future Grid Services.

7.2.1 Simpler Services: Computation delegation

The MapReduce Service is a fantastic tool, but it is not ideal in every use case; **MapReduce's limitations are well known in circumstances where the data pool to analyze is very limited in size** and, thus, a simple computation delegation service is more indicated in those situations. By removing the recruitment process and the coordination among multiple entities, **an Invoking Point can delegate a computation to a more powerful Node that will remotely execute the requested computation**.

7.2.2 More than computations: Storage sharing

Another wasted resource in users' Devices is storage space, frequently remaining unused; thus, another possibility for a Grid Service is to **allow Contributors to share said storage space**, implementing a **distributed file retrieval system correlated with replication and fault tolerance mechanisms**.

7.2.3 Reaching the physical world: Collective computing

Finally, **one particular trait of mobile devices is the presence of multiple sensors**. Given the participation of said devices to the Grid, **some services can offer the possibility to access said sensors in order to obtain physical world data** coming from the mobile devices of Contributors that allow such data retrieval.

Appendix A

The MapReduce paradigm

As previously discussed, Grid computing is a multipurpose tool aimed to offer devices' resources to perform various types of tasks. This project, despite being multipurpose in nature itself (meaning that organizes its structure keeping in mind the possibility to support additional services in the future), aims to specifically offer to a user the execution of a MapReduce service over the Grid.

Hence, this short appendix discusses the MapReduce paradigm, concluding the overview of the problem. Firstly, a definition and the history of such paradigm will be provided. Then, the programming model will be discussed, explaining the basic concepts needed to utilize MapReduce; following that, an overview of the master-slave architecture is provided. Continuing on, in an attempt to clarify as much as possible how the actual execution of a program that employs MapReduce works, an easy example of a MapReduce computation is displayed. To conclude, an analysis of the execution flow is presented.

Definition and history

MapReduce is a **popular programming model** designed to **easily process and generate large datasets on clusters of commodity machines**. Through this paradigm, a computation can be expressed in terms of a **map and reduce functions** while **the underlying system deals with communication, parallelization and error handling**, making it easy to use even for programmers who have no experience with distributed systems.

Google created this paradigm in **2003** in order to reduce development time and cost on their projects; after an analysis of the problem, Google's engineers noticed how **the majority of the computations in their products could be expressed through the map and reduce abstractions** that are typically present in **functional languages**. The MapReduce paradigm has been **used in a variety of Google's project**, including the indexing system used by the Google search engine [16].

Apache Hadoop, inspired by Google's work, integrates the MapReduce paradigm in its free-licensed framework, making it **one of the most used options** when it comes to applying distributed computations following this paradigm.

The programming model

In order to execute a MapReduce computation, it is required, as **input**, **a set of key/value pairs**. Said values are **modified through the Map and Reduce functions** and, ultimately, they produce as **output another set of key/value pairs**.

The Map and Reduce functions are **written by the user** but in the background, through the framework, they behave in the following way:

- **Map**: $(k1, v1) \Longrightarrow list(k2, v2)$

  The map function takes a single pair as input and produces a set of intermediate key/value pairs. The framework automatically merges the intermediate sets, grouping them using the keys. Said values are then passed as input to the Reduce function.

- **Reduce**: $(k2, list(v2)) \Longrightarrow list(v2)$

  The Reduce function uses the input provided by the automatic merge process performed by the framework. Every Reduce execution takes a pair composed of the intermediate key and a collection of values associated to that key; said pairs are provided using an iterator in order to work with collections that are too large to fit in memory. The values associated to the key are merged to form a possibly smaller set of values, typically resulting in one or zero output values produced as result (even though the function produces a list of values).

A programmer who implements a computation following this paradigm does not need to provide anything else but, **behind the scenes**, the framework performs additional

operations such as **Splitting** (that divides the input into smaller parts to be executed on the multiple workers), **Shuffling** (that merges the output of the individual Map functions) and **Collect** (that reunites results produced by the various workers).

Architecture

As anticipated in *section A*, MapReduce operations are performed over a cluster of commodity machines; such machines operate under a master-slave architecture, meaning that there is one node, the Master, that coordinates the execution of the algorithm across the worker nodes (or Slaves). *Figure A.1* shows how such architecture is handled inside Hadoop's MapReduce.

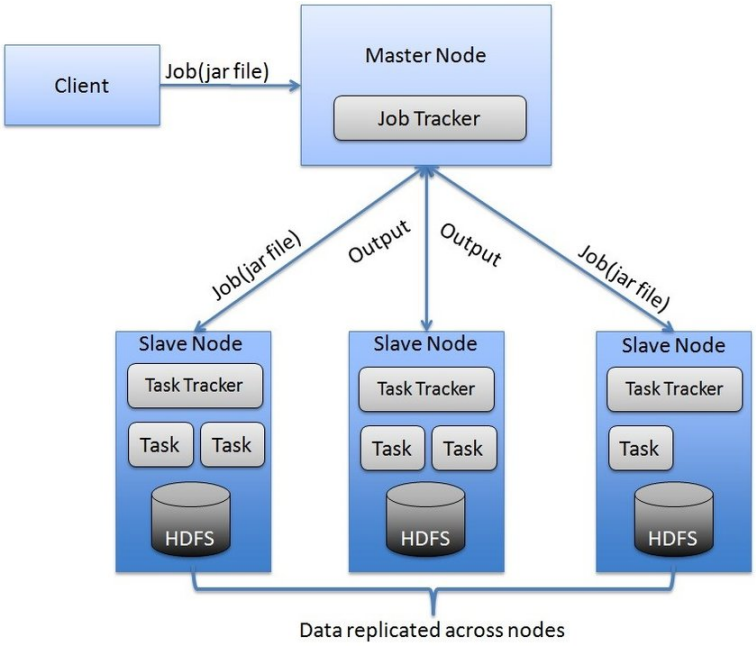


Figure A.1: Hadoop's MapReduce master-slave architecture[17]

A Client machine resides Outside the cluster and, when such machine needs to execute a MapReduce operation, firstly it needs to provide the Map and Reduce functions (written by the user who requests the task) to the Master (in Hadoop's case the code to execute is contained in a Jar file) which will propagate them to the Slaves. The Client also necessarily provides the data to process, whether sending them directly to the Master or specifying the source where to get them.

The Master machine has a Job Tracker through which it will coordinate the Slave machines; the worker machines, on the other hand, have a Task Tracker, through which

information about the ongoing task can be accessed by the Master. Each Slave Node will have a queue of pending Tasks (sent by the Master) that will be executed, and it has access to the underlying Hadoop Distributed File System (HDFS) for data replication.

Example

In this example (*figure A.2*), a simple word count will be performed: given a source of text, the algorithm will count the occurrences of every word appearing in such text.

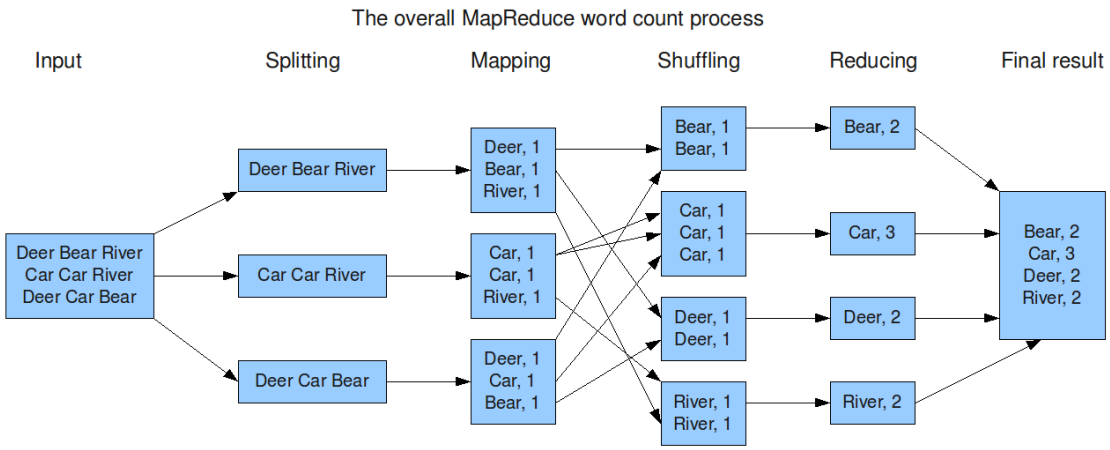


Figure A.2: Example of word count expressed through the MapReduce paradigm [18]

The process begins with splitting the input into smaller portions, in this case the text is split by row. On the resulting portions, the Map function is performed; in this example, every word (that is used as a key) is mapped to the value "1" (since it is one appearance of said word). The results are then grouped through the Shuffling operation, using the key as the grouping criteria. Finally, the Reduce function is executed, in different machines, on every group (the example sums the values in order to get the final number of occurrences for every word); the results of the Reduce operations is then collected, obtaining the final collection of key/value pairs.

Even though this example is displayed with a limited quantity of input and output data, the same mechanism can be expanded working with Big Data applications. The key here is the performance augmentation given by parallelizing the execution of the various computational steps across the cluster's machines participating in the MapReduce operation.

Execution flow

When it comes to understanding the actual flow of execution in a MapReduce cluster, two parameters need to be discussed:

- *The number of splits **M** (Map input)*
  This value determinates how many portions the input data will be divided into; this value should be set accordingly to have each portion between 16 and 64 Mb in size.

- *The number of intermediate key partitions **R** (Reduce input)*
  The intermediate key space is partitioned into R pieces using a partitioning function (e.g. *hash(key) mod R* [16]), distributing the Reduce invocations. R is usually a small multiple of the number of cluster machines that will be used.

The partitioning function, M and R can be set by the user. Typically, MapReduce operations performed at Google have M=200000 and R=5000 with 2000 machines [16].

The Master, acting as coordinator and intermediary among the Worker nodes, mainly keeps two data structures in memory:

- **Tasks' state**
  For every Map task and every Reduce task (for a total number of M + R), the Master keeps track of the current state, that can be either idle, in progress or completed.

- **Intermediate results location**
  The master stores the locations and sizes of the R intermediate file regions produced by the map task in order to communicate said data to the Reduce Workers that are in the "in progress" state.

Figure A.3 shows an example of execution flow (the enumeration in the following list corresponds to the numeric labels displayed in the figure) with M=5, R=2 and 6 machines (the R parameter has not an optimal value, but this is done only for explanatory reasons):

1. The MapReduce library on the User program side splits the input accordingly to the M parameter and starts many copies of the program on the cluster's machines. One of the machines acts as the Master and the others as Workers (*section A*). The Map

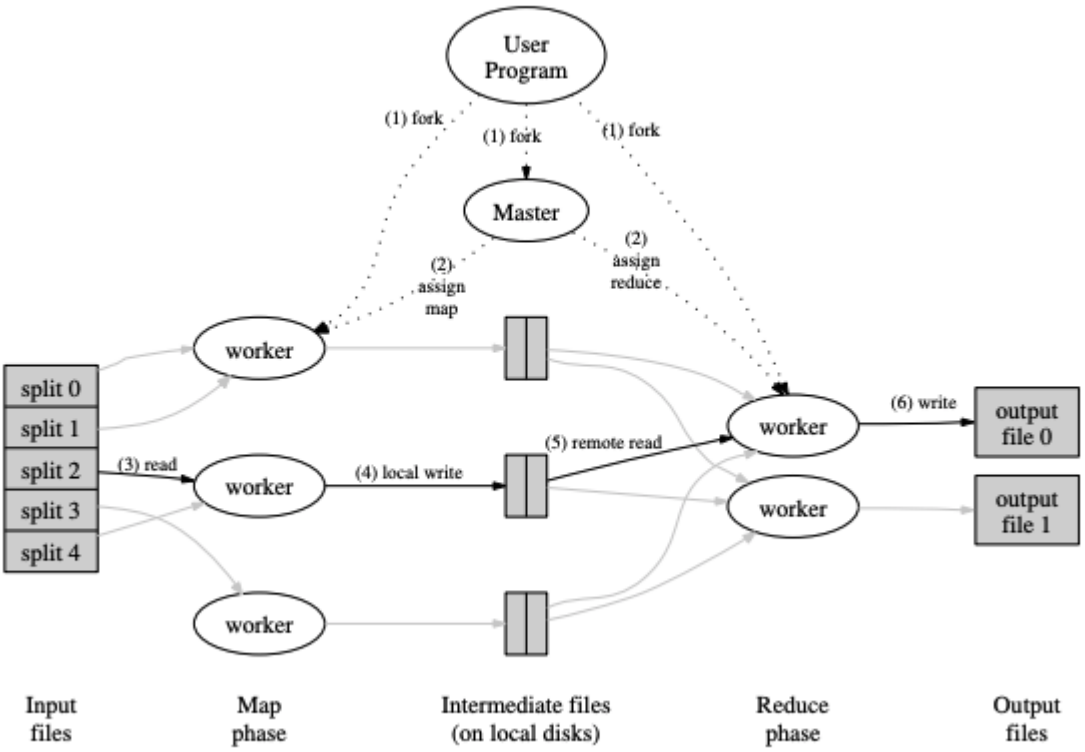


Figure A.3: MapReduce execution flow [16]

and Reduce functions' code is propagated to the machines and the User Program waits to be notified by the Master after the completion of the MapReduce tasks.

2. The Master assigns the total M + R tasks to the Workers, be it a Map task or a Reduce task.

3. A Worker that executes a Map task reads the corresponding input piece among the M parts. The key/value pairs are processed through the User-defined Map function and the intermediate results are buffered in memory.

4. Periodically, the buffered intermediate results are written on the local disk. Said results are partitioned locally in R regions following the partitioning function. After that, the Worker communicates the location of such results to the Master which, in turn, forwards this information to the Reduce Workers.

5. When a Worker is notified about the availability of new results, it reads them (using the location previously provided) from the correct Workers and, once it has all the

data for its region, it sorts such data using the intermediate key, grouping together data that belongs to the same key. If the intermediate data is too large to fit in memory, the sorting is done using the disk.

6. The Worker iterates over the data and, for each key, performs the Reduce function provided by the User. This produces a file containing all the Reduce results for the region handled by the Worker.

7. After all the Map and Reduce tasks are completed, the User program is notified, and it can resume its execution.

The final output of the R files can then be accessed as it is, combined into a single source, or used as input for a new MapReduce execution.

Bibliography

[1] "Introduction to Grid Computing". In: University of Chicago and University of Southern California, 2015, p. 19. URL: https://slideplayer.com/slide/4818943/.

[2] Raja Malleswara Rao Pattamsetti. *Distributed Computing in Java 9*. O'Reilly. URL: https://www.oreilly.com/library/view/distributed-computing-in/9781787126992/7478b64c-8de4-4db3-b473-66e1d1fcba77.xhtml.

[3] Foster Ian, Kesselman Carl, and Steven Tuecke. "The anatomy of the Grid: enabling scalable virtual organizations". In: *International Journal of High Performance Computing Applications* 15.3 (2001), pp. 200–222.

[4] Joannes Mongardini and Aneta Radzikowski. "Global Smartphones Sales May Have Peaked". In: *IMF Working Papers* 20 (May 2020).

[5] Thomas Phan, Lloyd Huang, and Chris Dulan. "Challenge: Integrating Mobile Wireless Devices into the Computational Grid". In: *Proceedings of the 8th Annual International Conference on Mobile Computing and Networking*. MobiCom '02. Atlanta, Georgia, USA: Association for Computing Machinery, 2002, pp. 271–278.

[6] Mustafa Sanver, Sathya Durairaju, and Ajay Gupta. "Should one incorporate Mobile-ware in Parallel and Distributed Computation?" In: Dec. 2003.

[7] Jason Wise. *Mobile and Desktop operating system market share stats for 2022*. https://earthweb.com/operating-system-market-share/. July 2022.

[8] Arvind Narayanan et al. "A First Look at Commercial 5G Performance on Smartphones". In: *Proceedings of The Web Conference 2020*. WWW '20. Taipei, Taiwan: Association for Computing Machinery, 2020, pp. 894–905. ISBN: 9781450370233. DOI: 10.1145/3366423.3380169. URL: https://doi.org/10.1145/3366423.3380169.

[9] Pijush Kanti Dutta Pramanik et al. “Power Consumption Analysis, Measurement, Management, and Issues: A State-of-the-Art Review of Smartphone Battery and Energy Usage”. In: *IEEE Access* 7 (2019), pp. 182113–182172. DOI: 10.1109/ACCESS.2019.2958684.

[10] Shivi Garg and Niyati Baliyan. “Comparative analysis of Android and iOS from security viewpoint”. In: *Computer Science Review* 40 (2021), p. 100372. ISSN: 1574-0137. DOI: https://doi.org/10.1016/j.cosrev.2021.100372. URL: https://www.sciencedirect.com/science/article/pii/S1574013721000125.

[11] Li Li et al. “CiD: Automating the Detection of API-Related Compatibility Issues in Android Apps”. In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2018. Amsterdam, Netherlands: Association for Computing Machinery, 2018, pp. 153–163. ISBN: 9781450356992. DOI: 10.1145/3213846.3213857. URL: https://doi.org/10.1145/3213846.3213857.

[12] Umar Kalim et al. “Mobile-to-Grid Middleware: An Approach for Breaching the Divide Between Mobile and Grid Environments”. In: *Networking - ICN 2005*. Ed. by Pascal Lorenz and Petre Dini. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 1–8. ISBN: 978-3-540-31956-6.

[13] F. Navarro et al. “Towards a Middleware for Mobile Grids”. In: May 2006, pp. 1–4. DOI: 10.1109/NOMS.2006.1687644.

[14] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, 2004.

[15] *What is a STUN/TURN Server?* https://blog.ivrpowers.com/post/technologies/what-is-stun-turn-server/. 2018.

[16] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters”. In: *Commun. ACM* 51.1 (Jan. 2008), pp. 107–113. ISSN: 0001-0782. DOI: 10.1145/1327452.1327492. URL: https://doi.org/10.1145/1327452.1327492.

[17] Rida Qayyum. “A Roadmap Towards Big Data Opportunities, Emerging Issues and Hadoop as a Solution”. In: *International Journal of Education and Management Engineering* 10 (Aug. 2020), pp. 8–17. DOI: 10.5815/ijeme.2020.04.02.

[18] *MapReduce*. https://whatsbigdata.be/mapreduce/. Accessed: 2022-08-30.