The University of Sheffield International faculty, CITY College

FINAL YEAR PROJECT

A Framework for Simulating Artificial Life on Distributed Systems

This report is submitted in partial fulfillment of the requirement for the degree of Bachelor of Science with Honours in Computer Science by

Filippidis Odysseas

June, 2019

Approved

Dr. Konstantinos Dimopoulos

A Framework for Simulating Artificial Life on Distributed Systems

by

Filippidis Odysseas

Supervisor

Dr. Konstantinos Dimopoulos

ABSTRACT

This paper describes a software project, which is a distributed framework for running Artificial Life simulations. Operating this type of simulations can be sometime very expensive in terms of computational power. This undermines the operation of the simulation and thus poses a problem to the user or developer of the simulation software. As such, a promising solution for this problem is to run this simulation on a distributed system, which will handle more efficiently the large computational load that usually comes with it. However, this is not a simple task and requires a lot of system configurations and changes in the software of the simulation. Therefore, this paper proposes an efficient and state of the art tool to solve this problem in a modern fashion. More specifically, this tool is a microservice-based framework which is able to run on any distributed environment.

Declaration

All sentences or passages quoted in this dissertation from other people's work have been specifically acknowledgement by clear cross-referencing to author, work and pages(s). I understand that failure to do this, amounts to plagiarism and will be considered grounds for failure in this dissertation and the degree examination as a whole.

It is also my understanding that intellectual rights of this work are shared equally between myself (the author of the dissertation), the supervisor of the project and The University of Sheffield International Faculty of CITY College, and that any kind of exploit of this work will need the consent of all parties involved.

Name: _____

Signed: _____

Date: _____

Acknowledgements

I want to thank Dr Dimopoulos for helping me with my dissertation and most importantly for having the patience to bare with me until the end of this project. I also want to thank my friends who supported me in this journey and finally the open source community that provided me all the tools that I needed in order to implement this project.

Contents

1 Introduction **1**

1.1 Problem Definition 2

1.2 Aim 2

1.3 Motivation 3

1.4 Objectives 4

1.5 Report Structure 4

2 Literature Review **6**

2.1 Background of study 6

2.1.1 Artificial Life 6

2.1.2 Genetic Algorithms 9

2.1.2.1 Parallel Genetic Algorithms 16

2.1.2.2 The Island Model Distributed Genetic Algorithm 22

3 Project Management **26**

3.1 Project Vision 26

3.1.1 Proposed Algorithm 27

3.2 Development process 28

3.2.1 Development Plan 30

4 First Iteration **32**

4.1 Analysis 32
4.2 Design 33
4.2.1 Theoretical Model 33
4.2.2 ”Island” Software Component 34
4.3 Implementation 35
4.4 Testing 36
4.4.1 Overview 36
4.4.2 Test Strategy 40
4.4.3 First Testing Activity 41
4.4.4 Second Testing Activity 42
4.4.5 Third Testing Activity 43

5 Second Iteration 46

5.1 Analysis 46
5.2 Design 47
5.3 Implementation 48
5.4 Testing 50
5.4.1 Test Strategy 50
5.4.1.1 Testing Activity 50

6 Third Iteration 54

6.1 Analysis 54
6.2 Design 54
6.3 Implementation 56
6.4 Testing 59
6.4.1 Test Strategy 59
6.4.2 Testing Activity 59

7 Fourth Iteration **63**

7.1 Analysis 63

7.1.1 Migration Policy 64

7.1.1.1 Migration Topology 64

7.1.1.2 Migration Frequency 65

7.1.1.3 Migration Rate 66

7.1.1.4 Migration Selection and Replacement 66

7.1.1.5 Copy-based or Transfer-based Migration 67

7.1.1.6 Synchronicity of Migration 67

7.1.1.7 Important Points 68

7.2 Design 69

7.2.1 Software Stack Architecture Design 69

7.2.1.1 Back-End Part 71

7.2.1.2 Front-End Part 72

7.2.2 WebSocket for Real-Time Communication 72

7.2.3 Migration Communication Design 74

7.2.3.1 RPC Application to Migration Methodology 78

7.3 Implementation 79

7.3.1 Software Architecture Implementation 79

7.3.2 Migration Communication Implementation 81

7.3.2.1 Simulation Island 82

7.3.2.2 Master Node 88

7.3.3 WebSocket Communication Implementation 91

7.4 Testing 93

7.4.1 Test Strategy 94

7.4.2 First Testing Activity 96

7.4.3 Second Testing Activity 103

7.4.4 Third Testing Activity 107

8 Evaluation **111**

8.1 Results Discussion 111

8.2 Benefits of Framework's Software Architecture 112

8.3 Problems Encountered 113

9 Conclusion **115**

9.1 Future Work 115

9.2 Final Thoughts 116

A Code Snippets of Final Software **A1**

A.1 Software Archtecture A1

A.2 Island Container A2

A.3 Master Node A7

A.4 Front-End GUI A11

B Testing Runs **B1**

B.1 First Run Screenshots B1

B.2 Second Run Screenshots B3

B.3 Third Run Screenshots B5

List of Figures

2.1 General Structure of a Genetic Algorithm 12

2.2 Terminology in Genetic Algorithms 13

2.3 Master-slave Parallel Genetic Algorithm 18

2.4 Fine-grained Parallel Genetic Algorithm 19

2.5 Coarse-grained Parallel Genetic Algorithm 20

2.6 Hierarchical Parallel Genetic Algorithms 21

2.7 Most widely used migration topologies. 23

3.1 Abstract idea of the expected project. 27

3.2 An abstract structure of the Evolutionary Prototyping model. 29

3.3 The detailed evolutionary prototyping model used for the project. 30

4.1 Theoretical Model of the framework. 34

4.2 Docker-CLI command to create the image. 42

4.3 Docker-CLI command to create the container. 43

4.4 TigerVNC command to connect to the windowing system of the island container. 44

4.5 The example simulation already running in the island container when VNC connection established. 45

5.1 Software architecture developed so far. 48

5.2 Docker CLI command used to check if RabbitMQ container is up and running. 51

5.3 Docker CLI command used to start the island container. 52

5.4 RabbiMQ web page that shows the created queue and its data. 52

5.5 RabbiMQ web page that shows the payload of the message inside the ”IslandQueue” queue. 53

6.1 Software architecture developed so far. 55

6.2 Docker CLI command used to check if RabbitMQ container and master node are up and running. 61

6.3 Web page served by the master node before any message received. 61

6.4 Web page served by the master node after a message received. 62

6.5 Terminal of the Processing framework that shows the messages exchanged. 62

7.1 Broadcast migration topology 65

7.2 Software architecture of the framework. 70

7.3 WebSocket communication between the system’s web page and master node. 73

7.4 RPC communication pattern as the underlying implementation of the broadcast migration topology. 77

7.5 RPC communication pattern for the migration process. 78

7.6 Commands that check the two preconditions of this testing activity. 98

7.7 Docker CLI command to deploy the software stack of the framerwork on the available distributed system. 98

7.8 First screenshot of the simulation’s first run. 99

7.9 Second screenshot of the simulation’s first run. 100

7.10 Third screenshot of the simulation’s first run. 101

7.11 Third screenshot of the simulation’s second run. 102

7.12 Second screenshot of the simulation’s third run. 102

7.13 Fourth screenshot of the simulation's third run. 103

7.14 Docker CLI command used to check which software stacks are up and running. 105

7.15 Docker CLI command used to scale the specified service by the specified number of containers. 105

7.16 Visualizer UI before scaling the system's simulation island component. . 106

7.17 Visualizer UI after scaling the system's simulation island component. . . 107

7.18 Docker CLI command used to check which software stacks are up and running. 109

7.19 Docker Machine CLI command used to shut down a Docker host. 109

7.20 Visualizer UI before shutting down the "myvm4" worker node. 110

7.21 Visualizer UI after shutting down the "myvm4" worker node. 110

B.1 First Screenshot of the simulation's first run. B1

B.2 Second Screenshot of the simulation's first run. B2

B.3 Third Screenshot of the simulation's first run. B2

B.4 Fourth Screenshot of the simulation's first run. B3

B.5 First Screenshot of the simulation's second run. B3

B.6 Second Screenshot of the simulation's second run. B4

B.7 Third Screenshot of the simulation's second run. B4

B.8 Fourth Screenshot of the simulation's second run. B5

B.9 First Screenshot of the simulation's third run. B5

B.10 Second Screenshot of the simulation's third run. B6

B.11 Third Screenshot of the simulation's third run. B6

B.12 Fourth Screenshot of the simulation's third run. B7

List of Tables

4.1 Test case for investigating the creation of the Docker Image. 41

4.2 Test case for investigating the creation of the Docker Container. 42

4.3 Test case for investigating the automatic start of the simulation on the container's boot. 44

5.1 Test case for investigating the communication between the island container and the RabbitMQ server. 51

6.1 Test case for investigating the communication between the island container and the master node. 60

7.1 All the possible cases that the master node may encounter during a migration operation. 96

7.2 All the possible cases that an island may encounter during a migration operation. 97

7.3 Test case for investigating the scalability feature of the framework. 104

7.4 Test case for investigating the fault-tolerance feature of the framework. . 108

List of Abbreviations

AE Artificial Entity.

AI Artificial Intelligence.

ANN Artificial Neural Network.

CAS Complex Adaptive System.

CLI Command Line Interface.

EA Evolutionary Algorithm.

EJS Embedded JavaScript.

FPGA Field Programmable Gate Array.

GA Genetic Algorithm.

GPU GraphicalProcessing Unit.

GUI Graphical User Interface.

MITM Man In The Middle.

OOP Object Oriented Programming.

QA Quality Assurance.

RPC Remote Procedure Call.

SDLC Software Development Life Cycle.

SSE Server-Sent Event.

UI User Interface.

VNC Virtual Network Computing.

Listings

4.1 Dockerfile used for the creation of the container that runs the simulation. . 35
4.2 Shell script that start all the needed services on the island container on boot. 36
5.1 Constructor of the ”ClientIsland” class. 49
5.2 Method ”sendMessage()” of the ”ClientIsland” class. 49
6.1 Web server implementation of the master node. 56
6.2 Implementation of the web app part of the master node. 56
6.3 Dockerfile used for the creation of the master node. 58
6.4 Implementation of the ”receiveMessage” method of the ”ClientIsland” class. 58
7.1 Docker-compose.yml file of the software system. 79
7.2 Code for sending a migration to the master node. 82
7.3 Code for requesting migrations from the other islands. 83
7.4 Code for receiving migrations from the other islands. 83
7.5 Code for checking the proper delivery of RabbitMQ messages and if they include any migration. 84
7.6 Code for correlating RabbitMQ messages and extracting data from them. 85
7.7 ”MigrationMessage” class that represents the migration entity. 86
7.8 The ”wm_startup.sh” shell script used by the simulation container to start programs on boot. 87
7.9 Main function of the master node. 88
7.10 Code for checking if the received RabbitMQ message is a migration or a request for migrations. 89
7.11 Code for properly inserting a migration to the sorted list of all the migrations. 89
7.12 Code for retrieving migrations to send as a response to an island. 90
7.13 Code for waiting WebSocket connections in the app.js file. 91
7.14 Code for sending data to the system’s web page through the WebSocket connection. 91
7.15 Code for rendering the web page before sending it to a client. 92
7.16 Code for connecting with the master node and receiving data from it. . . . 92
7.17 HTML body of the front-end’s web page. 93
A.1 Docker-compose.yml file of the software system. A1
A.2 ”ClientIsland” connection class. A2
A.3 ”app.js” JavaScript file that implements master node’s main functionalities. A7
A.4 ”package.json” file that contains the main configurations. A10
A.5 ”index.ejs” file that implements the html of the GUI. A11

Chapter 1

Introduction

Over the last few decades the discipline of Artificial Life attracts increasingly more attention, as new simulation technologies emerged along with advanced Artificial Intelligence (AI) techniques. The topic of Artificial Life is a field of study which has become subject of research for computer scientists over the recent years. Artificial Life, also known as ALife, is concerned with any aspect of life, such as behaviour, population and other characteristics of living organisms [1]. Popular examples of this concept are bacteria spread, a simple ant population and some other breeding and behaviours of virtual creatures. In the last few years, as mentioned before, advanced simulation technologies appeared and are extensively used in ALife scientific field. For that reason, this field can be and is often combined with various AI techniques, such as Genetic Algorithms (GAs) and Artificial Neural Networks[1] (ANN) [3]. A typical example of this matter would be a digital simulation of ALife that simulates the evolution of Artificial Entities (AE) with a brain, which is actually an ANN. This way, evolving the ANN of a AE, it will be able to adapt better to the environment and thus will probably produce offsprings for the next generation. The next generation will consist of AEs with evolved ANNs which will render them more enhanced versions of their predecessors. Hence, following the described way of breeding, the population of these virtual creatures will evolve in a similar way as a natural population of living organisms does.

Many digital simulations of ALife, similar to the example above, utilize a GA as a computational model for natural evolution [3]. In other words, the functionality and the overall behavior of an ALife simulation is mainly dictated by a specific GA. This is because, GAs are considered highly suitable for this task and thus play a central role in ALife digital simulations. The reason they are so applicable to ALife simulations is the fact that they use rules of natural selection to operate [4], the very same rules that these kind of simulations typically use in order to evolve their agents. Thus, an algorithm from the class of GAs is usually chosen and forms the backbone of the simulation software, as it provides the basic behavior expected from the simulation to have. GAs are a category of Evolutionary Algorithms (EAs) [5] and they are described in detail in section 2.1.2.

[1] A computational network mainly used for classification and pattern recognition, which is inspired from the neural network of the human's brain [2].

1.1 Problem Definition

Despite all that, there are two problems with regard to operating these type of simulations that have been noticed and addressed in this paper. The first one is related to the potentially high computational cost of the ALife simulations. That is, for some simulations the GA may require relatively high computational resources. For example, if in the ALife simulation from the example above, the ANN incorporated many layers or ,even worse, it was a Deep ANN[2], then this would impose intensive calculations to the simulation. Consequently, powerful computational resources would be needed in order to handle all these calculations. Thus, this type of simulations are computationally expensive and as more agents are added to the population, the more computational load the GA needs to deal with [5]. Hence, this problem poses performance limitations to the execution of some complex ALife simulations.

The second problem is encountered when the developers of these simulations try to solve the first problem. More specifically, most developers try to solve the problem of large computational loads by distributing it over a cluster of machines or any other distributed system. In this case, they usually face the problems of configuration overhead and many other complexities that come with this procedure. Thus, this problem has to do with the overall time-consuming practice of making the whole simulation a distributive application and then trying to operate it on a distributed environment in a proper way.

1.2 Aim

The main purpose of this paper is to address these two problems and propose a solution that resolves them in a modern and efficient way. Put differently, the aim of this project is to design, develop and test a framework for running ALife simulations on a distributed environment. In addition, this framework will be designed to solve these problems in an elegant and modern way by incorporating the latest state of the art cloud and web technologies.

At this point, it is important to realize that the current project will be built upon the previous works on the same project, which had produced a simulation software. More specifically, the previous works concentrated on creating an ALife simulation and it was successfully developed. Meaning, it is fully functional and able to simulate the natural evolution of organisms, following the Darwin's theory of evolution. In fact, it is the exact implementation of the example of Alife simulation mentioned before. That is, it is a digital ALife simulation, which combines the fields of Neural Networks and Genetic Algorithms. More precisely, the AEs in the simulation have a brain, which is actually a Neural Network, and they evolve according to a specific Genetic Algorithm. However, it is a usual single component software that is not able to run in distributed fashion and is far from being a

[2] A more advanced and complex type of Artificial Neural Networks with bigger amount of layers between the input and output layers [6].

framework for evolution simulation.

Therefore, this simulation will be used as a starting point in order to design and develop the expected distributed framework, which will eventually fulfill the main goal of this work.

1.3 Motivation

The main motivation of this work is to provide a useful software framework for running Artificial Life simulations. This framework will enable researchers to focus solely on the aspects of GA that they use to govern the behaviour of their Artificial Life simulation. They will not have to worry about the details of execution either on cloud or any other distributed system. The underlying algorithm of the framework will undertake the task of executing their simulation software on a distributed environment in a proper and efficient way.

This brings to the second motivation of the project, which is to provide computational efficiency to the user of the framework. That is, the expected distributed nature of the framework is intended to enable it to exploit the modern parallel computing. By doing so, it will maximize its efficiency and enable it to deal with much larger computational loads. The reason behind this motivation is the fact that currently parallel computing is considered very promising in terms of computational efficiency [7]. Additionally, nowadays parallel computing has become more commonplace in scientific computing than ever before. Consequently, they are more accessible now than in the past. Cloud computing is also on the rise these years, encouraging the researchers even more to use its pool of computing resources to support large computational loads of their simulations. Moreover, another reason for this motivation is the inherent parallel nature of GAs [8]. This trait makes the GAs a good fit for distributed applications, which boosts this motivation even more.

Finally, the third motivation has to do with the overall contribution of the framework to the discipline of ALife. Having said that, the expected framework will run evolution simulations, which can provide digital illustrations of natural biological phenomena. For instance, the simulation used in this case can be utilized as a tool in order to explore various general principles underlying natural life, such as natural life's evolution and its processes. As such, operating a simulation on the framework will enable a biology researcher to investigate the fundamental processes of living systems, that empower their evolution in an artificial environment. This way, the biology scientist will gain a deeper understanding of the complex information processing that occurs in such environments. Hence, the project provides a powerful tool which is able to reveal useful information regarding various aspects of life, if used properly as part of a scientific research. In that sense, the proposal of such a modern framework would be regarded as a significant contribution to the research community.

1.4 Objectives

In order to achieve the aim of the project, specific criteria need to be satisfied. In particular, there are four main objectives that the expected project needs to accomplish. These objectives are described below:

- **Scalability**: The framework needs to be scalable, since it is intended to solve the problem of large computational loads. That is, by scaling the framework over the available computational nodes of the cluster or any other distributed system, it will be able to handle more easily the simulations that are computationally intensive.
- **Fault tolerance**: The framework must also be fault tolerant, so that it is able to operate even in the cases of failure of one of its components. This way, the system will have the capability of delivering uninterrupted service in the face of node failures, that usually occur in a distributed system. Hence, the feature of fault tolerance will enable the framework to have high-availability in any distributed environment.
- **Portability**: The framework should be easily deployed in any distributed environment, either on a local cluster of machines or a cloud infrastructure. This will contribute to the ease of use of this framework. In other words, the more easily it is to deploy the framework anywhere, the more its usability is boosted.
- **Real-time behavior**: The framework should provide its users the ability to watch the operation and progress of their simulation in a real-time manner. In other words, the framework needs to be a real-time distributed application, so that its user interface (UI) can act as a dynamic and real-time monitoring system of the simulation. This way, the user will be enabled to watch live the progress and results of his simulation, through the user interface of the framework.

1.5 Report Structure

The report provides an extensive documentation for the project, which is designed to be coherent and clear. Therefore, it is organized in chapters, which represent phases of the project development. These chapters are also organized in sections and subsections, providing structured and coherent information to the reader. Also, additional material can be found in the appendices at the end of the report. Following is the detailed structure of the report:

1. **Chapter 1 (Introduction)**: Introduces the topic of the paper, highlights the problems noticed and proposes the solution along with the motivation behind it.
2. **Chapter 2 (Literature Review)**: Represents the research conducted for the project and thus provides a theoretical background, which describes the concepts involved in this project.

3. **Chapter 3 (Project Management)**: Describes the software development process that was followed and gives a detailed analysis of the project.
4. **Chapter 4 (First Iteration)**: Provides an abstract architecture of the system and also the first software component of that architecture.
5. **Chapter 5 (Second Iteration)**: Is concerned with further development of the first component.
6. **Chapter 6 (Third Iteration)**: Introduces the development of another software component of the system's architecture.
7. **Chapter 7 (Fourth Iteration)**: Describes the iteration where the final project is developed and tested.
8. **Chapter 8 (Evaluation)**: Discusses the results of this project and also highlights the strong points of the project's software architecture.
9. **Chapter 9 (Conclusion)**: Presents future work that is needed in order to improve the functionality and quality of the project and also the final thoughts on the product of this project.

Chapter 2

Literature Review

This chapter describes the literature review of this project, which demonstrates the research that needed to be conducted in order to be able to design and develop this project. As such, this review provides the theoretical background that is needed in order to grasp the concepts of Artificial Intelligence, upon which the project is built.

2.1 Background of study

2.1.1 Artificial Life

Artificial Life is a recognized discipline since the 1980's, that was coined by an American computer scientist Christopher Langton, and it is basically a field of study whose main purpose is to examine the essential properties of living systems [9]. This examination is achieved through human-made systems that imitate the characteristics of a living system [10]. In other terms, research in ALife complements the traditional biology sciences, which are naturally concerned with the investigation of living systems, by providing human-made simulations of natural life. This way, biology researchers are given the opportunity to study natural life beyond the constraints of what is known as "life-as-we-know-it". That is, using ALife tools they are able to investigate living systems in the larger context of "life-as-it-could-be" and finally understand the fundamental properties that constitute these systems.

The approach of "life-as-it-could-be" essentially means that there must be other examples of living systems in order to derive general principles that govern a living system [10]. To explain, up until now the traditional or theoretical biology has studied life which is based on carbon-chain chemistry. That is the only type of life that was available to the study, so it was difficult to derive general theories from only that single example. This is why till today theoretical biology has studied natural life only in the context of "life-as-we-know-it", since living systems, that are based on different physical chemistries, were

not available. Hence, the discipline of ALife tries to overcome this obstacle by providing alternative life-forms as new examples of living systems through man-made simulations.

According to Taylor and Jefferson [11], research in ALife organizes natural life in four basic levels of structure: the cellular level, the molecule level, the organism level and the population-ecosystem level. Knowledge at all these levels provides understanding of life in any depth. Therefore, each of the four levels is addressed by a specific type of simulation, which is based either on computers or other artificial media. More precisely, there are three main types of systems that operate as artificial simulations in ALife: the software systems, the hardware systems and the wetware systems. Each one of these types is used for one, sometimes two or three, particular levels of structure. That is, the molecular level is studied through wetware systems, the cellular and population levels are studied through software systems and the organism level through hardware systems. In the case at hand, the project to be developed is a framework for software systems. Put another way, it is a framework for software-based simulations.

Since the field of ALife is mainly concerned on imitating natural life, the key to achieving this imitation is to create life-like intelligent behaviors. Thus, ALife is mainly focused on the behavior of the agents, analysing the factors that make a behavior intelligent and adaptive. ALife also examines the ways that this intelligent behavior may emerge [1]. Based on the behavior-oriented approach of ALife, intelligence is defined in terms of observer behavior and self-preservation. More specifically, an intelligent agent in ALife simulation is the one that maximizes the chances of self-preservation and adaptation in the given environment. That is why ALife simulations are regarded as complex adaptive systems (CAS). Furthermore, according to Christopher Langton [10], this kind of behavior is emerged through a large number of organized interactions of the underlying parts of the living system. These parts are considered to be separate behaviors of non-living molecules of the natural system. Hence, life-like behavior is emerged out of local distributed interactions of behaviors, which is regarded as a bottom-up procedure.

It is also important to note that natural systems are fundamentally parallel [10]. In more detail, this bottom-up procedure of behavior generation takes place in a parallel fashion, since living systems are highly distributed and massively parallel. Consequently, the ALife simulations, that will model natural systems, must also incorporate these characteristics and operate in a highly distributed and parallel manner. That is why Pedro et al [12] in their paper suggest cloud computing to be utilized to support the operation of ALife software-based simulations. Because cloud provides virtual resources that can be used in parallel in order to enable the simulation to run in highly distributed and parallel fashion. Moreover, beneficial characteristics of cloud technology, such as message queueing and event-driven architecture, can significantly improve the adaptivity and sensitivity of a simulation's environment.

Apart from that, ALife simulations can also be delivered as cloud services through the internet [12]. That is, cloud technology provides the possibility to create services that serve as ALife frameworks. In other words, instead of creating ALife-as-a-framework, it is possible to create ALife-as-a-service. This paradigm shift will relieve users from the need to set up their own infrastructure and will also reduce the costs associated with that

infrastructure.

This can be taken further, as the concept of complex adaptive system, used in ALife simulations, can also be used in cloud computing [12]. That is, services in cloud infrastructures can be autonomous and adaptive, forming a self-managed computing infrastructure called autonomic computing. This way, the cloud infrastructure as a whole adopts complex adaptive behavior. Meaning, it will be able to self-heal, to reconfigure network connections, spawn new copies or delete them and other things. Additionally, services will be able to adapt to any cloud environment (e.g. containers, full-fledged virtual machines), to automatically migrate from one environment to another and to financially sustain themselves, through bitcoin mining for instance. All these adopted abilities are exploited without the need of a central authority. Instead peer to peer protocols, such as gossip protocols, are used to support these new functionalities, which means, the cloud infrastructure will continue to operate in a distributed manner.

The architecture of fog computing also exhibits some signs of complex adaptive systems. In particular, some key characteristics of fog computing, such as its decentralized, user-centric architecture and the employed collaborative protocols, remind the concept of complex adaptive systems. Therefore, findings in ALife research may also contribute to the fog computing field.

Artificial Life techniques are not only used for accurately modeling living systems in order to test biological hypotheses, they are also used in fields related to engineering [11]. The reason for that, as stated by Bonabeau and Theraulaz in their paper [13], is the fact that ALife techniques are highly suitable for systems that are too complex to tackle with traditional scientific tools. Hence, concepts of ALife are used in many fields of engineering to solve complex problems, despite the fact that the solution algorithm may be distantly related to any natural phenomena or biological system.

Apart from engineering though, ALife techniques can be used in many other scientific fields such as biology, computer science, art and philosophy [13]. For this reason ALife is regarded as an interdisciplinary field of study. However, wrong conclusions may be drawn when ALife concepts are used by researchers from solely one scientific field. For instance, computer scientists are generally not familiar with biological systems, something that will hinder the quality of the research. Thus, the key to the proper way of utilizing this approach is communication and cooperation between the involved fields.

Overall, Artificial Life is a relatively new discipline, which can be used in many other disciplines to either solve complex engineering problems or accurately model biological systems. The latter one is very important for theoretical biology, since many major findings may be discovered and thus reveal significant aspects of natural life.

2.1.2 Genetic Algorithms

Genetic algorithms (GAs) are adaptive search algorithms inspired by Charles Darwin's theory of evolution [14]. They belong to the broader family of Evolutionary Algorithms and they are typically used to find high-quality solutions for optimization and search problems. GAs are based on concepts of natural selection, where the individuals who can better adapt to their environment are able to survive and thus are selected for reproduction [4]. In other words, these algorithms simulate the notion of "the survival of the fittest" that exist in natural evolution. Thus, they are also used in Artificial Life as models of evolution [3].

According to Mitchel [15] Genetic Algorithms, as they are known today, were invented by John Holland in the 1960s. It was a momentous breakthrough as the previous versions of GAs did not have the operations of selection and crossover, instead they used only the operation of mutation to evolve the population. These earlier versions of GAs were first devised by Nils Barricelli and then further developed by Alexander Fraser and other biologists [14]. However, Baricelli's goal, before inventing the first GAs, was to develop Artificial Life, whereas Holland's goal was to understand the adaptivity of living organisms and systems to their surroundings. That being said, in 1975, with the help of his students and colleagues, Holland published his book "Adaptation in Natural and Artificial Systems" [14]. In this book, he describes GAs as an abstraction of biological evolution and uses them to provide a theoretical framework for adaption. This book was the first one to provide a theoretical foundation for computational evolution and many following theoretical works on GAs were based on this book, since it also presented the mathematics behind evolution.

In a general sense, a GA modifies a set of candidate solutions for a problem with GA operations until it finds the optimal set of solutions [14]. A candidate solution in the GA terminology is called "individual" and the whole set of candidate solutions is called "population". The stage of finding the optimal solutions is usually the stage where the fitness of the fittest individuals is stabilized and does not change with the successive GA modifications. This is what it is meant by saying a GA "converged" to a solution. Also, the GA operations, mentioned previously, are the three genetic operators exclusively used in GAs: selection, crossover and mutation. GA operators are inspired by natural selection and natural genetic variation and they are also called genetic operators [15]. These GA operators are responsible for producing a new population, that is based on the fittest individuals of the previous population [4], and they will be further described later in the text.

Having said all that, the population is first evaluated, assigning fitness values to each individual, and then the individuals with higher fitness value undergo recombination operations by the three genetic operators [14]. More precisely, the evaluation process involves testing the potential solutions to examine how well they solve the target problem and assigning them fitness values that correspond to their performance. This cycle of evaluating, modifying and finally producing a new population is called generation [4]. However, it is important to note that at the beginning a GA needs to initiate the first population, which is typically a randomly generated population of possible and feasible solutions. The reason

for this randomness is to initiate the search from a wide range of points in the solution space in order to ensure the search to be unbiased [4]. In addition, whenever new individuals are created by the genetic operations, a reinsertion plan is needed to dictate how these individuals will be inserted to the new population of the next generation [16]. Hence, there are two kinds of GAs that have different strategies for that matter: the generational GA and the steady-state GA [15]. In the case of the former, the population of the next generation is entirely composed of offspring of individuals from the previous generation. Whereas, in the case of the steady-state GAs, only a small portion of the next generation consists of new individuals, which represent the combination of fittest individuals from previous population and typically substitute individuals of the next population with the lowest fitness value. That being said, every next generation, of either type of GAs, is composed of individuals with higher average fitness. This way, after every generation the candidate solutions (individuals) are closer to the global optimal solution for the target problem.

However, there has to be a rule or a mechanism that will determine when the algorithm will stop, so that it does not run forever, trying to find better and better individuals. This way, termination criteria need to be in place whenever a Genetic Algorithm is used. Termination criteria are essentially stopping conditions, which are used as a guideline for stopping the algorithm when it reaches an acceptable solution that is close to the optimal solution for the problem at hand [17]. Nevertheless, due to the fact that termination criteria have different effects on various GAs, it is impossible to devise a general rule for optimal use of a termination criterion [18]. For this reason, finding a suitable termination criterion for a specific GA is considered a challenging task. Even so, the theoretical study on GAs suggests several desirable properties that a proper termination criterion needs to have [17] and they are the following:

- Should lead to satisfactory solutions.
- Should be applicable for any fitness function
- Should restrict the total number of strings searched to 2^L
- Should be easy to implement

Moreover, according to Safe et al. [19] there are three traditional termination criteria, which are used very often in GAs, especially the simple Genetic Algorithms (sGAs). These are the following:

- The maximum number of generations is reached
- The maximum number of fitness function evaluations is reached
- The probability of achieving notable changes in the successive generations is significantly low

Apparently, there are many more different termination criteria used in a GA, that are based on different concepts and conditions. That being said, Clausen et al. in their survey [18] classify all these possible termination criteria into seven categories. These categories are the following:

- Direct termination criteria
- Derived termination criteria
- Cluster-based termination criteria
- Operator-based termination criteria
- Performance indicator termination criteria
- Progress indicator termination criteria
- Termination criteria in hybrid MOEA

The first category is the simplest one (direct termination criteria), since it is based on predefined parameters. All the rest categories are based on adaptive stopping conditions and are more complex. It is also worthy to note that the current trends are towards the latter types of termination criteria, instead of the direct termination criteria [19].

Therefore, after the termination criteria are met, the algorithm stops and deliverers the found results to the user. These results are the fittest individuals of the final population, which represent the optimal potential solutions for the target problem. On the whole, the structure of a GA, described so far, can be expressed by the illustration in the figure 2.1.

Further to this, these potential solutions in a GA must be encoded in a chromosome-like data structure, which is usually a string of parameters [4]. So in essence, the data structure of an individual represents a “genetic structure” of a candidate solution or hypothesis. It is also important to point out the importance of the encoding in a GA, since the way that the potential solutions are encoded is one of the most significant factors in the success of a GA [15]. Hence, there have been many experimentations on that matter and currently there are several types of encoding, such as binary encodings, tree encodings, and gray encodings among others. The most usual though is the binary encoding, where each parameter value is converted to a fixed-sized bit string and then all of them are concatenated end-to-end, like genes in a DNA strand to form chromosomes [14].

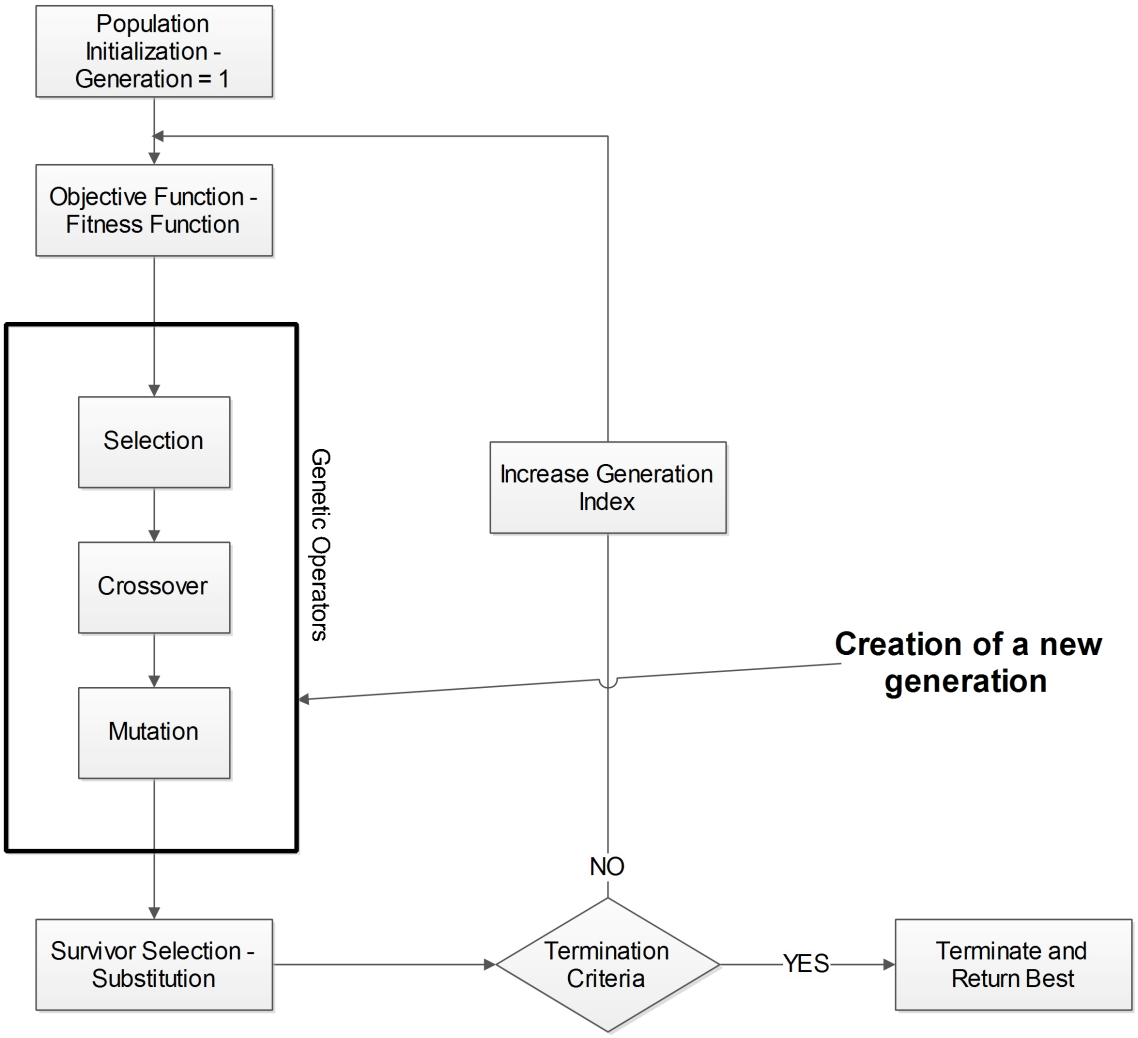


Figure 2.1: General Structure of a Genetic Algorithm

At this point it would be appropriate to describe the basic terms in the GA terminology, some of which has already been mentioned. These terms are the following:

- Chromosome
- Population
- Gene
- Allele
- Locus

A chromosome in a GA represents a candidate solution for the problem at hand and it is usually encoded as a bit string [14]. In addition, a chromosome serves as a "genetic

structure" or a blueprint of the potential solution, which many times is referred to as "individual" in the context of GAs instead of "chromosome". All the chromosomes, participating in a GA, constitute a population. Each one of these chromosomes consists of a set of genes, which represent parts or parameters of the solution [4]. More specifically, genes can be single bits or short blocks of adjacent bits, each one of which forms a particular part of the solution. Furthermore, an instance of a gene is expressed by a particular allele, which is usually 0 or 1 [15]. Put differently, an allele represents a value that a gene of a chromosome provides. For example, if the gene is regarded as a trait in an organism, such as skin color, then possible alleles for that gene could be "black", "red" or any other color. Furthermore, the exact position of the gene in the chromosome is called "locus" [15]. All these terms are depicted in figure below.

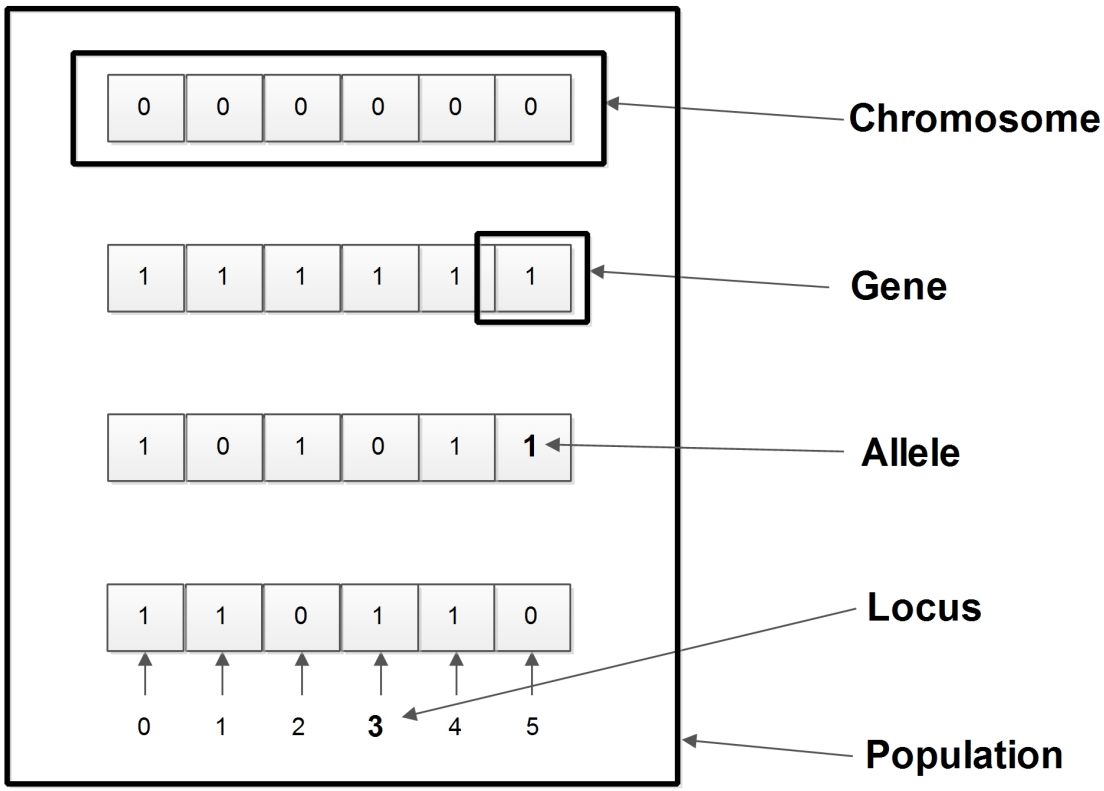


Figure 2.2: Terminology in Genetic Algorithms

The genetic operators, as mentioned before, are responsible for creating a new population of possible solutions. This creation process though entails the exploitation of promising solutions by the three genetic operators [4]. More particularly, the selection operator determines the individuals that will produce offsprings for the next generation and also the number of offsprings they will produce [15]. This GA operator follows the natural selection concept [4], which means that the more fit the individual is, the more likely it is to be selected for reproduction. There are several types of selection GA operators, each one of which uses different strategies to select the appropriate individuals for reproduction. The most common and widely used selection strategies are the "Roulette Wheel", "Rank Selection" and "Tournament Selection". After defining the individuals for reproduction, there needs to be a way to recombine these individuals in order to produce

new ones. This is where the crossover GA operator comes into the scene. This operator is responsible for extracting different parts of two individuals (portions of chromosomes) in order to combine them and produce new individuals for the next generation [4]. At this point, it is important to note that the crossover operation does not produce totally new chromosomes, since it only combines the ones that already exist. Thus, the idea of the crossover genetic operator is to create new individuals by combining (potentially promising) genes of already available chromosomes. There are many kinds of crossover operators in the literature, but two are the ones that are widely adopted: one site crossover and two site crossover. After the crossover operation, another GA operation follows to complete the creation of an individual. That is the mutation operation, which unlike the crossover operator, adds new information to the chromosome of the new individual. That is, the mutation GA operator provides new information to the search process of a GA by randomly modifying a gene in the chromosome of the individual being created [16]. The idea behind this process is to provide diversity to the new population, so that the algorithm does not get trapped in a local optima.

It is also important to describe the notions of objective and fitness functions and their differences as well. Objective function of a GA is used to assess the performance of an individual for the specific problem domain [16]. From that performance a fitness value is derived and assigned to the individual. The way that this value is derived from the objective function's raw performance measure is dictated by the fitness function. That being said, fitness function is used to map the performance measure of an objective function to a more appropriate performance measure, which is more expressive for the relative fitness of individuals in a GA [16]. For example, in the case of a minimization problem, the most fit individuals will have the lowest numerical value calculated by the objective function. Nevertheless, this value is not intuitive and is not appropriate for comparing the individuals in a GA population. For this reason, a fitness function is used in order to map or translate this measure into a more intuitive relative measure, which in this case would represent higher numerical values for fitter individuals. Another point to consider is the fact that the fitness function represents the only way for the developer to communicate his intentions to the powerful search methods of GAs, since it is the only component of the algorithm that decides how the individuals change over the generations [14]. Thus, it is crucial to communicate these intentions with precision, because, as Kinnear stresses in his book [20], the evolution of the population will certainly exploit the flaws and weaknesses of a fitness function.

Jenna Carr in her paper [14] points out the superiority of GAs over other traditional optimization methods. These optimization methods are categorized into three types: calculus-based, exhaustive search, and random. The methods of the first category usually utilize the "hill-climbing" approach and are applied only when derivatives are provided. The problem of the calculus-based methods though is that they cannot be used in practical problems, since they require the existence of derivatives. Apart from that, calculus-based methods are only capable of searching the local optima of a search space, which renders them useless in case where the neighbourhood of the global optimum is unknown. On the other hand, the exhaustive search optimization methods seem more attractive, since they are more simple and guarantee to find the best possible solution. These methods though are very inefficient, as they need to test each and every solution in the search space in

order to find the best one. In fact, exhaustive methods are the least efficient methods compared to the other two categories and they are usually impractical for the most real life problems, since their search space is too vast to explore using that approach. Lastly, the random optimization methods are considered to be the best choice out of the three categories, since they emerged after people saw all the weaknesses of the other two types of methods. More specifically, these methods pick different points of the search space in a random fashion and use them as a sampling. That sampling is a representative sampling of the search space, in which the random method will find the optimal solution. Therefore, random methods are definitely more efficient than the exhaustive ones, still though they cannot provide better results from the latter ones. That is, whether the found solution is close or far from the optimal solution is purely up to luck.

Genetic Algorithms are considered much better choice from these three types of optimization methods, as they do not have their shortcomings. For instance, GAs do not need any additional information about the problem in order to find optimal solutions for it, such as derivatives (as in the case of calculus-based methods). Instead, the only thing a GA needs is a numerical fitness value from the fitness function that the user defined [16]. Apart from that, GAs in general differ from all the other search and optimization methods in many different ways. Chipperfield et al and Mathew in their papers [4, 16] summarize them into the following four main differences:

- GAs focus on the coding of the solution set, instead of the solutions themselves.
- GAs search from different points of the search space, meaning they do not concentrate on a single solution.
- GAs do not require additional or auxiliary knowledge to operate, instead they use payoff information (fitness function).
- GAs utilize a probabilistic approach in their search, not a deterministic one.

Another important point to consider is the fact that GAs are algorithms that are designed to properly balance the exploration and exploitation of the search space [4]. In other words, they explore various points of the search space by initializing a population of random solutions at the beginning of the algorithm and also by the mutation operator. But, they also exploit promising points of the search space by the selection and crossover operators. Hence, GAs are able to balance the exploration and exploitation of the solution space and they also provide the ability to the user to control this balance, by manipulating the operators and other parts of the algorithm.

Overall, Genetic Algorithms are elegant and robust search algorithms, able to find quality solutions for difficult high-dimensional problems in a reasonable amount of time. However, the key to successfully utilize these algorithms is to properly define a GA representation of a potential solution and a meaningful fitness evaluation [4]. Therefore, by being able to define those two, GAs can be applied to many different problems and be useful for both problem-solving and modeling. Specifically, they can be used for scientific and engineering problems and also be applied to business, economics and entertainment

[14]. Few examples of their applications are Machine and robot learning, economic models and automatic programming among many others. In general terms though, as Mathew states in his paper [4], GAs are useful mainly in the following four cases:

- The search space of the problem is vast, complex and not well understood
- Domain or expert knowledge is difficult to utilize to narrow the search space
- There is no mathematical analysis of the problem
- Traditional optimization methods fail to solve the problem

In all those cases a GA is much more efficient than in any other case scenario. So, it is necessary to know when to utilize these algorithms and when not, in order to reap their benefits.

2.1.2.1 Parallel Genetic Algorithms

As Adamidis and Cantu-Paz point out in their papers [21, 22], despite the fact that Genetic Algorithms are generally effective search algorithms and able to find satisfactory solutions in a reasonable amount of time, they tend to face difficulties when applied to harder and more complex problems. To explain, these types of problems usually have large search space and thus demand a GA to use large population sizes and possibly costly evaluation functions. This typically results in increased time needed for a GA to find adequate solutions. Apart from this, even the found solutions at the end of the search, are usually far from optimal and thus are of low quality [23]. The reason for that is usually the fact that GAs are typically prone to "premature convergence", which is the case where the population of a GA is dominated by a suboptimal solution and thus all the following offsprings are very similar to that dominant solution. This is also the condition where the algorithm is said to be trapped in a local optimum [5]. Therefore, there have been many efforts to make a GA more effective and faster, when used for complex and demanding problems. As it seems, one of the most promising solutions to these two problems is the parallel implementation of GAs [22]. The reason for this is the fact that, as mentioned before, Genetic Algorithms are naturally prone to parallelism, since some operations are, to some extent, independent from each other [24]. Hence, the task of devising a parallel implementation of a GA should not be something difficult or complex, which makes the choice of parallel implementation for algorithm efficiency even more tempting.

In the literature, the ordinary GA, that was described in the previous section, is called "sequential GA" and the parallel implementation of it is called "parallel GA", also known as PGA. However, most people would consider a PGA as a parallel version of a sequential GA, which is far from true. In fact, a PGA is essentially a parallel algorithm, which is able to behave better than the sum of its separate sub-algorithms [24]. To elaborate more, the basic idea behind PGAs is the "divide and conquer" approach, according to which a task in the algorithm is divided into manageable chunks [22]. These chunks are then solved

simultaneously using multiple hardware or software components, such as processors or threads. This approach though is applied in many different ways to a GA for a successful parallel implementation. That is, some parallel implementations use a single population, while others use several relatively isolated subpopulations of the original one. Also, some use massively parallel computer architectures, while others use a cluster of computers that have noticeable computing power.

In addition, Luque and Alba introduce in their paper [5] the concept of decentralized or structured GAs, in order to describe PGAs. More specifically, they argue that in order to successfully parallelize a GA, it is often necessary to first structure it and then make the parallel implementation for it. That being said, there are several different ways to structure a GA, which will be described more thoroughly later in the text. It is also possible to use many different parallel implementations for the same structured GA. Another important point to make is that a structured GA can either be executed in parallel or not. Put differently, it is not necessary to make a parallel implementation for a structured GA, since its architecture already imposes a parallel execution of some tasks of the algorithm. This is also the reason why a structured GA can still perform well even when it is run on a non parallel computer architecture [24].

There are many different PGAs in the literature of GAs, but most of them fall into the following four main types:

- master-slave PGAs
- fine-grained PGAs
- coarse-grained PGAs
- hierarchical PGAs

These are the most common ones that are used very often both in the industry and research, out of which the fine and coarse grained PGAs have even more popularity. However, Luque and Alba in their work [5] introduce one more type of PGAs, which is the “independent runs PGAs”. Furthermore, each one of these types of PGAs are also called “models”. For example, the second type can also be referred to as “fine-grained model”.

That being said, the master-slave model, also known as ”global parallel GA”, uses a single population in its algorithm. However, the task of evaluating the individuals is distributed over a number of slave processors and thus is done in parallel [5]. In other words, this algorithm is identical to the sequential simple GA, with the only difference that fractions of the population are sent to different available processors for evaluation operations. This means that the solution space exploration is exactly the same with that of a simple GA. So, even if the number of available processors is increased, the same solutions would be evaluated. The only thing that would change is the execution time, which would be decreased. For this reason, the master-slave model is typically applied in cases where the evaluation function is computationally expensive and thus slow [22]. However,

this model suffers from the communication overhead problem, since communication bottleneck is created in the master processor as the number of slave processors increases. Hence, the more computationally expensive the evaluation function is, the more effective the master-slave model will be, since the communication overhead becomes unimportant in relation to the evaluation function time. Overall, master-slave PGAs use the master-slave paradigm (hence the name) where the population is stored in the master computing node and the evaluation operation is distributed among the slave computing nodes. More specifically, as illustrated in figure 2.3, the master sends a number of individuals along with the parameters for evaluation function to the slaves, which then compute their fitness values and return them back to the master. Furthermore, the master node controls the parallelization of the evaluation function and thus its distribution over the available slave nodes. As a whole, this model provides the advantage of computational speed, mentioned before, and also the fact that it can be easily applied to a simple GA, since there is no need to alter the search behavior of the algorithm, as previously explained [22]. On the other hand, apart from the communication overhead problem, another disadvantage of this model is that it uses a single population, which limits the search diversity of the algorithm [24].

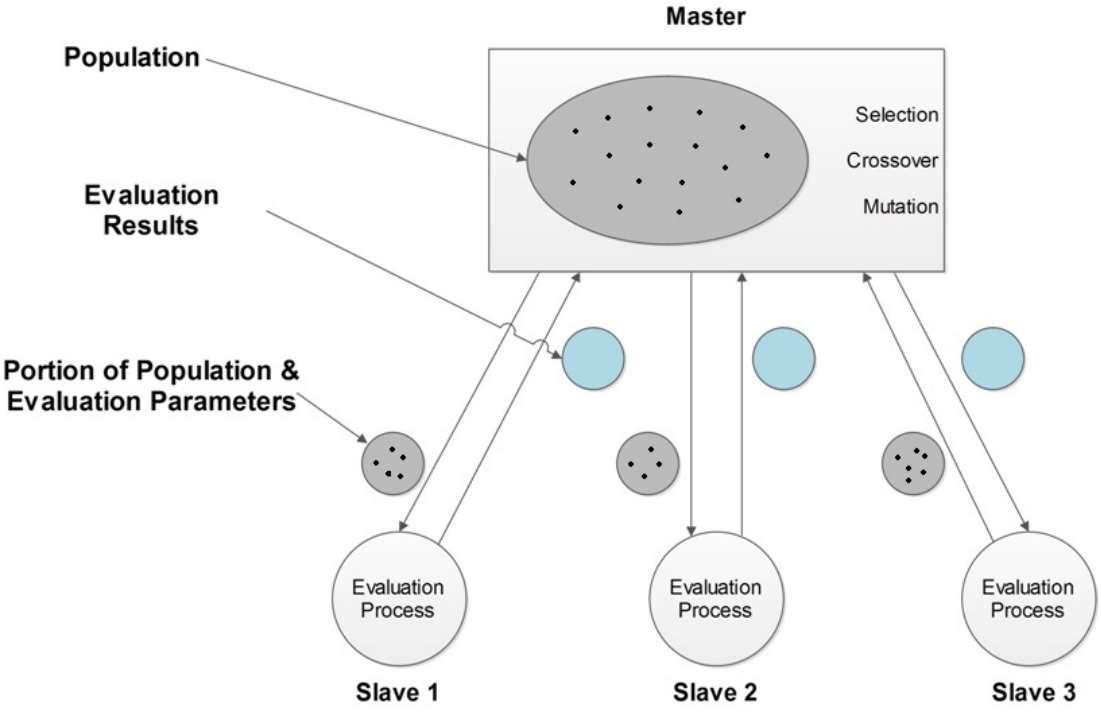


Figure 2.3: Master-slave Parallel Genetic Algorithm

The fine-grained PGAs, which are also called “cellular GAs” or “massively-parallel GAs”, use a single population as well. Nevertheless, the interaction between individuals is limited due to its spatial structure [22]. That is, the population is structured into multiple neighborhoods, where each individual is able to compete and mate with its direct neighbors. On the other hand, all these neighborhoods overlap, resulting in the diffusion of fit individuals over the entire population, which is the reason this model is also known as

diffusion PGA. Moreover, fine-grained PGAs were designed for massively parallel computer architectures, such as toroidal or 2-dimensional grids [5]. For this reason, nowadays the underlying hardware, used to support the operation of this model, is either graphical processing units (GPUs) or field programmable gate arrays (FPGAs). This way, each processing unit of this hardware is assigned with only one neighborhood, which is one or two individuals. The structure of this model is depicted in the figure below.

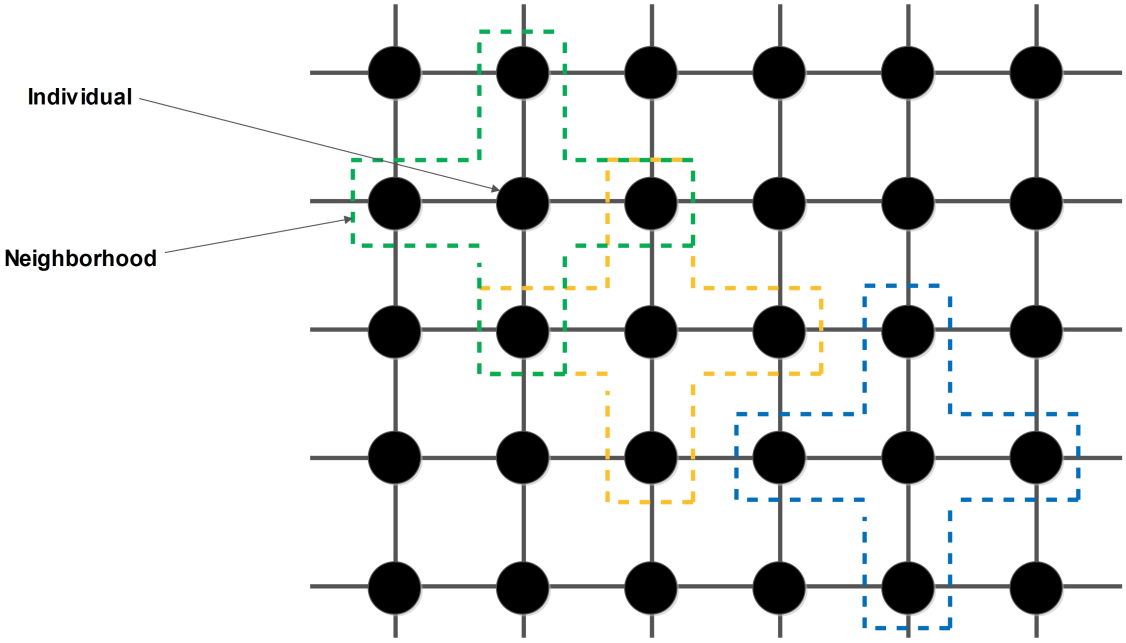


Figure 2.4: Fine-grained Parallel Genetic Algorithm

The coarse-grained PGAs, in contrast to the previous two models, use multiple populations in their structure. This type of PGA has many names in the literature, which are “distributed GA”, “multiple-population GA”, “multiple-deme GA” and “multi-deme GA”. As illustrated in figure 2.5, this model structures its whole population into a set of separate and independent subpopulations (or demes), of which in each one a simple GA is executed [5]. In other words, a coarse-grained PGA, using all the subpopulations available, executes simultaneously many Genetic Algorithms in order to explore efficiently the search space. Thus, the efficiency of the algorithm comes from the fact that multiple GAs are executed in parallel. This means that not only the time of execution is reduced, but also different regions of the search space are visited and explored, providing higher probability of finding a high-quality solution [24]. Moreover, coarse-grained PGAs introduce an additional phase of sparse communication between the demes, during which individuals migrate from one deme to another [22]. This exchange of individuals between the demes is the key characteristic of this model and is dictated by the migration policy, which is described in the next section. The purpose of this phase is to maintain and also introduce further diversity into each of the subpopulations of the algorithm, so that they can avoid getting stuck in a local optima. Furthermore, it is relatively easy to implement this model or convert a serial GA into a coarse-grained PGA. That is because only the migration process needs to be implemented into the program of the serial GA and the rest of its algorithm may remain the same [22]. It is also important to note that there are two

categories of multiple-deme GAs, depending on their migration topology: the Stepping-Stone and Island Models [25]. In Stepping-Stone models, the migration topology allows the individuals to migrate only to geographically nearby demes, while in Island Models the individuals are free to migrate to any other deme or follow some specific rules of any other migration topology. Finally, since the distributed nature of this project is based on the Island Model multiple-deme GA, the next section analyzes this model in more depth.

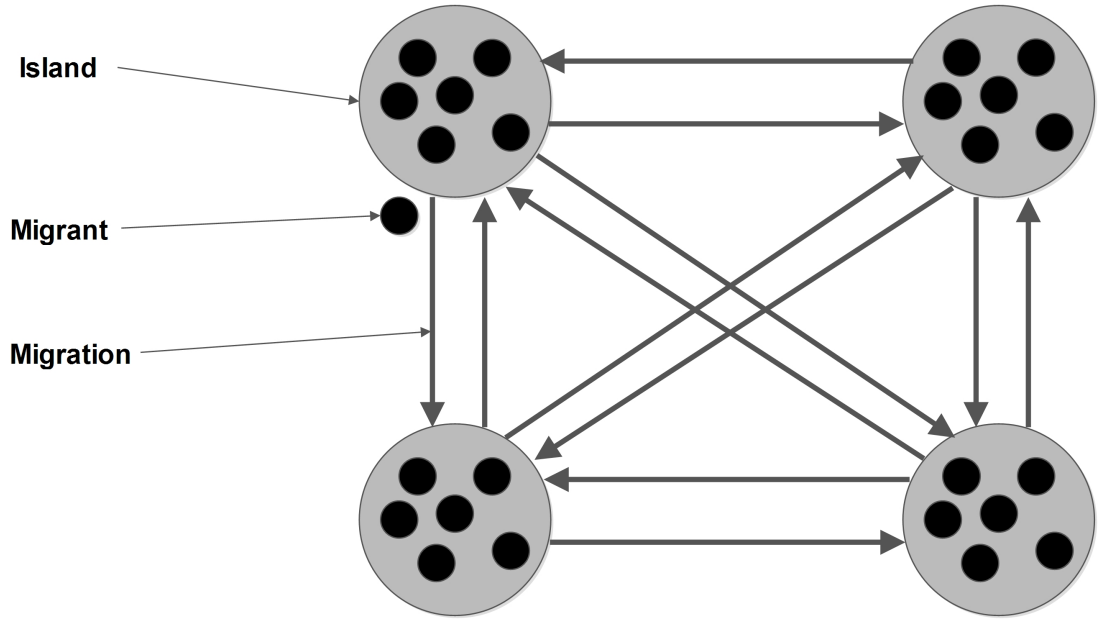
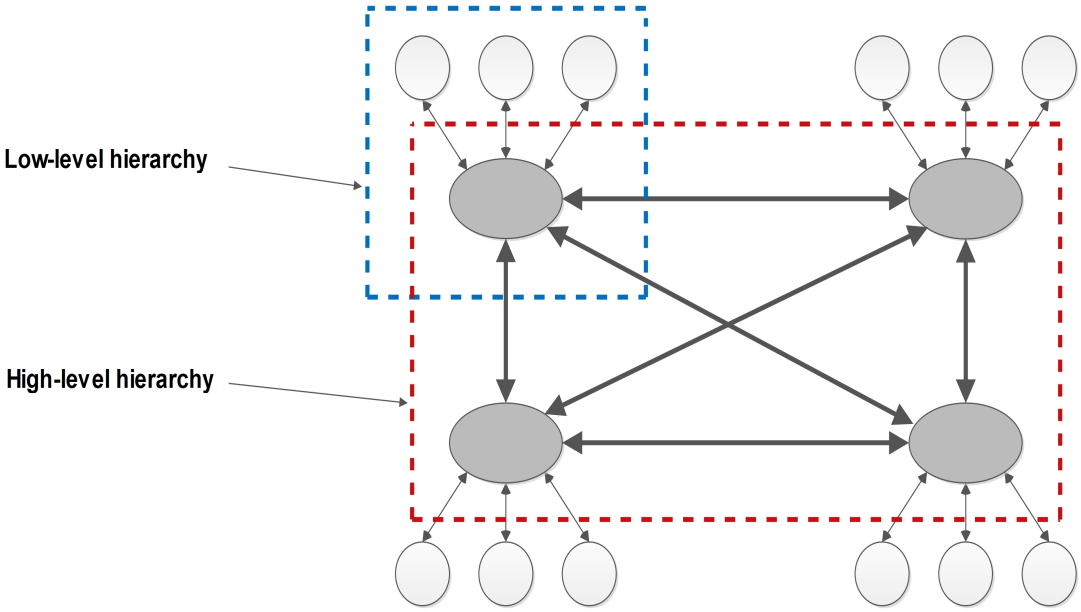


Figure 2.5: Coarse-grained Parallel Genetic Algorithm

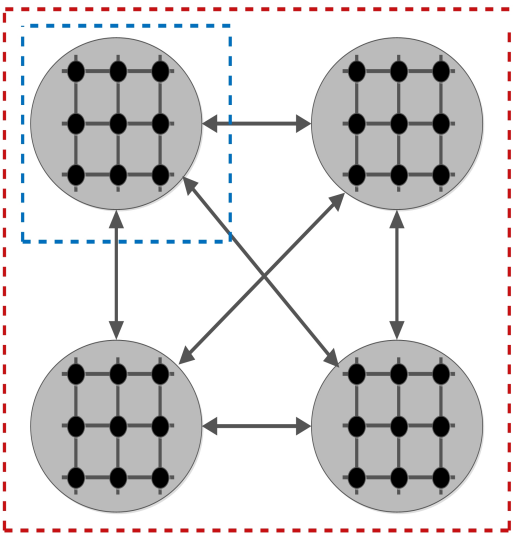
Similar to the coarse-grained model is the Independent Runs model, mentioned previously. More specifically, the structure of this model is exactly the same with that of a coarse-grained model, with the only difference the fact that there are no interactions between the subpopulations [5]. Therefore, the Independent Runs model has also the benefit of exploiting a pool of processors or any other parallel computer architecture. However, since there is no migration operation to improve its search quality, the main use case of this simple model is the rapid gathering of statistics of the problem at hand. This process is achieved by having in each run different initial conditions of the algorithm (population initialization). This attribute is very useful, as Genetic Algorithms are naturally stochastic and not deterministic. Nevertheless, it should be noted that, according to Cantu-Paz and Goldberg [26], the Independent Runs model is only efficient for short problems (not difficult) and also for small population sizes.

When some of these parallel models are combined, they form the hierarchical model, also known as “hybrid PGA”. That is, a hierarchical PGA is typically structured at two levels (hierarchies), in each one of which an ordinary PGA is used. The reason for this kind of structure is to combine and take advantage of the beneficial characteristics of the two simple parallel models, in order to optimize the quality of the search method [24]. As can be seen in figure 2.6, the upper level of this model is usually a coarse-grained PGA, most probably because of the available computer architecture that can support this structure.

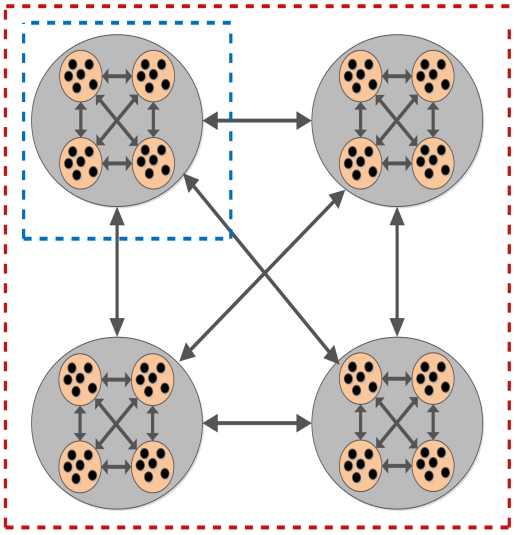
For instance, in figure 2.6a, the lower level uses the master-slave model, exploiting its parallelism and ability to handle computationally expensive evaluation functions. The upper level, on the other hand, uses the multiple-deme model to take advantage of its ability to retrieve different solutions from separate population evolutions, which explore different points of the search space. Therefore, the main advantage of this model is the fact that it is more efficient than any of its component PGAs alone [22]. However, as Luque and Alba state in their paper [5], hierarchical PGAs tend to add even more complexity to the already complex nature of the Parallel Genetic Algorithms, because additional parameters are needed for their implementation.



(a) Master-Slave model at the lower level and Coarse-grained model at the upper level.



(b) Fine-Grained model at the lower level and Coarse-Grained model at the upper level.



(c) Coarse-Grained model at both levels.

Figure 2.6: Hierarchical Parallel Genetic Algorithms

In conclusion, PGAs have all the advantages of sequential Genetic Algorithms, plus some of their own [24]. That is, the additional benefits that are introduced by the parallel models are their computation speed, ability to combine different parallel models and the fact that they tend to avoid suboptimal solutions. Overall, PGAs provide a superior numerical performance, even when executed on a single processor [5]. Nevertheless, as Alba and Troya point out in their paper [24], the drawback in using PGAs is the fact that they require complex design and a parallel computer architecture to run them. In any case, their advantages are more than appreciated by the research community and thus they are applied in a wide range of difficult problems. Overall, PGAs are successfully used in domains of scheduling, hardware design, Natural Language and Bioinformatics [5].

2.1.2.2 The Island Model Distributed Genetic Algorithm

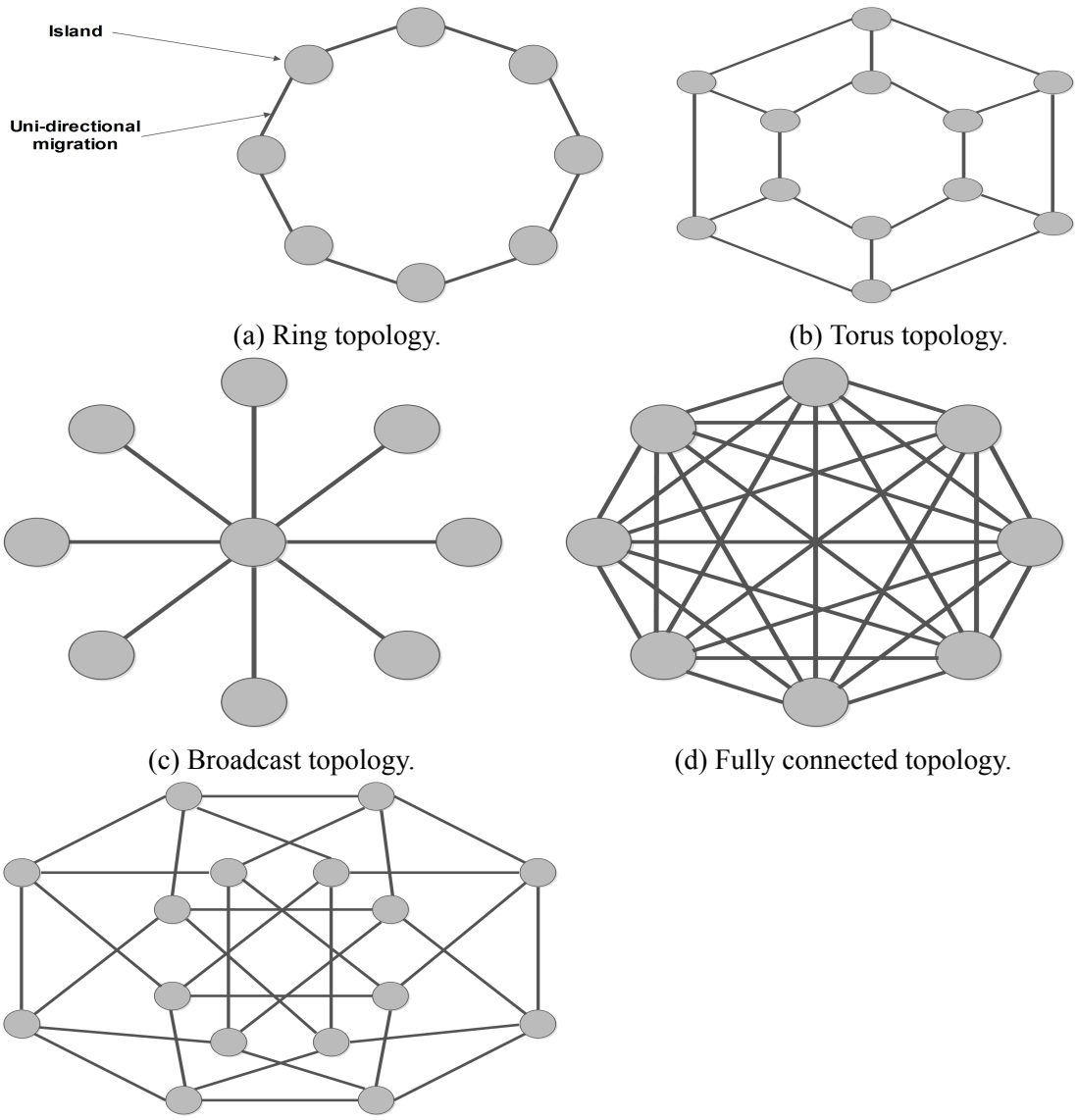
As mentioned in the previous section, the Island Model, also known as IMGA, is one of the two types of coarse-grained PGAs and its difference from the second is the way its demes interact with each other. Put differently, in Island Model PGA the demes are called "islands" and the interaction between them is free to be defined in whatever way by a migration policy, described later in the text. Furthermore, this model improves the performance of the algorithm by providing better balance between exploration and exploitation of the search space [27]. More specifically, the exploration is enhanced by the fact that multiple islands explore different parts of the solution space at the same time and also by the migration process. The exploitation, on the other hand, is enhanced by each island, that runs a separate GA and exploits the characteristics of promising individuals of the population, accommodated by that island. Therefore, combining these two attributes of the algorithm, the Island Model is able to deliver better solutions regarding their quality.

Migration policy is a set of rules, defined by the developer or user of the GA program, that guides the migration process of the model and also manipulates this balance between exploration and exploitation [28]. In other words, migration policy determines the communication phase between the islands, which enables it to bias the search process of the algorithm in any desirable way. Therefore, this policy, depending on its parameters, can also affect the convergence speed of the algorithm and thus reduce the computation time and effort [29]. However, the effects of the migration process is not fully comprehended yet, which is the reason the multiple-deme GAs in general are considered to be complex and difficult to understand [22].

In the literature of PGAs, six main parameters for migration policy can be found and they are the following:

- **Migration topology**: this parameter essentially determines the neighbors of an island. Meaning, for each island the migration topology decides to which destinations it can send information (migrants) to and which islands it can receive information from [30]. Also, depending on the topology and its proper usage, there are two possible effects on the performance of the algorithm, one being beneficial and the other

disadvantageous [27]. The beneficial one is the super-linear speed-up[1] of the algorithm due to information exchange and the other one is the BUS and CPU overhead due to required, but sometimes excessive, information flow. In any case though, this parameter, except for the convergence speed, it also affects the quality of the final results. The most popular and mostly used topologies are ring, torus, hypercube, broadcast and fully connected topology, which are illustrated in figure 2.7. What needs to be noted here is the fact that the sparser the topology is the slower the spread of information is [32]. In addition, it is a good practice to choose a migration topology according to the underlying native topology of the computer architecture, in order to fully take advantage of the hardware and thus maximize the efficiency of the algorithm. [33].



(a) Ring topology.

(b) Torus topology.

(c) Broadcast topology.

(d) Fully connected topology.

(e) Hypercube topology.

Figure 2.7: Most widely used migration topologies.

[1]The case where the speedup (increase in efficiency) is greater than the number of processors used, in terms of analogy [31].

- **Migration frequency**: this parameter, which is also known as "migration interval", determines the number of generations each island needs to perform to execute the migration process [33]. Put differently, each island undergoes migration operation every "x" number of generations. Important to note here is that it is a good practice to avoid using too high frequencies, since in that case the algorithm starts to lose its properties and starts to perform worse than a sequential GA [27].

- **Migration rate**: this parameter, also called "migration size", defines the quantity of individuals to be migrated from an island in a single migration process [33]. This quantity can be expressed either as a number or percentage of the island's population. That is, the migration rate is constrained by the size of the population. In regard to this matter, it is a good practice to use moderate values (below 25%) for this parameter, otherwise it will negatively affect the diversity of the populations [32].

- **Migration selection and replacement**: this parameter dictates the criteria upon which the migrants are selected in each island to be transferred to another island and also the individuals that will be replaced in the target island by those migrants [29]. That being said, there are four strategies for this parameter, each one of which is a combination of elitism and randomness. More specifically, the four possible selection and replacement strategies are the following:

  - Select the fittest migrants and replace the worst individuals in the target island.
  - Select the fittest migrants and replace random individuals in the target island.
  - Select random migrants and replace the worst individuals in the target island.
  - Select random migrants and replace random individuals in the target island.

  The choice of the strategy influences the search process of the algorithm, since it affects the balance between exploration and exploitation of the search space [33]. More particularly, when the strategy favors the elitism, as for example in the first strategy above, then it emphasizes the exploitation over the exploration, because promising individuals replace the "weak" individuals. This way, the more elitist the strategy is, the faster the convergence of the algorithm is [29]. So for instance, the strategy where random migrants replace random individuals in the target island, provides the slowest convergence.

- **Copy-based or Transfer-based migration**: this parameter defines which of the two types of migration will be used. That is, the migration in an Island Model is either copy-based or transfer-based [33]. The first one is the case where an individual after migration to a target island continues to exist in the source island where it came from. In other words, a copy of this individual were sent to the target island overwriting an individual that is selected to be replaced in that island. The second one is the case where a real exchange is taking place. This means that the migrant will cease to exist in the source island, so there will be no copies of that individual. Also, the migrant will not overwrite any other individual in the target island, instead it will occupy a place opened by an individual that will migrate from that island to another one. In this case it is worthy to note that the copy-based migration impacts

the diversity of the global population[2] by reducing it whenever a migration process occurs.

- **Migration synchronicity**: this parameter defines the migration synchronicity, which is either synchronous or asynchronous [27]. In case of synchronous migration, the exchange of migrants takes place at the same time for all islands, during which time all computations are blocked. In the other case of asynchronous migration, the exchange of individuals occurs whenever the source island is ready to do so, without considering the state of the target islands. In general, the models that use asynchronous migrations tend to be more efficient than their synchronous versions [33, 35].

Island Models, in general, can be either homogeneous or heterogeneous [30]. Homogeneous Island Model is the one that uses exactly the same GA with the same parameters (e.g. genetic operators) in each of its islands. On the other hand, heterogeneous Island Model differentiates its islands by using different parameters in each island, such as different genetic operators or different population size. The logic behind the heterogeneous models is to add even more diversity into the algorithm, in order to obtain better general structure of the final solution. Furthermore, this idea of heterogeneity was inspired by the "neutrality theory" [36]. Therefore, the homogeneous Island Models rely only on the stochastic way of diversification, whereas the heterogeneous ones, in addition to the stochastic approach, also use the differentiation of every island to add even more diversification.

[2]The sum of all the subpopulations that an Island Model GA has [34].

Chapter 3

Project Management

In this chapter the project vision is provided along with the algorithm, proposed for the core operation of the framework. Afterwards, the software development process model, that will be used to build the project, is thoroughly described. Also, a development plan, that this model will follow, is provided and analyzed. This way, reading this chapter, it becomes clear which steps were taken to build this project.

3.1 Project Vision

It is crucial to have an abstract idea about the nature of the project and its structure in order to successfully develop it. As such, a vision of the project needs to be defined, that will satisfy its aim and objectives, described in the "Introduction" chapter. Therefore, the vision of this project is to devise a novel framework for running Artificial Life simulations, which is able to operate on a distributed system. That is, this framework should essentially be a distributed application, which would allow the user to deploy it on any distributed environment with ease, without any configuration overhead. This way, the expected framework will be able to distribute the computational load of the simulation over the underlying distributed system, in order to enhance efficiency and operation smoothness of the simulation. This main idea of the project is depicted in figure 3.1.

With that being said, each computing machine of the distributed system will handle a portion of the computation needed for the framework. It will also be able to communicate information to the main framework operation or other computing machines. These machines can be any type of computing nodes, such as laptops or physical servers. Also, the software that will represent the main framework operation can either reside on another computing machine or the same one that runs a portion of its computational load.

Moreover, an important part of the vision is the incorporation of modern state of the art technologies, that will enable the framework to keep up with the latest advances in software development and also make it more easily deployable. Hence, the main idea

of the project also includes making the framework a modern distributed software stack, which will be able to scale to any extent. Apart from scalability though, by making the framework a distributed software stack, it will gain more beneficial characteristics. That is, it will also adopt many other virtues of cloud computing such as reliability, fault-tolerance, and portability among others.

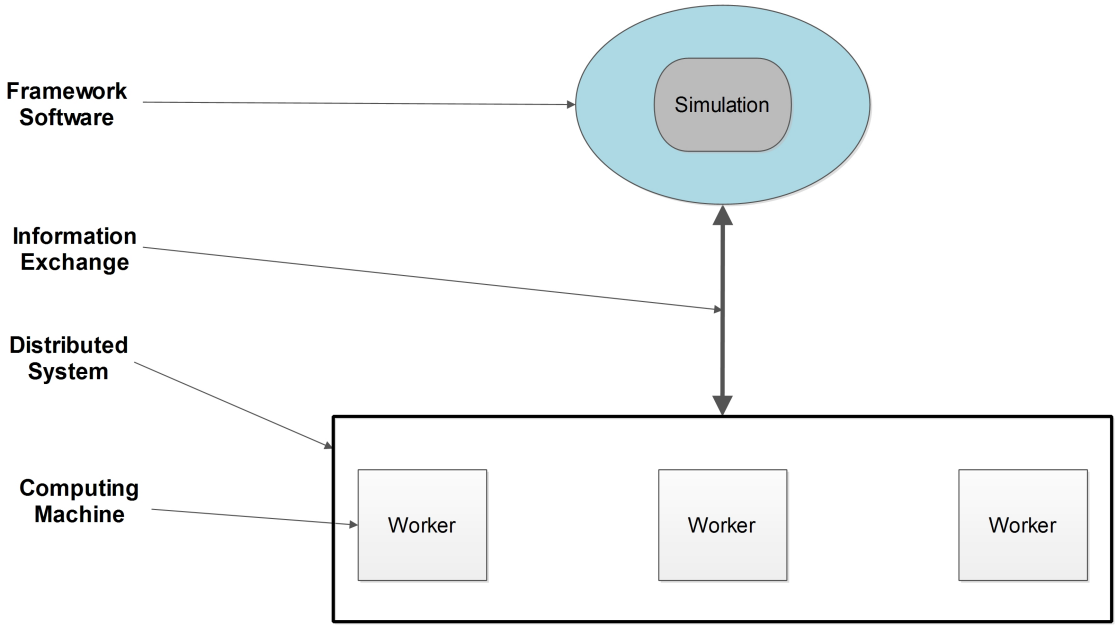


Figure 3.1: Abstract idea of the expected project.

3.1.1 Proposed Algorithm

In order to realize the vision previously described, a basic algorithm needs to be defined that will coordinate all the functionalities of the system. This algorithm will essentially serve as the core algorithm of the expected framework, which will dictate the way that the computational load will be distributed over the underlying distributed system. More particularly, it will be a Parallel Genetic Algorithm, since this type of algorithms are the ones that enable a population-based simulation to operate in a parallel fashion. However, there are many types of PGAs and each one of them applies different way of parallelization. Hence, an appropriate PGA needs to be selected that will undertake the role of the system's core algorithm and satisfy its objectives in the best possible way. As such, the algorithm that is proposed for this matter is the Island Model Distributed Genetic Algorithm, thoroughly described in the "Literature Review" chapter.

The reason for selecting this algorithm is mainly the fact that it is more simple than the other parallel models of GAs. This way, by avoiding the challenges that come with more complex PGAs, the risk of the project's development is mitigated. Also, another reason is that Island Model GAs are suitable for cloud computing [37]. To explain, the communication between islands in these algorithms is not continuous, but rather sparse.

Which means, this type of algorithms do not require fast underlying interconnection to support their communication process. In addition, the communication between the islands, apart from synchronous, it can also be asynchronous. Hence, since the communication part of these algorithms is not demanding, it is a good fit for the cloud environment where high-speed connection between computing resources is not always available. Finally, one more reason is the fact that, according to Gozali and Fujimura [38], Island Model GAs are generally suitable for complex problems, which can be easily scaled and which actually need to be scaled to be efficiently handled. This feature of Island Model GAs is suitable for this case because, since the project, as already stated, is envisioned to be a distributed software stack, it will be able to easily scale on any distributed environment, including cloud environment. Hence, in view of these reasons, the Island Model GA is considered the most appropriate algorithm for the current project.

3.2 Development process

In order to successfully realize the project vision, previously described, it is crucial to define the software development process model, according to which the software milestones of the project will be planned and executed. Software development model is a methodology for the development of a software project, which dictates the way that delivery cycles are handled [39]. As such, it is a very important part of the whole development process, as it affects the quality of the final software and its successful delivery on time.

With that being said, the development model that was chosen for the particular project is the evolutionary prototyping model. In this model a prototype of the expected software is firstly developed and is continuously improved until it satisfies the requirements given by the customer [40], as illustrated in figure . As such, the software in this case is developed in increments, so that frequent communication with the customer is feasible to solicit new requirements, according to which the prototype will be further refined and improved. The prototype, in this case, is the first incomplete version of the expected software system, which essentially consists of selected basic parts of it. Afterwards, the rest of the system's parts will be evolved from those basic parts, after every increment of the model using the customer's feedback. Furthermore, the selected basic parts, which make up the first prototype, are typically selected according to the degree of risk they pose in terms of software's risk management. Thus, high-risk areas of the software are selected to be prototyped in order to address the main risks early. This way, if these early risks are overcome, then the software can be evolved further with more confidence, otherwise no more effort will be spent on that prototype.

The reason this development process model was chosen is mainly the fact that this project is complex in terms of concepts involved and the combination of different state of the art technologies. The evolutionary prototyping model is used exactly for these types of software projects, wherein the degree of complexity is an issue [41]. Furthermore, this model is also used for projects wherein the requirements are not fully understood and may change in the future, which is also the case with the current project. That is,

since this project is complex, its requirements may be modified and adapted according to the customers feedback after every iteration. Therefore, the evolutionary prototyping development model seems valid for the nature of the current project and its requirements.

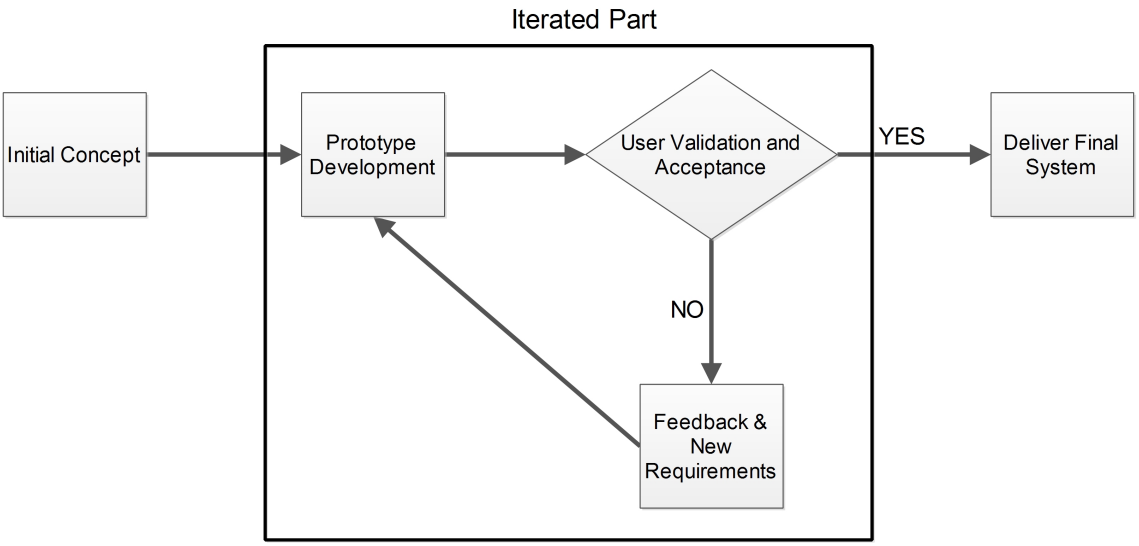


Figure 3.2: An abstract structure of the Evolutionary Prototyping model.

However, the "Prototype Development" part of the model (figure above) is not strictly defined in terms of the Software Development Life Cycle (SDLC) phases. To explain, that specific part is one of those that are continuously iterated and it may include any combination of different SDLC phases. For example, it may include phases of "Analysis", "Design", "Implementation" and "Testing". But, it may also include only the "Design" and "Implementation" phases and the other two defined only once before and after the parts that are iterated. Meaning, the "Analysis" phase may be defined once before the phases that are iterated and the the "Testing" phase after those phases. All these choices depend on the specific software to be developed and its requirements. In any case though, the iterated part of the model is not defined in the literature and that is why McConnell in his paper [40] refers to this part as "Refine prototype until acceptance". With that being said, this iterated part of the evolutionary prototyping model was adapted to the needs of this project and thus includes the "Analysis", "Design", "Implementation" and "Testing" SDLC phases. This part and the whole evolutionary prototyping model is depicted in detail in figure 3.3.

As for the validation and acceptance of each iteration, it will be accepted only if the aim of the project and also all of its objectives, which were defined in the "Introduction" chapter, are achieved. In addition, it is important to note that the "Testing" phase should not be confused with the "User Validation and Acceptance" phase. That is, the role of the "Testing" phase in this case is mainly to test the functionality of the software milestone, produced during that iteration, and also to find out if it can serve as the foundation for the next iterations. The role of the "User Validation and Acceptance" phase, on the other hand, is more general in nature. Meaning, during this phase it is decided if the software satisfies

the previously mentioned aim the objectives of the project. Thus, if the produced software is not validated, then another iteration is commenced and the new software produced from it is validated again. However, it is important to note that the role of this phase and also of the "Feedback and New Requirements" phase is mostly on theoretical level, since the number and the purpose of each iteration have already been determined. Which means, no real validation will take place at the end of each iteration. Put differently, these two phases practically have been carried out during the development plan, wherein these iterations were being defined. Finally, after all iterations have been completed, then "Evaluation" phase follows, during which a general discussion about the project's level of completeness and problems takes place.

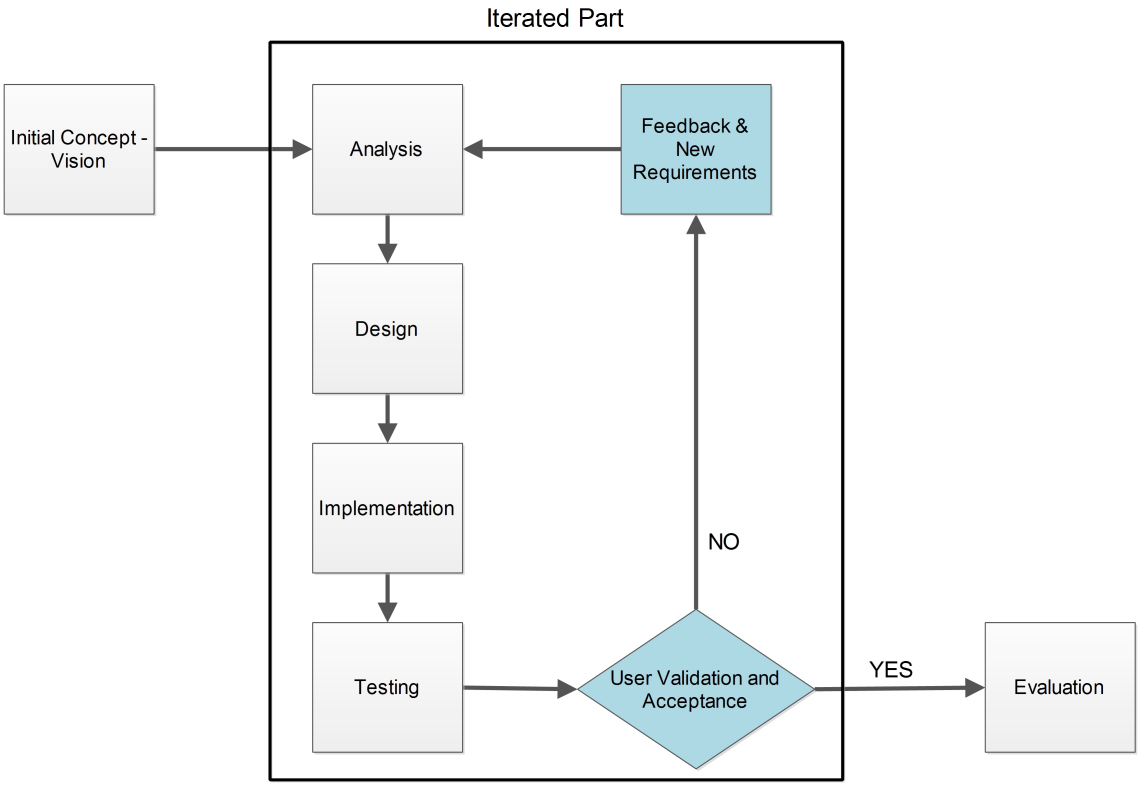


Figure 3.3: The detailed evolutionary prototyping model used for the project.

3.2.1 Development Plan

Since the development process model was defined, the next step was to plan the exact cyclical process of the development. More specifically, the iterations of the process needed to be abstractly planned, so that a basic idea of the development steps could be formed. Each of these steps is an iteration and it involves the development of a software's milestone. This way, after each iteration, the system will be improved incrementally and thus after building all the iterations, the software system will hopefully be evolved into the expected framework software. With that being said, the iterations that were planned for the development of the project are listed below:

1. **Iteration 1 (Theoretical model & first software component)**: in this stage an initial theoretical model is constructed to provide an abstract architecture of the system. Also, the first software component of the system is developed and tested in order to determine if it can serve as the first prototype that will form the basis for the next iterations.

2. **Iteration 2 (Interface class implementation)**: in this iteration the "ClientIsland" class is implemented and integrated into the simulation's software. This class serves as an interface between the user's simulation and the expected framework.

3. **Iteration 3 (Second software component implementation)**: in this stage another software component from the theoretical model, created in the first iteration, will be developed and tested.

4. **Iteration 4 (Final system delivery)**: in this final iteration the communication between all the software components of the system is implemented and tested.

Chapter 4

First Iteration

In this chapter the first iteration is presented and analyzed in detail, by separating it in sections, each one of which corresponds to a development phase. As such, all the development phases of the first iteration are thoroughly analyzed in order to describe the development of the first software component of the expected framework. Furthermore, in the last development phase, which is the testing part, an extended overview of all the software testing terms, used for the testing phase of the development, is provided. This way, the reader will be able to understand all the terms used in the testing phase and also the reason that certain software techniques and types were chosen.

4.1 Analysis

Since it is the first iteration, the main idea of the project needs to be analyzed in more detail, so as to define the problem more thoroughly. Thus, a theoretical model needs to be determined, which will act as a more detailed version of the abstract structure of the system, given in the "Project Vision" section. This model will serve as a starting point for all the iterations to be followed, since it will dictate the software components that need to be developed and also the technologies to be used.

Another problem that needs to be addressed in this iteration is the development of the first software component of the system. It was decided to first build a software component that will represent an island of the Island Model algorithm. The reason for this has to do with the fact that this part of the system is considered to pose the highest risk from all the other parts, since it is the main part of the Island Model GA. Meaning, all the other parts of the algorithm are related to the communication between the islands (migration process), which is only a matter of implementing the decided migration policy. Hence, by successfully developing this part of the algorithm, the biggest risk is mitigated. With that being said, the requirement for this software component is to run a simulation software automatically whenever it is created.

4.2 Design

4.2.1 Theoretical Model

The theoretical model that was designed in this iteration is depicted in figure 4.1. As can be seen from the figure, apart from the island components, the model also consists of some other software components. More specifically, these components are listed below:

- **Message Broker**: is the messaging middleware of the system, that undertakes the communication part of it, which is actually the migration process of the algorithm.
- **Server**: The server in this case is destined for two tasks, one of which is to take the role of the master node that will interact with the islands and the other is to serve a web page to the user.
- **Client**: is that web page, served by the server component, that essentially provides the Graphical User Interface (GUI) to the user.

Each one of these components will play a significant role in realizing this theoretical model. Furthermore, this model serves as the initial abstract architecture of the expected system and thus will form the basis for the next iterations, by dictating the software components that need to be developed.

Also, as mentioned before, the expected project is going to be a distributed software stack, that will be able to operate on any distributed environment. For that reason, the software system to be developed will use the Docker's container technology, since this technology is widely used nowadays to build software stacks. Furthermore, Docker containerization technology ensures the portability of the system. Put differently, every software component of the system is packaged into a Docker container and then all these containers are organized into a multi-container software system. This way, this software is enabled to run on any computer architecture, be it distributed or not. As such, each one of the software components, included in the theoretical model, will reside in a container. Additionally, the multi-container nature of the system boosts its scalability by making it very easy to scale any of its components. Meaning, in a traditional software system various network and software configurations would be required in order to scale any part of it.

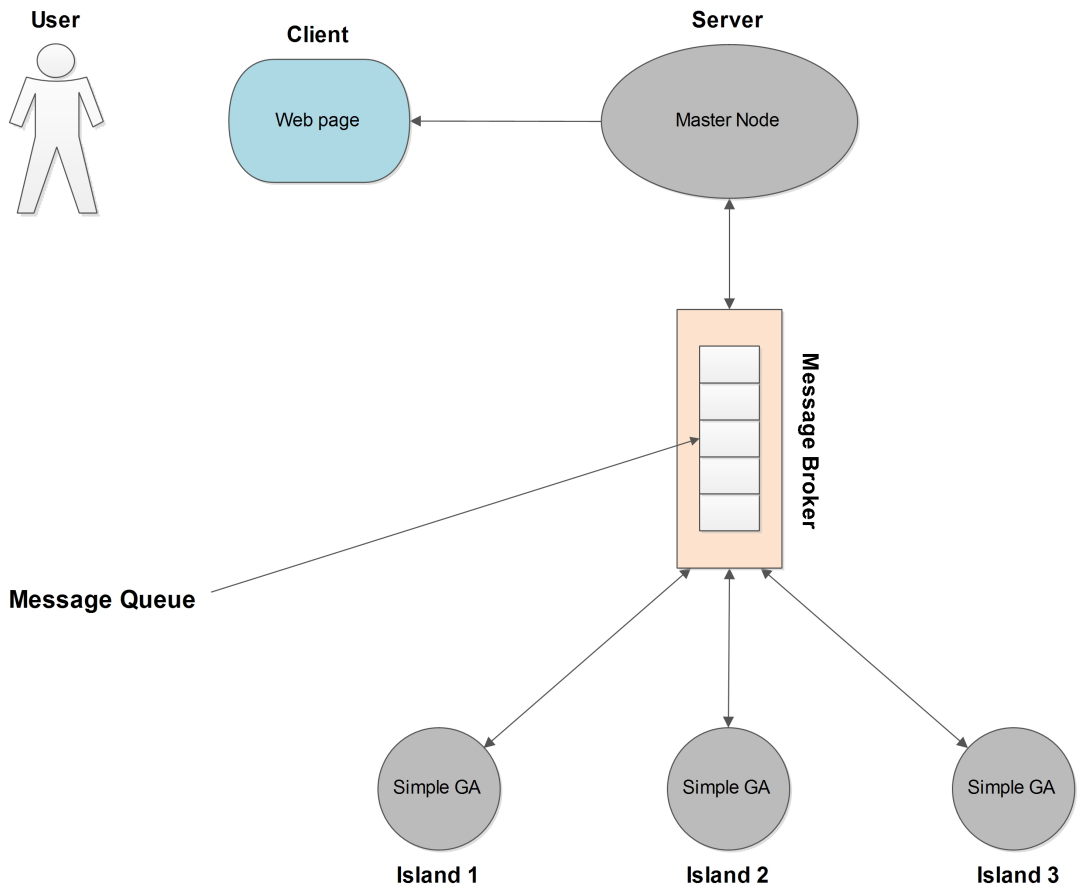


Figure 4.1: Theoretical Model of the framework.

The reason Docker container technology was chosen to facilitate the portability feature of the system is the fact that at the moment it is the most flexible and lightweight container-based virtualization technology in the market [42]. This way, the created software stack will be able to be easily deployed on any distributed environment. Also, another reason is the fast scalability that this technology provides. That is, Docker's ability to scale its software system outperforms that of the traditional virtualization technologies, since it can scale the amount of containers in a very fast way with a single Command Line Interface (CLI) command [43]. Finally, the last reason is the wide-spread use of this technology and its popularity in general [42]. This means that there is a mature documentation of Docker technology and a strong forum support that will mitigate the development difficulty and thus the risk that comes with it.

4.2.2 "Island" Software Component

Since Docker containerization technology is going to be used for the development of this project, the first step is to containerize the first to be developed part of it, which was decided in the Analysis phase. This part is the island component of the system, but to be more precise, it is actually the software simulation of the user. As such, the simulation that

the user wants to run on the expected framework, needs to be containerized first in order to be integrated to it and run in a distributed manner, as intended. However, since in this case there are no real users wishing to run their simulations on this framework, there was a need for an example simulation, which will also be useful for testing and evaluation purposes. Hence, for this matter it was decided to use the software simulation produced from the previous works on this project. That is, the simulation sketch, developed in Processing Java framework, will be used as an example simulation to test and evaluate the expected framework.

4.3 Implementation

To run the simulation sketch on the system, as previously said, the simulation needs to be containerized first. More specifically, it needs to be dockerized, since Docker containerization technology is used in this case. Hence, in order to dockerize the given simulation, a Dockerfile needs to be created first. To clarify, Dockerfile is a text file, which has instructions on how the Docker image should be built [44]. As such, when the Dockerfile is being built, the Docker engine (Docker daemon) reads the file from top to bottom and builds the Docker image accordingly. The Docker image is a file, comprised of a series of layers, which is used to create a container and execute code in it. Thus, to create a Docker container, the Docker engine first creates an image from the given Dockerfile and then creates the container from that image. All these tasks need to be instructed by the developer through the Docker CLI, in order to be executed by the Docker engine. With that being said, the Dockerfile that was used to create the Docker image for the software component that runs the simulation is the one depicted below.

```
FROM consol/ubuntu-xfce-vnc
MAINTAINER dyssos29@gmail.com

EXPOSE 5901
ENTRYPOINT ["/dockerstartup/vnc_startup.sh"]
CMD ["--wait"]
```

Listing 4.1: Dockerfile used for the creation of the container that runs the simulation.

After the creation of that image, the expected docker container, that will run the user's simulation, can be built from it (it will be called island container from now on). However, that is not enough, since that container also needs to automatically start running the simulation whenever it is created or started, as stated in the requirement of the "Analysis" phase. In other words, the simulation of the user needs to be automatically launched whenever the container, that it resides in, is newly created or started after stopping. This way, each time an island container is started or newly created, no intervention is needed from the side of the user, in order to start his simulation on that container. Instead, his simulation will be automatically started each of those times. But, in order to achieve that, the island container needs to be properly configured. That is, the shell script, which is used in that container to start services on boot, was modified to also start the simulation on boot. This shell script is written in the "`wm_startup.sh`" file, which resides in the container's file

system, and it is illustrated below.

```
#!/usr/bin/env bash
### every exit != 0 fails the script
set -e

echo -e "\n------------------ startup of Xfce4 window manager
    ------------------"

### disable screensaver and power management
xset -dpms &
xset s noblank &
xset s off &

/usr/bin/startxfce4 --replace > $HOME/wm.log &
/headless/Desktop/ExecutableSketch/SimpleOrganisms_Evolution &
sleep 1
cat $HOME/wm.log
```

Listing 4.2: Shell script that start all the needed services on the island container on boot.

As can be seen from the script, in line 13 the shell command for simulation launching is given. Without this line, the simulation would not run at its container's boot. The user would have to penetrate into the container's CLI and issue that command manually. However, with this configuration, the launching part of the user's simulation is automated.

4.4 Testing

4.4.1 Overview

In order to determine if the developed system (or a part of it) functions as expected, software testing needs to be conducted. In general terms, software testing is an activity to investigate whether a program behaves the way that it was designed to behave without any unpredictable actions [45]. In more technical terms, software testing is used for detecting missing requirements, errors and bugs in the source code of the program and any other defects it may have [46]. However, it should be noted that software testing is only able to show the presence of these defects and not their absence [47]. In other words, no matter how good or extensive the testing of a software is, it will not be able to prove the absence of the defects in that software.

With that being said, there are many different software testing techniques applied for different types of software at different development phases. However, in a very abstract view, they are divided into manual and automation testing [48]. The manual software testing, as its name suggests, is conducted in a manual way, having the developers to test the software by hand. This is more of an exploratory process and it involves writing test cases or detailed testing steps in order to verify the software's compliance with its requirements from the end user's point of view. The automation software testing, on the

other hand, is a process where the software is tested in an automatic fashion. Meaning, the software is tested by test scripts, written by developers or testers, and test results are generated automatically by software testing tools, such as Selenium.

However, in a more detailed view, software testing types can be categorized in levels according to the level of construction of the software under test. As such, as stated by Spillner et al in their paper [49], there are four levels of software testing:

- **Unit testing**: is the case where the smallest testable portion of the software system is being tested. It is also called "component testing", since the components of the software are usually its smallest testable portions.
- **Integration testing**: is when the interaction between those software components is tested after their separate unit testing. That is, the interface and also the data flow between these components are examined.
- **System testing**: is the testing level where the whole system is tested in regard to its behaviour. More specifically, the system is tested to verify its compliance to the requirements specification, use cases and other high-level descriptions. Moreover, on this level of testing the system's interaction with the underlying operating system and other resources is also checked.
- **Acceptance testing**: is the level of testing which is conducted after all of the rest testing levels, which are in the development process, have already taken place. As such, in this case the completed software system is not tested by its developers, but rather by the customer or intended users. This way, during acceptance testing it is decided if the software product can be released or not.

Furthermore, during all these levels different types of software testing can be conducted. That is, according to Graham et al [50], there are mainly five types of software testing, which are listed and described below:

- **Functional testing**: is the type of software testing where the main functionalities of the system are tested. Put differently, the software system is examined to find out if it satisfies its functional requirements. Additionally, functional testing is usually done in unit and integration testing, but in general it is done at all testing levels.
- **Non-functional testing**: is the type of software testing which is mainly related to the quality of the software. Which means, this type of testing evaluates the non-functional aspects of the software, that are not addressed by the functional testing. In this case, the software under test needs to be measured on a scale of measurement, instead of being characterized as good, bad or best, which are subjective and vague terms. This type of testing includes performance testing, load testing, reliability testing, portability testing and scalability testing among others. Furthermore, it can be conducted at all levels of testing.

- **Structural testing**: is the type of software testing where the structure of the software is tested. In case it is the software component under test, then its internal structural aspects are examined, though in case it is the software system, then its architecture is examined. In any case, during this type of testing, the software is examined by covering its structural elements, which is usually the source code. For this reason, structural software testing is often performed in the unit and integration levels, since there is a good tool support for code coverage.

- **Change-related testing**: is the type of testing that is related to the changes made to the software during its development. These changes might be fixes from the previous testing or new functionalities. With that being said, every time the software undergoes a change, it is important to make sure that this change not only fixed the previous problem, but it also did not cause any adverse side effects. As such, in order to investigate those two cases, two categories of change-related testing are used: confirmation testing for the first case and regression testing for the latter case. The confirmation testing validates that the last change applied to the software did fix the flaw that it was supposed to fix. The regression testing, on the other hand, is used to examine if any unexpected adverse side effects, mentioned before, are caused from this last change. In detail, during regression testing all the previous tests are ran again in order to make sure that the software still meets the requirements (like it did till that time) and it did not regressed to a state where it has more defects than before.

- **Maintenance testing**: is the type of testing which is performed in the phase where the software is fully functional and already deployed in production. As such, this type of testing occurs whenever the software needs to be corrected, changed or extended. It is important to note here that maintenance testing should not be confused with the maintainability testing, which examines how easily maintainable is the system. Moreover, during maintenance testing the same testing process steps are applied, meaning the software may be tested at any level (unit, integration etc.) depending on its size and risk. However, there are several areas of this type of testing that are very different from all the other types.

All these testing types though are planned and conducted according to an approach or technique. With that being said, there are two main techniques that dictate the way that a software test is performed: the black-box testing and white-box testing [51].

The black-box software testing technique, as its name suggests, treats the software as a “black box”, because there is no access to the source code. To elaborate more, the tester is only aware about the fundamental aspects of the software and not its internal logical structure [52]. As such, it can be said that during this testing technique the behavior of the software is tested from the user’s point of view. This way, the tester can verify if the software operates in an expected way or it exhibits some unexpected behavior. Put differently, during this type of testing, the software’s functionality is validated in terms of the requirements specifications and not its source code. Furthermore, this technique is usually applied to the functional type of testing, but it can be applied to other types as well (apart from structural testing) at any level of testing. Finally, there is not only one

black-box technique, instead there are several of them, some of which are equivalence partitioning, boundary value analysis and state transition testing.

The white-box testing technique, unlike the former one, is mainly concerned with the internal logic and source code of the software under test [53]. Which means that internal structure and source code need to be accessible in this case, which is the reason it is also known as transparent-box, open-box and clear-box testing. As such, this technique is performed from the developers point of view to find logical and implementation errors in the software through some coverage criteria, the most crucial of which is the decision coverage. Moreover, white-box technique is usually utilized in the structural type of software testing, but it may be used in other types as well. As for the levels of testing, it is typically applied in the unit, integration and system levels. Finally, this technique is also categorized into several types, some of which are the desk checking, basic path testing and data flow testing.

However, apart from all these types of software testing, there also needs to be a test strategy at place. Test strategy is essentially a plan that dictates the approach to testing software. Put differently, it describes the exact way that the software will be tested, including the type, technique and level of software testing, test objectives and many more [54]. Also, it is important to point out the difference between test plan and test strategy, since many developers confuse these two terms. Hence, test plan describes an abstract vision on what needs to be done during the testing process, whereas test strategy describes the way that vision will be achieved. As such, test strategy is usually a subsection of a test plan, but it also can be written separately in another document. With that being said, test strategy document typically comprises the following information:

- **Scope and overview**: defines who will review and who will approve this document. Also, it describes the testing activities along with their timelines.
- **Test approach**: describes many different types of information related to the way that the software will be tested, among which are process of testing, testing levels, types of testing and testing techniques.
- **Test environment**: describes the required configuration or setup for the testing environment, such as operating system, and also the test data backup and restore strategy.
- **Tools for testing**: describes test automation and test management tools, which might be open source or commercial.
- **Risk analysis**: describes all the estimated risks and a plan to mitigate them.
- **Review and approval**: describes the review and approval of all the test activities. This is done by the business, project management and development team.

This document is considered very important as far as Quality Assurance[1] (QA) activities are concerned, as it guides the development team on how to minimize risk and

[1] All the activities which verify that a software's quality requirements are fulfilled [49].

improve the quality of the software through specific steps.

4.4.2 Test Strategy

The size and content of the testing strategy that was written and used for this iteration is adapted to the needs of the software created so far. Meaning, it does not have all the information that a traditional test strategy have. Thus, the topics of "tools for testing", "risk analysis" and "review and approval" are omitted in this test strategy. With that being said, the test strategy that was followed in this iteration is the following:

- **Scope and overview**: The software under test in this case will be the island container. More specifically, the first testing activity will be to test the Dockerfile to find out if a Docker image can be created from it. Then, the second testing activity will be to test the created image to see if the expected island container can be created from it. Finally, the third is to check if the simulation starts to operate automatically on boot, since it is the main requirement defined in the "Analysis" phase.
- **Test approach**:
  - Manual software testing was chosen to be used, since the technologies, used for the realization of the project, are relatively new. Which means, there are no standard testing scripts or testing tools to perform the software testing in an automatic way. As such, the cost of writing testing scripts or searching an appropriate testing tool would be too high in terms of time. Thus, in order to lower this cost and hence mitigate the development risk, the software will be tested manually by using test cases. As a matter of fact, manual testing will be used in all of the iterations of this project, since there will be no significant change in the nature of the software.
  - Furthermore, for this iteration black-box testing technique will be used, since no code was written in this phase. As such, the behavior of the software will be examined through a test case.
  - Also, functional type of testing will be used in this iteration, because change-related, maintenance and structural testing are not applicable. The reason they are not applicable for this case is because it is the first iteration and thus there are no previous versions of the software in order to perform change-related or maintenance testing. Moreover, as mentioned before, no code was written, so that is why structural testing is not applicable as well. As for the non-functional testing, it is not used in this iteration because the main goal in this phase is to test the basic functionalities of the system and not its performance characteristics. For that matter, neither the rest of the iterations utilize this type of testing for the same reason.
  - Finally, since the software under test is the first software component developed for the system, the level of the software testing will be unit level.

- **Test environment**: The docker container is going to be tested on Linux Ubuntu 18.04.3 LTS operating system, using the command line interface.

4.4.3 First Testing Activity

The first testing activity, as mentioned before, is to examine the Dockerfile created in the "Implementation" phase. In order to do that, Docker CLI needed to be used to issue the appropriate command for building the image. Hence, the test case created and used for this purpose is illustrated in the table 4.1. In that table, it can be seen that the test case was successful and thus the image was created successfully from the given Dockerfile. Furthermore, the Docker CLI command that is mentioned in the table is the following: "docker build -t dyssos29/ubuntu-processing .". The "-t" flag is used to tag the image to be created by a name and the full stop at the end indicates the path, which is the current path. This is how the Docker engine finds the Dockerfile, through the path indicated at the end of the command. That is the reason in the "Preconditions" column it is stated that the Dockerfile should be in place, meaning it should be exactly at the path, indicated at the end of the command. However, indicating the Dockerfile's path is not enough, since a Docker engine is also needed to actually do the underlying tasks of finding the Dockerfile and creating the image. That is why Docker engine is another precondition stated in the same column.

Testing Activity 1 - Docker Image Creation					
Test Case	**Preconditions**	**Test Steps**	**Expected Results**	**Actual Results**	**Pass/Fail**
Issue Docker CLI command to build the image and check response.	Docker engine installed and Dockerfile in place.	1 - Run the Docker command.	Newly created Docker image.	Newly created Docker image	Pass

Table 4.1: Test case for investigating the creation of the Docker Image.

The exact procedure of the testing activity is illustrated in figure 4.2, where the command for image creation is issued. As can be noticed from the figure, after issuing the command, the process steps defined in the Dockerfile are executed, meaning it will take a while for the Docker engine to build the image. Also, from the figure it can be verified that the Dockerfile is in the current path and thus the command execution will not have problems related to its path.

However, it was also necessary to check if the image was indeed created, in order to fill in the "Actual Results" column of the previous table. As such, a new Docker CLI needed to be issued in order to list the images that exist in the current Docker engine. As can be seen in the same figure , the result of that command, which is "docker images", shows one Docker image, which is the intended one. Thus, the creation of the expected Docker image was successful.

```
root@ody-Lenovo-B50-80:~/Desktop/Project/Software/IslandProgram# ls
Dockerfile  SimpleOrganisms_Evolution
root@ody-Lenovo-B50-80:~/Desktop/Project/Software/IslandProgram# docker build -t dyssos29/island-image .
Sending build context to Docker daemon  258.6MB
Step 1/5 : FROM consol/ubuntu-xfce-vnc
 ---> a86ae35bf09a
Step 2/5 : MAINTAINER dyssos29@gmail.com
 ---> Running in 3a44f3a03fb0
Removing intermediate container 3a44f3a03fb0
 ---> 184ca1926473
Step 3/5 : EXPOSE 5901
 ---> Running in 3f5b2b5595e0
Removing intermediate container 3f5b2b5595e0
 ---> f5e78c1e241a
Step 4/5 : ENTRYPOINT ["/dockerstartup/vnc_startup.sh"]
 ---> Running in e2c409c28ec8
Removing intermediate container e2c409c28ec8
 ---> 3cc7f8a96fc1
Step 5/5 : CMD ["--wait"]
 ---> Running in 36648542c038
Removing intermediate container 36648542c038
 ---> bf026334ab64
Successfully built bf026334ab64
Successfully tagged dyssos29/island-image:latest
root@ody-Lenovo-B50-80:~/Desktop/Project/Software/IslandProgram# docker images
REPOSITORY                          TAG                 IMAGE ID            CREATED             SIZE
dyssos29/island-image               latest              bf026334ab64        2 minutes ago       1.18GB
```

Figure 4.2: Docker-CLI command to create the image.

4.4.4 Second Testing Activity

The second testing activity, as it is already known, is to investigate the created image to find out if the expected container can be produced from it. As such, the test case for the image examination is shown in table 4.2. This test case is also successful, as can be seen from that table, which means that the container was created with any problems.

Testing Activity 2 - Docker Container Creation					
Test Case	**Preconditions**	**Test Steps**	**Expected Results**	**Actual Results**	**Pass/Fail**
Issue Docker CLI command to create the container and check response.	Docker engine installed and Docker image in place.	1 - Run the Docker command.	Newly created Docker container.	Newly created Docker container	Pass

Table 4.2: Test case for investigating the creation of the Docker Container.

Furthermore, for this investigation Docker CLI command, once again, was needed. Thus, to create that container the command that was used is “docker run -d –name island-container2 -p 5904:5901 dyssos29/island-image”, as illustrated in figure . In that figure, it can be seen that the command “docker images” confirms that the needed image is in place, in order to validate the defined precondition in the table of this test case.

To elaborate more, from the “run” word in the command it is already obvious that as soon as the container is created, it will be up and running. Also, the “-d” flag stands for “detached”, which means that the container will run in separate thread and will not affect

```
root@ody-Lenovo-B50-80: ~
File  Edit  View  Search  Terminal  Help
root@ody-Lenovo-B50-80:~# docker images
REPOSITORY                      TAG                 IMAGE ID            CREATED             SIZE
dyssos29/island-image           latest              bf026334ab64        5 hours ago         1.18GB
dyssos29/ubuntu-processing      latest              82828a8ea8b6        22 hours ago        1.51GB
ubuntu                          16.04               c6a43cd4801e        40 hours ago        123MB
dyssos29/express-server         6                   4269ad9607e3        3 months ago        124MB
dyssos29/express-server         5                   abdd33bf6b49        3 months ago        124MB
dyssos29/express-server         4                   0a0e7ce3a3ba        3 months ago        124MB
express-server                  3                   7acaa0985681        3 months ago        124MB
<none>                          <none>              0d31eb310878        3 months ago        124MB
express-server                  2                   d75c163b3ab6        3 months ago        124MB
dyssos29/ubuntu-processing      9                   7dee665d7b69        3 months ago        5.06GB
timbru31/node-alpine-git        latest              3280efbb041c        6 months ago        97.1MB
dyssos29/ubuntu-processing      8                   2cbfec23643c        7 months ago        4.24GB
dyssos29/ubuntu-processing      7                   abedb2d3d1b2        7 months ago        4.24GB
dyssos29/ubuntu-processing      6                   828de138760b        7 months ago        4.24GB
dyssos29/ubuntu-processing      5                   1e3c6e12e10c        7 months ago        3.74GB
dyssos29/ubuntu-processing      4                   f60e5f823b95        7 months ago        3.5GB
netresearch/node-webserver      latest              57a4c0886cc4        7 months ago        79.6MB
rabbitmq                        3-management        ac759a4f2d38        7 months ago        226MB
dyssos29/ubuntu-processing      <none>              4fe9deae3da8        8 months ago        1.39GB
hello-world                     latest              fce289e99eb9        11 months ago       1.84kB
consol/ubuntu-xfce-vnc          latest              a86ae35bf09a        13 months ago       1.18GB
dockersamples/visualizer        <none>              8dbf7c60cf88        2 years ago         148MB
smebberson/alpine-nginx-nodejs  latest              1050c092d580        3 years ago         110MB
centurylink/dockerfile-from-image latest            970eaf375dfd        4 years ago         19.2MB
nacyot/vnc                      latest              f27917b61bb5        5 years ago         1.02GB
root@ody-Lenovo-B50-80:~# docker run -d --name island-container2 -p 5904:5901 dyssos29/island-image
f00d06879db689ad4c5cd8375b87b6ac6b9e21ea9e3b120768b3b6e0e2178a6
root@ody-Lenovo-B50-80:~# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS                          NAMES
f00d06879db6        dyssos29/island-image   "/dockerstartup/vnc_..."   6 seconds ago       Up 4 seconds        6901/tcp, 0.0.0.0:5904->5901/tcp   island-container2
root@ody-Lenovo-B50-80:~#
```

Figure 4.3: Docker-CLI command to create the container.

the current unix shell. Moreover, the name “island-container2” is given to the container through the flag “–name” and also port binding has been defined through the “-p” flag. The port binding allows to bind any port from the side of the host to the port that exists inside the container. So, in this case the port “5904” of the laptop was bound to the port “5901” of the container. Lastly, at the end of the command, the name of the image was given in order to define the image from which the container will be created.

In order to check the results of this command and then compare them with the expected results defined in the table, the “docker ps” command was issued to list all the running containers. The reason for choosing this command is the fact that, as stated before, the island will run automatically right after its creation and thus will be in the list of all running containers. Hence, from the figure above, it can be seen that the expected island container is up and running. Consequently, the container was successfully created from the given image.

4.4.5 Third Testing Activity

In this testing activity the goal is to find out if the example simulation starts automatically on boot, after making the needed configurations. Hence, the test case created and used for this goal is shown in table 4.3. This test case, as may be seen, is also successful, meaning the simulation runs automatically each time the island container is created.

This time Virtual Network Computing (VNC) technology was used to perform this testing activity. Particularly, TigerVNC tool, which is an implementation of VNC technology, was utilized to connect to the container’s windowing system. Therefore, as depicted in figure 4.4, the TigerVNC command “vncviewer localhost:5901” was issued to connect to the running island container. After issuing that command, as shown in the same figure, VNC authentication was required in order to connect to container’s windowing system.

Testing Activity 3 - Automatic Simulation Operation					
Test Case	**Preconditions**	**Test Steps**	**Expected Results**	**Actual Results**	**Pass/Fail**
Connect to the island container and check if the simulation is already running.	Island container up and running	1 - Run the VNC command to connect to the container. 2 - Insert the correct password.	Simulation up and running	Simulation up and running	Pass

Table 4.3: Test case for investigating the automatic start of the simulation on the container's boot.

That shows that the system also incorporates some security measures. Finally, the command "docker ps" was issued, once again, in the beginning of the testing activity to ensure that the island container is up and running and thus fulfill the precondition of this test case.

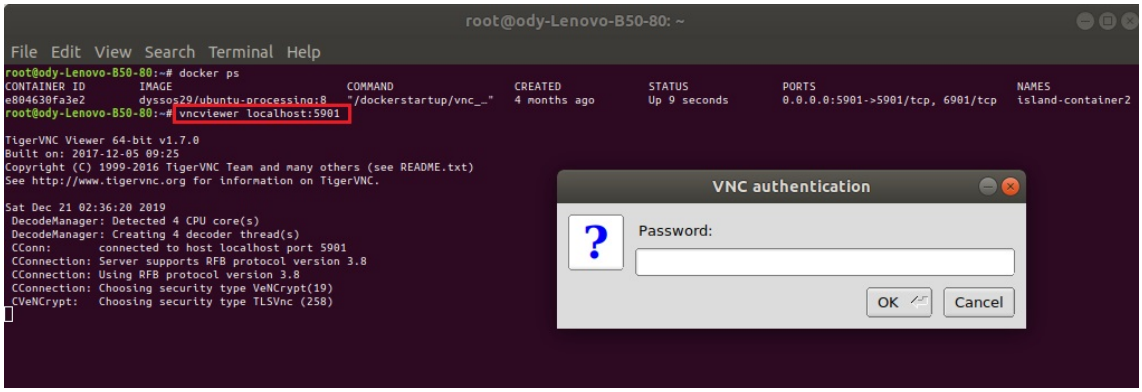


Figure 4.4: TigerVNC command to connect to the windowing system of the island container.

After putting the password and successfully connecting to the island container, the simulation was found already up and running, as expected. This is illustrated in figure 4.5, where one can also notice the desktop behind the simulation which is what the windowing system, installed in the container, provides.

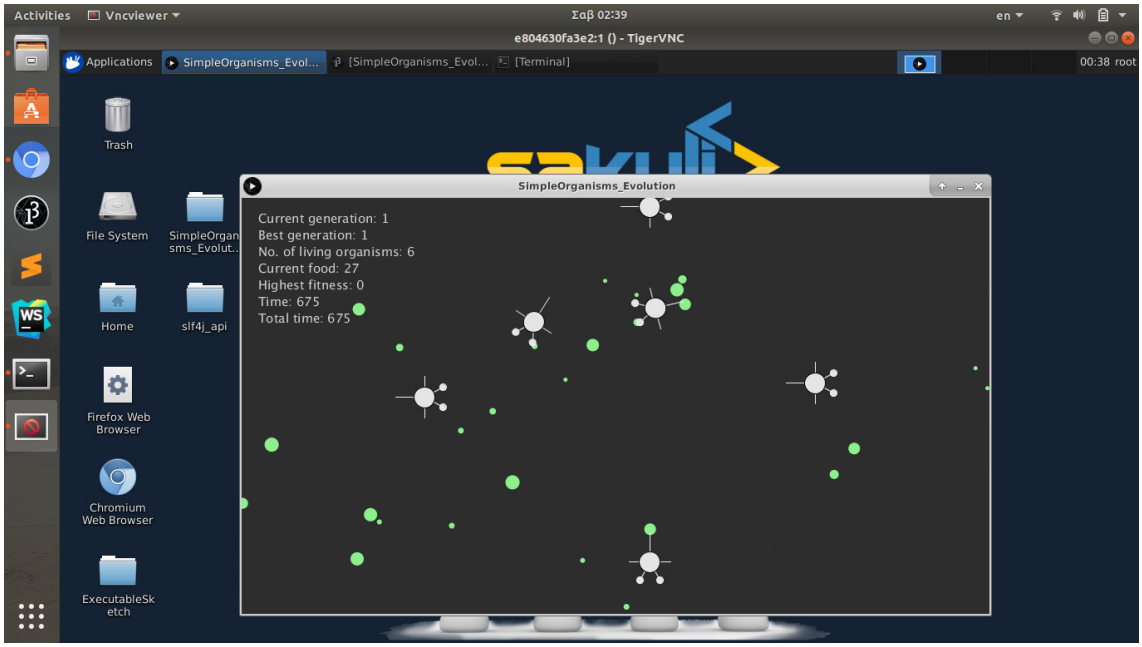


Figure 4.5: The example simulation already running in the island container when VNC connection established.

Chapter 5

Second Iteration

In this chapter the second iteration is being presented by providing thoroughly analyzed development phases that are described in different sections. This iteration is concerned with the development of an interface class that connects the simulation with the expected framework.

5.1 Analysis

In this iteration, the problem that needs to be addressed is the development of the interface class, which will act as the connection between the framework and the user's simulation. This interface class will be named "ClientIsland" and it will be part of the simulation software. At this point it should be clarified that for now the "ClientIsland" class is not intended to be an interface class strictly in terms of Object Oriented Programming (OOP) paradigm. That is to say, it will not be a class which only describes the actions that the objects (of the classes that implement that interface class) can do, through the use of static constants and abstract methods. Instead, it will be an ordinary class, that will simply play the role of a connection class. However, the terms "interface class" and "connection class" will be used interchangeably in the text and indicate the same thing, which is the ordinary class of "ClientIsland". The reasons for using an ordinary class, instead of a real interface class, are explained in the "Evaluation" chapter.

Furthermore, at this stage a message broker, defined in the theoretical model of the previous chapter, will be chosen to be incorporated in the software system. The main reason for incorporating this software component in this iteration and not later, is the fact that it will be used to test the "ClientIsland" interface class. In other words, the interface class will be tested in terms of its communication with the message broker. As such, the requirement for the "ClientIsland" interface class is for it to be able to communicate data to the message broker in a robust way.

5.2 Design

Before designing the way that the class, mentioned above, will be developed, it was crucial to decide the exact message broker to be used for this project. Hence, one of the most basic criteria for choosing the right message broker service was the popularity and industry acceptance of that service. The reason for this is the fact that a message broker that is popular, from the perspective of industry adoption, obviously has a strong community and a clear documentation. Those two attributes play a very important role in the development of this project, since they will help to mitigate any development risk that is related to that message broker. As such, currently there are two message brokers that stand out in the market: RabbitMQ and Kafka [55]. Therefore, it came down to choosing one of these two message brokers.

First of all, it is important to point out that both RabbitMQ and Kafka are very successful and robust middleware services in distributed systems. However, each one of them has different features, since they have been designed for different purposes [56]. For this reason, when it comes to choosing one of the two, it is necessary to take into consideration the exact use cases. In other words, it is important to know what purpose the message broker will be used for and if its features and unique qualities are suitable for that purpose. As such, RabbitMQ was chosen to be used in this case because the features and capabilities of this tool are more suitable for the needs of this project than those of Kafka.

To explain, there are three main reasons that RabbitMQ was preferred over Kafka and they are listed below:

- RabbitMQ is multi-protocol, which means it can support various protocols, instead of only one [56]. That is, apart from the AMQP, which is the main protocol used in RabbiMQ, it can also use other protocols such as STOMP and MQTT. Kafka, on the other hand, is able to support only one protocol, which is the binary protocol over TCP [57]. Therefore, since using a tool which supports more than one protocol reduces the risk of protocol incompatibility with other services, for this reason it was decided to use RabbitMQ. In other words, by choosing a multi-protocol tool for the project, the development risk is minimized.
- RabbitMQ server provides a user-friendly and helpful GUI (web page), which enables the user to monitor the data exchange in real-time and also to control various aspects of the message broker, such as queues and exchanges among others [56]. On the other hand, Kafka needs to use a separate tool for that reason, which is the Kafka Tool. Incorporating this tool though, adds up the overall configuration overhead. As such, for monitoring and testing purposes, the easy to use GUI of the RabbitMQ server was more appealing.
- RabbitMQ is more suited for RPC (Remote Procedure Call) style of communication, due to its features of ”correlation ID” and ”reply-to”. This fact is very important for the current project, because all the internal communications of the system will be based on the RPC communication paradigm, as explained in the last iteration.

Now that the message broker of the system is defined, the “ClientIsland” connection class can be implemented. As mentioned before, it will be part of the simulation software and its main responsibility will be to communicate with the RabbitMQ server. This idea is illustrated in figure 5.1, where, till now, that class is shown to send data to the RabbitMQ server. In the next iterations though, the “ClientIsland” class will also receive data from the message broker.

Furthermore, it can also be noticed from the same figure that the RabbitMQ server resides in a Docker container as well. That RabbitMQ docker container was created from the official RabbitMQ Docker image, which can be found in the Docker registry (Docker Hub). In other words, that image was already developed and ready to be used in production environment. This way, no time was spent for the dockerization process of the RabbitMQ message broker.

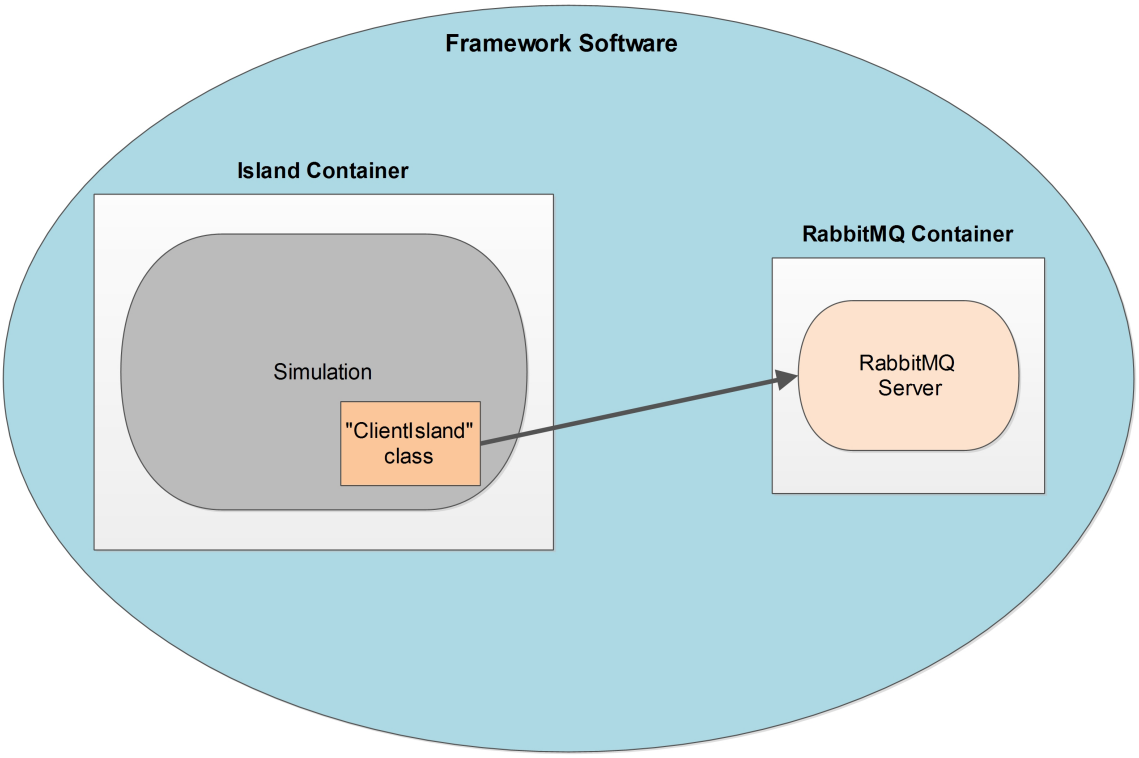


Figure 5.1: Software architecture developed so far.

5.3 Implementation

The “CliendIsland” class was implemented using Java programming language, since it will be incorporated in a simulation that was developed using the Processing framework, which is a Java framework. Furthermore, since it will be interacting with the RabbitMQ broker, it will need a Java client library of this message broker. With that being said, after including all the needed libraries in the class, the next step was to implement the constructor, which is shown in the code snippet below. As can be noticed, the role of that

constructor is to establish the connection with the RabbitMQ server, which is necessary in order to interact with it.

```
public ClientIsland()
{
  factory = new ConnectionFactory();
  factory.setHost("ody-rabbit");
  try
  {
    connection = factory.newConnection();
  }
  catch (IOException e)
  {
    System.out.println("Error in creating connection: " + e.
  getMessage());
  }
  catch (TimeoutException t)
  {
    System.out.println("Error in creating connection: " + t.
  getMessage());
  }

  try
  {
    channel = connection.createChannel();
    channel.queueDeclare(QUEUE_NAME, false, false, false, null);
  }
  catch (IOException e)
  {
    System.out.println("Error in creating channel: " + e.getMessage()
  );
  }
}
```

Listing 5.1: Constructor of the ”ClientIsland” class.

Afterwards, the method “sendMessage” was implemented so that it is possible to utilize it in another class by using the object of the “ClientIsland” class. In that method, the message is created and sent to the predefined queue, as can be seen in the code snippet below.

```
public void sendMessage() throws IOException, NullPointerException
{
  String message = "Hello from island container!";

  channel.basicPublish("", QUEUE_NAME, null, message.getBytes());
}
```

Listing 5.2: Method ”sendMessage()” of the ”ClientIsland” class.

5.4 Testing

The testing part in this iteration and the next two will be concerned only with the test strategy and the testing activities. This is because the overview of all the software testing concepts was given in the first iteration. Hence, from now on the test strategy and testing activities will be directly analyzed in each iteration.

5.4.1 Test Strategy

The software testing strategy that was chosen to be followed in this iteration includes the same type of information with the previous one. What needs to be clarified though is that this test strategy is described in relation to the test strategy of the previous iteration. In other words, the description includes only the parts of the strategy that are different from the previous one. All the other parts that are the same will not be included in the description, but they might be mentioned there. With that being said, the test strategy used for this iteration is the following one:

- **Scope and overview**: The software under test, once again, will be the island container. That is, it will be tested in terms of its ability to send data to the RabbitMQ message broker, since it is the requirement defined in the "Analysis" phase. As such, there will be only one testing activity that will test the robustness of the aforementioned interaction between the island container and RabbitMQ server.
- **Test approach**:
  - Black-box testing technique will be used in this iteration as well, because the code that was written is too simple and thus does not require extensive investigation. As such, the behavior of the system will be tested instead, in order to reveal unexpected software activities, if there are any.
  - Furthermore, functional type of testing was chosen to be used, as the other types were, once again, not applicable. That is, although this is the second iteration, not much software was developed. For this reason the change-related and maintenance testing were not applicable. Also, for the same reason neither the structural type of testing was applicable.
  - Finally, the software will be tested on the level of integration testing, as there are now two software components and the interaction between them is what is being examined.

5.4.1.1 Testing Activity

In this testing activity the communication between the island container and the RabbitMQ message broker is examined. That is, this activity includes sending data from the island

container to the RabbitMQ server and checking if the server received that data. The data that is being sent is a plain “String” data, which is “Hello from island container!”. This procedure is carried through the test case that is shown in table 5.1. As can be seen from that table, the test case was successful, which means that the data, sent from the island container, was received successfully by the RabbitMQ server.

Testing Activity - RabbitMQ Communication					
Test Case	**Preconditions**	**Test Steps**	**Expected Results**	**Actual Results**	**Pass/Fail**
Send a ”String” data to the RabbitMQ server and check if it was delivered.	RabbitMQ container up and running.	1 - Start the island container. 2 - Open the GUI of RabbitMQ server and monitor the incoming data.	The arrived data is the exact ”String” data sent from the island container.	The arrived data is the exact ”String” data sent from the island container.	Pass

Table 5.1: Test case for investigating the communication between the island container and the RabbitMQ server.

Furthermore, as shown in the table above, the precondition for the test case is the container of RabbitMQ message broker to be up and running. This can be checked using the Docker CLI command “docker ps”. After running that command, the container of RabbitMQ message broker was found to be up and running. This can be seen in the figure below, where the only container running was the one with the name “ody-rabbit”.

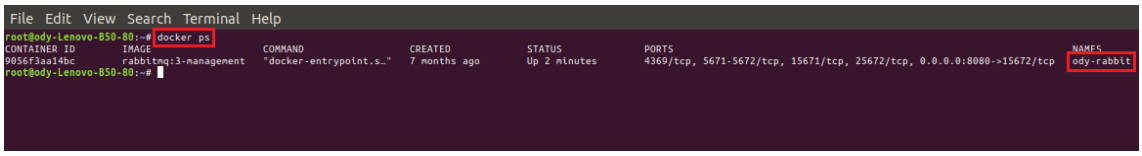


Figure 5.2: Docker CLI command used to check if RabbitMQ container is up and running.

Afterwards, the island container needed to be started, so that it can send the “String” data to the RabbitMQ server. Once again, the Docker CLI was used to achieve this. More particularly, the “docker start island-container2” command was used to start the island container, as shown in figure 5.3. It should also be noted that the simulation software was programmed to send the “String” data after completing a generation. The reason for this is the convenience of knowing when the data will be sent.

After starting the island container, the GUI, provided by the RabbitMQ server, needed to be used in order to monitor the queues and incoming data, if any. This way, if the correct data arrives to the message broker, then the test case is successful. Hence, that GUI is a web page served by the RabbitMQ server and it can be accessed using the IP of the machine which that server runs on. Since in this case the RabbitMQ server runs locally

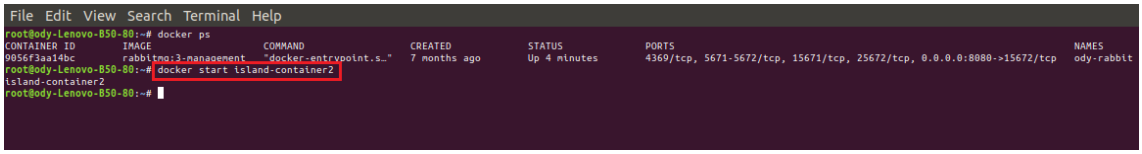


Figure 5.3: Docker CLI command used to start the island container.

on the Docker engine installed on the Linux Ubuntu18.04.3 LTS, the IP address that was used is “0.0.0.0” or ”localhost”. Furthermore, the correct port needed to be used in order to access that page and that port is “8080”. With that said, that GUI web page is illustrated in the figure below.

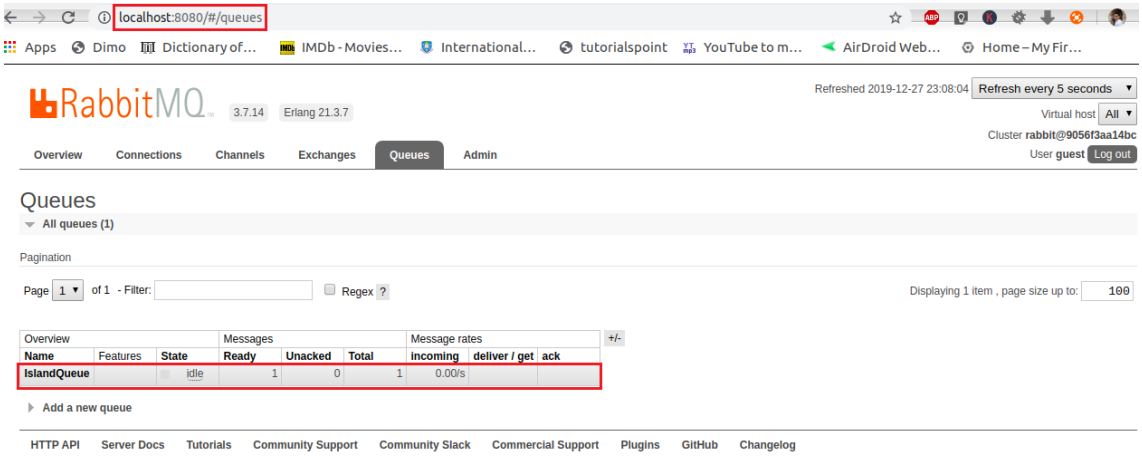


Figure 5.4: RabbiMQ web page that shows the created queue and its data.

It can be seen from the illustration above that a queue with the name “IslandQueue” was created and it had already one message. Thus, it needed to be checked if that message is the same that the island container was supposed to send. So, when the payload of that message was checked, it contained the expected “String” data which is “Hello from island container!”, as can be seen in figure 5.5. Therefore, the test case was successful, which means the system, developed till now, can proceed to the next iteration.

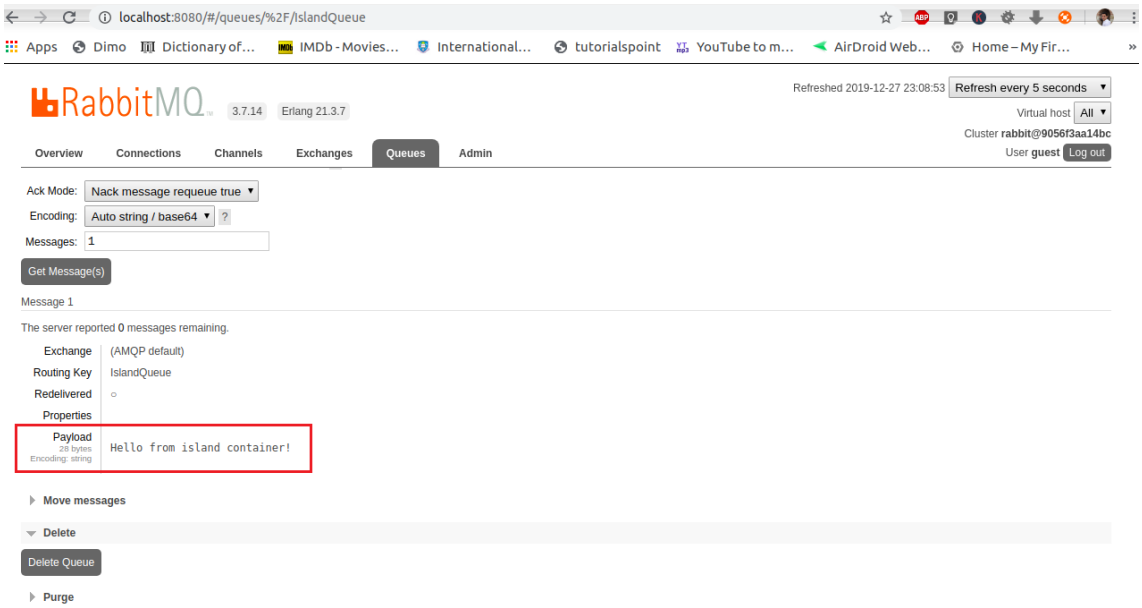


Figure 5.5: RabbiMQ web page that shows the payload of the message inside the ”IslandQueue” queue.

Chapter 6

Third Iteration

This chapter describes the second iteration by breaking it down into development phases, as in previous chapters. As such, each development phase is thoroughly analyzed in separate section. That said, in this iteration a new software component is developed and added to the software system of the framework, developed till now. Then, that software component is tested in terms of its integration to the system.

6.1 Analysis

This iteration is concerned with the problem of developing the master node, defined in the theoretical model of the first iteration. In general, the master node of the final iteration will be assigned various responsibilities, which are of vital importance for the software system. One of them, as already mentioned, is being a web server. Consequently, it will be the most critical part of the framework's software. In this iteration though, it will be designed to do the simple task of exchanging "String" data with the island container, developed in the previous iterations. The reason for that is the fact that this simple implementation of data exchange will act as the foundation for any more complex communication scheme that the final system will be decided to use. Therefore, the requirement for this iteration is the master node to be able to correctly exchange "String" data with the island container.

6.2 Design

Since it is required from the master node component to be able to exchange data with the island containers, it should be designed to first be able to communicate with the RabbitMQ broker. This is because in this case, RabbitMQ broker is the middleware through which the components of the system will communicate with each other. As such, in order for the master node to send data to an island container, it needs to first send that data to the

RabbitMQ broker. Afterwards, the island container needs to consume that data from the broker in order to receive it. The same procedure takes place also in the reversed case of island container sending data to the master node. Hence, this is the way that the data will be exchanged between the master node and island container, which can be seen in the figure below.

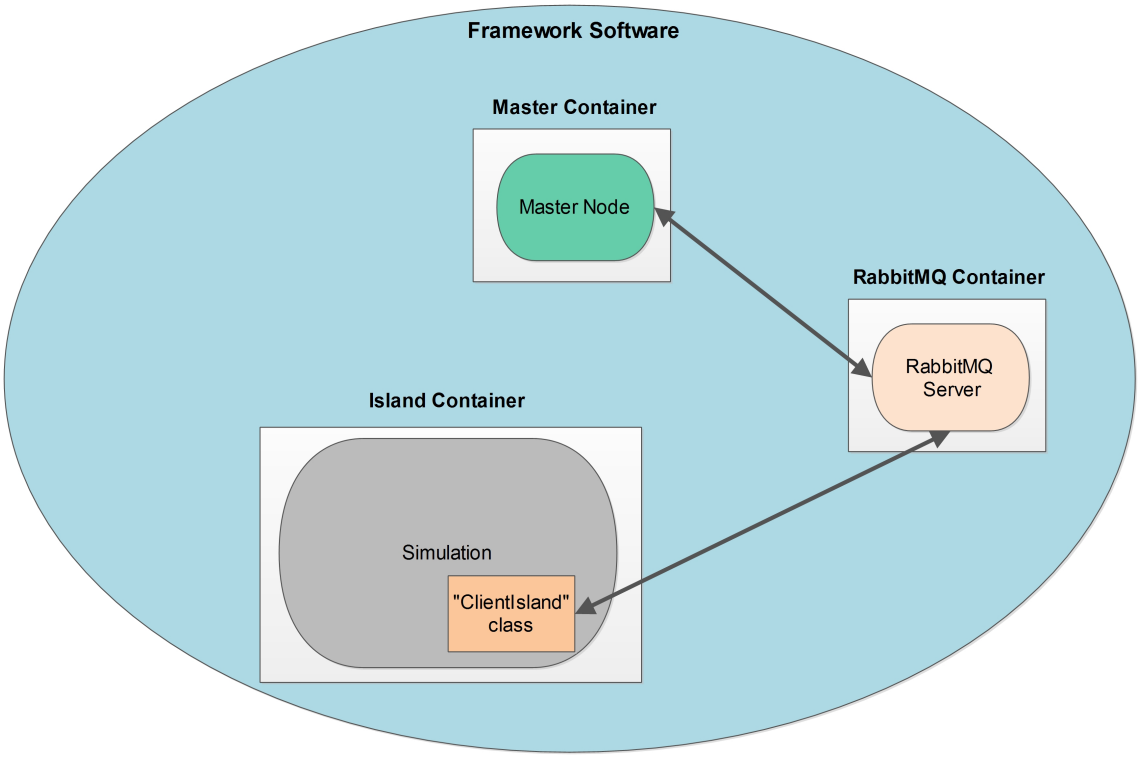


Figure 6.1: Software architecture developed so far.

The master container, illustrated in the same figure, will be created from a Docker image, which will be shown in the next section. Moreover, it can also be noticed that this time the island container also receives data, except from sending, since the master node is designed to send data to it.

Also, at this stage the exact technology, that would be used for the master node development, needed to be decided. As such, for the development of the master node, since it is also a server, Node.js was chosen. The main reason for that is the fact that Node.js is intended for real-time applications that run in a distributed fashion [58], which is the case at hand. Furthermore, in the view of the current system, this event-driven way of Node.js operation will enable the system to notify the master node about the evolution processes happening in the island containers in real time. Besides, it is more memory efficient than other server side tools, as it operates on a single thread using non-blocking I/O calls, instead of spawning a thread for every connection consuming system RAM. However, this technology is not good with heavy computation that can choke up its single thread, which would block all the incoming requests until the said computation was completed. But that will not happen in this case, since the computation is offloaded to the island containers.

Furthermore, in order to develop the server faster, a specific Node.js framework was used, the Express.js framework. However, the main reason for using this framework is to avoid writing boilerplate code and focus on the logic of the web application [59]. That is, by avoiding writing the boilerplate code the application is more robust and reliable, because writing boilerplate code is more error prone due to increased complexity [60]. This way, by using the available boilerplate code from this framework, the development risk is mitigated even more.

6.3 Implementation

Due to the specifications of the system and its complexity, the master node or the web server of the system needs to be developed manually. For that reason, the system can not adopt an already developed open source Node.js web server, that can be found as an official Docker image in Docker Hub. Hence, the master node was manually developed in Node.js server-side programming language. However, the development of this component, apart from the web server, involved the implementation of all the functionalities of the master node, related to the Island Model GA operations. As such, due to the development of all those functionalities, this software component can be regarded as a web application. In this iteration though, that web app will only include the implementation for the connection to the RabbitMQ broker and the process of receiving and sending messages. Moreover, except for the master node, the island container also needed to be modified, since at this stage both software components take part in the message exchange procedure. With that being said, the first step was to implement the web server part of the master node, which can be seen in the code snippet below.

```
const express = require('express');
const http = require('http');

const app = express();

const server = http.createServer(app).listen(3005, function(){
  console.log('Express server listening on port ' + 3005);
});
```

Listing 6.1: Web server implementation of the master node.

As can be noticed from the code above, the implemented server listens on port “3005”. However, it can listen to any other available port, thus the choice was rather arbitrary. It is also important to note that this port is the one used to directly access the server. Meaning, in the stage of dockerization of this component, which will be explained later in the text, another port may be used to access the docker that will accommodate the master node.

The next step was to write code for the connection to the server of the RabbitMQ message broker and the procedures of receiving and sending messages. This code is illustrated in the code snippet below.

```
amqp.connect('amqp://ody-rabbit', function(error0, connection) {
```

```
2     if (error0) {
3         console.log('Error in connection: ' + error0.msg);
4         throw error0;
5     }
6     console.log('Connected to RabbitMQ');
7     connection.createChannel(function(error1, channel) {
8         if (error1)
9             throw error1;
10
11        channel.assertQueue(fromQueue, {durable: false});
12        console.log(" [*] Waiting for messages in %s. To exit press
    CTRL+C", fromQueue);
13
14        channel.consume(fromQueue, function(msg) {
15            receivedMessage = msg.content.toString();
16            console.log(" [x] Received: %s", receivedMessage);
17
18            channel.assertQueue(toQueue, {durable: false});
19            toSendMessage = 'Answer to "' + receivedMessage + '": Hello
     from master!';
20      channel.sendToQueue(toQueue, Buffer.from(toSendMessage));
21            console.log(" [x] Sent: %s", toSendMessage);
22        }, {
23            noAck: true
24        });
25    });
26 });
27
28 app.get('/', (req, res) => res.send('Message from RabbitMQ broker: ' +
    receivedMessage));
```

Listing 6.2: Implementation of the web app part of the master node.

As shown in the code above, the first thing to do was to establish connection to the message broker. This can be done using the correct IP address or alias of the machine accommodating it, which in this case is the “amqp://ody-rabbit”. Afterwards, a channel needed to be created and thus it was implemented using the method “createChannel” in line 7. Then, it was necessary to create the queue from which the master node will consume messages and it was created using the method “assertQueue” in line 11. Nonetheless, in order to consume messages from that queue, the method “consume” was used in line 14. After consuming the message, the master node needs to respond with a message of its own, as required in the “Analysis” phase. Hence, it was necessary to use the “assertQueue” once again in line 18, in order to define the queue to which messages will be sent, since the same queue must not be used for consuming and publishing messages. After that, the “sendToQueue” method needed to be used in line 20, in order to send the message to the predefined queue. As for the last line 28, the master node outputs the received message to the web page that is served to the user, since it is also a web server.

After the master node was developed, it needed to be dockerized, since the software system is a multi-container application, where all the software components are software containers. Thus, in order to dockerize this software component, a Dockerfile needed to be written from scratch. As such, the Dockerfile written and used for the development of

the master node is the one shown below.

```
FROM timbru31/node-alpine-git
WORKDIR /app
COPY package.json /app
RUN npm install
COPY . /app
EXPOSE 3005
CMD ["npm","start"]
```

Listing 6.3: Dockerfile used for the creation of the master node.

As for the island container, the code of the “ClientIsland” class needed to be extended so that it can also receive messages, apart from sending. This way, by modifying and extending both of the software components, the message exchange process can be completed. As such, the code that was added to the connection class is illustrated below.

```
  public void receiveMessage() throws IOException
  {
    channel.basicConsume(QUEUE_NAME_FROM, true, new DefaultConsumer(
   channel) {
      @Override public void handleDelivery(String consumerTag, Envelope
    envelope, AMQP.BasicProperties properties, byte[] body)
      {
        try
        {
          String message = new String(body, "UTF-8");
          System.out.println("[x] Received: '" + message + "'");
        }
        catch (UnsupportedEncodingException e)
        {
          System.out.println("UnsupportedEncodingException error in
   receiving: " + e.getMessage());
        }
      }
    });
  }
```

Listing 6.4: Implementation of the ”receiveMessage” method of the ”ClientIsland” class.

As can be seen from the code snippet above, the code that was added is actually the “receiveMessage” method, whose task is to consume messages from the queue defined in the final variable[1] “QUEUE_NAME_FROM”.

All things considered, by making all these modifications to both the master node and island container, the current system is enabled to circulate data between these two software components.

[1] Variable that is constant, meaning it cannot be changed [61].

6.4 Testing

6.4.1 Test Strategy

As in the previous iteration, the test strategy of the current iteration is described in relation to the previous test strategy. Hence, following is the description of this test strategy:

- **Scope and overview**: As in the previous iteration, the software under test in this one includes more than one software components. That is, the RabbitMQ message broker, island container and master node software components of the current system are being tested. More particularly, these components will be tested in terms of communication. This communication mainly involves the process of exchanging messages, since it is the requirement defined in the "Analysis" phase. As such, the testing activity in this iteration is concerned with testing the robustness of "String" data exchange between the two software components.
- **Test approach**:
  - In this case as well, black-box testing technique will be used. This is because, once again, the code, written in this iteration, was too simple to invest time for white-box testing. This way, by investing time more wisely, the development risk of this project is mitigated in terms of development time. As such, the functionality of the system, developed so far, will be tested to find out if there are any unexpected behaviors, using test cases.
  - Furthermore, functional testing was chosen to be used for the testing phase of this iteration, because it was more preferable from the other types of testing. To explain, till now not many changes were made to the code from the previous iterations, that is why it was decided to not invest time for change-related and maintenance testing. Moreover, the structural type of testing was not chosen because, as mentioned before, the code written till now is not too complex and thus this type of testing was not worthy of incorporating. Besides, black-box testing was considered enough to reveal any errors and bugs for this iteration.
  - Finally, the software in this iteration will be tested on integration level of testing, once again. The reason for this is that in the current system there are three software components and the communication between them is what is being tested.

6.4.2 Testing Activity

This testing activity examines the communication between the island container and master node through the RabbitMQ message broker. In other words, the message broker upholds the data exchange between the other two components. As such, the correctness of this data exchange is being examined. With that being said, the exact procedure of the data

exchange includes having the island container send “String” data to the master node and then having the master node automatically respond by sending back different “String” data. More specifically, the island container will first send (or publish) the “Hello from island container!” message to the RabbitMQ server, as in the previous iteration, and then the master node will receive (or consume) that data from there. After receiving the message sent from the island container, the master node will automatically respond with its own “String” data, which is “Hello from master node!”, and publish it to the RabbitMQ server. Finally, the island container will consume that message and the activity will be completed. This message exchange procedure takes place within the frame of the test case shown in the table below.

Testing Activity - Data Exchange					
Test Case	**Preconditions**	**Test Steps**	**Expected Results**	**Actual Results**	**Pass/Fail**
Send a ”String” data from island container to the master node, check if it was delivered and backwards.	RabbitMQ container and master node up and running.	1 - Start the island container. 2 - Monitor the incoming data from the web page of the master node and the simulation terminal of the island container.	Correct ”String” data received from the two software components.	Correct ”String” data received from the two software components.	Pass

Table 6.1: Test case for investigating the communication between the island container and the master node.

As can be seen from the table above, the test case was successful. Meaning, the data exchange between the island container and master node, using the RabbitMQ broker, took place without any problem. Also, the correct messages were arrived to each of the software components at the right time.

Before starting the activity, it was necessary to check the preconditions of the test case, as in the previous iterations. As such, these preconditions, as shown in table 6.1, are the RabbitMQ and master node containers to be up and running. Therefore, the two containers were launched and in order to validate that they were indeed up and running, the Docker CLI command “docker ps” was issued. After running that command, the containers of RabbitMQ (ody-rabbit) and master node (iteration3-express) were found to be up and running and thus the preconditions were validated. This can be seen in figure 6.2.

After checking the preconditions, the first test step of this testing activity, which is to start the island container, was made. This step initiated the whole message exchange procedure between the two software components, since the data communication between them was programmed to be automated. To explain, as in the previous iteration, the island

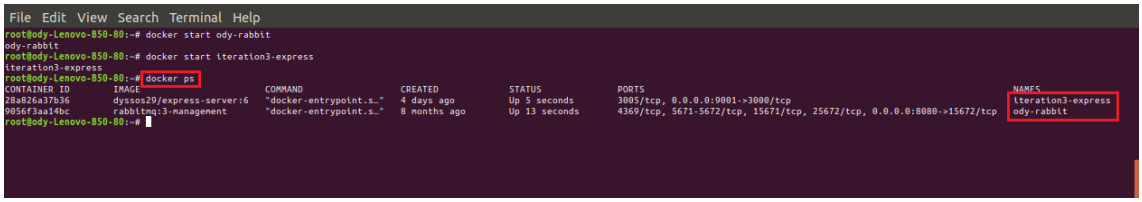


Figure 6.2: Docker CLI command used to check if RabbitMQ container and master node are up and running.

container sends the message automatically without any intervention, using the "sendMessage" method of the "ClientIsland" class. Afterwards, the master node as well is programmed to send respond in an automatic manner, after it consumes the message from the island container. Therefore, by starting the island container, the data exchange procedure begins and ends when that container receives response from the master node container.

After starting the island container, the next step was to monitor both of the software components to find out if the exchange of messages occurs as expected. However, for someone to be able to monitor those two components, they need to provide some monitoring tools. With that being said, the master node uses the web page, which it can serve as a web server, in the role of an output interface that displays the received message. The island container on the other hand, uses the terminal provided by the Processing framework, by which the example simulation was developed.

Since the master node would be the first one to receive a message, it was the first one to monitor, using its web page that was mentioned before. That web page can be seen in figure 6.3, from which it can be noticed that no message was received and consequently no message was sent as a response.

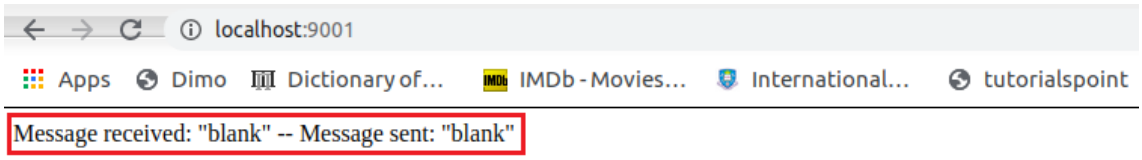


Figure 6.3: Web page served by the master node before any message received.

After a while though, the master node received a message as expected and this time the web page displayed different content, which can be seen in figure 6.4. As shown in that figure, the message that was received is exactly the one that the master node was expected to receive. Thus, that message was successfully transferred from the island container to the RabbtiMQ message broker and from there to the master node.

Figure 6.4: Web page served by the master node after a message received.

In the same figure it can be noticed that the master node sends a message automatically as a response. Hence, now the island container needs to be checked in order to find out if it received that message. With that being said, as illustrated in figure 6.5, the island container successfully received the message that was sent from the master node. Hence, that message as well successfully travelled from the master node to the RabbitMQ broker and from there to the island container.

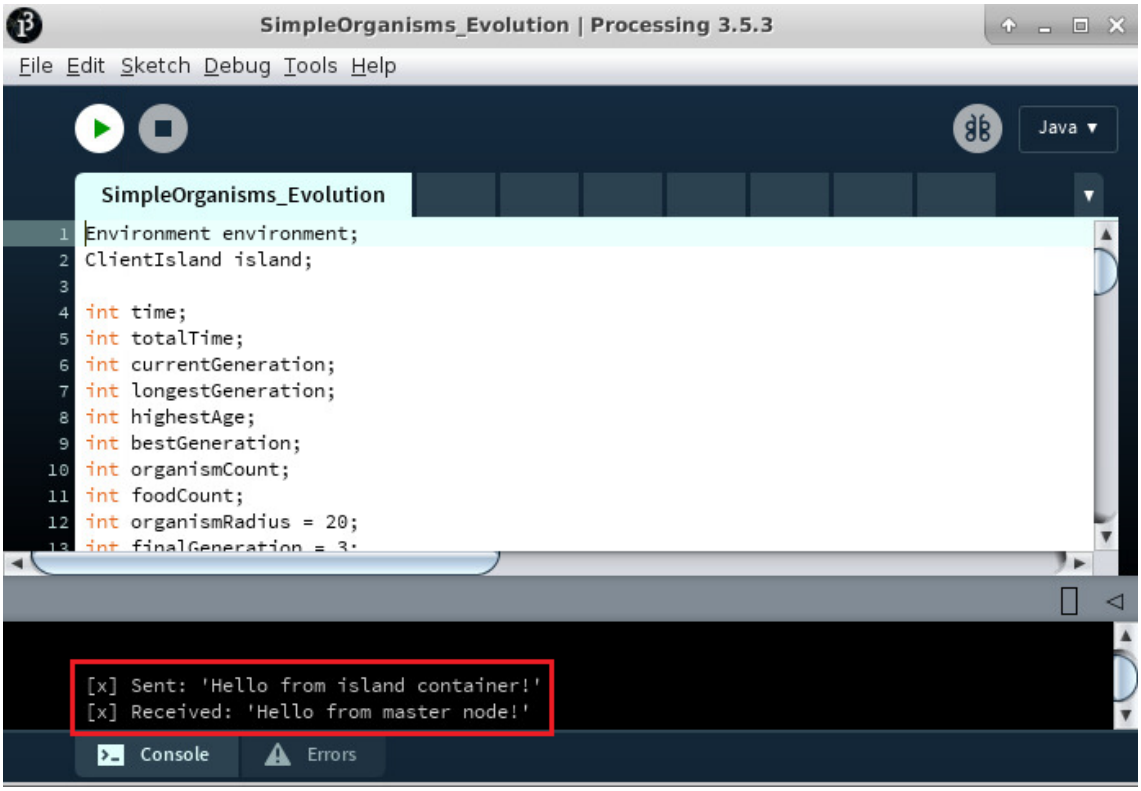


Figure 6.5: Terminal of the Processing framework that shows the messages exchanged.

From that terminal it can also be noticed that before receiving any message, the simulation sent the “Hello from island container!” message as planned from the beginning. Therefore, that message was indeed sent from the island container, which means the whole message exchange procedure executed exactly as expected. For that reason the test case of this testing activity was successful.

Chapter 7

Fourth Iteration

7.1 Analysis

Since it is the last iteration, all the remaining development problems need to be tackled in order to complete and deliver the final version of the project. All these problems will be regarded as requirements for this iteration, since this is the "Analysis" phase. As such, there are mainly three problems that remain to be solved. One of them is to make the current system a software stack, as visualized from the beginning of the project. The other problem to be solved is to equip the framework with real-time behavior, that will enable its users to watch live the progress of their simulation. Finally, the last one is to define a migration policy for the PGA that the framework uses as a core algorithm.

Regarding the first problem, by upgrading the existing software into a software stack, using the Docker containerization technology, will grant the system the beneficial traits of scalability, fault-tolerance and portability. These three features are essentially three of the four objectives that the expected system needs to fulfill. Thus, the solution of this problem will be confirmed by the successful validation of these objectives. With that being said, making this framework a software stack means that its software will consist of multiple software containers which will not operate in an isolated manner, as was the case till now, but rather in a unified manner. More particularly, each software component will be represented by a Docker service, which is able to spawn as many software containers of that component as needed. Thus, the software stack will comprise several Docker services which will operate in a unified manner, since they will be orchestrated by the Docker swarm orchestration tool. As such, the first requirement assigned for this iteration is to incorporate all these characteristics to the final software system, so that eventually it can be a software stack.

Next, the second problem is concerned with equipping the framework with the previously mentioned real-time behavior, which is actually another objective the project needs to fulfill. As such, the solution of this problem as well will be confirmed by the successful validation of the real-time behavior objective. With that being said, this problem is related

to the communication between the Graphical User Interface (GUI) of the system and the master node. That is, in order to enable a user to monitor the progress of his simulation in real-time fashion, the GUI needs to be updated with new data (from the master node) in the same fashion. Therefore, a technology needs to be chosen that will enable real-time data exchange between the GUI, which in the current system is a web page, and the server, which is also the master node.

As for the third problem, it is concerned with the definition, design and development of the migration policy of this framework's core algorithm, which is the Island Model GA. As such, the migration policy of the algorithm needs to be firstly defined and then applied to the current system. It is also important to note that the internal communication part of the framework is exclusively made up of migration operations that follow a specific migration policy. This means that the migration policy governs all the internal communications of the framework and thus it plays a very important role.

With that being said, the migration policy of the Island Model GA will be defined in this section as a third requirement for the system. In other words, since this section describes the "Analysis" phase, the parameters will be described in terms of what needs to be done. This way, after defining all the parameters of the migration policy, the next "Design" phase will describe the way they will be adapted to the architecture of the system. This in turn will make it possible to easily incorporate all those parameters into the software of the system in the "Implementation" phase. As such, the next section provides in-depth description of these parameters.

7.1.1 Migration Policy

In this section, all the parameters of migration policy are described thoroughly, explaining the reason they were chosen, along with their pros and cons. Furthermore, some important points, regarding those parameters, are given at the end of the section. However, before proceeding with the migration parameters, it is important to note that according to Cantu-Paz [22] the study of migration policy in IMGAs is still in its infancy and not fully understood. Thus, the below parameters were chosen according to the available information reported in the literature.

7.1.1.1 Migration Topology

Broadcast topology will be utilized, as it is the most suitable one for the particular system. The reason mainly has to do with the architecture of the framework's system. That is, any other topology would require more complicated architecture and additional components for neighbor definition and discovery. For example, in the case of ring topology, the system would be required to be able to define the two neighbors of an island and to properly add a new island in the ring topology. This would require some additional software components or more code to be written in the "ClientIsland" class for checking and monitoring the

neighbors. Therefore, it was decided to choose a topology that would would be simpler for the current system in order to avoid this architectural complexity.

There are of course some practical advantages of the broadcast topology that benefit the software system. These are the relatively low diameter of the edges and the fact that the number of edges grow linearly as more islands are added [27]. In other words, this topology alleviates the communication overhead problem of the migration process. Nevertheless, the broadcast topology has also a notable disadvantage, which is the communicational or processing bottleneck that occurs in the center node, something that to some extent negates the advantages mentioned before [27]. The center node in this case is the Master node of the system which, among other functionalities, also provides a broadcast point for this topology, as depicted in figure 7.1. This problem though is dealt with one of the components of the software architecture, which is the RabbitMQ component. In that respect, RabbitMQ is capable of handling large amounts of data due to its high throughput and scalability properties [56], making this tool perfect fit to solve this problem.

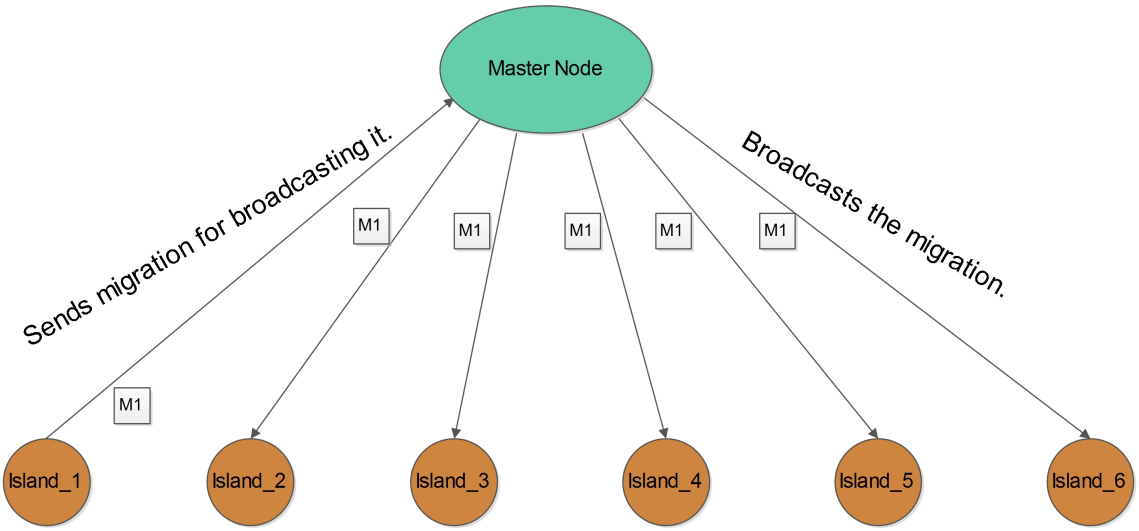


Figure 7.1: Broadcast migration topology

7.1.1.2 Migration Frequency

High migration frequency was chosen for the system, meaning migrations will occur after every generation. The main reason is the long generation time, since the GA runs on the Processing framework which needs time to demonstrate the learning process of the organisms. Which means, there will not be many generations due to possible time constraints. Furthermore, this approach is also supported by Skolicki in his paper [30], stating that low migration frequency should be avoided when the time is limited.

On the other hand, Skolicki and De Jong in their paper [32] have observed that usually the performance is significantly decreased with high migration frequencies. This is because of the "conquest" phenomenon, where some islands dominate or "overwrite" other ones before any successful recombination of solutions could be reached. This way fre-

quent migrations also decrease the diversity between the islands, since the exchange of individuals occurs so soon that the islands start to exchange the same individuals.

Therefore, in order to alleviate this problem, it was decided to incorporate a policy that would determine if the island will migrate individuals or not. According to this policy, the migration will occur only if the fittest individual has fitness value higher than that of the fittest individual from the previous generation. The idea was taken from the observations of some preliminary system tests, where some times fittest individuals had the same fitness values with the previous ones. Meaning that there were occasions were no progress was achieved from one generation to another.

7.1.1.3 Migration Rate

Migration rate of 5% was chosen because large rates diminish the performance and excessively small ones tend to not affect the diversity [32]. The large migration rates worsen the performance of the algorithm because they tend to overwrite the target population, leaving no room for mixing the genetic material. On the other hand, small rates also negatively affect the performance as they slow the convergence because the migrants will most probably die before they affect the target population [29].

Having said that, the reason for a relatively small migration size is to slow the convergence of the algorithm, since it will be already fast due to the high migration frequency.

7.1.1.4 Migration Selection and Replacement

In the literature of Island Models it is known that migrating fittest individuals increases the selection pressure[1] [33] and contributes to faster convergence of the algorithm [29]. Furthermore, this practice boosts the exploitation part of the algorithm, since the ”strong” characteristics of the fittest individuals are exploited in order to produce the next generation of individuals. Apart from all these, another reason to follow this strategy is to simply avoid migrating weak individuals, since they will negatively impact the evolution of the target island [33]. On the other hand, by migrating random individuals, the selection pressure is lowered and as a result the convergence of the algorithm is slowed down. Moreover, this practice increases the local diversity of the target’s island population and prevents the “conquest” effect, where dominant individuals cause premature convergence [33]. In other words, this strategy boosts the exploration part of the algorithm.

With that being said, the strategy devised and used for this system is to send three individuals, one of which will be the fittest individual from the last generation and the other two random individuals. The reason for choosing these quantities of individuals is related to the migration rate, which was previously described. That is, the population size of the example simulation is 60. This means that, since the migration rate is set to 5%,

[1] Degree of emphasis on selecting the fittest individuals, instead of weaker or random ones [62].

then the migration size will be 3. Furthermore, the reason for choosing this analogy of fittest and random individuals is to maintain performance balance. To clarify, having in mind the aforementioned theory of migrating either fit or random individuals, the reason for including both of these individuals in the migration process is to maintain balance between exploitation and exploration of the algorithm. However, the reason for sending more random individuals is to compensate for the high convergence speed, caused by the high migration frequency, which was previously chosen to be used. So overall, this strategy for migration selection and replacement contributes to the balance of the convergence speed and diversity.

7.1.1.5 Copy-based or Transfer-based Migration

The transfer-based migration is selected to be used, because copy-based migration reduces the diversity of the local population on every migration occurrence [33]. This way, the higher diversity, provided by the transfer-based migration, also contributes to solve the problem with the high migration frequency.

7.1.1.6 Synchronicity of Migration

Asynchronous migration was chosen for the framework's core algorithm, as it is specifically applicable to the asynchronous nature of the system. More particularly, the time duration of a generation in every island is not fixed. Which means, every island will commence the migration process at different points of time. Thus, this approach is only feasible in asynchronous versions of the Island Model.

For this reason, RabbitMQ message broker is utilized to support the internal communication part of the system, which as mentioned before, exclusively consists of the migration operations of the algorithm. That is, this message broker is especially suitable for the asynchronous communication between the software components of the system, due to the asynchronous nature of the AMQP protocol used in it [63]. Therefore, for the particular software stack, it undertakes the task of asynchronously transferring migrants between the islands.

Apart from that, asynchronous migration process provides some considerable benefits on its own. One of them is the fact that in case of synchronous migration, synchronizing the execution and migration process in a distributed computing environment most probably requires a centralized approach to govern these executions. This approach according to Biscani et al [27] will inflict significant additional computation costs and may also affect the scalability of the algorithm. Island Models that use asynchronous migration, on the other hand, do not have these type of problems and thus avoid those computation costs. This holds especially when the communication between islands is implemented in an asynchronous way. That is, communication does not block the computation on the target island, exactly as in the current system.

Another significant benefit of asynchronous migration is that according to Alba and Troya [35] algorithms which use asynchronous migration perform better from the ones that do not. The reason for this is the fact that the asynchronous version of the algorithms seemed much more consistent in their work, whereas the performance of the synchronous ones mostly depended on the migration policy. In their paper they also showed that asynchronous migration reduces the execution time of the algorithm, since it does not block the execution at every migration step.

On the other hand, it is important to note that there is one disadvantage when using asynchronous migration. Biscani et al [27] in their work emphasize that the centralized synchronization mechanism, present in Island Models that use synchronous migration policy, enables the system to monitor and control the algorithm much easier. The reason for this is that, contrary to the asynchronous migration, in the case of the synchronous one there is global synchronization between the islands. Thus, for the asynchronous versions of Island Model this makes it harder to know the global state of the algorithm at any point of time. Which means, it is not a simple task for instance to monitor the algorithm in order to stop it whenever a particular value of the objective function is reached. This is also the reason algorithms that use asynchronous migration are not suited for theoretical studies, as indicated by Barbosa et al in their paper [33]. Fortunately though, the purpose of this project is not to control or monitor the migration process. Thus, there are no practical disadvantages of using asynchronous migration in the algorithm.

7.1.1.7 Important Points

Considering the parameters above, it is important to realize that the main disadvantage of the chosen migration policy is the high migration frequency. Nonetheless, that was a necessary choice due to the nature of the example simulation, as explained before. This disadvantage though, is mitigated by four design and parameter decisions. Namely, one of the design decisions to alleviate the problem is the policy, described in migration frequency parameter, which decides whether the migration operation will be performed or not. The other three are the small migration rate parameter, selecting more random individuals to migrate and the transfer-based migration, instead of the copy-based one. This way, the convergence speed and the local diversity will be balanced.

Also, another important thing to note is that the broadcast topology described above is applied on a theoretical level to the system. Meaning, the underlying implementation of this topology required communication techniques that are not usually utilized for broadcast communication pattern. The main reason for this is the asynchronous nature of the software system. These communication techniques are thoroughly analyzed in section 7.2.3.

7.2 Design

The first task of this phase is to tackle the problem of upgrading the existing software system into a software stack. In order to do that though, the existing software needs to be properly structured. As such, the next section is concerned with the design of the final system's architecture according to which the software components will be structured. This way, by incorporating that architecture and some additional technologies, the software system of the framework will be evolved into a distributed software stack able to operate on any distributed environment.

The next task is to define the technology or protocol that will grant the framework the real-time behavior, mentioned in the ”Analysis” phase as one of the requirements. More specifically, that technology needs to be able to establish a real-time communication between the GUI and master node of the system.

The last thing to design is the migration policy of the algorithm, which was defined as a requirement in the “Analysis” phase. In this case, it is important to point out that most of the defined parameters of the migration policy are straightforward and thus can be directly implemented in the “Implementation” phase. To be more precise, the only parameter that needs to be designed, in order to be easily implemented, is the parameter of the migration topology. The reason for this is the fact that the implementation of the chosen migration topology is the most complex part of the system, for reasons that will be explained later in the text. As such, the design of the migration topology is thoroughly analyzed in section 7.2.3.

7.2.1 Software Stack Architecture Design

As already known, the software stack of the framework's system will consist of several different software components. Each one of these components is a vital part of the system's theoretical model, defined in the first iteration. This way, the system's software may also be considered as a multi-component software, that can distribute its software components over a cluster of machines or any other distributed system. However, that software needs to be scalable, portable and fault-tolerant, since these are the project's objectives. This is where the Docker technology comes in and solves those three problems by using its containerization technology, as explained in the first iteration. As such, to be more precise, the multi-component software of the system is actually a multi-container software, because of the incorporated Docker technology. Nonetheless, the containers of this multi-container software are required to be properly structured and operate in a unified manner, as stated in the “Analysis” phase. Thus, by doing so, the software of the system will be evolved into the expected software stack, able to easily scale its containers on any system, distributed or not.

With that being said, all the software components of the current system will be structured according to the software architecture, illustrated in figure 7.2. This architecture

is based on the theoretical model, presented in the first iteration (figure 4.1), which provided more abstract architecture of the system. Put differently, that model was used as a theoretical foundation for the construction of this, more detailed, software architecture. Furthermore, the Docker Swarm orchestration tool, as mentioned before, will be used in order to make all the containers of the software system operate in a cooperative and unified fashion. However, this is not a simple task, since it requires configuring a cluster of computing machines and also writing a configuration file, something that the "Implementation" phase throws light upon.

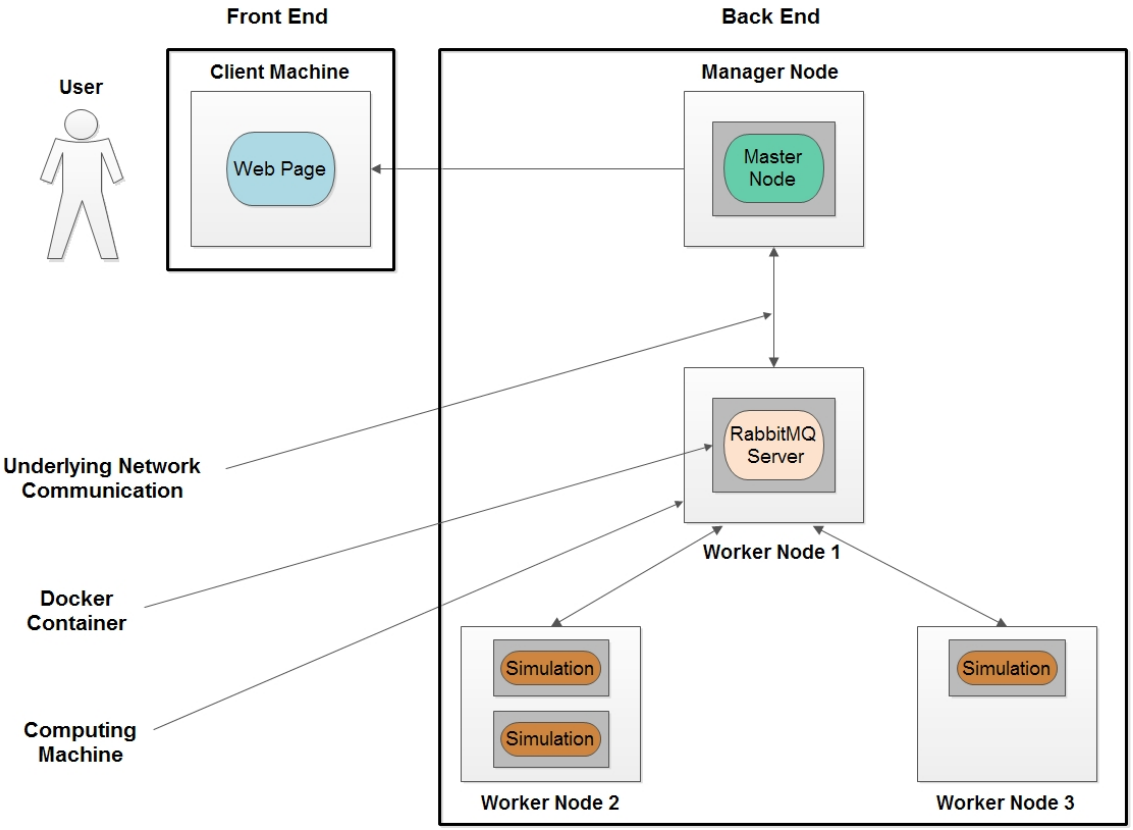


Figure 7.2: Software architecture of the framework.

As can be noticed from the figure above, the multi-container software of the framework distributes its containers over the cluster of available computing machines. That is, each machine of the cluster can accommodate one or more software containers of the system. Also, the architecture of the software is divided into two parts, which are the back-end and front-end parts. That being said, the back-end part of the system is basically the cluster of computing machines or any other distributed system, which undertakes the operation of the framework. More specifically, the core algorithm of the framework, which is the Island Model PGA, actually runs on that distributed system. Meaning, the operation of a simple GA on each island, their interaction with the master node and the queuing process of the message broker, they all take place on the back-end part of the system. The front-end part, on the other hand, is actually a web page served by the master node, since it is also a web-server. Hence, that web page acts as a Graphic User Interface (GUI) in

this software system, which shows the user various information about the progress of his ALife simulation and also its final results. These two parts are analyzed in more depth in the next two sections.

7.2.1.1 Back-End Part

The distributed system of the back-end part is governed by the Docker Swarm orchestration tool. This way, every computing machine is employed in the Swarm cluster, where it is labeled as “node”, apart from the one that handles them which is labeled as “manager”. Furthermore, each machine in that cluster can be either a physical or a virtual one. The only thing that is required from the computing machine to participate in the Swarm cluster is the Docker engine. Meaning, a computer needs to have the Docker engine installed in its operating system in order to be able to join the Docker Swarm cluster. Moreover, each container of the software stack belongs to a service, more precisely the Docker Service, which was mentioned in the ”Analysis” phase. To clarify, a Docker Service is a specific application that is being packaged and it can have one or more containers. As such, the island of the framework’s algorithm is one of the services employed in the system and it can have one or more containers, which can span over multiple machines of the underlying distributed system. Thereby, this service can have as many containers as the number of islands needed for the core algorithm of the framework.

To elaborate more, in the back-end part one can notice that the architecture follows the Master-Slave programming paradigm, exactly as in the theoretical model it is based on. As such, each island, which is a slave in this case, can not directly interact with another island. Instead, it needs to communicate with the master node first in order to interact and send data to other islands, as described in section 7.2.3. This whole communication between the islands and the master node is undertaken by the RabbitMQ message broker presented in the second iteration. Hence, all the messages that are sent between master and slaves are queued in that broker first and then carefully delivered to their destination. This way, the message broker of the system ensures the reliability of the communication, since all the messages are secured within the queues of the broker and thus are never lost.

Moreover, the master node component of the system, as already mentioned in the previous iteration, undertakes various important responsibilities, which makes it a vital part of the system. As such, apart from the responsibility of serving a web page to the user, another of its responsibilities is to gather the results from the core algorithm of the system, which is the Island Model GA. That is, after all the islands have finished running all their generations, they will send their results to the master node. Also, another responsibility of the master node is to undertake the migration process between the islands, which means it will exchange data with the island containers, according to an algorithm, also described in section 7.2.3.

7.2.1.2 Front-End Part

As for the front-end part of the system, the web page, presented to the user, is served from a server, which uses the Node.js technology. This modern server-side technology is highly suitable for the particular system, not only because of its event-driven nature, as explained in the previous iteration, but also because it operates in a non-blocking fashion. Meaning, the non-blocking feature of Node.js technology enables the asynchronous communication between the islands, since this feature enables a Node.js server to concurrently serve many clients without blocking none of them. This way an island can asynchronously perform its migration step whenever it needs to, without blocking its entire execution.

In addition to that, the web server also utilizes the WebSocket technology, which grants the framework the advantage of real-time communication, which is also one of the project's objectives. That said, by using the contemporary WebSocket protocol, the server of the system is able to update the data in the web page in real-time fashion. This way, the user of the system will be able to watch live the update of data in the web page, whenever new data are arrived from the server. This technology is thoroughly described in the next section. With that being said, using these two technologies, the framework software enables its users to watch the progress of their ALife simulation in a real-time manner.

7.2.2 WebSocket for Real-Time Communication

As already mentioned, the WebSocket technology is used in the software system of the framework to fulfill one of its objectives, which is to equip the system with real-time monitoring feature. As such, the modern WebSocket technology is utilized to enable real-time communication between server and client. To clarify, WebSocket is a new technology, provided as a feature of the modern HTML5, which introduces real-time, desktop application behavior to web apps [64]. More particularly, it provides real-time, bi-directional, full-duplex and persistent communication between server and client, as illustrated in figure 7.3. This way, the Node.js server (master node) of this system can send Rabbitmq messages from the back-end part of the system to the client (web page) in real-time fashion.

However, there are several different ways to achieve this. With that being said, the reason this technology was chosen is the fact that the other ways and techniques are not as efficient because they rely solely on HTTP requests [65]. To explain, communication techniques that rely on HTTP request, apart from code complexity, they introduce significant overhead in terms of resource consumption and bandwidth. The reason is that HTTP request and response headers contain lots of additional, unnecessary header data and thus cause high latency [64]. Additionally, in order for these techniques to simulate full-duplex communication, they need to establish two connections, one for the downstream and the other for upstream. Thus, the task to maintain and coordinate those two connections involves additional complexity and resource consumption. Such techniques are AJAX short or long polling, Comet polling and Server-Sent Event (SSE). Compared to

WebSocket communication protocol, all these previous technologies mimic the real-time communication fashion, by really using HTTP requests behind the scenes, as mentioned before. WebSockets, on the other hand, provide actual two-way open connection between server and client, with much lower latency, without all the previous additional overhead.

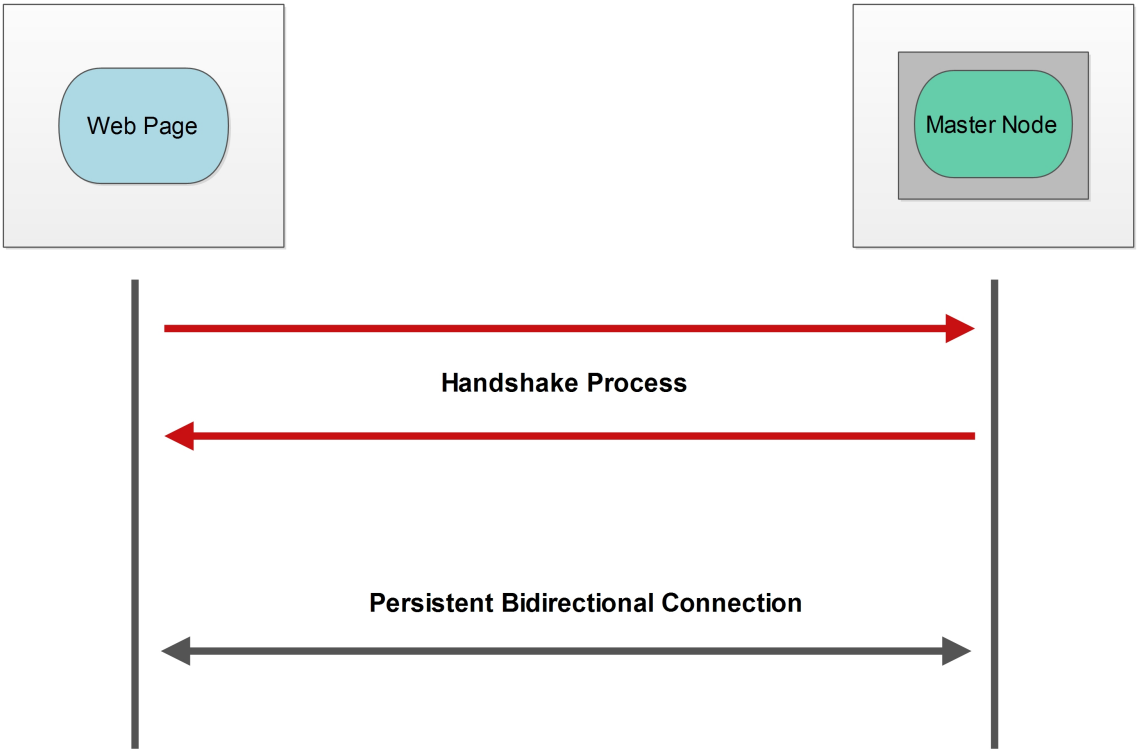


Figure 7.3: WebSocket communication between the system's web page and master node.

Furthermore, in terms of security, WebSockets are better option than the previous techniques [66]. AJAX polling, for example, is susceptible to Man In The Middle (MITM) attacks, since each request is a new TCP connection which travels through the Internet infrastructure. In WebSocket case, on the other hand, after the Handshake phase, shown in figure 7.3, a persistent connection is established, which makes it far more challenging to intercept in between. Also, WebSockets enforce frame masking, while streaming the data, which makes it difficult to probe the data and thus adds more security to the communication provided by this technology.

It is also important to note that Socket.IO library was used for the implementation of the WebSocket protocol. The reason for using this Node.js library is to simplify the WebSocket programming and make the implementation of that technology more robust, as was the case with the Express.js framework. In detail, apart from the simplicity of code and the already available boilerplate code provided by the Socket.IO library, it also equips the system with some useful features [67]. That is, this library is able to use fallback transport technologies, which in the case of direct implementation of WebSocket is not available. This way, every user can have real-time experience using the framework, no matter which protocol is available for transportation in the system of that user. Furthermore, it enables the system to establish connection in the presence of proxies and load balancers. Some-

thing that is not possible when using directly WebSocket protocol. Therefore, in addition to the real-time communication objective, all these features of Socket.IO library fulfills also the portability objective of the system even more.

7.2.3 Migration Communication Design

As mentioned in the ”Analysis” phase, the migration operation will use the broadcast topology for the particular algorithm. However, it was also mentioned that the underlying implementation of this topology is totally different from what someone would expect. The main reason for this, as already mentioned, is the fact that due to the asynchronous nature of the system, it was not possible to implement directly the communication pattern of the broadcast topology. However, there is also another reason for this, which has to do with the algorithmic aspect of the migration topology and it will be described in more depth later in the text. Therefore, due to these two reasons, other techniques and communication patterns needed to be involved in order to implement the theoretical communication pattern of the broadcast migration topology. For this reason the implementation of this migration topology is the most complex part of the framework's software system. Lastly, before proceeding with the details of the communication design, it is important to point out that from now on all the migrants sent from an island, along with some other information, will be called as ”migration” for simplicity reasons.

The second reason for not being able to directly implement the broadcast topology, mentioned previously, is mainly related to the migration topology itself. That is, the problem mostly has to do with the algorithmic aspect of the migration topology. In particular, the problem is the fact that, since each island receives and broadcasts a migration after every generation, then in case of fairly big number of islands, each one of them will be receiving too many migrations from all the other islands. As a result, an island at the end of every generation will have to choose a migration to receive from a big number of migrations, arrived to that island. Otherwise it would have to receive and incorporate all the arrived migrations from other islands. This would impair the performance and cause unexpected behavior of the algorithm, since the incoming migrants would constitute a bigger part of the population, especially as the number of islands grows.

For this reason, it was decided to broadcast only the best two migrations, among the ones sent to the master node, in terms of fitness value of their fittest migrant. This way, every island would receive only two migrations to choose from and incorporate in their population. The precise number of migrations to be broadcasted, which is two, is justified and explained later in the text. This idea of broadcasting the best migrants from all the islands is also supported and implemented by Grefenstette [68] and Marin et al [7] in their papers.

In order to implement the above idea of propagating the globally best two migrations to all the islands, all the migrations, sent to the master node, needed to be arranged in a sorted list. Hence, sorting the migrations according to their fitness value would simplify the process by broadcasting the first two migrations (from the ”best” side of the sorted

list). However, as mentioned many times, the asynchronous version of migration process is used in the system, which means that sorting the migrations should not be a synchronous process. Having said that, the sorting process can not be executed at fixed points of time, instead it needs to be executed in asynchronous fashion. Thus, it was decided that the master node will sort the list of migrations in real time whenever it receives a new migration. This way, the list of migrations will always be sorted and available at any point of time.

The reason for choosing to broadcast exactly two migrations to every island is associated with two of the several cases that an island deals with whenever it receives migrations. The first one is the case where the best of the two received migrations is the one that was sent from the particular island itself. In that case it will accept the second best migration of the two in order to avoid incorporating its own migrants. Besides, accepting the second one contributes to the island's diversity maintenance, since it provides individuals from another island [38]. This way, in case of accepting the second best migration, it helps the island to avoid premature convergence, despite the lower fitness value it provides. The other case is where the best of the two migrations happens to have the same fitness value with that of the fittest individual of the island's last generation. In this case also the island will choose the second migration to incorporate for the same reason of diversity maintenance.

Given these points, it was decided to use request-response communication paradigm in order to realize the above methodology of migration. More particularly, the Remote Procedure Call (RPC) communication pattern, which resembles the client-server pattern, was chosen to represent this communication paradigm. The reason for choosing this type of communication paradigm is the fact that it was considered the most suitable one for the described methodology of migration. To explain, since this methodology is actually a modified version of the broadcast topology, the RPC communication pattern was considered the most applicable one to realize these modifications.

In RabbitMQ, RPC is one of many communication patterns used to support interprocess communication and it is primarily based on the request-response communication model. Hence, using RPC the island sends the migration to the master node as a request and receives the best two islands as a response. This communication procedure is analyzed in more depth in the next section. Furthermore, it is important to realize that the master node sorts the migration list right before sending the two best migrations. With that being said, the communication between each island and the master node is dictated by the following procedure:

1. An island sends its migration as a request to the master node.
2. The master node receives the migration and immediately sorts the migration list by putting the received migration in the correct position, according to its fitness value.
3. The master node sends the first two migrations from the list (which are the best).
4. The island receives the two migrations as a response from the master node and chooses one to incorporate in its population.

The third and fourth steps should be further analysed in order to examine all the cases of the communication procedure that occurs during a migration operation. As such, regarding the third step, the master node actually deals with one of the following three possible cases.

- The case where the master node has not yet received any migration from any island. In that case, whenever it receives the first migration from an island, it responds with an empty message. This happens so that the master node does not send back that island's own migration, since there are no other received migrations.
- The case where the master node has only one migration in its list and receives a request for migrations from an island (which will be explained later in the text). In that case, it will send only one migration, since it is the only one in its current list of migrations.
- The last case is the typical one where the master node has more than one migrations in its list and thus send two migrations as a response whenever an island sends a migration or a request.

As for the fourth step, the typical case is where an island receives two migrations from the master node as a response and it chooses one to incorporate into its population. In that case, the main criterion is to choose the migration that provides the fittest individual of the two. However, there are other cases that an island needs to deal with whenever it commences a migration operation. More specifically, each island will face one of the four following cases:

- The case where the island happens to send the globally best migrant or where the best of the two migrations it received has the same fitness value with that of the fittest individual of the island's last generation. In this case, as already mentioned, the island incorporates the second best migration of the two received.
- The typical case where the island sends a migration as a request and receives two migrations as a response and chooses the best one, according to the main criterion mentioned above. In this case, the best of the two migrations is not the one sent from the same island, as in the previous case.
- The case where the island does not send any migration, because the fittest individual of the current generation is not better than the one of the previous generation, as explained in the migration frequency parameter. In this case, instead of a migration, the island will send a request to the master node in order to only receive migrations from the other islands. Afterwards, upon receiving two migrations as a response from the master node, the island operates the same way as in the two previous cases.
  - There is a subcase in this occasion, where only one migration is sent from the master node and thus that migration is the only one it can incorporate. Even so, the island in this case does not incorporate that one migration. The reason for this is the fact that, according to the previous explanation, when an island

makes a request, it means that it has already sent at least one migration. This means that the one migration that the island has received as a response to its request is the one the island itself previously sent. Thus, the island proceeds to the next generation without incorporating any migrants.

- The case where the island is the first one to launch the migration process. In this case, after sending a migration or a request for migrations, it will get an empty response, because, as previously explained, there are no other migrations from other islands to receive. Consequently, the island will continue to the next generation without embodying any migrants.

In summary, whenever an island typically sends a migration to the master node, it either receives two migrations as a response or none. However, whenever it sends a request for migrations, it either receives two migrations or one, which it does not incorporate anyway.

Given these points, the RPC communication pattern, described so far, is illustrated in the figure below, showing the communication procedure in an abstract manner.

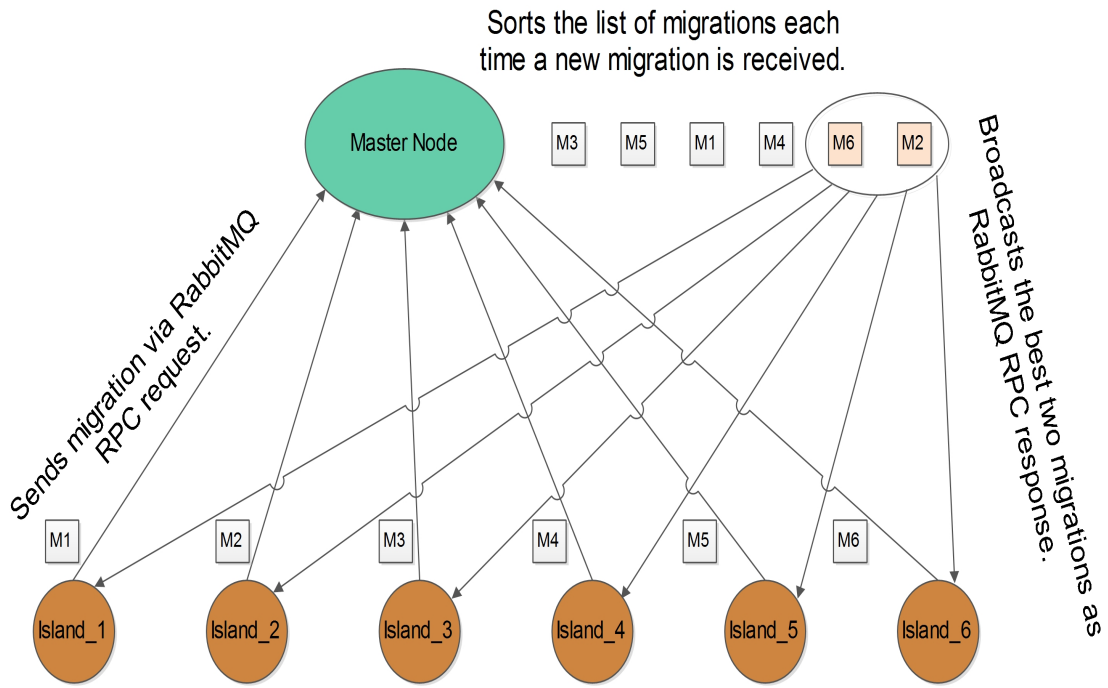


Figure 7.4: RPC communication pattern as the underlying implementation of the broadcast migration topology.

Finally, one last thing that needs to be pointed out is that the term ”migration”, which is used in this section, will be expressed as an object in the ”Implementation” phase. That is, in this iteration a new class will be developed, the ”MigrationMessage” class. This class will represent the migration entity, that is being exchanged between the islands of the algorithm and the master node.

7.2.3.1 RPC Application to Migration Methodology

As mentioned in the previous section, the communication between islands and master node, which actually facilitates a migration operation, will be implemented by the RabbitMQ's RPC pattern. In general, Remote Procedure Call, also known as a subroutine call, is a high-level interprocess communication paradigm typically used in distributed systems and client-server based applications [69]. This technique is also used in RabbitMQ communication as another message exchange pattern. With that being said, a simple implementation of this pattern was enough to uphold the migration process of the framework's Island Model GA. Therefore, the RPC pattern designed and applied to the algorithm's migration operations is illustrated in the figure below.

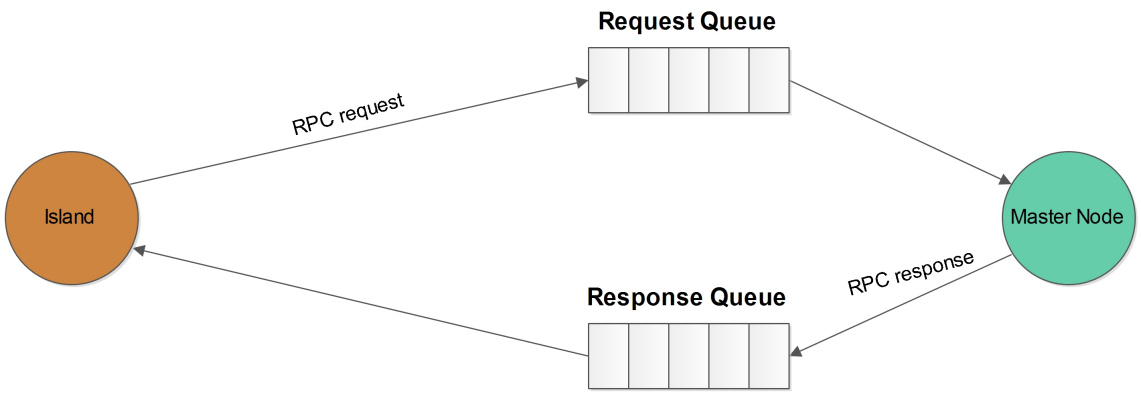


Figure 7.5: RPC communication pattern for the migration process.

As can be seen in the pattern above, there is a separate queue for both request and response messages of the communication procedure. Both of these messages usually contain the migrants to be sent and some additional information for the communication process. It is also important to note that the two queues are securely located in the RabbitMQ broker, ensuring the reliability of the software system.

Also, something that ensures this reliability even more is the involvement of the "correlationId" property, which is part of the additional information mentioned before and it represents a unique value. The role of this property is to identify the correct response that correlates to a request. This eliminates the problem of false association between a request and a response and makes the communication more robust. With that being said, every time an island sends an RPC request, it creates a random unique value, puts it in the "corralationId" property and assigns it to that request. Afterwards, when a response is arrived, its "corralationId" property is compared to that of the request last sent, if they are the same then the response is validated, otherwise it is discarded. Furthermore, apart from the "correlationId" value, the island also sends, in a property called "replyTo", information that specify the exact queue where the response message should be sent. In other words, it sends the name or ID of the response queue, shown in figure 7.5, ensuring the correct routing of messages from the master node.

7.3 Implementation

For the development process of this iteration, the top-down approach was used. This approach is a programming paradigm that suggests to firstly tackle the high level parts of the development process, which is usually to design a class diagram or a system architecture. As such, the first thing to do was to structure and organize the software components properly in order to build the software architecture, designed in the previous phase. By doing so, the first requirement of the iteration, which is to upgrade the system into a software stack, was fulfilled. The next task to do was to implement the migration policy, defined in the "Analysis" phase. Since this task included implementing all the parameters of the migration policy, eventually it fulfilled the requirement of the migration topology as well. At this point it should be noted that from all these parameters, only the implementation of the migration topology was analyzed in this section. The reason for this is that, as already mentioned, all the other parameters were too straight forward to implement. Finally, the last task to do was to incorporate the WebSocket technology and implement a real-time communication between the GUI and the master node of the system. This way, the requirement of the real-time behavior was also accomplished. One last thing to mention before proceeding with the details of implementation is that all the code (files and classes) of the final iteration can be found in the appendices of the paper.

7.3.1 Software Architecture Implementation

In order to properly structure the software components of the system and ultimately build the architecture shown in figure 7.2, a specific configuration file needed to be created. This configuration file is the docker-compose.yml file, which is used to create multi-container applications in Docker environment. More particularly, the docker-compose.yml file defines the software components and the number of their containers that will be used in the application. It defines them by instructing the Docker engine to download and utilize the Docker images (packaged applications) that correspond to those software components. Therefore, the docker-compose.yml file that was used to construct the software architecture, designed for the system, is depicted below.

```
version: "3.7"

services:
  master:
    image: dyssos29/express-server:6
    deploy:
      replicas: 1
      restart_policy:
        condition: on-failure
      placement:
        constraints: [node.role == manager]
    ports:
      - "9001:3005"
    networks:
      - webnet
```

```
16    simulation:
17      image: dyssos29/ubuntu-processing:9
18      deploy:
19        replicas: 2
20        restart_policy:
21          condition: on-failure
22        placement:
23          constraints: [node.role == worker]
24      ports:
25        - "5901:5901"
26      networks:
27        - webnet
28    visualizer:
29      image: dockersamples/visualizer:stable
30      ports:
31        - "8080:8080"
32      volumes:
33        - "/var/run/docker.sock:/var/run/docker.sock"
34      deploy:
35        restart_policy:
36          condition: on-failure
37        placement:
38          constraints: [node.role == manager]
39      networks:
40        - webnet
41    ody-rabbit:
42      image: lovelysystems/rabbitmq-swarm-cluster
43      hostname: "{{.Service.Name}}.{{.Task.Slot}}.{{.Task.ID}}"
44      deploy:
45        restart_policy:
46          condition: on-failure
47        update_config:
48          parallelism: 2
49          delay: 10s
50        placement:
51          constraints: [node.role == worker]
52      environment:
53        - RABBITMQ_ERLANG_COOKIE=abc
54        - RABBITMQ_USE_LONGNAME=true
55        - RABBITMQ_MNESIA_DIR=/var/lib/rabbitmq/mnesia
56        - RABBITMQ_PLUGINS_EXPAND_DIR=/var/lib/rabbitmq/mnesia/plugins-
    expand
57        - SERVICE_NAME={{.Service.Name}}
58        - SLOT={{.Task.Slot}}
59        - MASTER_SLOT=1
60      ports:
61        - "5672:5672"   # amqp
62        - "15672:15672" # web ui
63      networks:
64        - webnet
65
66  networks:
67    webnet:
```

Listing 7.1: Docker-compose.yml file of the software system.

The “services”, written in the snippet above, are the Docker services, which represent the software components of the system as already mentioned. As such, there are four such services in the docker-compose.yml file: master, simulation, visualizer and ody-rabbit. Each one of them are defined in detail, using the appropriate parameters. These parameters constitute a very important part of a docker-compose.yml file, because they are the ones that determine how the software will distribute its components over the available distributed system. Apart from this though, they are also used to determine some other configurations related to each of the services involved. That is, these configurations, except for the mandatory one which is the “image”, may be the “deploy”, “ports”, “networks” and “volumes”, as shown in the same code snippet.

As for the roles of these services, the “simulation” service is the one that represents the software component where the island containers run. This software component from now on will be called ”simulation island”, because till now the name of the container was used to address this component (island container). As such, this service will spawn as many containers as the number of islands the algorithm will be configured to operate on. Next, the “master” service represents the software component of the master node. Then, the “ody-rabbit” service represents the RabbitMQ message broker of the system, which serves as a middleware through which all the internal messages pass. Lastly, the “Visualizer” service is an open-source tool, that can be found in Docker Hub and is mainly used to display all the Docker services running on a Docker swarm. As such, in this case it is used for testing and monitoring purposes, as described later in the “Testing” phase.

Furthermore, the described docker-compose.yml file is used to launch the multi-container software of the framework, which now is a full-fledged software stack, on the available underlying distributed system. More specifically, in the ”manager” node, the Docker-cli command “docker stack deploy -c docker-compose.yml evolutionapp” launches the software stack on the Docker Swarm cluster. The “evolutionapp” word in the command is the name of the stack, defined each time the command is written. This way, the Docker engine deploys the framework on the given cluster, using this file.

Overall, this docker-compose.yml file facilitates the main structure or skeleton of the software stack and servers as a foundation for further development. It can also be said that it actually implements the Island Model GA of the system, since it structures the software in an appropriate and convenient way for the particular algorithm.

7.3.2 Migration Communication Implementation

For the implementation of the algorithm’s migration topology, which is the broadcast topology, the software components that were involved are the master node and the simulation island. As such, this topology was facilitated by implementing the appropriate communication between those two components. More specifically, this communication was implemented in accordance to the data exchange instructions of the “Design” phase.

7.3.2.1 Simulation Island

On the part of the simulation island component, the connection class “ClientIsland” was further implemented in order to establish the required communication with the master node. More particularly, additional code was written on that class to properly send and receive migrations from the master node, which are the first and fourth steps of the communication procedure described in the previous phase (section 7.2.3). Thus, the code for sending a migration to the master node is illustrated in the code snippet below.

```
1  public void sendMigration(MigrationMessage migration) throws
     IOException, NullPointerException
2  {
3    receivedMigrationMessages = new MigrationMessage[]{};
4    correlationId = UUID.randomUUID().toString();
5    AMQP.BasicProperties properties = new AMQP.BasicProperties
6                .Builder()
7                .correlationId(correlationId)
8                .replyTo(responseQueueName)
9                .build();
10   String json = serializeMigrationMessage(migration);
11
12   channel.basicPublish("", REQUEST_QUEUE_NAME, properties, json.
     getBytes());
13 }
```

Listing 7.2: Code for sending a migration to the master node.

As can be seen from the code above, the “correlationId” and “replyTo” properties are utilized for reliability and robustness purposes, described in section 7.2.3.1. It can also be noticed that the migration object being sent is firstly converted into JSON format. With regard to that, there are two reasons that the migrations of the system’s core algorithm are exchanged in that format. The one is the fact that since a Node.js server is used in the system, it becomes much easier to convert data in JSON format into server’s native Javascript objects, as the Node.js language is actually Javascript [70]. This way, the complexity of the code is reduced and thus it becomes less error prone. The other reason is the fact that JSON in general is a lightweight format [71], which contributes to reducing the communication overhead inside the framework’s system.

Moreover, in the same code snippet, the two queues, involved in the system’s RPC communication pattern, can be noticed. More particularly, these two queues are the “Request Queue” and “Response Queue”, mentioned in section 7.2.3.1. To explain, the final variable “REQUEST_QUEUE_NAME” represents the queue that receives the migrations from all the islands and is actually the “Request Queue” of the previously mentioned RPC pattern. As for the “Response Queue”, it is represented by the variable “responseQueueName”, which is assigned to the “replyTo” property. It is important to note here that the name (or ID) of the “Request Queue” is common for all the islands of the Island Model GA and does not change, which is the reason it is assigned to a final variable. The name of the “Response Queue”, on the other hand, is unique for every island. This strategy was considered the most suitable one to express and implement the RPC communication pattern, designed in the previous phase.

However, apart from sending migrations, an island may also only request migrations from the other islands without sending its own, as already explained in section 7.2.3. In this case, the island sends an empty JSON data to the “Request Queue”, as can be seen from the code snippet below.

```
1  public NeuralNetwork[] requestMigrations() throws Exception
2  {
3    NeuralNetwork[] receivedNNs = {};
4    receivedMigrationMessages = new MigrationMessage[]{};
5    correlationId = UUID.randomUUID().toString();
6    AMQP.BasicProperties properties = new AMQP.BasicProperties
7                .Builder()
8                .correlationId(correlationId)
9                .replyTo(responseQueueName)
10               .build();
11   String request = "{}";
12
13   channel.basicPublish("", REQUEST_QUEUE_NAME, properties, request.
     getBytes());
14   receivedNNs = getReceivedNeuralNetworks();
15
16   return receivedNNs;
17 }
```

Listing 7.3: Code for requesting migrations from the other islands.

In line 11 of the code above is that empty data, which is sent exactly the same way as in the previous case, using the properties “correlationId” and “replyTo”. As already known, this procedure happens in case where the fittest individual of the last generation is not better than the one of the previous generation.

As for receiving migrants, the method “getReceivedNeuralNetworks” was implemented for that purpose, which can be seen in line 14 of the previously shown code. The name of this method relates to the design and implementation of the example simulation, which uses Artificial Neural Networks (ANNs) to evolve its population. To explain, each organism in that simulation uses an ANN as its brain, which guides its behavior and evolution overall. As such, what is really being migrated and incorporated into the population of an island is the “brain” of that organism, which is the reason that method is named this way. Hence, below is the illustration of that method.

```
public NeuralNetwork[] getReceivedNeuralNetworks() throws Exception
{
  NeuralNetwork[] receivedNNs = {};

  if (checkIfMigrantsArrived())
  {
    if (bestFitnessValue == receivedMigrationMessages[0].
  getBestFitnessValue())
    {
      if (receivedMigrationMessages.length > 1)
      {
        receivedNNs = receivedMigrationMessages[1].
  getMigrantNeuralNetwors();
```

```
          System.out.println("Chose the second best migration, which is : " + receivedMigrationMessages[1].getBestFitnessValue());
        }
        else
          System.out.println("The island received only one migration message and did not choose it because it had the same fitness value with the island's best fitness value so far.");
      }
      else
      {
        receivedNNs = receivedMigrationMessages[0].getMigrantNeuralNetwors();
        System.out.println("Chose the best migration, which is : " + receivedMigrationMessages[0].getBestFitnessValue());
      }
    }

    return receivedNNs;
  }
```

Listing 7.4: Code for receiving migrations from the other islands.

From the code above it can be noticed that before anything else, the method checks if any migration indeed arrived. For that purpose it utilizes the “checkIfMigrantsArrived” method, which will be explained later in the text. After that, the method examines the fitness value of the fittest migrant arrived to check if it is the typical case where it is not the one sent from the same island. If that is not the case, then it checks if the migrations arrived are two or one so that in case they are two, it chooses the second one to incorporate into the population. Otherwise the island container continues to the next generation without incorporating any migrant.

Regarding the “checkIfMigrantsArrived” method, that was previously mentioned, it is used as a utility method in the “ClientIsland” class. The purpose of this method is to check the proper delivery of messages, that contain migrations, from the RabbitMQ broker, using the “correlationId” property. This is achieved using the “processMessage” method, that will be described later in the text. With that being said, in case the message is properly delivered, it also checks if that message contains any migration. If that message is empty, then no migrations are received and thus the method returns “false” as in the case of improper delivery. Thus, the code of this method is illustrated below.

```
  private boolean checkIfMigrantsArrived() throws Exception
  {
    BlockingQueue<String> response  = new ArrayBlockingQueue<String>(1);

    channel.basicConsume(responseQueueName, consumer);
    QueueingConsumer.Delivery deliver = consumer.nextDelivery();

    if (processMessage(deliver, response))
    {
      long deliveryTag = deliver.getEnvelope().getDeliveryTag();
      channel.basicAck(deliveryTag, true);

```

```
13        if (receivedMigrationMessages.length == 0)
14          return false;
15        else
16          return true;
17      }
18
19      return false;
20    }
```

Listing 7.5: Code for checking the proper delivery of RabbitMQ messages and if they include any migration.

It is also important to note that in case the message, sent from the RabbitMQ broker, is delivered properly, then the method sends an acknowledgement back to the broker to notify it about that. This way, the RabbitMQ broker can discard that message from its memory. In general, the acknowledgement in that case is sent regardless if any migration is included in the delivered message. The command that sends that acknowledgement can be seen in line 11 of the code snippet above.

As for the previously mentioned "processMessage" method, it is also used as a utility method in that class. This method is the one that correlates the received messages to the ones that were sent, using the "correlationId" property. It is also the one that extracts the data from the received message and in case of failure to do that, it returns false, as in the case of failed correlation. The rest of the code are intended for monitoring and debugging purposes. Below is the illustration of this method.

```
1   private boolean processMessage(QueueingConsumer.Delivery delivery,
     BlockingQueue<String> response) throws Exception
2   {
3     if (delivery.getProperties().getCorrelationId().equals(
     correlationId))
4     {
5       //System.out.println("Correct correlation id.");
6       if (response.offer(new String(delivery.getBody(), "UTF-8")))
7       {
8         String responseStr = response.take();
9         if (responseStr.equals("{}"))
10          System.out.println("The message is empty.");
11        else
12        {
13          receivedMigrationMessages = deserializeMigrationMessages(
     responseStr);
14          if (receivedMigrationMessages.length == 1)
15            System.out.println("Received the following migrants: " +
     receivedMigrationMessages[0].getBestFitnessValue());
16          else
17            System.out.println("Received the following migrants: " +
     receivedMigrationMessages[0].getBestFitnessValue() + ", " +
     receivedMigrationMessages[1].getBestFitnessValue());
18        }
19
20        return true;
21      }
```

```
      else
      {
        System.out.println("Error: the insertion to the blocking queue was not successful.");
        return false;
      }
    }
    System.out.println("Wrong correlation id.");
    return false;
  }
```

Listing 7.6: Code for correlating RabbitMQ messages and extracting data from them.

Furthermore, as already mentioned, the “MigrationMessage” class was developed in this iteration to realize the migration entity that is being sent from one island to the other. The objects of this class though, apart from the migrants (ANNs), include also some additional information. Namely, they also include information about the fitness value of the best migrant they carry, the number of the generation their migrants come from and the ID of the container that accommodates the island of those migrants. All these can be seen in the code snippet shown below.

```
class MigrationMessage
{
  private String nodeId;
  private int generationNumber;
  private int bestFitnessValue;
  private NeuralNetwork[] migrantsNN;

  public MigrationMessage(){}

  public MigrationMessage(String nodeId, int generationNumber, int bestFitnessValue, NeuralNetwork[] migrantsNN){
    this.nodeId = nodeId;
    this.generationNumber = generationNumber;
    this.bestFitnessValue = bestFitnessValue;
    this.migrantsNN = migrantsNN;
  }

  public String getNodeId(){
    return nodeId;
  }

  public int getGenerationNumber(){
    return generationNumber;
  }

  public int getBestFitnessValue(){
    return bestFitnessValue;
  }

  public NeuralNetwork[] getMigrantNeuralNetwors(){
    return migrantsNN;
  }
}
```

Listing 7.7: ”MigrationMessage” class that represents the migration entity.

Finally, in order for the simulation island software component to be properly synchronized with the other components at the startup time, the “wm_startup.sh” shell script needed to be extended. To explain, whenever a container of the simulation island component (“simulation” service) is launched, it is programmed to automatically connect to the RabbitMQ message broker component (”ody-rabbit” service). However, the message broker component at that exact time may not be launched or ready yet. This may happen because when a docker-compose.yml file is used to launch a multi-container software stack, the Docker services specified in that file are not built in any specific order.

Even so, Docker technology provides commands like “depends_on”, “links” and others, which can regulate the dependency order of the services, employed in the same docker-compose.yml file [72]. However, it needs to be clarified that none of these commands checks if the container of a service is ready to operate as needed. Instead, they check if the container is running or not, without knowing if it is still in “booting” phase. Put differently, none of the previously mentioned commands waits for the container to complete its boot phase and get into the “ready” state, for availability reasons [72]. In spite of that, there are tools that can achieve this waiting process, such as “wait-for-it”, “wait-for” and “dockerize”. For the current project, the “dockerize” tool was used to enable the simulation island wait for the RabbitMQ broker to be ready. As such, this tool was installed in the simulation container and the “wm_startup.sh” shell script was extended by incorporating the command shown in line 13 of that script, which is illustrated below.

```
1  #!/usr/bin/env bash
2  ### every exit != 0 fails the script
3  set -e
4
5  echo -e "\n------------------ startup of Xfce4 window manager
       ------------------"
6
7  ### disable screensaver and power management
8  xset -dpms &
9  xset s noblank &
10 xset s off &
11
12 /usr/bin/startxfce4 --replace > $HOME/wm.log &
13 dockerize -wait tcp://ody-rabbit:5672 -timeout 50s &&
14 /headless/Desktop/ExecutableSketch/SimpleOrganisms_Evolution &
15 sleep 1
16 cat $HOME/wm.log
```

Listing 7.8: The ”wm_startup.sh” shell script used by the simulation container to start programs on boot.

7.3.2.2 Master Node

Further implementation of this component was needed to establish the RPC based communication with the simulation island, which is vital for the development of the migration topology. As such, the "app.js" file was further developed to be able to receive migrations from all the island, sort them according to their fitness value and send as a response the top two of them. With that being said, the code written for these tasks is illustrated below.

```
promiseRetry(function (retry, number) {
  console.log('Attempt number: ' + number);

  return amqp.connect('amqp://ody-rabbit')
      .catch(retry);
}).then(function(conn) {
  console.log('Connected to rabbitmq.');
  return conn.createChannel();
}).then(function(ch) {
  return ch.assertQueue(requestQueue, { durable: true}).then(function(
    ok) {
    return ch.consume(requestQueue, function(msg) {
      if (msg !== null) {
        let migrationStr = msg.content.toString();
        console.log(" [x] Received %s", migrationStr);
        processRPCrequest(migrationStr);

        console.log('Length of the migration objects array: ' +
    receivedMigrations.length);
        io.sockets.emit('migration', receivedMigrations);

        let jsonToSend = JSON.stringify(getMigrationsToSend());
        console.log('Json that is being sent: ' + jsonToSend);

        ch.sendToQueue(msg.properties.replyTo, Buffer.from(jsonToSend),
     {correlationId: msg.properties.correlationId});
        ch.ack(msg);
      }
    });
  });
}).catch(console.warn);
```

Listing 7.9: Main function of the master node.

As can be seen from the code above, the master node initially tries to connect to the RabbitMQ broker, using the "promiseRetry" function. That function is actually a Node.js module, which is used to retry functions until they return the desired results. In this case, that module is used to connect to the message broker, by retrying as many times as needed until the connection is established. This process is similar to the one that the simulation container uses in order to connect to the message broker through multiple attempts. The difference is that the latter one uses the "dockerize" tool for that task.

After establishing a connection with the message broker, each time the master node receives a migration from an island, it processes it by using the "processRPCrequest"

method, which will be described later in the text. This process mainly has to do with the sorting procedure. After that, the master node sends the previously sorted list to the front-end (web page), using the WebSocket technology, as will be described later in the text. Lastly, the master node retrieves the right migrations from the sorted list of all the migrations and sends them to the island from which the most recent migration was sent. The retrieval of the migrations to be sent is carried out by the "getMigrationsToSend" method, which will also be described later in the text.

Regarding the last step, in order for the master node to send the response of those migrations properly, it also needs to utilize the "correlationId" and "replyTo" properties. This can be seen in line 23 of the listing 7.9. By doing so, it sends the response to the correct island, using the "replyTo" property, and correlates the response to the correct corresponding request, using the "correlationId" property. Moreover, as can be seen in line 24 of the same code snippet, the master node sends an acknowledgement to the RabbitMQ broker to notify it about the successful delivery of the request it replies to.

As for the previously mentioned "processRPCrequest" method, its main task is to check if the message is a migration from an island or a request from it. The latter case is related to the occasion where the island does not send any migration, instead it requests migrations from the other islands. Thus, in case the received message is an empty JSON object, it is a request to which the master node responds normally by sending the top two migrations. Otherwise, the master node puts the received migration to the sorted list of all the migrations and then responds the same way as in the previous case. All these can be seen in the code snippet illustrated below.

```
function processRPCrequest(aMigrationString)
{
  let migrationObj = JSON.parse(aMigrationString);

  if (aMigrationString != '{}')
    addToSortedList(migrationObj);
  else
  {
    isRequest = true;
    console.log('Received empty message');
  }
}
```

Listing 7.10: Code for checking if the received RabbitMQ message is a migration or a request for migrations.

The process of inserting that migration into the sorted list of migrations should be carried out in a proper manner, so that the list remains sorted. For that purpose, the "addToSortedList" method was created, which can be seen in line 6 of the above code. This method is nothing more than a typical algorithm of adding a value to a sorted list and its code is shown in the code snippet below.

```
function addToSortedList(migration)
{
  if (receivedMigrations.length == 0)
    receivedMigrations.push(migration);
```

```
  else
  {
    console.log('length: ' + receivedMigrations.length);
    if (receivedMigrations[receivedMigrations.length - 1].
   bestFitnessValue > migration.bestFitnessValue)
      receivedMigrations.push(migration);
    else
    {
      for(let i = 0; i < receivedMigrations.length; i++)
      {
        if (receivedMigrations[i].bestFitnessValue == migration.
   bestFitnessValue)
          break;
        else
        {
          if (receivedMigrations[i].bestFitnessValue < migration.
   bestFitnessValue)
          {
            receivedMigrations.splice(i, 0, migration);
            break;
          }
        }
      }
    }
  }
}
```

Listing 7.11: Code for properly inserting a migration to the sorted list of all the migrations.

Finally, regarding the "getMigrationsToSend" method, as already noted, it undertakes the task of retrieving the migrations that will be sent as a response to an island. Basically, there are three cases when it comes to sending a response:

- The case where the island is the first one to send a migration and thus it will get an empty response. The reason for this is that since there are no other migrations in the master node, that first migration is the only one that can be sent as a response. However, sending back its own migration as a response is not a viable option, which is why instead an empty JSON object is sent back.
- The case where an island requests migrations from other islands and there is only one migration in the sorted list at the time. In this case, that one migration is sent as a response.
- The typical case where there is more than one migrations in the sorted list of the master node. In this case, the top two migrations are sent as a response, either the island sent a migration or simply a request.

The code of this method is illustrated in the code snippet below.

```
function getMigrationsToSend()
{
  let migrationsToSend = [];
```

```

  if (receivedMigrations.length == 1 && !isRequest)
    return {};
  else
  {
    if (receivedMigrations.length == 1)
      migrationsToSend.push(receivedMigrations[0]);
    else
      migrationsToSend.push(receivedMigrations[0], receivedMigrations
    [1]);
  }

  return migrationsToSend;
}
```

Listing 7.12: Code for retrieving migrations to send as a response to an island.

7.3.3 WebSocket Communication Implementation

In order to implement the real-time communication between the system's master node and web page, the code of both software components needed to be extended in a proper way. As such, in the master node case, the code of the app.js file was extended as needed. More specifically, in that file, code was written to wait for WebSocket connections, which is shown in the code snippet below.

```
const io = socket(server);
io.on('connection', function(socket){
  console.log('Socket connection established with status: ' + socket.
    connected);
});
```

Listing 7.13: Code for waiting WebSocket connections in the app.js file.

Moreover, in the same file code was written to send data to any client that is connected to the server through the WebSocket protocol. More specifically, the system's web page, which represents its front-end part, is the client in this case that connects to the Node.js server of the back-end part and receives these data. This code can be seen in the code snippet below.

```
    return ch.consume(requestQueue, function(msg) {
      if (msg !== null) {
        let migrationStr = msg.content.toString();
        console.log(" [x] Received %s", migrationStr);
        processRPCrequest(migrationStr);

        console.log('Length of the migration objects array: ' +
    receivedMigrations.length);
        io.sockets.emit('migration', receivedMigrations);
```

Listing 7.14: Code for sending data to the system's web page through the WebSocket connection.

In the code illustrated above, the last line is actually the one that executes the task of sending data to the web page. It can also be noticed that this data is the "receivedMigrations" variable, which is the one that contains all the migrations sent to the master node arranged in a sorted list. This means that whenever there is a change in that list, the master node updates the web page by sending the whole list again, instead of sending only the change. The reason for this strategy is the fact that in the latter case, more complex code and additional computations would be required in order to apply that change on the list located in the web page.

In general though, every time a new client is connected to the Node.js server (master node), the server renders a web page, using EJS (Embedded JavaScript) template, fills it with data and sends it to that client. The data, that the rendered web page is filled with, is the same data the server sends to its clients whenever there is a change in the list of migrations, as previously explained. The code of this rendering procedure is located in the "index.js" file which is illustrated below.

```
const express = require('express');
const router = express.Router();

/* GET home page. */
router.get('/', function(req, res, next) {
  const migrations = req.app.locals.migrations;
  res.render('index', { title: 'Evolution Process', migrations:
   migrations});
});

module.exports = router;
```

Listing 7.15: Code for rendering the web page before sending it to a client.

As for the front-end part, code was written to connect to the Node.js server (master node) and also to receive data from it. The code written for both of these tasks is located in the file socket.js and is illustrated below.

```
const socket = io.connect('http://192.168.99.100:9001');
var migrations = document.getElementById('migrations');

socket.on('migration', function(data){
    migrations.innerHTML = '';
    data.forEach(function (element) {
        migrations.innerHTML += '<p>Island id: ' + element.nodeId + ',
   Generation: ' + element.generationNumber + ', Best individual: ' +
   element.bestFitnessValue + '</p>';
    })
});
```

Listing 7.16: Code for connecting with the master node and receiving data from it.

This socket.js file is a Javascript file used in the HTML code of the web page, which is the reason for putting all these code into that file. With that being said, the first line of the code above executes the task of sending the connection request to the Node.js server, by using its IP address and port number. The rest of the code executes the task of receiving

data from that server and incorporate it into the HTML content of the web page. This data, as mentioned before, is a list of all the migrations sent from all the islands to the master node, sorted according to their fitness value.

Regarding the HTML code of the web page, as already mentioned, it is developed according to the EJS templating engine used by the Node.js server. As such, the HTML code of the front-end's web page is located in the "index.ejs" file. The code snippet below shows the body of that HTML code.

```
<body>
  <h1>Island Model PGA Framework</h1>
  <h2>Results of the <%= title %>.</h2>
  <h3>Real-time Migrations.</h3>
  <div id="migrations">
    <% if (migrations.length > 0){ %>
      <% migrations.forEach(function(item){ %>
      <p>Island id: <%=item.nodeId %>, Generation: <%=item.
 generationNumber %>, Best individual: <%=item.bestFitnessValue %></
 p>
      <% }); %>
    <%  } %>
  </div>
  <script src="/socket.js"></script>
</body>
```

Listing 7.17: HTML body of the front-end's web page.

Finally, as can be observed, the implemented WebSocket communication between these two software components is unidirectional, from the master node to the web page. The reason for this is the fact that currently there is no need for the system to use other types of communication, according to its objectives and the requirements specified in the "Analysis" phase.

7.4 Testing

Since it is the last iteration where the software system has been fully developed, some objectives of the project will also be checked and validated along with the requirements specified in this iteration. That is, one of the requirements, specified in the "Analysis" phase, will be tested in terms of two objectives, defined and assigned to this project in the "Introduction" chapter. More specifically, the requirement of making the software system a multi-container software stack will be tested by validating the objectives of scalability and fault-tolerance. The reason for this is that, as already noted in the "Analysis" phase, the Docker technology of the upgraded software stack, due to its nature, will automatically grant the system the traits of scalability, fault-tolerance and portability. Hence, the successful validation of scalability and fault-tolerance objectives will in turn indicate the successful achievement of the requirement of upgrading the system into a software stack. Regarding the second requirement of this iteration, which is the real-time behavior of the software, it will not be tested in this phase. Meaning, the objective of real-time behavior

will not be examined during the whole development of this project. Moreover, neither the portability objective will be tested in this project. The reasons for that will be explained in the "Evaluation" chapter. As for the third requirement, which is actually the implementation of the migration operation, it will be tested purely in terms of the chosen migration policy of the system's Island Model GA. Put differently, only the algorithmic aspect of the framework will be examined in that case.

7.4.1 Test Strategy

As in the preceding two iterations, the test strategy of this iteration is described in relation to the previous test strategy. Meaning, only the parts of the strategy, that are different or carried out differently from the previous one, will be included in this section. This is done so that all the other parts, that are the same, are not repeated again. With that being said, the test strategy used for this iteration is the following one:

- **Scope and overview**: The software under test this time is the final software of the framework, as it is the last iteration. As such, the objectives of the project will also be involved in the testing process, as previously explained. Therefore, there will be three testing activities in this phase. The first testing activity will examine the algorithmic aspect of the frameworks software. Then, the other two activities will be concerned with examining two of the project's objectives, the successful validation of which will mean the successful fulfillment of the first requirement of this iteration. Namely, the requirement of upgrading the framework's software into a multi-container distributed software stack will be fulfilled in case those two objectives are successfully validated. With that being said, these three activities are listed in more depth below:

    1. The first testing activity will investigate how well the Island Model GA was applied to the software of the framework. In other words the correctness of that genetic algorithm will be tested in terms of its migration policy. This will be carried out by monitoring the terminals of both the master node and an island container, in order to examine the migrations sent and received by these two components.
    2. The second testing activity will test the scalability of the framework by scaling one of its software components. More specifically, the simulation island component of the system will be scaled by one container.
    3. The last activity will test the fault-tolerance feature of the framework, by shutting down one computing node of the cluster. That computing node will be one of the two nodes that run containers of the simulation island component.

- **Test approach**:

    – Black-box testing will be used in this iteration as well. There are two reasons for choosing this testing technique, which this time are not related to the fact that the source code is too simple or small in amount. That said, the first reason

is related to the cost of testing. That is, this cost has to do with time and effort, since currently there is a fairly large amount of source code and thus inspecting all these code through white-box testing would be time-consuming. To explain, the white-box testing technique is an exhaustive and time-consuming procedure in general [73]. As such, by avoiding that testing technique, the development risk is mitigated. The second reason is the fact that since it is the last iteration, higher level of testing is intended to be used, instead of the more detailed lower level testing. Consequently, in this case black-box testing technique was preferred over the white-box technique, as the latter one is more suitable for lower levels of testing, where quality of code is examined [49].

– Functional type of testing will be used for the first testing activity and non-functional type for the other two. The reason for using the functional type of testing for the first testing activity is the fact that, as noted before, it examines the algorithmic aspect of the system. This aspect though, since it is directly related to the core algorithm of the framework, represents the main functionality of the system. Thus, taking that fact into account, that testing activity examines if the system satisfies its functional requirements, which justifies the choice of functional type of testing. As for using the non-functional testing type for the other two testing activities, the reason is the fact that the goal of these two testing activities is to validate the objectives of scalability and fault-tolerance. More particularly, the objective of scalability will be inspected by the scalability test and the objective of fault-tolerance by reliability test. These two types of tests though are considered non-functional types, which is the reason for choosing the particular type for the last two testing activities. Regarding the other types of testing, the change-related type is not applicable in this case because no modifications were applied to the system so far. Instead, the source code and the structure of the system was being upgraded till now. Meaning, most of the source code was being replaced by new code, more suitable for the system's needs at the time. Therefore, the final software does not contain any code from the previous iterations, apart from the code related to the RabbitMQ connection and the boilerplate code of Express.js framework. As such, the final version of the framework will be tested as if it is the first prototype, which is the reason change-related testing type will not be used. Also, the maintenance testing is not applicable as well, since the final software is not used extensively yet. Therefore, there are no requirements for any changes or extensions to the software of the framework and thus no maintenance was needed. As regards the structural type of testing, it was not chosen to be used for the same reasons the white-box testing technique was not preferred.

– Finally, the framework will be tested on the level of system testing, since it is its final version. That is, its compliance to the specified requirements will be examined, along with its objectives.

7.4.2 First Testing Activity

The first activity, as already mentioned, examines the correctness of the framework's Island Model GA algorithm. More specifically though, the migration policy part of that algorithm is under inspection in this activity. However, as stated in the "Analysis" phase, out of all the parameters of the specified migration policy, the migration topology parameter is the one that was the most hard to implement, since it constitutes the most complex part of the system. As such, the specified migration topology parameter is what is really being tested in this testing activity.

As already known, that migration topology was specified to be the broadcast topology. That topology though was implemented using the RPC communication pattern. Thus, the migration topology will be examined by investigating the RPC governed message exchange between an island container and the master node. As such, since this message exchange is actually the algorithm's migration operation, its investigation will be achieved by testing all the cases that both an island container and the master node face during this operation. These cases are thoroughly described in section 7.2.3 of the "Design" phase and also illustrated in tables 7.1 and 7.2.

With that being said, the master node deals with three possible cases during a migration operation. These cases are described and illustrated in the table below.

Master Node Cases			
Test Case	**Expected Results**	**Actual Results**	**Pass/Fail**
1. Receives its first migration from an island.	Responds with an empty message.	Responds with an empty message.	Pass
2. Has only one migration in its list and receives a request from an island.	Responds by sending that one migration.	Responds by sending that one migration.	Pass
3. Has more than two migrations in its list and receives a migration or a request from an island.	Responds by sending two migrations.	Responds by sending two migrations.	Pass

Table 7.1: All the possible cases that the master node may encounter during a migration operation.

An island container, on the other hand, deals with 5 possible cases whenever it receives a response from the master node during a migration operation. These cases are described and shown in table 7.2.

Island Cases			
Test Case	**Expected Results**	**Actual Results**	**Pass/Fail**
1. Sends the globally best migration or the received migration has no better fitness value.	Incorporates the second best migration of the two received.	Incorporates the second best migration of the two received.	Pass
2. The typical case where it sends a migration and receives two migrations, where the best migration has higher fitness value of the island's fittest individual.	Incorporates the best of the two received migrations.	Incorporates the best of the two received migrations.	Pass
3. Sends request for migrations, instead of a migration, and receives two migrations as a response.	Behaves according to the rules of the two previous cases.	Behaves according to the rules of the two previous cases.	Pass
4. The same case as the case 3 above, with the only difference that one migration is received, instead of two.	Proceeds to the next generation without incorporating that one.	Proceeds to the next generation without incorporating that one.	Pass
5. Sends the first migration to the master node and thus receives an empty message from the master node.	Proceeds to the next generation without incorporating any migration.	Proceeds to the next generation without incorporating any migration.	Pass

Table 7.2: All the possible cases that an island may encounter during a migration operation.

As can be seen from the two tables above, all these possible cases, that both an island container and the master node may encounter during a migration operation, were successfully covered. Meaning, in all of these cases, the software of the framework acted as it was expected to act.

In order to cover and test all these cases though, three simulation runs were needed. To explain, the example simulation, which is a Processing sketch created in the previous works of this project, was run three times on the framework in order to cover and examine all the cases. However, before these runs were made, some preconditions needed to be checked. These preconditions were not mentioned in the two tables above because they are the same for both of them. With that being said, these preconditions are two. The one is the docker-compose.yml file to be in the same directory where the Docker CLI command "docker stack deploy -c docker-compose.yml evolutionapp" will be issued. That command is used to deploy and launch the framework on any available distributed system. The other precondition is the distributed system on which the framework will be

deployed, which in this case is the swarm cluster, to be available. As for the test steps, there was only one step in this case, which is to issue that command in order to deploy the framework and make those runs. This test step was not mentioned in the above tables for the same reason of being the same for both of them.

With that being said, in order to check the two preconditions, firstly the "ls" command was issued to verify that the docker-compose.yml file is in the current directory. Then, the Docker Machine CLI command "docker-machine ls" was issued in order to verify the Docker swarm cluster, created for the framework's software to be run on it, is available for use. The execution of these commands is illustrated in the figure below.

```
File  Edit  View  Search  Terminal  Help
root@ody-Lenovo-B50-80:~/Desktop/Project/Software# ls
docker-compose.yml  EvolutionProject  IslandProgram  MasterProgram  RabbitmMQ-ClientLibs
root@ody-Lenovo-B50-80:~/Desktop/Project/Software# docker-machine ls
NAME    ACTIVE   DRIVER       STATE     URL                         SWARM   DOCKER     ERRORS
myvm1   -        virtualbox   Running   tcp://192.168.99.100:2376           v19.03.5
myvm2   -        virtualbox   Running   tcp://192.168.99.101:2376           v19.03.5
myvm3   -        virtualbox   Running   tcp://192.168.99.102:2376           v19.03.5
myvm4   -        virtualbox   Running   tcp://192.168.99.103:2376           v19.03.5
root@ody-Lenovo-B50-80:~/Desktop/Project/Software#
```

Figure 7.6: Commands that check the two preconditions of this testing activity.

As can be seen from the figure above, the docker-compose file was at the currently directory and the swarm cluster was available for use. As such, the two preconditions were successfully validated.

After checking the preconditions, the one test step of this testing activity needed to be made in order to make the first run of the example simulation. That step is to deploy the framework on the available swarm cluster, issuing the aforementioned command "docker stack deploy -c docker-compose.yml evolutionapp". The figure below illustrates this command and the resulting procedure that follows afterwards.

```
root@ody-Lenovo-B50-80:~/Desktop/Project# cd Software/
root@ody-Lenovo-B50-80:~/Desktop/Project/Software# ls
docker-compose.yml  EvolutionProject  MasterProgram  RabbitmMQ-ClientLibs
root@ody-Lenovo-B50-80:~/Desktop/Project/Software# docker stack deploy -c docker-compose.yml evolutionapp
Creating network evolutionapp_webnet
Creating service evolutionapp_ody-rabbit
Creating service evolutionapp_master
Creating service evolutionapp_sketch
Creating service evolutionapp_visualizer
root@ody-Lenovo-B50-80:~/Desktop/Project/Software#
```

Figure 7.7: Docker CLI command to deploy the software stack of the framerwork on the available distributed system.

As can be seen in that figure, after running the command, the Docker engine firstly creates the specified network for the Swarm cluster and then launches all the Docker services indicated in the docker-compose file.

After deploying the framework and making the first run of the example simulation (Processing sketch) on it, some screenshots were taken in order to capture some of the cases that were covered in this run. However, the purpose of the first screenshot was to illustrate in detail all the parts of it so that it is clear which part is what. Also, another reason for taking that screenshot was to show the successful connection of both the master node and the island container to the RabbitMQ message broker. As such, that screenshot is shown in the figure below.

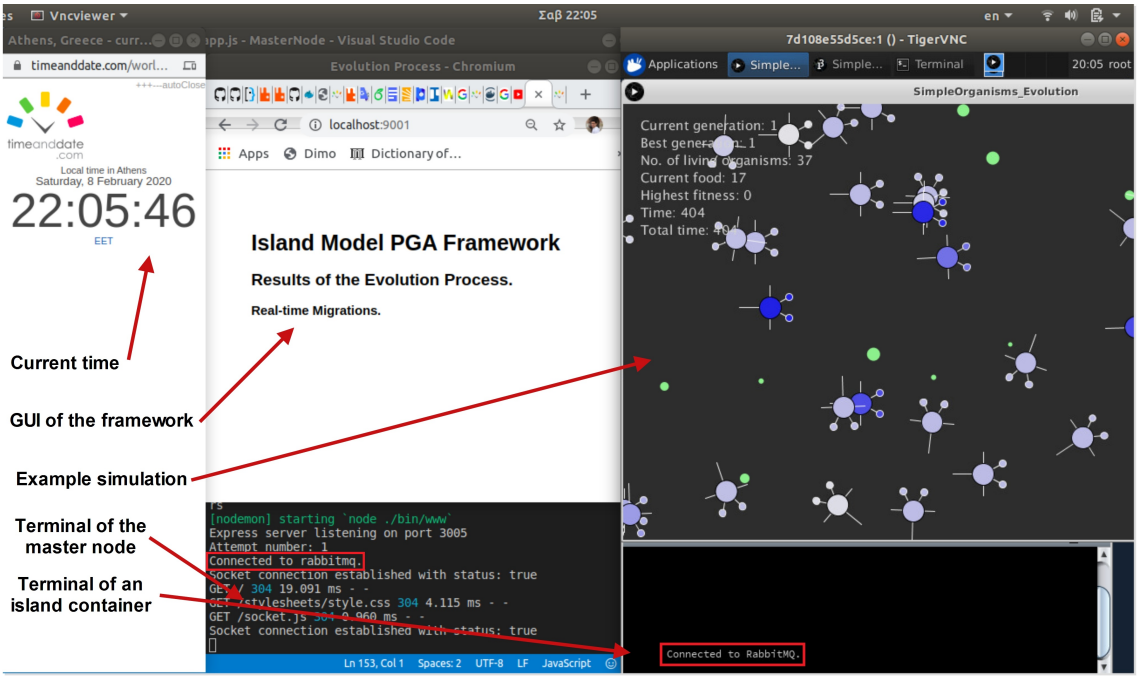


Figure 7.8: First screenshot of the simulation's first run.

In the screenshot above, apart from the successful connection to the RabbitMQ broker, five more things can be discerned:

- The example simulation of the previous works, which is a Processing sketch that runs on an island container.
- The GUI of the system, which is a web page that is being served by the Node.js server (master node).
- The terminal of the island container, which displays what is being sent and received from it and also some additional information about its algorithmic behavior.
- The terminal of the master node, which displays the same information with the other terminal.
- The display of current time, whose purpose is to provide some sense time and reality to the reader of this paper, so that he understands how and when this data exchange between an island container and the master node occurs.

It is also important to note that not all the screenshots of the three simulation runs will be shown in this section. Instead, they all will be provided in the appendices of this paper. As such, in this section, only the screenshots that show the coverage of some cases, presented in the two previous tables, will be illustrated.

With that being said, the second screenshot of this run shows the moment where the system of the framework covers the case 1 of the master node and 5 of the island. Those numbers indicate the cases to be tested which are shown in tables 7.1 and 7.2. That screenshot is illustrated in detail in figure 7.9. In that figure, the successful coverage of those two cases is clearly indicated by the terminals of the two parts and also by showing the order of the steps taken in the particular migration operation. Furthermore, the fitness value of the fittest individual of the previous generation is shown in the running Processing sketch of the simulation. The reason for this is to demonstrate that along with the two random individuals, the island container also sends the fittest individual of the previous generation.

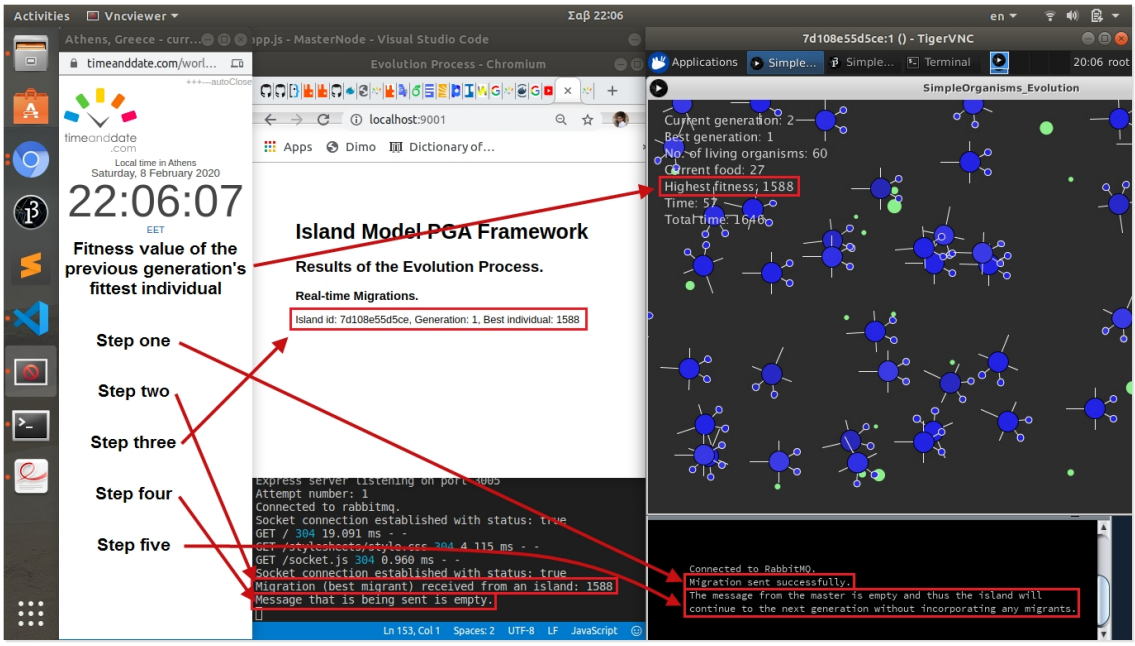


Figure 7.9: Second screenshot of the simulation's first run.

Next, the third screenshot of the first run shows the moment where the system covers the case 3 of the master node and 1 of the island. The specific screenshot, which is illustrated in figure 7.10, indicates the successful coverage of those two cases the same as the previous one, showing the two terminals and the order of steps. Moreover, the fourth screenshot of this run is not shown in this section because it shows the moment where the system covers again the same cases as before (3 and 1).

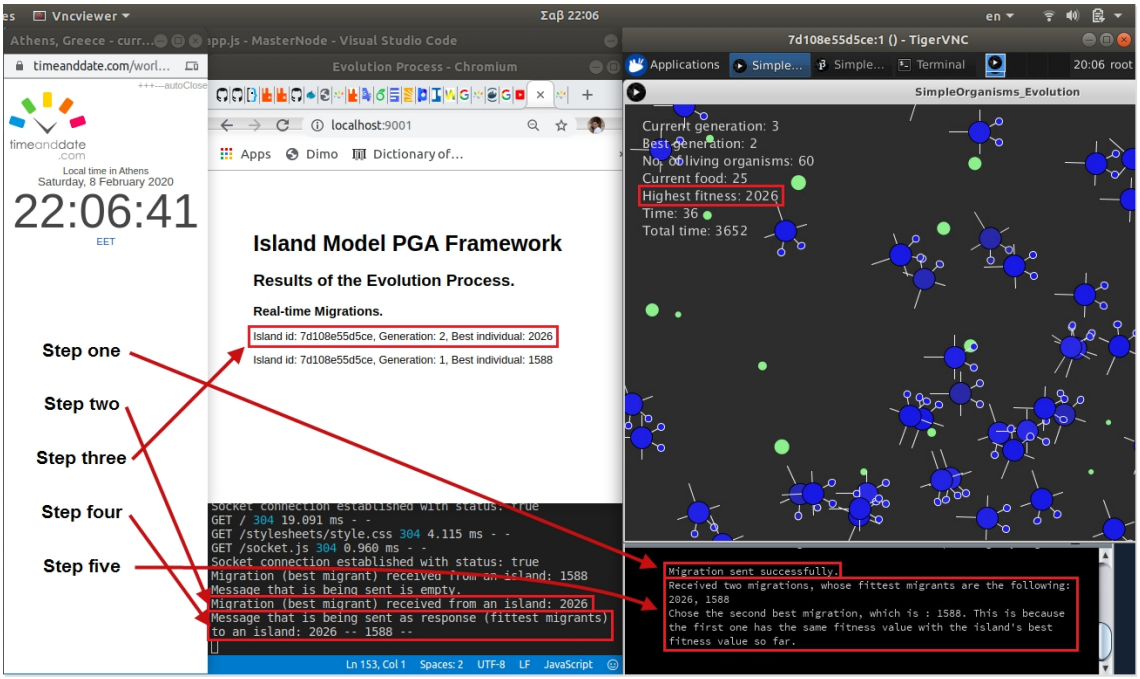


Figure 7.10: Third screenshot of the simulation's first run.

Afterwards, the second run of the simulation needed to be run in order to cover the other cases. As such, after running the simulation for the second time, four screenshots were taken as in the first time. However, the first one is not shown in this section because it showed the same information of RabbitMQ connection as the first screenshot of the previous run did. The second screenshot of this run is also not shown because it demonstrated exactly the same case that the second screenshot of the previous run demonstrated. Moreover, neither the fourth screenshot is shown because it demonstrated the same case as the one the third screenshot did. Therefore, only the third screenshot of the second run is shown in this section and it shows the moment where the system covers the case 2 of the master node and 4 of the island. That screenshot is illustrated in figure 7.11.

After the second run, the final third run of the simulation needed to be made in order to test the remaining two cases, which are the ones that an island may face. Hence, as in the previous run, four screenshots were taken after making this run. However, two of them are shown, the second and fourth screenshots to be more precise. As for the other two, the first one is not shown for the same reason as the first screenshot of the previous run was not shown. The third screenshot is not shown because it demonstrated the same case as the one the second screenshot of this run did.

As such, the second screenshot demonstrates the moment where the system covers the case 2 of the island and it is shown in figure 7.12. As can be seen from that figure, when the island sent its migration to the master node, the latter already had received three migrations from another island. Thus, the migration, sent from the island at hand, was added to the master node's sorted list of migrations.

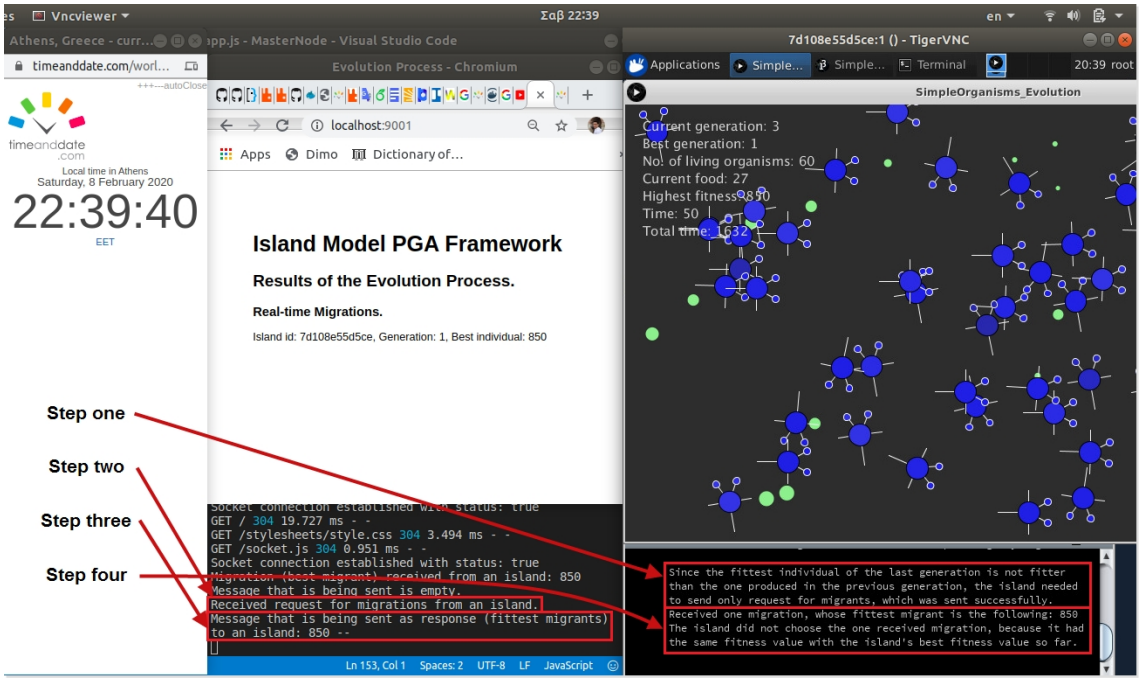


Figure 7.11: Third screenshot of the simulation's second run.

However, as the number of migrations, sent from different islands, increases, it will be difficult to discern which migration was sent from which island. For this reason, as can be noticed from figure 7.12, along with the rest of the additional information, the GUI of the framework also displays the ID of the island from which a migration was sent. This way, it is much easier to discern and identify the migrations that are shown in that list.

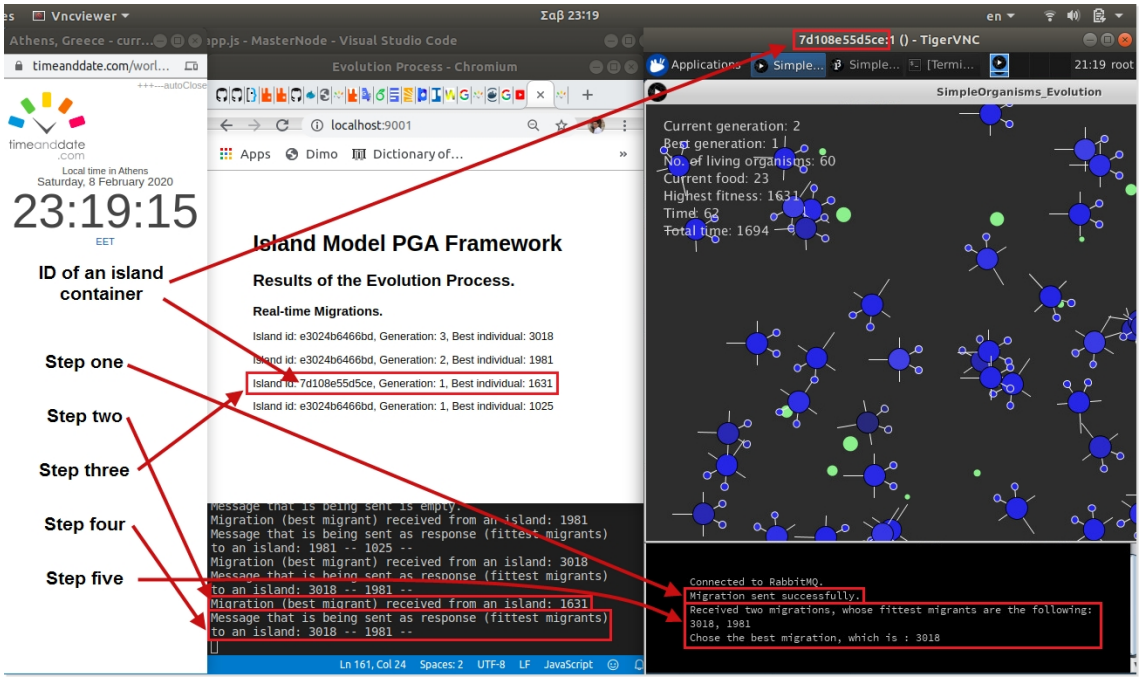


Figure 7.12: Second screenshot of the simulation's third run.

Finally, the fourth screenshot of the simulation's third run shows the moment where the system covers the last remaining case 3 of the island and it is illustrated in figure 7.13. However, as one can notice from that figure, the simulation does not run anymore. The reason is that the fourth screenshot captures the moment where the simulation completes its third and final generation, after which it stops operating. Nevertheless, before the simulation stops, it executes a migration operation exactly the same way as in any other case with the only difference of not continuing to the next generation. This means that the terminal of the island container, in which the simulation was running, will show the data exchange of that last migration operation the same way as in the previous generations.

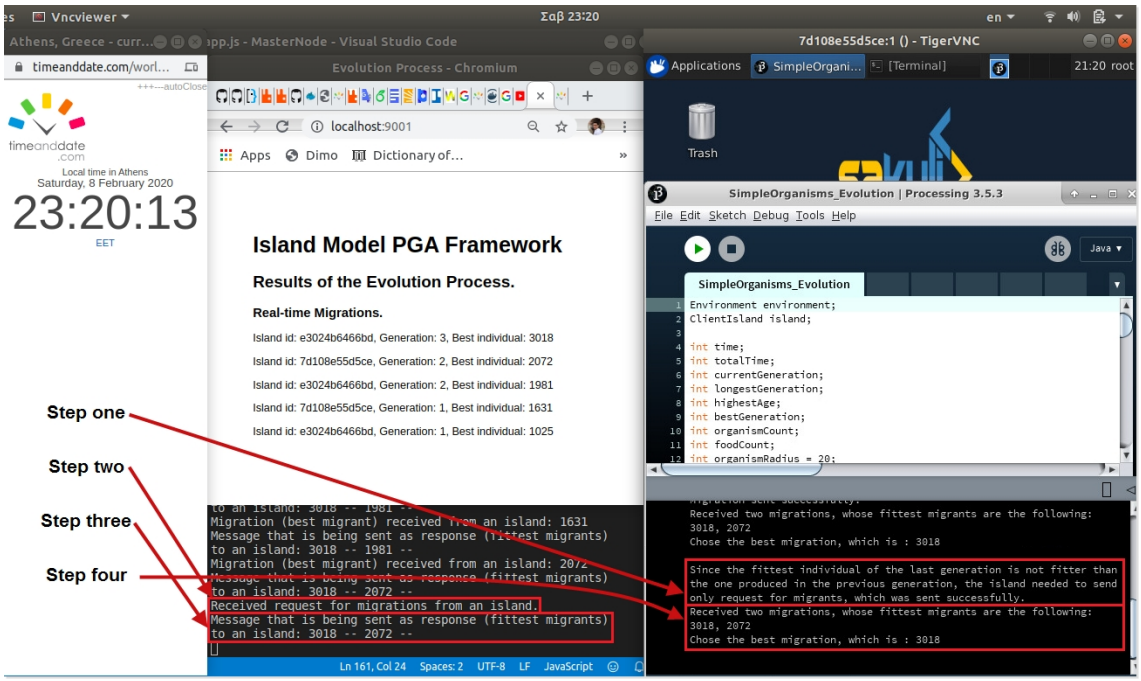


Figure 7.13: Fourth screenshot of the simulation's third run.

On the whole, all the cases of both the master node and island island were tested and showed no problem. This means that, considering all the above observations, the RPC governed message exchange between the master node and an island container was successfully investigated and no problems were found. Therefore, the migration topology of the Island Model GA, which is the broadcast topology, was successfully tested without showing any algorithmic defects.

7.4.3 Second Testing Activity

The second activity, as mentioned before, is to examine the scalability feature of the framework's final software. In order to do that, the simulation island component of the software needs to be scaled by one container. In other words, if initially that component is operated by two island containers, then after the scaling procedure, it will be operated by three

island containers. This scaling process can be achieved by issuing the Docker CLI command ”docker service scale evolutionapp_simulation=3”. That command actually defines the total number of containers of the specified Docker service, which in this case is 3. Moreover, the name ”evolutionapp_simulation” of the specified service is made up of the name ”evolution”, which is the name of the software stack, and the name ”simulation”, which is defined in the docker-compose.yml file and represents a Docker service. This way, all the Docker services in a Docker environment, apart from their name, can be also distinguished by the name of the software stack they belong to. With that being said, the test case created and used for this purpose is illustrated in the table below.

Testing Activity 2 - Scalability					
Test Case	**Preconditions**	**Test Steps**	**Expected Results**	**Actual Results**	**Pass/Fail**
Scale the simulation island component by one container.	The multi-container software stack of the framework to be up and running.	1 - Issue the appropriate Docker CLI command to scale the simulation island component by one container. 2 - Monitor the Visualizer UI to find out if the simulation island component is scaled or not.	Three, instead of two, island containers to be shown in the UI of the Visualizer component.	Three, instead of two, island containers to be shown in the UI of the Visualizer component.	Pass

Table 7.3: Test case for investigating the scalability feature of the framework.

As can be seen, the test case, demonstrated in the above table, was successful. Meaning, the simulation island component was indeed scaled by one container without any problem. This was confirmed by using the Visualizer software component of the framework’s system. To explain, that component, as already mentioned in section 7.3.1, is an open source tool that is used to show the way the Docker services of the software stack at hand distribute their containers over the specified Docker swarm. This way, using Visualizer, it is possible to see which Docker containers from which services run on which nodes of the underlying Docker swarm cluster. Therefore, with the help of this tool it was easy to monitor the distribution of the framework’s software stack and thus to confirm the successful scaling procedure.

However, as in the previous iterations, the preconditions of the test case needed to be checked, before starting the test activity. This test case has only one precondition, which is the software stack of the framework to be up and running, as shown in table 7.3. Hence, the framework was deployed and launched the same way as in the first testing activity.

However, in order to check if the framework was indeed launched, the Docker CLI command "docker stack ls" needed to be issued from the manager node of the employed swarm cluster. After running that command, the software stack of the framework was found to be up and running with the specified name of "evolutionapp", as can be seen in figure 7.14. Thus, the precondition of this test case was successfully validated.

Then, in order to validate that the software stack is up and running, the Docker CLI command "docker stack ls" needed to be issued from the manager node of the employed swarm cluster. After running that command, the software stack of the framework was found to be up and running with the specified name of "evolutionapp", as can be seen in figure 7.14. Thus, the precondition of this test case was successfully validated.

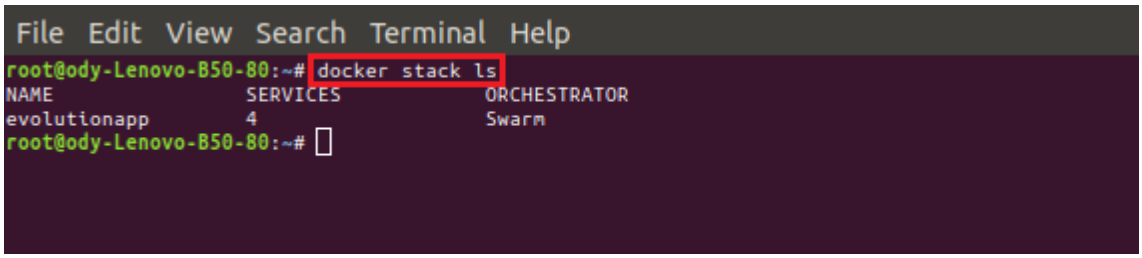


Figure 7.14: Docker CLI command used to check which software stacks are up and running.

Afterwards, the first step of the test case needed to be made, which is to scale the simulation island component by one container, issuing the "docker service scale evolutionapp_simulation=3" command. Right after the execution of this command, details of the scaling process, such as progress and verification, are shown via the terminal, as can be seen in the figure below.

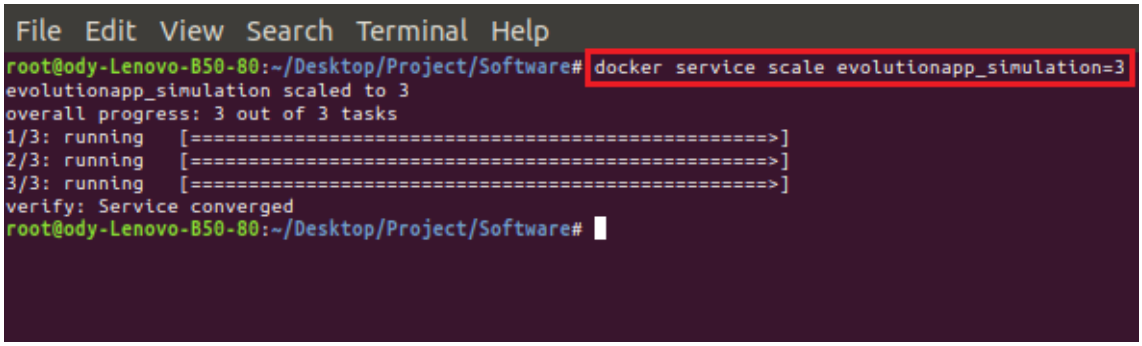


Figure 7.15: Docker CLI command used to scale the specified service by the specified number of containers.

The next step was to monitor the Visualizer's UI in order to verify that the scaling process was conducted as needed. As such, before issuing the command to scale the simulation island component, the UI was showing two containers of that component up and running, as can be seen in figure 7.16. Apart from that, in the same figure, one can notice that the Visualizer shows all the nodes of the swarm cluster, along with their roles

(manager/worker). For example, the virtual machine by the name of "myvm1" is the manager node of the particular swarm cluster and all the others are workers. Furthermore, it can be seen that the Visualizer and master components are running on the manager node (myvm1) and all the others on the worker nodes, exactly as was planned in the "Design" phase.

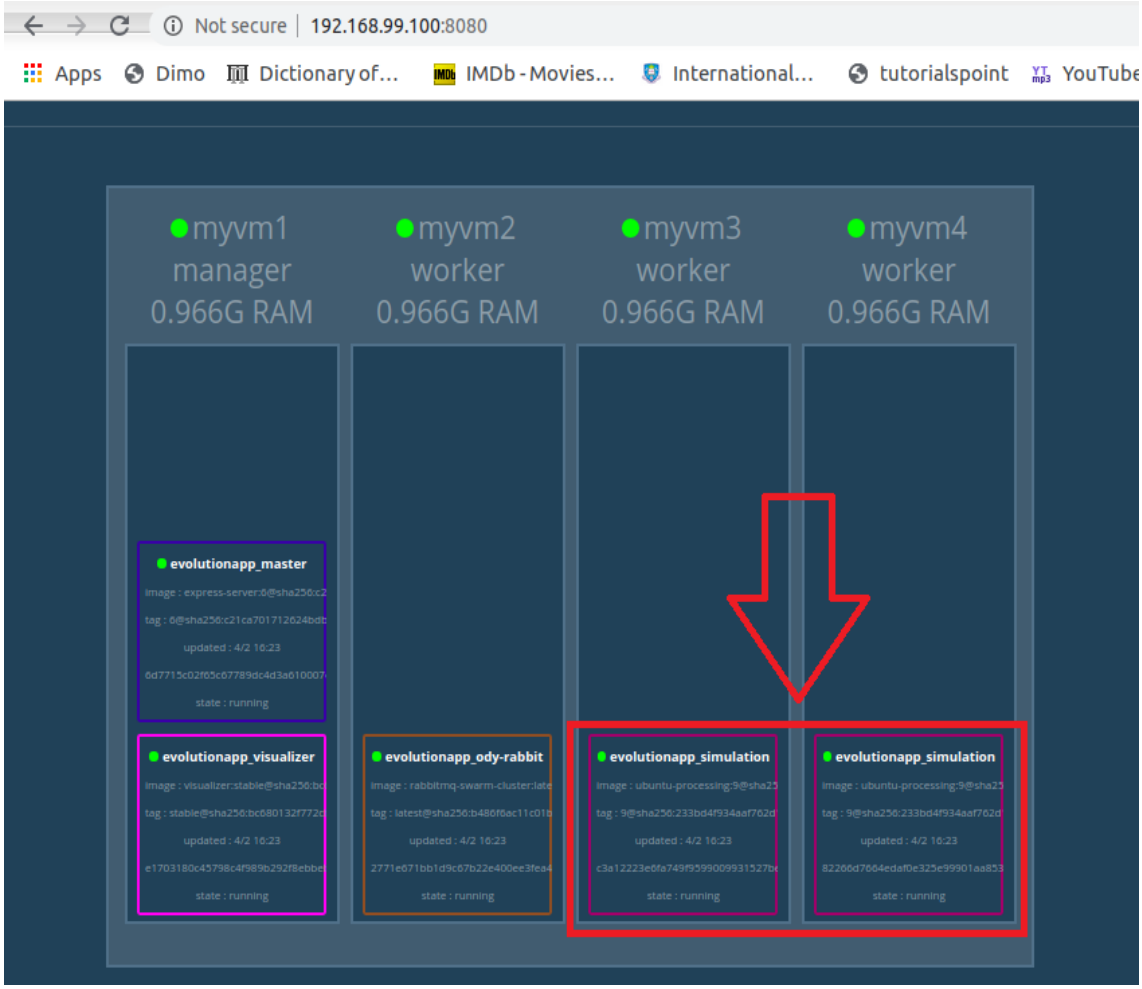


Figure 7.16: Visualizer UI before scaling the system's simulation island component.

After issuing that command though, the UI of the Visualizer component was showing three containers of the simulation island component, as illustrated in figure 7.17.

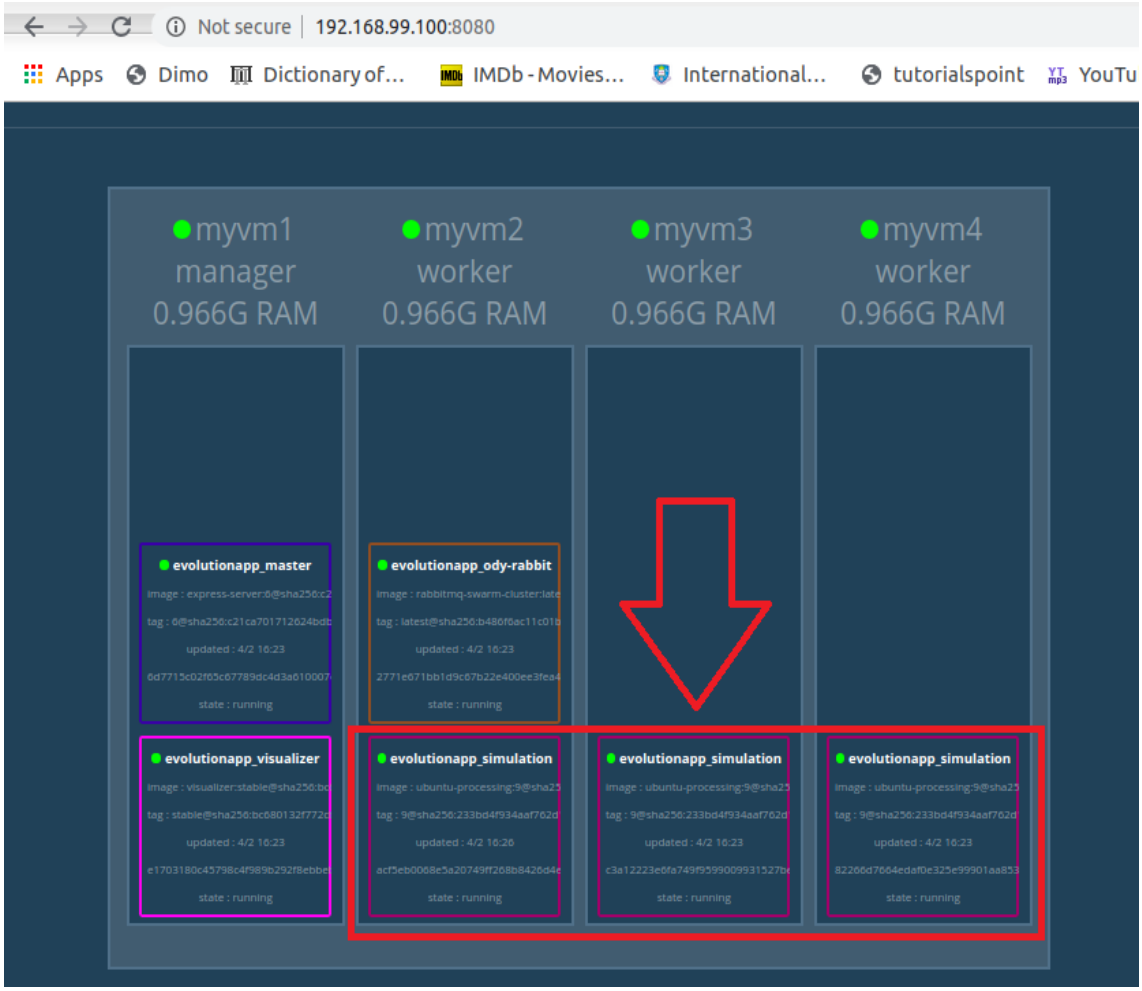


Figure 7.17: Visualizer UI after scaling the system's simulation island component.

Therefore, according to the above observations, the scaling process of the system's simulation island software component was carried out without any problem. As such, the scalability objective of this project was successfully fulfilled.

7.4.4 Third Testing Activity

The second activity of this phase is to inspect the fault-tolerance feature of the framework. This will be done by shutting down a worker node in the swarm cluster. More specifically, it will be one of the worker nodes on which the simulation island component runs its containers (island containers). Thus, by shutting down this node the framework's software will be tested in terms of reliability and tolerance against technical failures. That is, the software of the framework is expected to regenerate the containers which were running on the node that was shut down (or failed, if it was a real case scenario).

The particular node will be shut down using, this time, the Docker Machine CLI command. This is because all the nodes of the swarm cluster are essentially virtual machines

that have Docker Engine installed on them, which makes them Docker hosts. The docker hosts, in general, are created and then easily remotely controlled by the Docker Machine tool[2] [74], which is the reason the CLI of this tool is used. As such, the exact Docker Machine CLI command that needs to be issued in order to shut down the particular worker node is "docker-machine stop myvm4". Thus, the test case that was created and used for the purpose of this testing activity, and where that Docker Machine command was used, is the one shown in the table below.

Testing Activity 3 - Fault-Tolerance					
Test Case	**Preconditions**	**Test Steps**	**Expected Results**	**Actual Results**	**Pass/Fail**
Shut down a worker node of the swarm cluster.	The multi-container software stack of the framework to be up and running.	1 - Issue the appropriate Docker Machine CLI command to shut down the worker node on which an island container runs. 2 - Monitor the Visualizer UI to find out if the island container of that node was regenerated.	The island container of that node to be shown up and running again in the Visualizer's UI.	The island container of that node to be shown up and running again in the Visualizer's UI.	Pass

Table 7.4: Test case for investigating the fault-tolerance feature of the framework.

As can be seen from the table above, this test case was also successful. This means that the island container, which was running on the specific worker node, was regenerated by the framework's dockerized software without any problem, after that node was shut down. Apparently, that container was regenerated in another worker node, since this is the way that Docker technology handles computing node failures. Finally, the success of this test case was confirmed, once again, by the Visualizer tool, as will be shown later on.

Before starting this testing activity though, the preconditions of the test case needed to be checked. The precondition of this test case is the same as in the previous one, which is the software stack of the framework to be up and running, as shown in table 7.4. As such, the framework was deployed as in the previous testing activities and then checked if it was indeed up and running using the "docker stack ls" command once again. After issuing that command, the framework's "evolutionapp" software stack was found to be up and running, as expected. This can be seen in figure 7.18. Hence, the precondition of this test case was successfully validated.

[2] A tool, provided by Docker technology, that is used to easily provision and remotely manage Docker hosts, which are either local or located in a cloud infrastructure [74].

```
File  Edit  View  Search  Terminal  Help
root@ody-Lenovo-B50-80:~# docker stack ls
NAME                SERVICES            ORCHESTRATOR
evolutionapp        4                   Swarm
root@ody-Lenovo-B50-80:~#
```

Figure 7.18: Docker CLI command used to check which software stacks are up and running.

Afterwards, the first step of the test case was made, which is to shut down the worker node that runs an island container in order to imitate technical failure. To do that, as mentioned before, the command "docker-machine stop myvm4" needed to be issued. Thus, the execution of this command is illustrated in the figure below.

```
File  Edit  View  Search  Terminal  Help
root@ody-Lenovo-B50-80:~/Desktop/Project/Software# docker-machine stop myvm4
Stopping "myvm4"...
Machine "myvm4" was stopped.
root@ody-Lenovo-B50-80:~/Desktop/Project/Software#
```

Figure 7.19: Docker Machine CLI command used to shut down a Docker host.

The second step was to monitor the UI of the Visualizer in order to check two things. The first one was to confirm that the particular node was indeed shut down and then the other one was to verify if the island container, which was running on that node, was regenerated. As such, before issuing the command, all the Docker hosts were in active state and two island containers were running separately on a worker node, as shown in figure 7.20. However, after issuing the command to shut down the "myvm4" worker node, the UI shows that node to be in inactive state and its island container to be up and running again, but this time on the other worker node. This can be seen in figure 7.21.

In conclusion, according to these observations, the framework's software was able to tolerate the failure of a computing node by regenerating the software part that was running on that node. As such, the fault-tolerance objective of the project was successfully achieved.

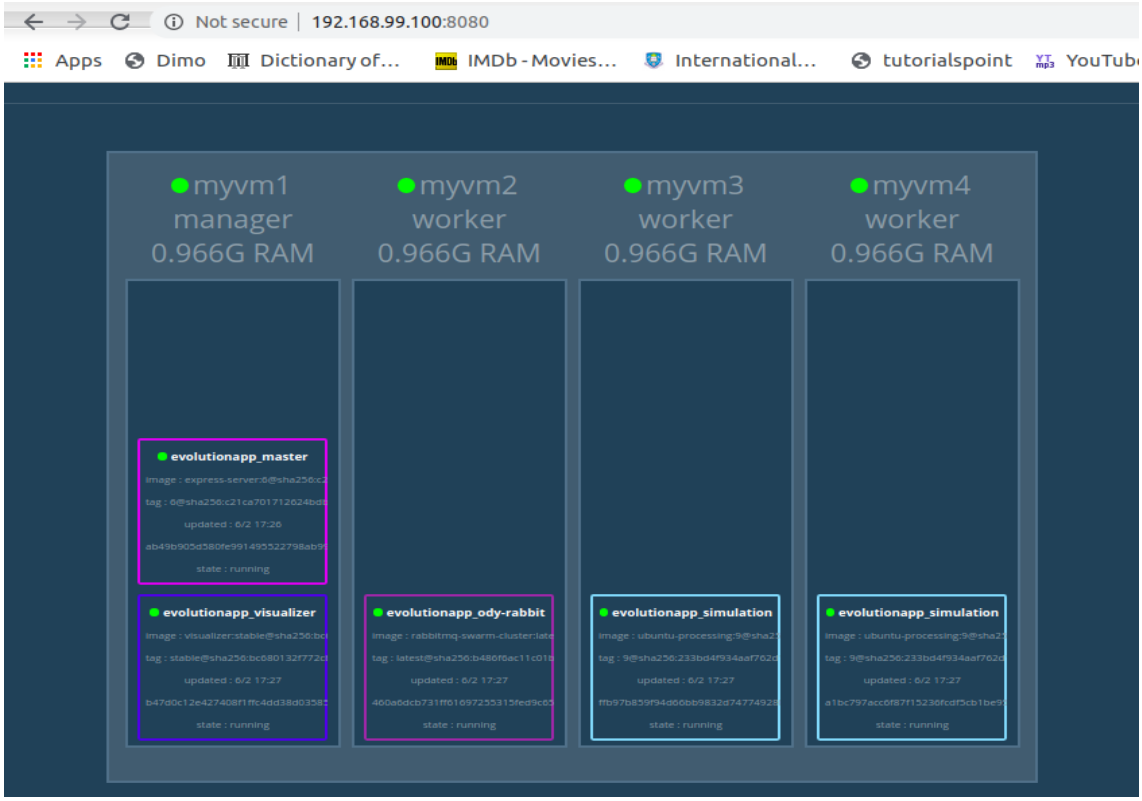


Figure 7.20: Visualizer UI before shutting down the ”myvm4” worker node.

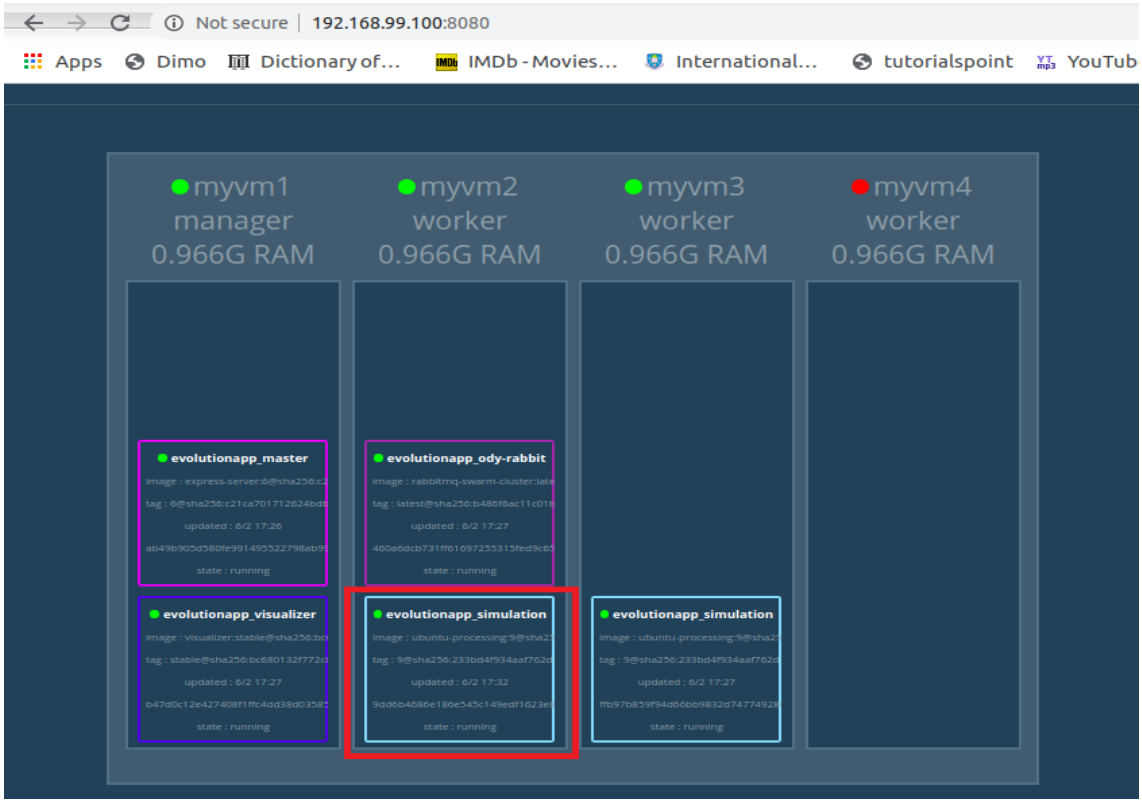


Figure 7.21: Visualizer UI after shutting down the ”myvm4” worker node.

Chapter 8

Evaluation

8.1 Results Discussion

After four iterations of development, using the evolutionary prototyping development model, eventually the final version of the expected framework was delivered. This version of the framework was thoroughly tested in the last iteration and according to the results, the aim of this project was successfully achieved. Meaning, the final product of this project is a functional framework for running ALife simulations on a distributed environment. As such, the finished simulation software not only is able to run in distributed way over any number of machines, but it is also empowered with all the features of the employed Docker technology. Namely, these features are scalability, fault-tolerance, reliability and portability.

Regarding the objectives assigned to this project, in actual practice, all the four objectives have been achieved. To explain, since the final version of the framework is built using the Docker technology and it is fully functional, this means that it is automatically granted the Docker's features of scalability, fault-tolerance and portability. Hence, since these three features are actually the project's objectives, then they are achieved. In addition to that, the features of scalability and fault-tolerance were successfully tested in the last iteration and showed no problems. This implies that the portability feature, although it was not tested, should have also been adopted by the framework, as the other two features have been.

The reason the portability feature was not tested, as the scalability and portability features were, is the cost that comes with testing this feature. Meaning, testing the portability feature has a cost in terms of time, since another cluster needs to be established or a cluster in a cloud infrastructure needs to be configured for this purpose. As such, this cost posed a development risk to the project and thus testing this feature was not an option.

As for the real-time behavior objective, it was also not tested in the course of this project. The reason is, once again, the cost of time. That is, testing this feature is a time-

consuming process, due to the complexity of this kind of testing procedure. Meaning, certain types of diagrams need to be involved for this purpose, such as the Message Sequence Chart. For this reason, neither the objective of real-time was validated in order to avoid the implicated development risk. However, as mentioned before, this objective is practically accomplished, since the framework indeed operated in a real-time fashion during the the first testing activity of the testing phase, thanks to the employed WebSocket technology.

Apart from the objectives though, there are also problems in the functionality of the delivered framework. That is, the final product of this project is not a fully fledged framework yet, in the sense that a simulation can be easily adapted and connected to the framework, so that it can run on it. As such, many changes need to be done in the software of a simulation, so that it is able to run on the current version of the framework, as will be explained later in the "Conclusion" chapter.

Also, another thing, important to point out at this point, is that the framework is not able yet to output the final results of the user's simulation. To explain, after the simulation, that runs on the framework, stops operating, its results are not collected by the framework and thus can not be shown to the user via its GUI. However, this is not something that can not be achieved and thus it can be assigned for the following works on this project, which will also be explained in the "Conclusion" chapter.

Finally, the last thing to point out is that the main key to achieving these results was to use one of the Parallel Genetic Algorithms (PGAs) as the software's core algorithm. As such, the specific PGA that was chosen to manifest the distributed nature of the framework is the Island Model GA, which was thoroughly described in section 2.1.2.2. For this reason, the framework is able to run only according to the Island Model GA rules. This in turn means that it is able to run only the ALife simulations that use a Genetic Algorithm, mainly because these types of algorithms operate by using "generations".

8.2 Benefits of Framework's Software Architecture

The software architecture presented and described in this paper, can actually be regarded as a microservice architecture. This type of architecture is a novel architectural paradigm inspired from the well-known (more heavyweight) Service Oriented Architecture (SOA) [75]. As such, each Docker service in the software stack can be viewed as a microservice, since it is a totally autonomous and independent software unit. Therefore, the framework software inherits all the benefits that come with incorporating the microservice architecture approach [76]. Some of these key benefits are listed below:

- Loosely coupled structure that promotes the "separation of concerns" concept. This way, each software component is a distinct unit of functionality with its own logic and thus can be dealt with in total separation from the rest of the system.
- Language neutral development approach, which allows the usage of different pro-

gramming languages for each software component. This promotes the concept of "best of breed alternatives".

- Failure isolation, since each microservice is independent and thus its failure will not affect the whole software stack. This promotes the reliability of the system, which is also enhanced by the fault tolerance granted from the Docker technology.

Apart from these, the framework also has the advantage of having portability and scalability features provided, once again, by the Docker technology. This way, the docker containers can operate on any machine that can run the Docker engine. Also, the framework's microservices (Docker services) can scale to any number of containers and thus each node of the underlying distributed system can accommodate as many containers as needed. Finally, in case of using a cloud infrastructure, the number of nodes of the employed cluster can be scaled as well. Therefore, in that case, the whole system can be scalable at both levels (software and hardware).

8.3 Problems Encountered

Apart from the problems related to the results of this project, there were also problems during the development of this project. These problems were encountered during the design and development phases. Following is the detailed list of these problems:

- There was a problem with finding the right technologies to realize the vision of the project. That is, it took too much time and effort to research and try many different technologies in order to find out the ones that are fit to fulfill the objectives of the project.
- There was a problem with launching the simulation on the Docker container in an automatic manner. In order to solve this problem, a considerable amount of time was needed in order to find the right docker container and also modify its startup script.
- There was a problem synchronizing the Docker services so that one can wait till the other was in ready state. Once again, a lot of time was needed in order to find out that this problem can be solved using the dockerize tool.
- There was a problem with sending and receiving RabbitMQ messages using the "ClientIsland" class in the example simulation. More particularly, since the example simulation is a Processing sketch, it could not work with the latest Java versions. Thus, lambda expressions, which are extensively used in the RabbitMQ Java client library, could not be used. Thus, in order to solve this problem, other techniques needed to be involved instead.
- There was a problem in implementing the broadcast migration topology. Meaning, much research and effort was put in order to figure out the problem and ending up with the decision of using the RPC communication paradigm.

On the whole, apart from all the above problems, there was a huge development risk from the beginning of this project. The reason is the fact that the nature of the expected framework demanded the involvement and smooth combination of various state of the art technologies. Something that, first of all, is not easily achieved and secondly does not give the certainty that this is feasible.

Chapter 9

Conclusion

9.1 Future Work

The final product of this project is still not in a state where it can be easily used by a user, who wants to run his digital ALife simulation in a distributed manner. To explain, the example simulation, which as already known is a Processing sketch, was run on the produced framework by adapting it in a hard coded manner. Meaning, some parts of that simulation's software needed to be modified so that it can fit the way in which the framework's genetic algorithm operates. For that reason, there is still a lot of work that needs to be done, so that the framework can really operate as expected from it. As such, the future work that needs to be done on this framework is listed below:

- To improve the containerization of the simulation. Meaning, to make it easier to containerize a simulation and make it run on the framework.
- The user must know how to exactly use the framework and what needs to be done in order to run his simulation on that framework. For this reason, it is important to create documentation that will guide the user on what needs to be done for that matter.
- To make the "ClientIsland" connection class a real interface class in terms of OOP. Also, incorporate somehow the "Migration" entity in that class or make a separate interface class.
- To further develop the island containers in order to enable them to notify the master node that they have finished running the simulation, before ceasing their operation. This way, the master node will be able to deliver the final results to the user.

Nevertheless, the finished project established a promising foundation for the next researchers to work on and develop the proposed simulation framework.

9.2 Final Thoughts

In the final analysis, the current project represents the first attempt to create a distributed simulation framework and achieve the objectives that were specified in the “Introduction” chapter. Thus, the product of this project should be regarded as a prototype and should be used as a basic infrastructure, that provides the fundamental building blocks, needed for its further development.

References

[1] L. Steels, "The artificial life roots of artificial intelligence," *Artificial Life*, vol. 1, pp. 75–110, 1993.

[2] S. Agatonovic-Kustrin and R. Beresford, "Basic concepts of artificial neural network (ann) modeling and its application in pharmaceutical research," *Journal of pharmaceutical and biomedical analysis*, vol. 22, no. 5, pp. 717–727, 2000.

[3] M. Mitchell and S. Forrest, "Genetic algorithms and artificial life," *Artificial life*, vol. 1, no. 3, pp. 267–289, 1994.

[4] T. V. Mathew, "Genetic algorithm," *Report submitted at IIT Bombay*, 2012.

[5] G. Luque and E. Alba, *Parallel genetic algorithms: theory and real world applications*. Springer, 2011, vol. 367.

[6] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural networks*, vol. 61, pp. 85–117, 2015.

[7] F. J. M. Martín, O. Trelles, and F. S. Hernández, "Genetic algorithms on lan-message passing architectures using pvm: Application to the routing problem," in *PPSN*, 1994.

[8] G. A. Sena, D. Megherbi, and G. Isern, "Implementation of a parallel genetic algorithm on a cluster of workstations: Traveling salesman problem, a case study," *Future Generation Computer Systems*, vol. 17, no. 4, pp. 477–488, 2001.

[9] B. L. Keeley, "Artificial life for philosophers," *Philosophical Psychology*, vol. 11, no. 2, pp. 251–260, 1998.

[10] C. Langton, "Artificial life," Los Alamos National Lab., NM (USA), Tech. Rep., 1987.

[11] C. Taylor and D. Jefferson, "Artificial life as a tool for biological inquiry," *Artificial Life*, vol. 1, no. 1_2, pp. 1–13, 1993.

[12] J.-J. Merelo-Guervós, P. A. Castillo, and M. García-Valdez, "Alife in the clouds: A short review of applications of artificial life to cloud computing and back," *Frontiers in Robotics and AI*, vol. 4, p. 64, 2018.

[13] E. W. Bonabeau and G. Theraulaz, "Why do we need artificial life?" *Artificial Life*, vol. 1, no. 3, pp. 303–325, 1994.

[14] J. Carr, "An introduction to genetic algorithms," *Senior Project*, vol. 1, no. 40, p. 7, 2014.

[15] M. Mitchell, *An introduction to genetic algorithms*. MIT press, 1998.

[16] A. Chipperfield, P. Fleming, H. Pohlheim, and C. Fonseca, "Genetic algorithm toolbox for use with matlab," 1994.

[17] D. Bhandari, C. Murthy, and S. K. Pal, "Variance as a stopping criterion for genetic algorithms with elitist model," *Fundamenta Informaticae*, vol. 120, no. 2, pp. 145–164, 2012.

[18] S. N. Ghoreishi, A. Clausen, and B. N. Jørgensen, "Termination criteria in evolutionary algorithms: A survey.," in *IJCCI*, 2017, pp. 373–384.

[19] M. Safe, J. Carballido, I. Ponzoni, and N. Brignole, "On stopping criteria for genetic algorithms," in *Brazilian Symposium on Artificial Intelligence*, Springer, 2004, pp. 405–413.

[20] K. E. Kinnear, W. B. Langdon, L. Spector, P. J. Angeline, and U.-M. O'Reilly, *Advances in genetic programming*. MIT press, 1999, vol. 3.

[21] P. Adamidis, "Review of parallel genetic algorithms bibliography," 1994.

[22] E. CANTU-PAZ, "A survey of parallel genetic algorithms," 1998.

[23] J. Smith and K. Sugihara, "A tool for design of distributed genetic algorithms on www," 1997.

[24] E. Alba and J. M. Troya, "A survey of parallel distributed genetic algorithms," *Complexity*, vol. 4, no. 4, pp. 31–52, 1999.

[25] R. Baraglia and R. Perego, "Parallel genetic algorithms for hypercube machines," in *International Conference on Vector and Parallel Processing*, Springer, 1998, pp. 691–704.

[26] E. Cantú-Paz and D. E. Goldberg, "Are multiple runs of genetic algorithms better than one?" In *Genetic and Evolutionary Computation Conference*, Springer, 2003, pp. 801–812.

[27] M. Ruciński, D. Izzo, and F. Biscani, "On the impact of the migration topology on the island model," *Parallel Computing*, vol. 36, no. 10-11, pp. 555–571, 2010.

[28] G. Duarte, A. Lemonge, and L. Goliatt, "A new strategy to evaluate the attractiveness in a dynamic island model," in *2018 IEEE Congress on Evolutionary Computation (CEC)*, IEEE, 2018, pp. 1–8.

[29] E. Cantú-Paz, "Migration policies, selection pressure, and parallel evolutionary algorithms," *Journal of heuristics*, vol. 7, no. 4, pp. 311–334, 2001.

[30] Z. Skolicki, "An analysis of island models in evolutionary computation," in *Proceedings of the 7th Annual Workshop on Genetic and Evolutionary Computation*, ser. GECCO '05, Washington, D.C.: ACM, 2005, pp. 386–389. DOI: `10.1145/1102256.1102343`. [Online]. Available: `http://doi.acm.org/10.1145/1102256.1102343`.

[31] E. Alba, "Parallel evolutionary algorithms can achieve super-linear performance," *Information Processing Letters*, vol. 82, no. 1, pp. 7–13, 2002.

[32] Z. Skolicki and K. De Jong, "The influence of migration sizes and intervals on island models," in *Proceedings of the 7th annual conference on Genetic and evolutionary computation*, ACM, 2005, pp. 1295–1302.

[33] T. Tavares Magalhães, E. Krempser, and H. Barbosa, "Migration policies to improve exploration in parallel island models for optimization via metaheuristics," Jan. 2015. DOI: 10.20906/CPS/CILAMCE2015-0147.

[34] H. He, O. Sỳkora, and A. Salagean, "Various island-based parallel genetic algorithms for the 2-page drawing problem.," in *Parallel and Distributed Computing and Networks*, 2006, pp. 316–323.

[35] E. Alba and J. M. Troya, "Influence of the migration policy in parallel distributed gas with structured and panmictic populations," *Applied Intelligence*, vol. 12, no. 3, pp. 163–181, 2000.

[36] M. Toussaint and C. Igel, "Neutrality: A necessity for self-adaptation," in *Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No. 02TH8600)*, IEEE, vol. 2, 2002, pp. 1354–1359.

[37] Y. Gong and A. Fukunaga, "Distributed island-model genetic algorithms using heterogeneous parameter settings," in *2011 IEEE Congress of Evolutionary Computation (CEC)*, IEEE, 2011, pp. 820–827.

[38] A. A. Gozali and S. Fujimura, "Localized island model genetic algorithm in population diversity preservation," in *2018 International Conference on Industrial Enterprise and System Engineering (IcoIESE 2018)*, Atlantis Press, 2019.

[39] O. Elgabry. (2017, Mar 18). Software engineering — software process and software process models (part 2), Medium, [Online]. Available: https://medium.com/omarelgabrys-blog/software-engineering-software-process-and-software-process-models-part-2-4a9d06213fdc (visited on 12/05/2019).

[40] S. McConnell, *Rapid Development: Taming Wild Software Schedules*, ser. Developer Best Practices. Microsoft Press, 2010, ISBN: 9780735646360. [Online]. Available: https://books.google.gr/books?id=nbrae6-9XWAC.

[41] Prototyping model in software engineering: Methodology, process, approach, Guru99, [Online]. Available: https://www.guru99.com/software-engineering-prototyping-model.html (visited on 12/08/2019).

[42] T. Combe, A. Martin, and R. Di Pietro, "To docker or not to docker: A security perspective.," *IEEE Cloud Computing*, vol. 3, no. 5, pp. 54–62, 2016.

[43] T. Vase, "Advantages of docker," 2015.

[44] Docker overview, Docker Inc, [Online]. Available: https://docs.docker.com/v17.09/engine/docker-overview/ (visited on 12/11/2019).

[45] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing*. John Wiley & Sons, 2011.

[46] What is software testing? introduction, definition, basics & types, Guru99, [Online]. Available: https://www.guru99.com/software-testing-introduction-importance.html (visited on 12/13/2019).

[47] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press, 2016.

[48] Rajkumar. (2019, Nov 13). What is software testing – definition, types, methods, approaches, SoftwareTestingMaterial.com, [Online]. Available: https://www.softwaretestingmaterial.com/software-testing/ (visited on 12/16/2019).

[49] A. Spillner, T. Linz, and H. Schaefer, *Software testing foundations: a study guide for the certified tester exam*. Rocky Nook, Inc., 2014.

[50] D. Graham, E. Van Veenendaal, and I. Evans, *Foundations of software testing: ISTQB certification*. Cengage Learning EMEA, 2008.

[51] Software testing | basics, GeeksforGeeks, [Online]. Available: https://www.geeksforgeeks.org/software-testing-basics/ (visited on 12/13/2019).

[52] M. E. Khan, F. Khan, *et al.*, "A comparative study of white box, black box and grey box testing techniques," *Int. J. Adv. Comput. Sci. Appl*, vol. 3, no. 6, 2012.

[53] S. Nidhra and J. Dondeti, "Black box and white box testing techniques-a literature review," *International Journal of Embedded Systems and Applications (IJESA)*, vol. 2, no. 2, pp. 29–50, 2012.

[54] A. Imam. (2018, Nov 28). What is test strategy, and how to write a test strategy document, TestLodge, [Online]. Available: https://blog.testlodge.com/what-is-test-strategy/ (visited on 12/16/2019).

[55] (2019, Feb 22). Message broker analysis, Jubeen Shah, [Online]. Available: https://www.jubeenshah.com/projects/message-broker-analysis/ (visited on 12/22/2019).

[56] P. Dobbelaere and K. S. Esmaili, "Kafka versus rabbitmq," *arXiv preprint arXiv:1709.00333*, 2017.

[57] Kafka protocol guide, Apache Software Foundation, [Online]. Available: https://kafka.apache.org/0100/protocol.html#protocol_network (visited on 12/22/2019).

[58] T. Capan. Why the hell would i use node.js? a case-by-case tutorial, Toptal, [Online]. Available: https://www.toptal.com/nodejs/why-the-hell-would-i-use-node-js (visited on 12/30/2019).

[59] P. Sauve. (2018, May 28). Create new express.js apps in minutes with express generator, SitePoint, [Online]. Available: https://www.sitepoint.com/create-new-express-js-apps-with-express-generator/ (visited on 12/30/2019).

[60] R. Mhetre. (2019, Mar 02). What is boilerplate code? why & when to use the boilerplate code? FullStackWorld, [Online]. Available: https://fullstackworld.com/post/what-is-boilerplate-code-why-when-to-use-the-boilerplate-code (visited on 12/30/2019).

[61] Final keyword in java, JavaTpoint, [Online]. Available: https://www.javatpoint.com/final-keyword (visited on 01/26/2020).

[62] T. Back, "Selective pressure in evolutionary algorithms: A characterization of selection mechanisms," in *Proceedings of the first IEEE conference on evolutionary computation. IEEE World Congress on Computational Intelligence*, IEEE, 1994, pp. 57–62.

[63] M. Rostanski, K. Grochla, and A. Seman, "Evaluation of highly available and fault-tolerant middleware clustered architectures using rabbitmq," in *2014 federated conference on computer science and information systems*, IEEE, 2014, pp. 879–884.

[64] Html5 websocket: A quantum leap in scalability for the web, websocket.org, [Online]. Available: `https://www.websocket.org/quantum.html` (visited on 01/14/2020).

[65] V. Pimentel and B. G. Nickerson, "Communicating and displaying real-time data with websocket," *IEEE Internet Computing*, vol. 16, no. 4, pp. 45–53, 2012.

[66] V. Pterneas, *Getting Started with HTML5 WebSocket Programming*. Packt Publishing Ltd, 2013.

[67] Websocket vs socket.io, EDUCBA, [Online]. Available: `https://www.educba.com/websocket-vs-socket-io/` (visited on 01/15/2020).

[68] J. J. Grefenstette, "Parallel adaptive algorithms for function optimization," 1981.

[69] B. H. Tay and A. L. Ananda, "A survey of remote procedure calls," *SIGOPS Oper. Syst. Rev.*, vol. 24, no. 3, pp. 68–79, Jul. 1990, ISSN: 0163-5980. DOI: `10.1145/382244.382832`. [Online]. Available: `https://doi.org/10.1145/382244.382832`.

[70] About node.js, OpenJS Foundation, [Online]. Available: `https://nodejs.org/en/about/` (visited on 01/24/2020).

[71] Introducing json, [Online]. Available: `https://www.json.org/json-en.html` (visited on 01/24/2020).

[72] Control startup and shutdown order in compose, Docker, [Online]. Available: `https://docs.docker.com/compose/startup-order/` (visited on 01/26/2020).

[73] Black box testing vs. white box testing: Key differences, Guru99, [Online]. Available: `https://www.guru99.com/back-box-vs-white-box-testing.html` (visited on 02/01/2020).

[74] Docker machine overview, Docker, [Online]. Available: `https://docs.docker.com/v17.12/machine/overview/` (visited on 02/05/2020).

[75] P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis, and S. Tilkov, "Microservices: The journey so far and challenges ahead," *IEEE Software*, vol. 35, no. 3, pp. 24–35, 2018.

[76] D. Jaramillo, D. V. Nguyen, and R. Smart, "Leveraging microservices architecture by using docker technology," in *SoutheastCon 2016*, IEEE, 2016, pp. 1–5.

Appendix A

Code Snippets of Final Software

A.1 Software Archtecture

```
version: "3.7"

services:
  master:
    image: dyssos29/express-server:6
    deploy:
      replicas: 1
      restart_policy:
        condition: on-failure
      placement:
        constraints: [node.role == manager]
    ports:
      - "9001:3005"
    networks:
      - webnet
  simulation:
    image: dyssos29/ubuntu-processing:9
    deploy:
      replicas: 2
      restart_policy:
        condition: on-failure
      placement:
        constraints: [node.role == worker]
    ports:
      - "5901:5901"
    networks:
      - webnet
  visualizer:
    image: dockersamples/visualizer:stable
    ports:
      - "8080:8080"
    volumes:
      - "/var/run/docker.sock:/var/run/docker.sock"
    deploy:
```

```
      restart_policy:
        condition: on-failure
      placement:
        constraints: [node.role == manager]
    networks:
      - webnet
  ody-rabbit:
    image: lovelysystems/rabbitmq-swarm-cluster
    hostname: "{{.Service.Name}}.{{.Task.Slot}}.{{.Task.ID}}"
    deploy:
      restart_policy:
        condition: on-failure
      update_config:
        parallelism: 2
        delay: 10s
      placement:
        constraints: [node.role == worker]
    environment:
      - RABBITMQ_ERLANG_COOKIE=abc
      - RABBITMQ_USE_LONGNAME=true
      - RABBITMQ_MNESIA_DIR=/var/lib/rabbitmq/mnesia
      - RABBITMQ_PLUGINS_EXPAND_DIR=/var/lib/rabbitmq/mnesia/plugins-
    expand
      - SERVICE_NAME={{.Service.Name}}
      - SLOT={{.Task.Slot}}
      - MASTER_SLOT=1
    ports:
      - "5672:5672"   # amqp
      - "15672:15672" # web ui
    networks:
      - webnet

networks:
  webnet:
```

Listing A.1: Docker-compose.yml file of the software system.

A.2 Island Container

```
import com.rabbitmq.client.*;
import com.rabbitmq.client.impl.recovery.*;
import com.rabbitmq.client.impl.*;
import com.rabbitmq.utility.*;
import com.rabbitmq.tools.jsonrpc.*;
import com.rabbitmq.tools.json.*;
import com.rabbitmq.client.impl.nio.*;

import org.slf4j.event.*;
import org.slf4j.helpers.*;
import org.slf4j.*;
import org.slf4j.spi.*;
import org.slf4j.impl.*;

```

```
import com.google.gson.*;
import com.google.gson.stream.*;
import com.google.gson.reflect.*;
import com.google.gson.internal.*;
import com.google.gson.internal.reflect.*;
import com.google.gson.internal.bind.*;
import com.google.gson.internal.bind.util.*;
import com.google.gson.annotations.*;

import java.io.*;
import java.util.concurrent.TimeoutException;
import java.util.UUID;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.ArrayBlockingQueue;

public class ClientIsland
{
  private ConnectionFactory factory;
  private Connection connection;
  private Channel channel;
  private QueueingConsumer consumer;
  private final String REQUEST_QUEUE_NAME = "rpc_request_queue";
  private String responseQueueName = "";
  private Gson jsonMaker;
  private String correlationId;
  private MigrationMessage[] receivedMigrationMessages = {};
  private int bestFitnessValue;

  public ClientIsland() throws  TimeoutException
  {
    factory = new ConnectionFactory();
    factory.setHost("ody-rabbit");
    try
    {
      connection = factory.newConnection();
    }
    catch (IOException e)
    {
      System.out.println("Error in creating connection: " + e.
    getMessage());
    }

    try
    {
      channel = connection.createChannel();
    }
    catch (IOException e)
    {
      System.out.println("Error in creating channel: " + e.getMessage()
    );
    }

    try
    {
      responseQueueName = channel.queueDeclare().getQueue();
    }
```

```
    catch (IOException e)
    {
      System.out.println("Error in creating anonymous queue: " + e.
  getMessage());
    }

    consumer = new QueueingConsumer(channel);
    jsonMaker = new Gson();
  }

  public void updateBestFitnessValue(int bestFitnessValue)
  {
    this.bestFitnessValue = bestFitnessValue;
  }

  public String getNodeId()
  {
    String nodeId = "";

    try
    {
      File file = new File("/etc/hostname");
      BufferedReader br = new BufferedReader(new FileReader(file));
      nodeId = br.readLine();
      br.close();
    }
    catch (IOException e)
    {
      System.out.println("Error in retrieving the node id from the
  hostname file: " + e.getMessage());
    }
    catch (NullPointerException p)
    {
      System.out.println("Null pointer exception in retrieving the node
   id from the hostname file: " + p.getMessage());
    }

    return nodeId;
  }

  public void sendMigration(MigrationMessage migration) throws
  IOException, NullPointerException
  {
    receivedMigrationMessages = new MigrationMessage[]{};
    correlationId = UUID.randomUUID().toString();
    AMQP.BasicProperties properties = new AMQP.BasicProperties
                .Builder()
                .correlationId(correlationId)
                .replyTo(responseQueueName)
                .build();
    String json = serializeMigrationMessage(migration);

    channel.basicPublish("", REQUEST_QUEUE_NAME, properties, json.
  getBytes());
  }

```

```
120   public NeuralNetwork[] requestMigrations() throws Exception
121   {
122     NeuralNetwork[] receivedNNs = {};
123     receivedMigrationMessages = new MigrationMessage[]{};
124     correlationId = UUID.randomUUID().toString();
125     AMQP.BasicProperties properties = new AMQP.BasicProperties
126                 .Builder()
127                 .correlationId(correlationId)
128                 .replyTo(responseQueueName)
129                 .build();
130     String request = "{}";
131
132     channel.basicPublish("", REQUEST_QUEUE_NAME, properties, request.
      getBytes());
133     receivedNNs = getReceivedNeuralNetworks();
134
135     return receivedNNs;
136   }
137
138   public NeuralNetwork[] getReceivedNeuralNetworks() throws Exception
139   {
140     NeuralNetwork[] receivedNNs = {};
141
142     if (checkIfMigrantsArrived())
143     {
144       if (bestFitnessValue == receivedMigrationMessages[0].
      getBestFitnessValue())
145       {
146         if (receivedMigrationMessages.length > 1)
147         {
148           receivedNNs = receivedMigrationMessages[1].
      getMigrantNeuralNetwors();
149           System.out.println("Chose the second best migration, which is
       : " + receivedMigrationMessages[1].getBestFitnessValue());
150         }
151         else
152           System.out.println("The island received only one migration
      message and did not choose it because it had the same fitness value
       with the island's best fitness value so far.");
153       }
154       else
155       {
156         receivedNNs = receivedMigrationMessages[0].
      getMigrantNeuralNetwors();
157         System.out.println("Chose the best migration, which is : " +
      receivedMigrationMessages[0].getBestFitnessValue());
158       }
159     }
160
161     return receivedNNs;
162   }
163
164   private boolean checkIfMigrantsArrived() throws Exception
165   {
166     BlockingQueue<String> response  = new ArrayBlockingQueue<String>(1)
      ;
```

```

    channel.basicConsume(responseQueueName, consumer);
    QueueingConsumer.Delivery deliver = consumer.nextDelivery();

    if (processMessage(deliver, response))
    {
      long deliveryTag = deliver.getEnvelope().getDeliveryTag();
      channel.basicAck(deliveryTag, true);

      if (receivedMigrationMessages.length == 0)
        return false;
      else
        return true;
    }

    return false;
  }

  private boolean processMessage(QueueingConsumer.Delivery delivery,
   BlockingQueue<String> response) throws Exception
  {
    if (delivery.getProperties().getCorrelationId().equals(
   correlationId))
    {
      //System.out.println("Correct correlation id.");
      if (response.offer(new String(delivery.getBody(), "UTF-8")))
      {
        String responseStr = response.take();
        if (responseStr.equals("{}"))
          System.out.println("The message is empty.");
        else
        {
          receivedMigrationMessages = deserializeMigrationMessages(
   responseStr);
          if (receivedMigrationMessages.length == 1)
            System.out.println("Received the following migrants: " +
   receivedMigrationMessages[0].getBestFitnessValue());
          else
            System.out.println("Received the following migrants: " +
   receivedMigrationMessages[0].getBestFitnessValue() + ", " +
   receivedMigrationMessages[1].getBestFitnessValue());
        }

        return true;
      }
      else
      {
        System.out.println("Error: the insertion to the blocking queue
   was not successful.");
        return false;
      }
    }
    System.out.println("Wrong correlation id.");
    return false;
  }

```

```
  private String serializeMigrationMessage(MigrationMessage migration)
  {
    return jsonMaker.toJson(migration);
  }

  private MigrationMessage[] deserializeMigrationMessages(String
    migrationJson)
  {
    return jsonMaker.fromJson(migrationJson, MigrationMessage[].class);
  }
}
```

Listing A.2: ”ClientIsland” connection class.

A.3 Master Node

```
const createError = require('http-errors');
const express = require('express');
const path = require('path');
const cookieParser = require('cookie-parser');
const logger = require('morgan');
const http = require('http');
const amqp = require('amqplib');
const socket = require('socket.io');
const promiseRetry = require('promise-retry');

const indexRouter = require('./routes/index');
const usersRouter = require('./routes/users');

const app = express();

// view engine setup
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'ejs');

app.use(logger('dev'));
app.use(express.json());
app.use(express.urlencoded({ extended: false }));
app.use(cookieParser());
app.use(express.static(path.join(__dirname, 'public')));

app.use('/', indexRouter);
app.use('/users', usersRouter);

// catch 404 and forward to error handler
app.use(function(req, res, next) {
  next(createError(404));
});

// error handler
app.use(function(err, req, res, next) {
  // set locals, only providing error in development
  res.locals.message = err.message;
```

```
  res.locals.error = req.app.get('env') === 'development' ? err : {};

  // render the error page
  res.status(err.status || 500);
  res.render('error');
});

const server = http.createServer(app).listen(3005, function(){
  console.log('Express server listening on port ' + 3005);
});

const io = socket(server);
io.on('connection', function(socket){
  console.log('Socket connection established with status: ' + socket.
    connected);
});

var receivedMigrations = [];
const requestQueue = 'rpc_request_queue';
var isRequest = false;

function addToSortedList(migration)
{
  if (receivedMigrations.length == 0)
    receivedMigrations.push(migration);
  else
  {
    // console.log('length: ' + receivedMigrations.length);
    if (receivedMigrations[receivedMigrations.length - 1].
    bestFitnessValue > migration.bestFitnessValue)
      receivedMigrations.push(migration);
    else
    {
      for(let i = 0; i < receivedMigrations.length; i++)
      {
        if (receivedMigrations[i].bestFitnessValue == migration.
    bestFitnessValue)
          break;
        else
        {
          if (receivedMigrations[i].bestFitnessValue < migration.
    bestFitnessValue)
          {
            receivedMigrations.splice(i, 0, migration);
            break;
          }
        }
      }
    }
  }
}

function getMigrationsToSend()
{
  let migrationsToSend = [];

```

```
  if (receivedMigrations.length == 1 && !isRequest)
    return {};
  else
  {
    if (receivedMigrations.length == 1)
      migrationsToSend.push(receivedMigrations[0]);
    else
      migrationsToSend.push(receivedMigrations[0], receivedMigrations
    [1]);
  }

  return migrationsToSend;
}

function printBestMigrantsBeingSent(migrationsBeingSent)
{
  let bestMigrants = '';

  for(let i = 0; i < migrationsBeingSent.length; i++)
  {
    bestMigrants += migrationsBeingSent[i].bestFitnessValue + ' -- ';
  }

  return bestMigrants;
}

function processRPCrequest(aMigrationString)
{
  let migrationObj = JSON.parse(aMigrationString);

  if (aMigrationString != '{}')
  {
    console.log("Migration (best migrant) received from an island: %s",
     migrationObj.bestFitnessValue);
    addToSortedList(migrationObj);
  }
  else
  {
    isRequest = true;
    console.log('Received request for migrations from an island.');
  }
}

promiseRetry(function (retry, number) {
  console.log('Attempt number: ' + number);

  return amqp.connect('amqp://ody-rabbit')
      .catch(retry);
}).then(function(conn) {
  console.log('Connected to rabbitmq.');
  return conn.createChannel();
}).then(function(ch) {
  return ch.assertQueue(requestQueue, { durable: true}).then(function(
    ok) {
    return ch.consume(requestQueue, function(msg) {
      if (msg !== null) {
```

```
        let migrationStr = msg.content.toString();
        processRPCrequest(migrationStr);

        // console.log('Length of the migration objects array: ' +
    receivedMigrations.length);
        io.sockets.emit('migration', receivedMigrations);

        let migrationsToSend = getMigrationsToSend();
        let jsonToSend = JSON.stringify(migrationsToSend);
        if (jsonToSend != "{}")
          console.log('Message that is being sent as response (fittest
    migrants) to an island: ' + printBestMigrantsBeingSent(
    migrationsToSend));
        else
          console.log('Message that is being sent is empty.');

        ch.sendToQueue(msg.properties.replyTo, Buffer.from(jsonToSend),
     {correlationId: msg.properties.correlationId});
        ch.ack(msg);
      }
    });
  });
}).catch(console.warn);

app.locals.migrations = receivedMigrations;

module.exports = app;
```

Listing A.3: ”app.js” JavaScript file that implements master node’s main functionalities.

```
{
  "name": "masternode",
  "version": "0.0.0",
  "private": true,
  "scripts": {
    "start": "nodemon ./bin/www"
  },
  "author": "ofilippidis@citycollege.sheffield.eu",
  "dependencies": {
    "amqplib": "0.5.3",
    "cookie-parser": "~1.4.4",
    "debug": "~2.6.9",
    "ejs": "~2.6.1",
    "express": "~4.16.1",
    "http-errors": "~1.6.3",
    "morgan": "~1.9.1",
    "nodemon": "^1.19.2",
    "promise-retry": "^1.1.1",
    "socket.io": "2.2.0"
  }
}
```

Listing A.4: ”package.json” file that contains the main configurations.

A.4 Front-End GUI

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= title %></title>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/socket.io
   /2.2.0/socket.io.dev.js"></script>
    <link rel='stylesheet' href='/stylesheets/style.css' />
  </head>
  <body>
    <h1>Island Model PGA Framework</h1>
    <h2>Results of the <%= title %>.</h2>
    <h3>Real-time Migrations.</h3>
    <div id="migrations">
      <% if (migrations.length > 0){ %>
        <% migrations.forEach(function(item){ %>
        <p>Island id: <%=item.nodeId %>, Generation: <%=item.
   generationNumber %>, Best individual: <%=item.bestFitnessValue %></
   p>
        <% }); %>
      <%  } %>
    </div>
    <script src="/socket.js"></script>
  </body>
</html>
```

Listing A.5: ”index.ejs” file that implements the html of the GUI.

Appendix B

Testing Runs

B.1 First Run Screenshots

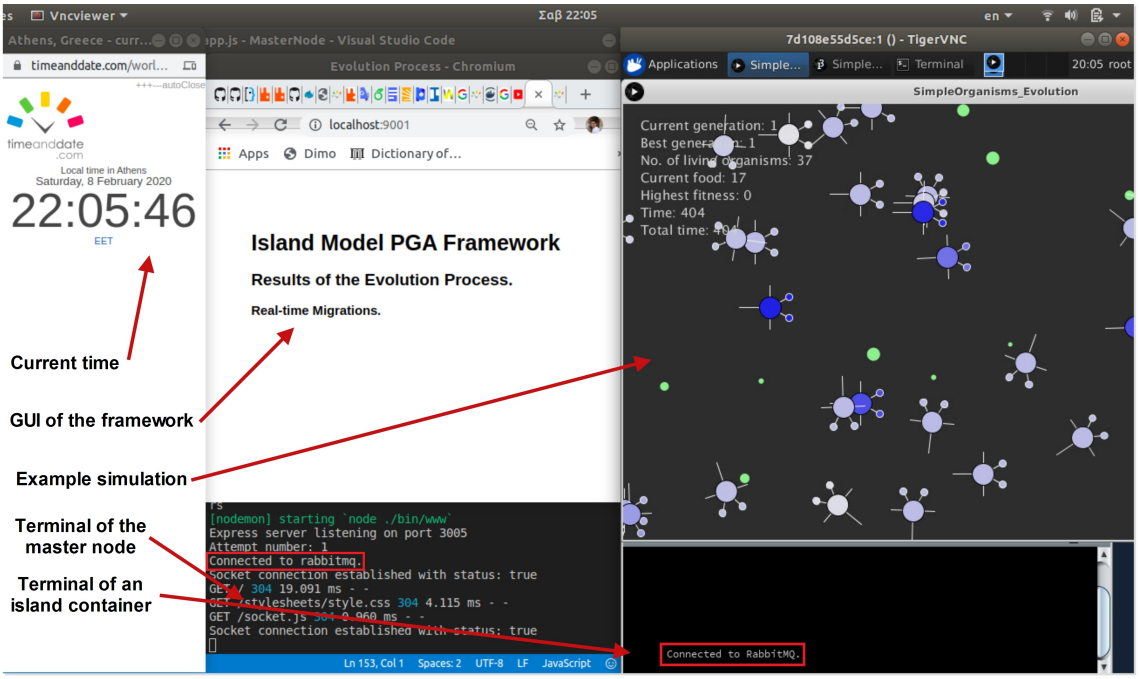


Figure B.1: First Screenshot of the simulation's first run.

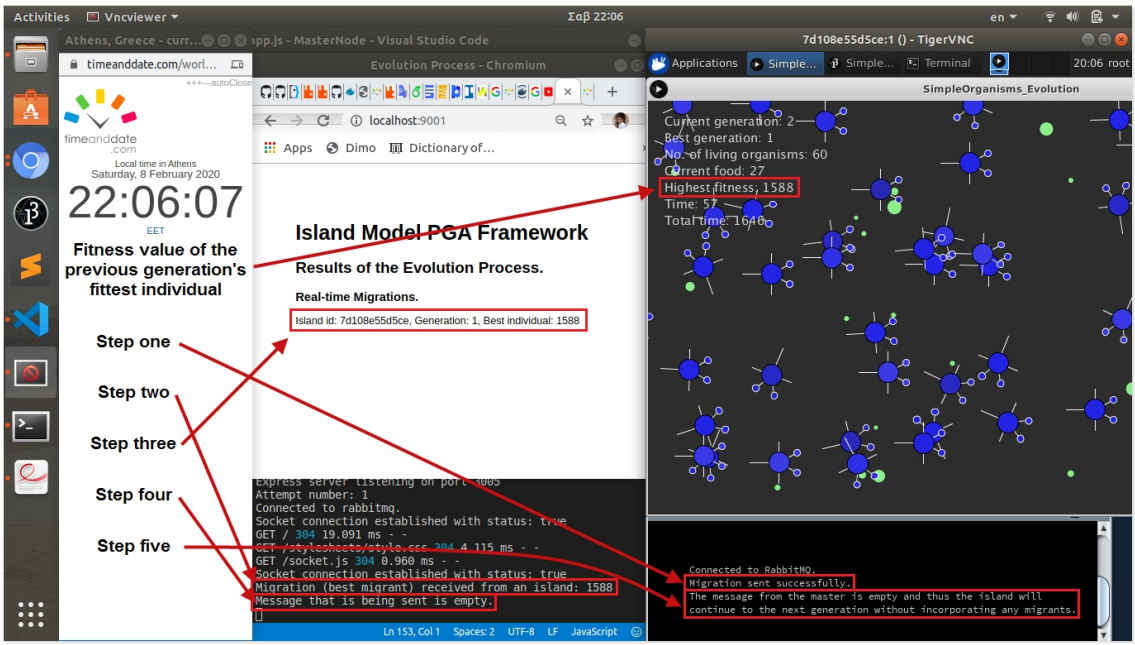


Figure B.2: Second Screenshot of the simulation's first run.

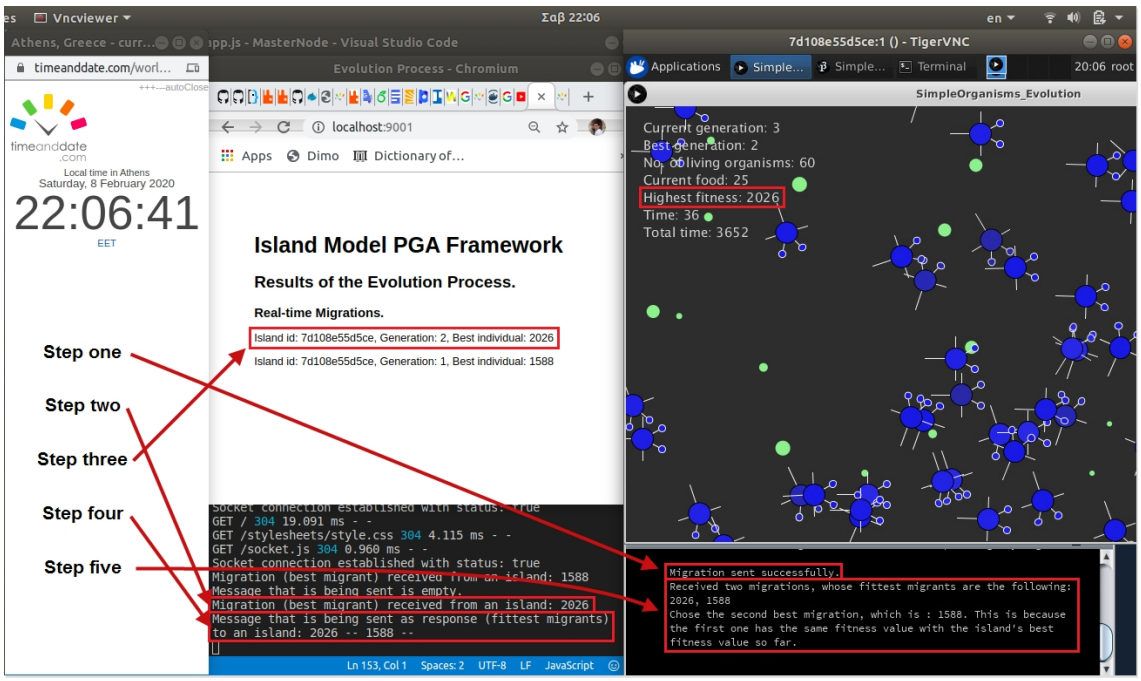


Figure B.3: Third Screenshot of the simulation's first run.

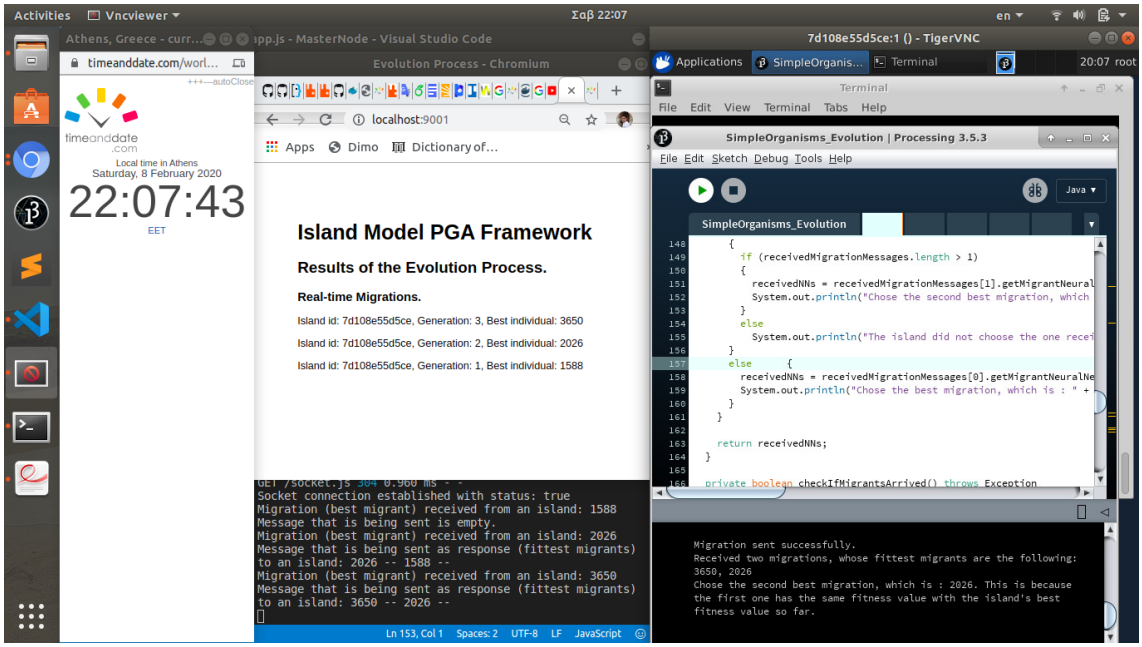


Figure B.4: Fourth Screenshot of the simulation's first run.

B.2 Second Run Screenshots

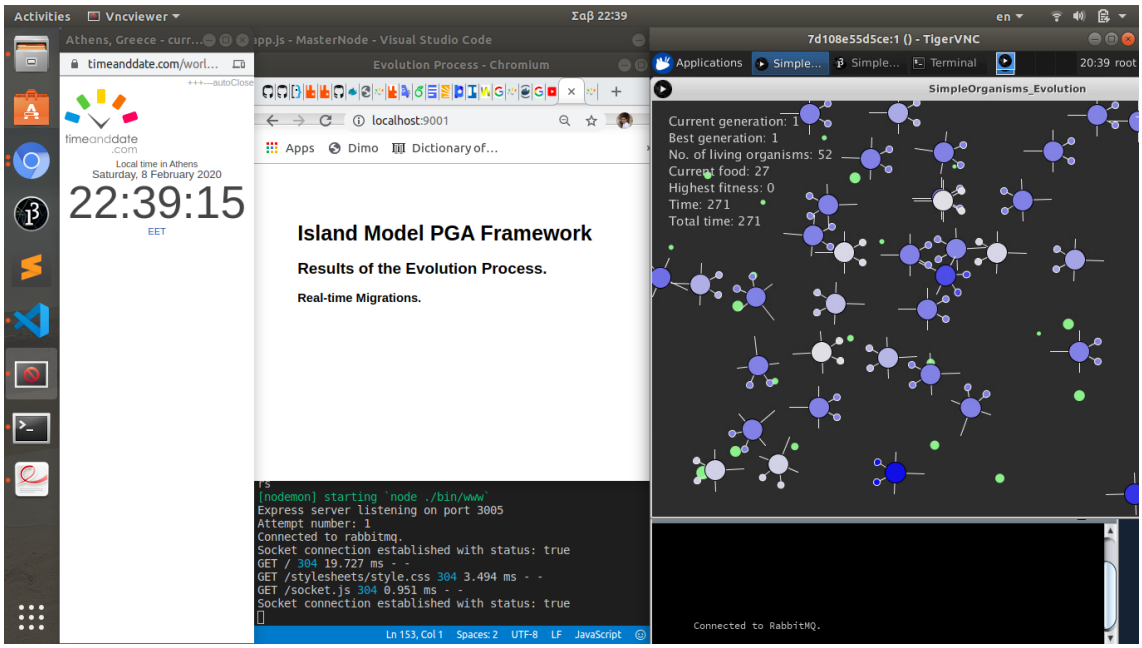


Figure B.5: First Screenshot of the simulation's second run.

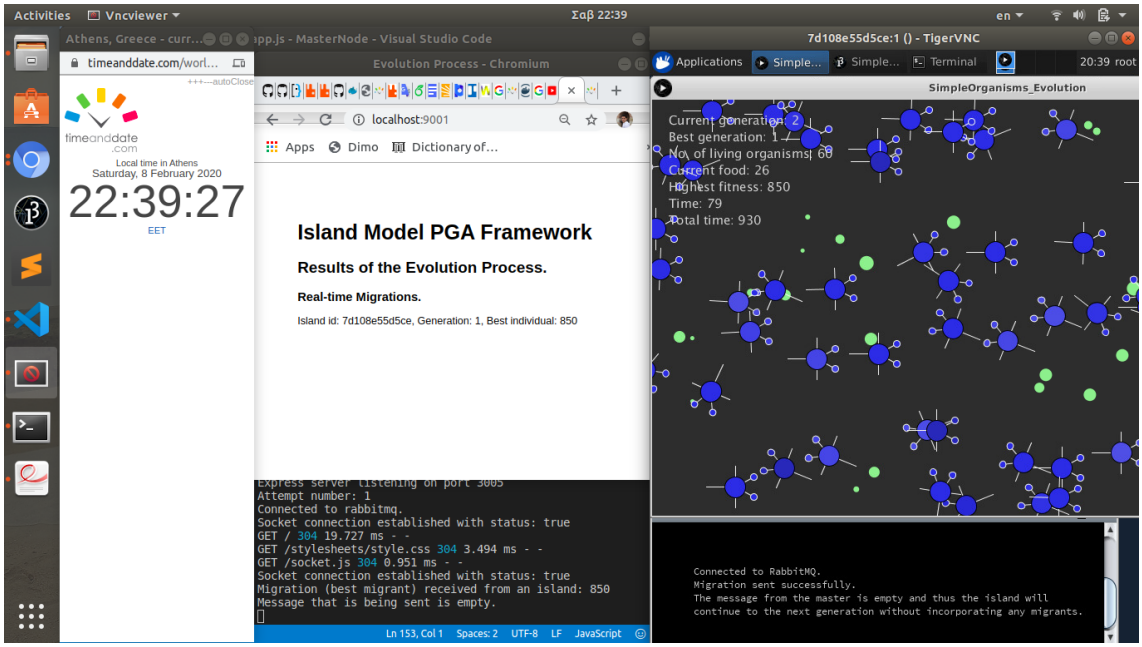


Figure B.6: Second Screenshot of the simulation's second run.

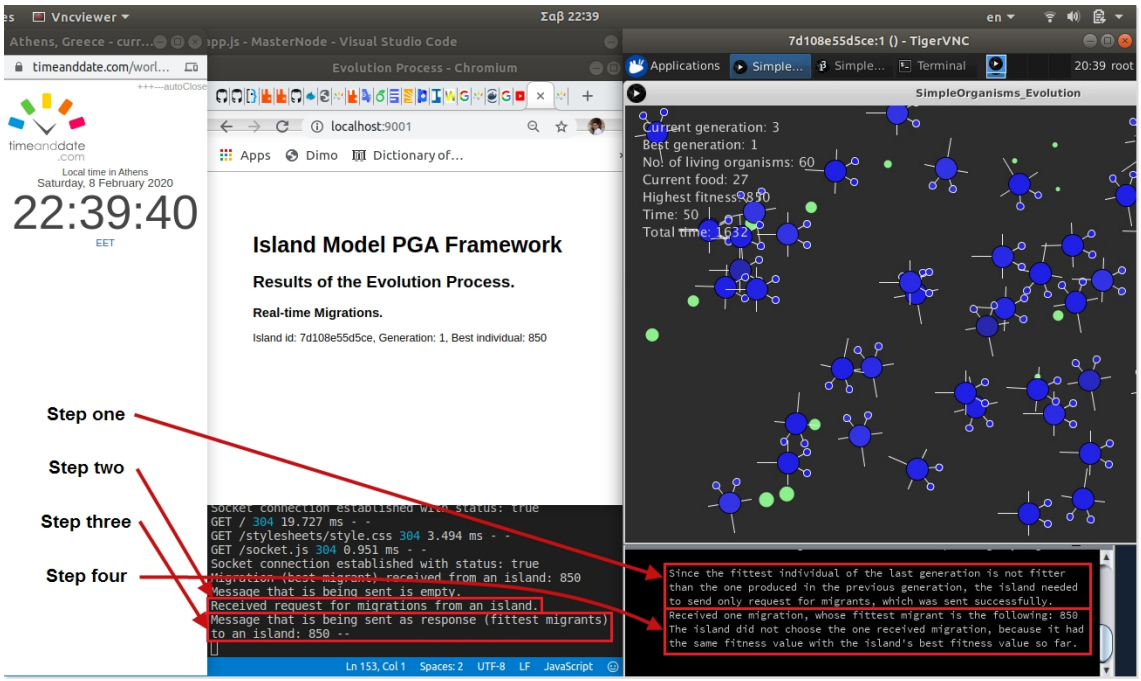


Figure B.7: Third Screenshot of the simulation's second run.

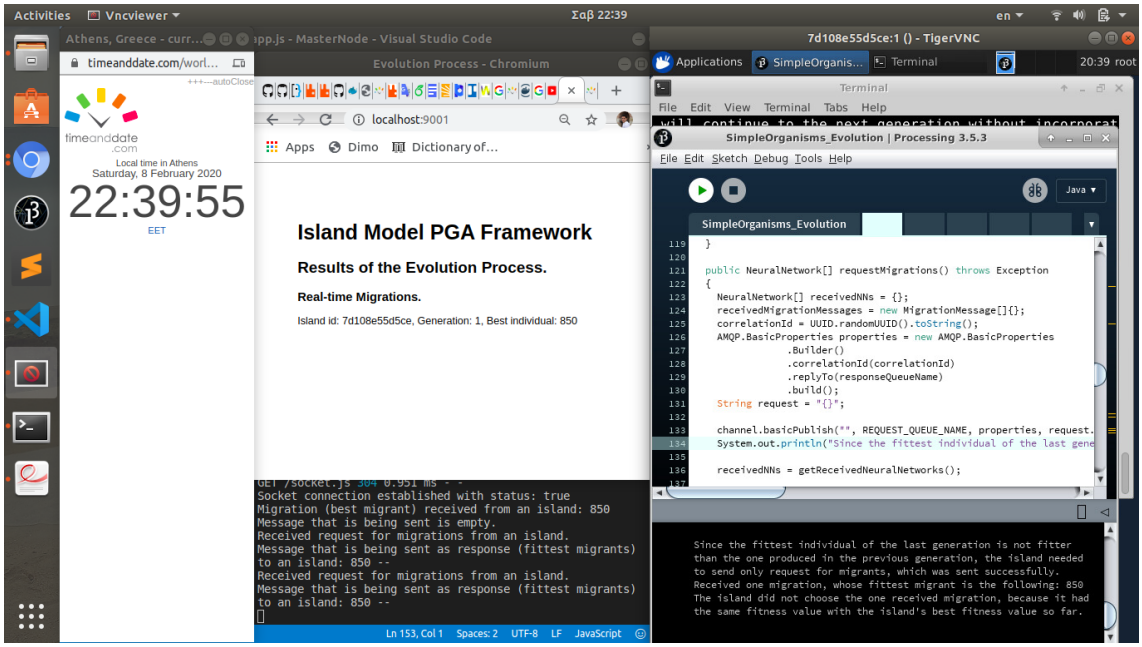


Figure B.8: Fourth Screenshot of the simulation's second run.

B.3 Third Run Screenshots

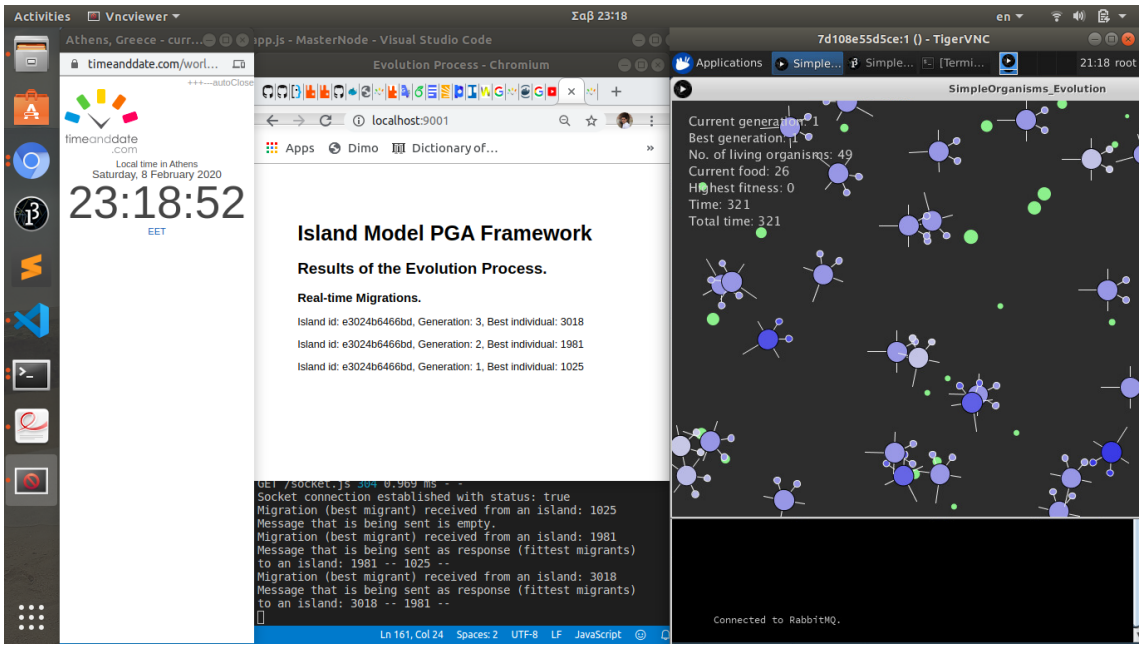


Figure B.9: First Screenshot of the simulation's third run.

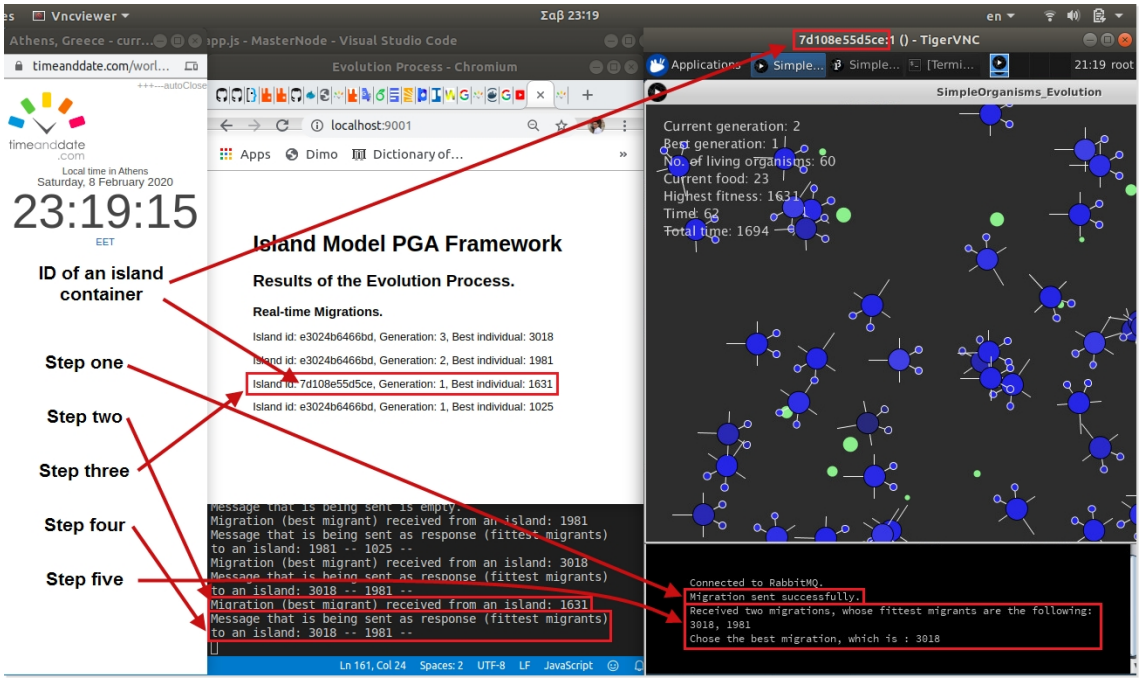


Figure B.10: Second Screenshot of the simulation's third run.

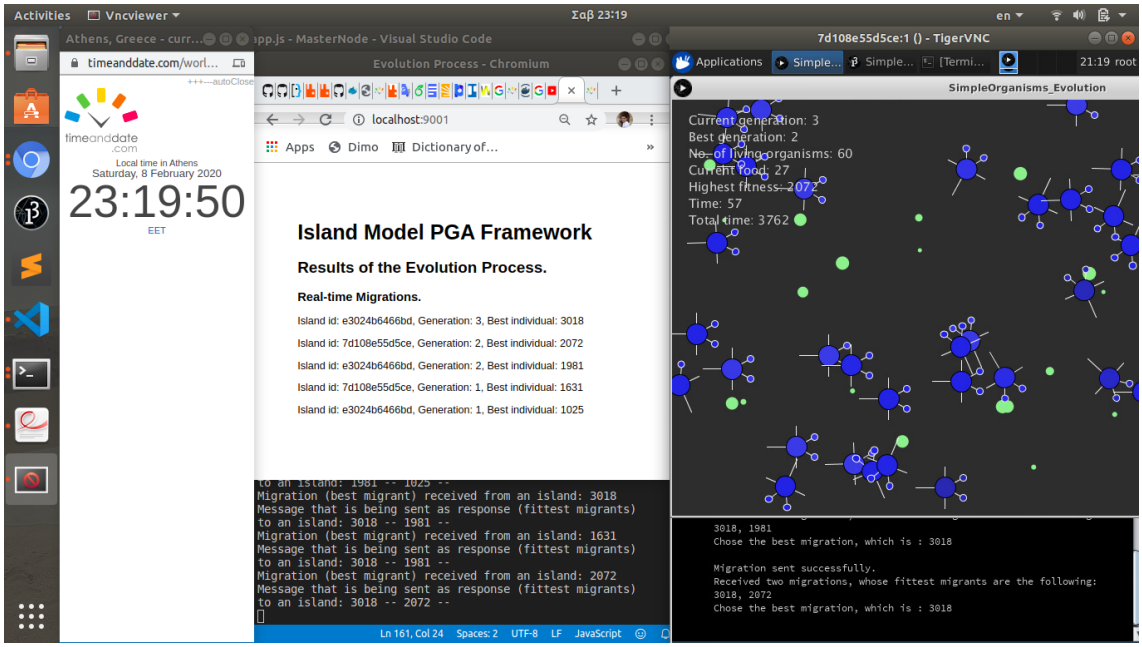


Figure B.11: Third Screenshot of the simulation's third run.

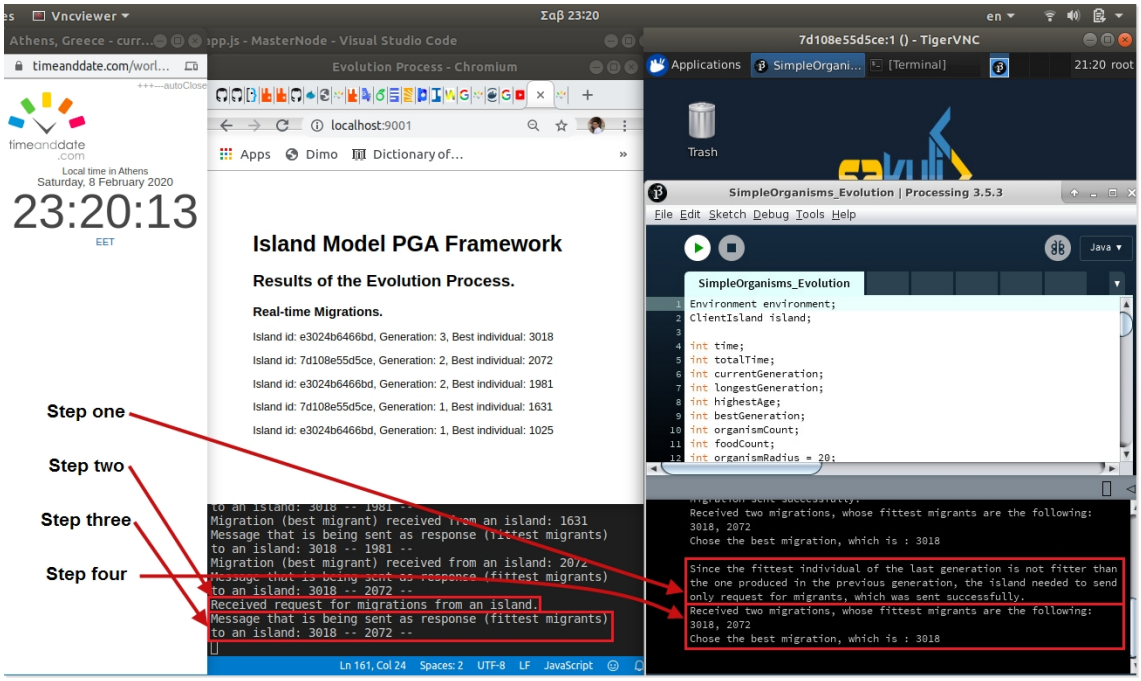


Figure B.12: Fourth Screenshot of the simulation's third run.