HAW HAMBURG

BACHELOR THESIS
Bruno Guiomar

Development and Evaluation of a Distributed Computing System for Evolutionary Algorithms

Faculty of Engineering and Computer Science
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE
WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Bruno Guiomar

Development and Evaluation of a Distributed Computing System for Evolutionary Algorithms

Bachelor thesis submitted for examination in Bachelor´s degree
in the study course *Bachelor of European Computer Science*
at the Department Computer Science
at the Faculty of Engineering and Computer Science
at University of Applied Science Hamburg

Supervisor: Prof. Dr. Christian Lins
Supervisor: Prof. Dr. Thomas Clemen

Submitted on: 23. July 2024

Bruno Guiomar

Thema der Arbeit

Entwicklung und Evaluation eines verteilten Systems für die Berechnung von evolutionären Algorithmen

Stichworte

Evolutionäre Algorithmen, Meta-EAs, Genetische Programmierung, Verteiltes Rechnen, Verteilte Auswertung, Inselmodell, DECS

Kurzzusammenfassung

Evolutionäre Algorithmen und ihre adaptive Natur machen sie besonders effektiv bei der Bewältigung vielfältiger und dynamischer Probleme in verschiedenen Bereichen. Sie können jedoch leicht komplex und ressourcenintensiv werden, was ihre Effizienz erheblich einschränkt. Eine weit verbreitete Strategie zur Bewältigung dieses Problems ist die Verteilung von Aufgaben auf mehrere Rechenressourcen, um die Leistung zu verbessern. In dieser Arbeit werden die Auswirkungen verschiedener Verteilungsmethoden auf die Effizienz von evolutionären Algorithmen untersucht. Ein **D**istributed **E**volutionary **C**omputing **S**ystem (DECS) wurde entwickelt, um eine technische Studie durchzuführen, die die Effektivität der Verteilungsmethoden Distributed Evaluation und Island Model zusammen mit ihren Konfigurationen analysiert, um zu verstehen, wie ihre Vorteile voll genutzt werden können. Das vorgeschlagene System wurde dann einer Bewertung der Benutzererfahrung unterzogen, um sicherzustellen, dass die Anwendungsfälle erfüllt werden. Diese Forschung zeigt eine klare Korrelation, mit Einschränkungen, zwischen der Menge an Rechenressourcen und der benötigten Zeit zur Lösung eines Problems bei Anwendung der verteilten Bewertungsmethode. Beim Inselmodell ist jedoch eine komplexere und subtilere Beziehung zwischen den verschiedenen Konfigurationen und der evolutionären Effizienz zu beobachten, die sich eindeutig positiv auswirkt. Zusammenfassend lässt sich sagen, dass beide Modelle im Vergleich zu lokalen, zentralisierten Ausführungen eine erhebliche Leistungsverbesserung darstellen und das vorgeschlagene System seine ursprünglichen Ziele tatsächlich erreicht hat.

Bruno Guiomar

Title of Thesis

Development and Evaluation of a Distributed Computing System for Evolutionary Algorithms

Keywords

Evolutionary Algorithms, Meta-EAs, Genetic Programming, Distributed Computing, Distributed Evaluation, Island Model, DECS

Abstract

Evolutionary algorithms and their adaptive nature make them particularly effective in tackling diverse and dynamic problems across various fields. However, they can easily become complex and resource-demanding, posing a considerable setback to their efficiency. A widely used strategy to tackle this issue involves the distribution of tasks across multiple computing resources to improve performance. This thesis studies the impact of different distribution methods on the efficiency of evolutionary algorithms. A **D**istributed **E**volutionary **C**omputing **S**ystem (DECS) was developed to perform a technical study that analyzes the effectiveness of the Distributed Evaluation and Island Model distribution methods, together with their configurations, in order to understand how to fully leverage their benefits. The proposed system was then subjected to a user experience evaluation to ensure the fulfillment of its use cases. This research shows a clear correlation, with limitations, between the amount of computational resources and the required time to solve a problem when applying the distributed evaluation method. However, a more complex and subtle relation between the various configurations and evolutionary efficiency is observed in the island model, with clear positive outcomes. In conclusion, both models pose a considerable performance improvement, in contrast with local centralized executions, and the proposed system effectively achieved its initial objectives.

Contents

List of Figures

List of Tables

1 Introduction

Evolutionary algorithms (EAs) are a heuristic-based approach inspired by the way biological life evolves through the process of reproduction, mutation, recombination, natural selection, and survival of the fittest. They are expected to provide non-optimal but good quality solutions to problems whose resolution is impracticable by exact methods. While these algorithms have been traditionally designed with single-processor systems in mind, the current technological landscape includes a high variety of frameworks and devices that twist and shift this paradigm in many different directions. In the execution of evolutionary algorithm methods, a large amount of computational resources are required, and in most cases, they are directly related to the processing time required to find a solution. There are several reasons covered in this thesis that explain this high demand. However, a straightforward common approach solves most of them, which involves distributing the workload across multiple computers and executing all computations simultaneously in parallel. With the introduction of this technique, several distribution models were conceived and introduced over the years with different approaches and objectives. There are two main groups of distribution models covered in this thesis, based on their optimization objectives. Some models try to optimize computing performance by having multiple processing units simultaneously compute tasks for a common problem, as in the example of the *Distributed Evaluation* model in Section 3.1.1. On the other hand, some models try to optimize evolutionary efficiency, reaching better solutions earlier, as in the example of the *Island Model* (Section 3.1.2).

These distribution models are often complex and require a certain structure or configuration to achieve positive outcomes. An in-depth understanding of the model's behavior when subjected to different configurations and environments is essential to conceiving high-performance systems and making the most of these distribution models. The objective of this research was inspired by the central research question:

How do different distribution methods impact the efficiency of evolutionary algorithms?

There are some scientific contributions and frameworks that allow evolutionary algorithms to be computed distributively, however, they are usually complex, require in-depth programming skills, and are not user-friendly. In addition, the bridge between evolutionary algorithm processing and distributed computing is often not solid, which results in an urge to search for monolithic system approaches such as multithreading, which generally perform worse for complex EA problems. To tackle this issue, a **D**istributed **E**volutionary **C**omputing **S**ystem **(DECS)** system was developed and experimentally evaluated in this thesis. Its objective is to establish a robust platform that aims to achieve easy user interaction while building solid distributed processing transparency that simplifies the internal process of task distribution and support for several types of EAs and configurations. While aiming for adaptability, the two-component system, composed of a central entity (DECS) and multiple processing nodes (DECS-Slave), can adapt itself to handle different topologies and settings, effectively distributing tasks to achieve high efficiency.

This thesis is structured in the following chapters:

Chapter 2 provides a knowledge base, introducing the basic concepts of evolutionary algorithms and the sub-types explored in this work, followed by an introduction to some complex evolutionary problems and their specificities. An introduction to distributed computing is also made, including the architectures, models, and main technologies used. Finally, introductory concepts are presented and analyzed in the context of Graphical User Interfaces, Software Security, and testing.

Chapter 3 analyzes the state-of-the-art distribution models for evolutionary algorithms included in this work and several other evolutionary frameworks.

Chapter 4 describes the proposed system design, including the global and internal architectures of each entity in the system, together with an overview of concrete software security considerations.

Chapter 5 outlines the system's main components, implemented features, and protocols. A brief overview of the deployment possibilities is made, followed by a description of the system's software testing suite.

Chapter 6 provides a complete description of the technical study as part of the research procedure. This chapter analyzes the planned methodology for the study, including the different test suites and the respective results.

Chapter 7 provides an overview of the user experience study, which evaluates the proposed system in a user-side context and its use cases.

Chapter 8 establishes a discussion of the research results from chapters 6 and 7 and formulates concrete conclusions.

The thesis concludes in ***Chapter 9*** with a summary and future work.

2 Conceptual Background

This chapter establishes the knowledge base for the contents explored in this work. First, a basic introduction to *Evolutionary Algorithms* is established, with a particular focus on the types of algorithms explored in this thesis. Afterwards, some specific problems with complex features are analyzed and their theoretical concepts explained. Thereafter, an introduction to distributed computation is made, analyzing different methods and technologies used in DECS. A brief introduction to the concepts of graphical user interfaces is made followed by an analysis of some vulnerabilities and considerations on software security. Finally the software testing techniques and technologies used in this work are presented.

2.1 Evolutionary Algorithms

Evolutionary algorithms are understood as population-based stochastic direct search algorithms that, in some sense, mimic the natural evolution process, as defined by Charles Darwin in [7]. The idea behind this process relies on the principle of natural selection, where the best individuals are selected for reproduction, combining their best features and passing them on to the next generation, leading to a positive evolution, and the weakest individuals are deprived of reproduction, preventing their features from spreading to the next generation. Points in the search space are considered individuals or solution candidates, which form a population.

The invention and development of the first evolutionary algorithms are nowadays attributed to a few pioneers who independently suggested four related approaches. Fogel et al. (1965) introduced *Evolutionary Programming* (EP), aiming at evolving finite automata and later solving numerical optimization problems [25]. Holland (1973) presented *Genetic Algorithms* (GA), using binary strings, which were inspired by genetic code found in natural life, to solve combinatorial problems [17]. *Evolution Strategies* (ES) as proposed by Rechenberg (1971) [29] and Schwefel (1975) [36] were motivated by engineering

problems and thus mostly used real-valued representation. *Genetic Programming* (GP), suggested by Koza (1994), emerged in the early 1990s [21].

2.1.1 Evolutionary Cycle

The evolution process on EAs follows an iterative cycle that can be generally defined by Figure 2.1.

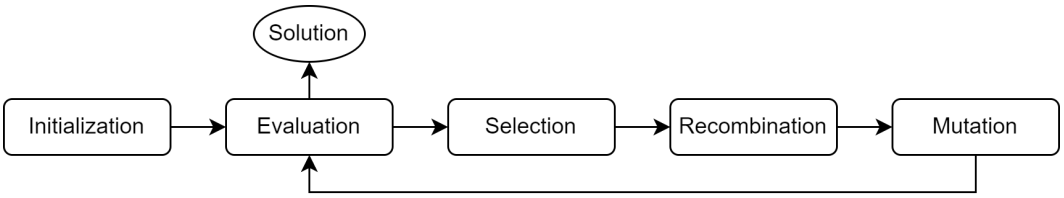


Figure 2.1: Evolutionary Cycle

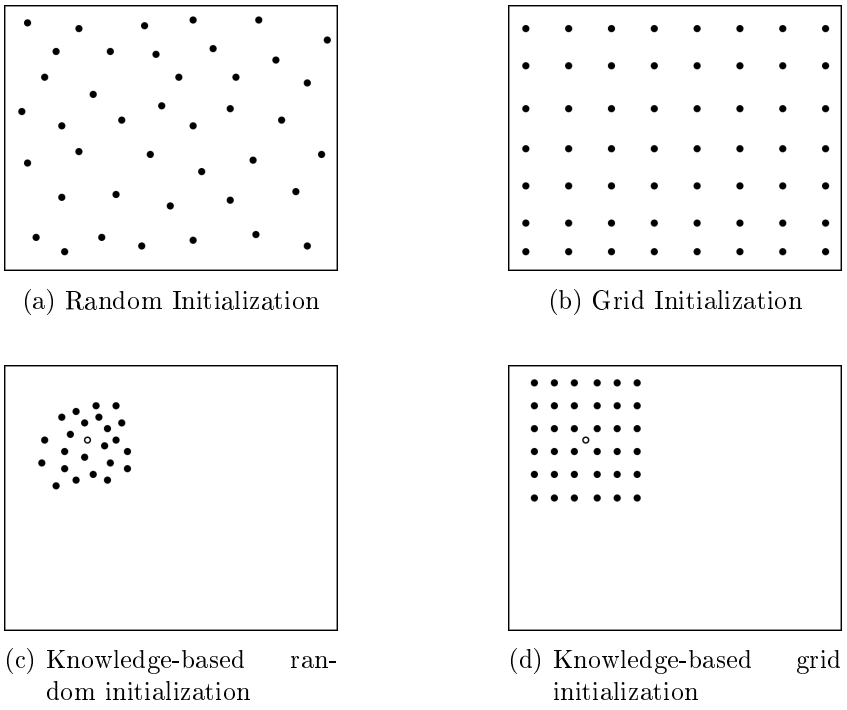
In the **Initialization** phase of the evolutionary cycle, the initial values for each individual are defined. Figure 2.2 represents the various methods that can be employed to generate the initial population [45, p.20,21].

In most cases, where there is little or no knowledge about the problem's solution, the population can be initialized with random values, ideally using a uniform distribution. This creates a simple and straightforward initialization method called *Random Initialization*, illustrated in Figure 2.2a, which ensures good coverage of the search space and increases the potential for breeding better solutions.

In some cases, less search space coverage is required, and a simple grid layout, which disposes individuals uniformly across the search space, is enough to find a solution. The *Grid Initialization* method, illustrated in Figure 2.2b, implements this concept and can be more efficient than random initialization, as generating a large amount of random numbers can be demanding.

When specific knowledge about the objective function is known and there is an understanding of what a reasonably good solution looks like, it is possible to incorporate it into the initialization procedure. Previous solutions can be integrated into the search space by assigning them to individuals. Placing the remaining ones randomly distributed (Figure 2.2c) or organized in a grid (Figure 2.2d) around those solutions can improve the evolutionary efficiency by focusing on key areas of the search space. However, this can prevent

the population from advancing beyond average quality to achieve high-quality solutions by narrowing the search space exploration. For some problems with specially complex features, such as multimodal functions and their multiple local optima, as explained in Section 2.2.3, this initialization method might not be as effective as random initialization.



(a) Random Initialization

(b) Grid Initialization

(c) Knowledge-based random initialization

(d) Knowledge-based grid initialization

Figure 2.2: Gene Initialization Methods [45, p.21]

The choice of initialization methods depends on the specific study context. While random initialization is mostly used in general EA investigations, for real-world applications, it is usually recommended to incorporate as much expert knowledge as possible.

In the **Evaluation** phase, each individual receives a quantitative quality characterization that is calculated using a predefined fitness function and expressed in a numerical value. This function measures how well an individual solves a given problem. For example, in a genetic algorithm minimizing a mathematical function, fitness might be determined by how closely an individual's output is to zero.

To ensure efficiency and performance in this phase, it is important to optimize the fitness function since it will be executed numerous times on each cycle. There are problems where

the fitness function is naturally complex, as further analyzed in Section 2.2.2, and it is imperative, especially in such cases, to meticulously consider its optimization in order to avoid unnecessary resource consumption.

In this phase, the cycle termination condition is verified by comparing the fitness values of each individual against the fitness of the solution. Once a solution is found, the cycle stops, and the algorithm successfully solves the problem.

Selection is a crucial process in the Evolutionary Cycle that both removes individuals with a low fitness value and drives the population towards better solutions, defining how the algorithm updates the population from one iteration to the next. The survival rate of an individual can be controlled by selection pressure. Excessive pressure tends to force convergence towards a narrow portion of the search space, potentially causing premature stagnation on a suboptimal solution. Conversely, insufficient pressure leads to slow convergence, prolonging the search process [27, p.14,15].

The *Tournament Selection* is a commonly used method to select the best individual from a group. Each tournament begins with a random selection of k individuals, whose fitness is then compared, with the highest fitness individual being assigned to the slot. This process is repeated for each slot in the population of the next generation. This way, it is possible to filter the best individuals and increase the population efficiency on each generation. The tournament size, typically set to two individuals and occasionally extended to five, prevents premature convergence that can be caused by excessive selection pressure [45, p.22]. Figure 2.3 shows the tournament selection process with $k = 3$.

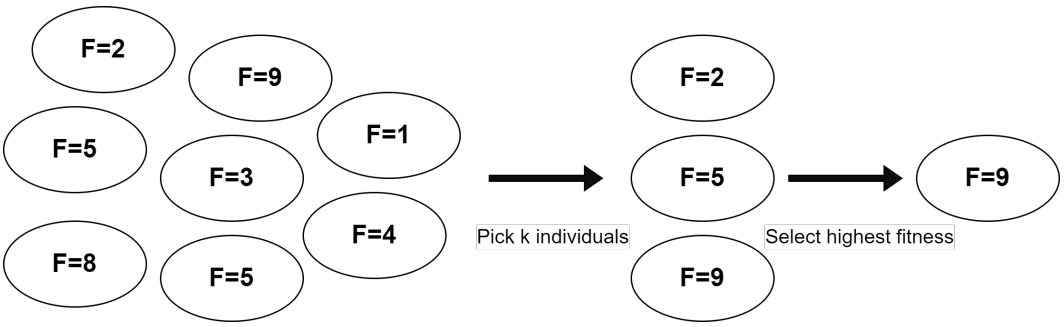


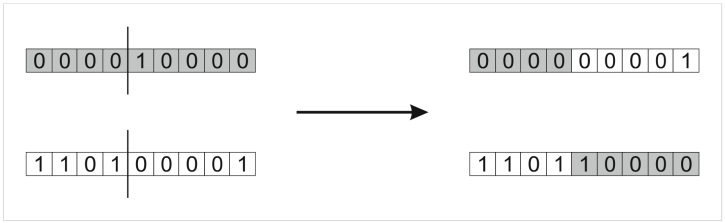
Figure 2.3: Tournament Selection Diagram

There are multiple selection operators suitable for different scenarios, but the tournament selection operator stands out for its ease of implementation, swift results, minimal com-

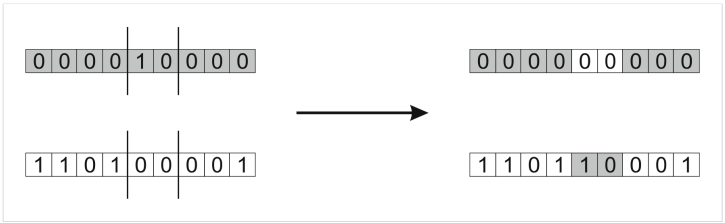
putational demands, and reliance on limited parameters. Consequently, it has become one of the most commonly used selection operators.

Recombination is considered one of the most important features in evolutionary algorithms, where a new individual solution is created from the information contained within two (or more) parent solutions [45, p.15,16], [38, p.45,46].

For binary individuals, the standard form of recombination starts with two parents and creates two children, with the possibility of extending this number. *One-point Crossover* works by choosing a random number, splitting both parents at this point, and creating two children by exchanging the tails (Figure 2.4a). Since one-point crossover can be easily generalized to *n-point crossover* (Figure 2.4b), where the chromosome is fragmented into multiple segments, each comprising contiguous genes. Offspring are then generated by selected alternating segments from the parents.



(a) One-point Crossover



(b) n-point Crossover

Figure 2.4: Crossover Types

Individuals can have several representations, such as integers or real values, which require specific recombination operators. Detailed information about other operators and methods can be found in the literature [10].

Mutation is the generic name given to variation operators that use only one parent and create one child by applying some kind of randomized change to the representation.

The form taken depends on the encoding used, as does the meaning of the associated parameter, which is often introduced to regulate the intensity or magnitude of mutation. Depending on the given implementation, this can be mutation probability, rate, step size, or even others [45, p.14,15], [38, p.49].

Mutation operators in EAs introduce diversity into the sampled population and are used in an attempt to avoid local minima. This is done by preventing the individuals from becoming too similar to each other, thus slowing or even stopping convergence to the global optimum. Most EAs adopt strategies that prioritize diversity over solely selecting the fittest individuals when generating subsequent generations. Instead, a randomized or semi-randomized approach is often adopted, with a bias towards individuals of higher fitness.

In binary-encoded algorithms, the most common mutation operator considers each gene separately and allows each bit to flip (i.e., from 1 to 0 or 0 to 1) with a small probability pm. The number of values changed is not fixed but depends on the sequence of random numbers drawn, so for an encoding of length L, on average, $L*pm$ values will be changed. Figure 2.5 illustrates the effect of a bit-flip mutation operator.

Figure 2.5: Bit-flip Mutation Operator

The mutation rate parameter has a direct effect on the evolution process. A lower mutation rate is less likely to disrupt good solutions, which can help to achieve a population in which all members have high fitness. A higher mutation rate helps to find one highly fit individual and is used when the potential benefits of ensuring good coverage of the search space outweigh the cost of disrupting copies of good solutions.

2.1.2 Genetic Algorithms

Genetic algorithms (GAs) belong to the larger class of evolutionary algorithms and are a type of computational optimization technique inspired by the principles of natural selection and genetics [38, p.35,36].

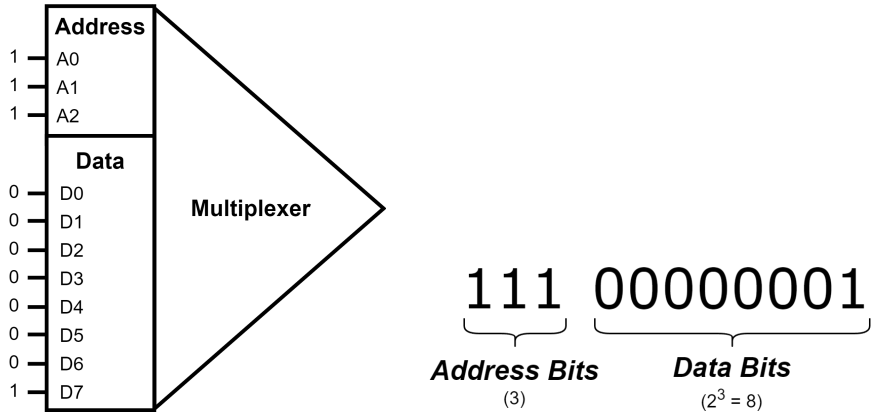
High-quality solutions to optimization and search problems can be found using GAs by iterating through the evolutionary cycle explained in Section 2.1.1. Individuals are often called phenotypes, and their set of properties, genotypes, or chromosomes.

2.1.3 Genetic Programming

Genetic Programming (GP) algorithms were initially developed to explore how computers could learn to solve problems autonomously without specific programming instructions. The GP technique is an evolutionary algorithm closely related to *Genetic Algorithms*, further explained in Section 2.1.2. When compared with GAs, some divergences become apparent. GP typically codes solutions as tree-structured, variable-length chromosomes, while GAs generally make use of fixed-length and structured chromosomes. GP usually includes a domain-specific syntax that dictates the permissible and meaningful configurations of information on the chromosome. In contrast, GA chromosomes generally lack any syntax constraints. GP employs genetic operators that maintain the syntax of its tree-structured chromosomes during the reproduction process. The use of this flexible coding system allows the algorithm to perform structural optimization. This can be useful for the solution of any engineering problem [38, p.141-143].

In 1985, Cramer [6] developed one of the first tree-structured GAs for basic symbolic regression. However, it was Koza in 1992 [20] and 1994 [22] who was largely responsible for the popularization of GP within the field of computer science. His GP algorithm, coded in LISP, was applied to a wide range of problems, including symbolic regression, robotics, games, and classification. Since this initial work, interest in the field has grown, with the first international conference on GP held at Stanford University in 1996 (GP'96). While still dominated by computer scientists, engineering applications have begun to appear.

The **Boolean n-bit Multiplexer** represents a series of single-step supervised learning problems, such as the 6-bit and 11-bit versions, which are conceptually modeled after the function of an electronic multiplexer (MUX), illustrated in Figure 2.6a. A MUX is a device that selects one of several input signals and outputs it. The 11-bit multiplexer problem, illustrated in Figure 2.6, involves generating a dataset of random 11-bit strings. For each bit string, the output value is determined by the three address bits and the value of the corresponding register bit they reference.



(a) 11-bit Multiplexer (b) Bit-string Input

Figure 2.6: 11-bit Multiplexer Problem

Figure 2.6b describes a possible problem with bit-string input. Each binary digit in this string represents a unique feature, which can be either 0 or 1. These features, also known as attributes or independent variables, include the first three as address bits. In the 11-bit multiplexer problem, the address 111 points to the register bit with ID = D7. The value of the referenced register bit is 1, so the class for this instance is 1. Depending on the context, the class of an instance might also be called the endpoint, action, phenotype, or dependent variable. The n-bit multiplexer problem is often used in benchmark tests due to its advanced scalability. As n increases, the complexity of the problem grows exponentially, providing difficulty control for different applications. This problem remains a valuable test case for studying the efficiency, robustness, and scalability of evolutionary algorithms.

2.1.4 Meta-EA

Optimization methods such as genetic algorithms and differential evolution rely on various parameters that influence their behavior and effectiveness in optimizing a given problem. Selecting these parameters manually can be a challenging and time-consuming task that is always susceptible to human misinterpretations of the success of the individual parameters.

A solution to this problem can be achieved by using a Meta-Evolutionary Algorithm. These algorithms are used to optimize the parameters of a second one by adding an evolutionary layer (Meta-Optimizer) surrounding the algorithm of the main problem

(Optimizer) [38, p.74]. Figure 2.7 illustrates the concept of Meta-optimization.

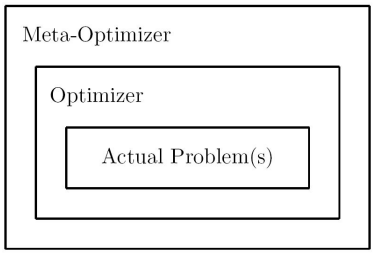


Figure 2.7: Meta-optimization concept

In practical terms, the population of the Meta-Optimizer is formed by individuals that represent the evolutionary algorithm that is trying to solve the actual problem (Optimizer) with a variable set of parameters that are being optimized.

To test an individual, the Optimizer algorithm is executed with its parameters N times. The mean best fitness over those N times becomes the fitness of the Meta-Optimizer individual holding those parameters.

2.2 Complex Evolutionary Problems

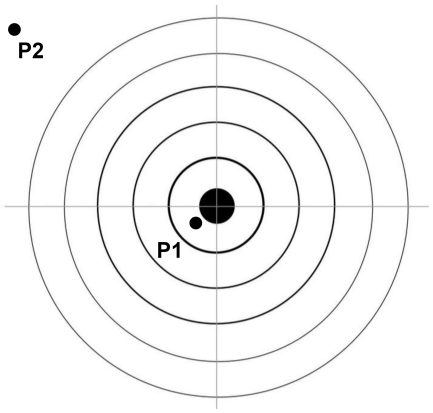
Each evolutionary problem has its own set of characteristics and challenges, ranging from complex fitness evaluation to a treacherous search space full of obstacles that can hinder the search for a solution or even intricate problem representations. This section covers the challenges posed by problems with a complex fitness evaluation in Section 2.2.2, and multimodal problems in Section 2.2.3. There are many more aspects that make a problem difficult to solve and can be further analyzed in the literature [11].

2.2.1 Fitness Function Aspects

In the context of numerical problems, a fitness function is typically defined explicitly through a mathematical equation. Yet, when it comes to real-world scenarios, problems are usually not well defined, leaving their representation at the discretion of the EA designer. The primary criterion is that the fitness function effectively evaluates individuals,

ensuring that the most optimal solution receives the highest fitness score. Failure to establish such a robust fitness function may lead to the erroneous selection of individuals in the formation of subsequent generations. There are still aspects that can lower the effectiveness of a fitness function, such as Plateaus, Ridges, Local Optima, and much more.

Plateaus [45, p.28] can happen when the fitness numerical representation is not precise enough. This aspect can be better perceived with a practical example represented by Figure 2.8. Assuming that a controller for a crossbow-shooting robot is being evolved. A simple fitness function could be defined such that the fitness of the individual is 1 if the robot hits the central target and 0 otherwise. In this setup, an individual that slightly misses the central target **(P1)** will receive the same fitness as an individual that completely misses the target **(P2)**. Hence, the EA is practically performing a random search if no individual can hit the central target in the initial population. This evaluation can be improved by incorporating the distance between the arrow and the central target as a contributing factor for the fitness function. Figure 2.8b shows a comparison between the rudimentary Plateau and improved fitness functions, represented by Tables 2.8i and 2.8ii, respectively.



(a) Target

(i) Plateau Fitness Function

Point	Fitness
P1	0
P2	0

(ii) Enhanced Fitness Function

Point	Fitness
P1	0.9
P2	0

(b) Fitness Function Comparison

Figure 2.8: Shooting Robot Problem

Ridges [45, p.31,32] in the fitness landscape are caused by the correlation among the problem parameters in the fitness function. To avoid a reduction in fitness and positively evolve individuals while handling ridges, it is necessary to change multiple problem pa-

rameters simultaneously, which can be a complex task. There are three solutions to overcome this problem. The fitness function can be rewritten, removing the correlation among the parameters. The problem can be handled by various extensions of the basic algorithms or through specialized alteration operators. The latter has been vastly investigated and further information can be found in literature [35] and [40].

Local optima [45, p.32,33] are a largely common phenomenon in real-world applications. It is represented by points in the search space where the current solution is the best among neighboring solutions, yet may not be globally optimal. The problem arises because, for the algorithm to positively evolve, it is necessary to explore further neighbor points and accept worse fitness values to leave the local optimum. A search space exploration can be achieved by increasing the ratio of mutation operators, which create more distinct solutions, but to achieve greater efficacy they should be combined with other techniques, such as Simulated Annealing, among others. Since deterministic local techniques only use one current solution to create new candidate solutions, these algorithms are largely affected by local optima. This often creates stagnation points where the algorithm can't leave the local optimum and effectively continue the evolution process. Problems with many local optima are called multimodal in the literature, and they are further described in Section 2.2.3.

Usually, the most time-consuming task in EAs is the evaluation phase, often influenced by fitness function complexity. This indicates that the optimization of the fitness function itself can be a relevant step in the performance enhancement of evolutionary algorithms.

2.2.2 Complex Fitness Evaluation

In numerical problems, the fitness evaluation process is often exclusively the calculation of the fitness function with the individual's variables. This can be considered relatively simple and lightweight when compared with other types of fitness evaluation, such as simulation-based fitness functions or event meta-evolutionary problems.

There are problems where the success of an individual cannot be directly mapped by a simple function. These problems are often related to real-world scenarios and can only be evaluated within a complex simulation of the environment. Considering that a simulator cannot achieve perfect accuracy, and a solution obtained from a simulation may not maintain its performance when applied in the real system. One solution to this problem can be achieved by introducing some noise in the simulated environment and thereby

hoping that the evolved individual will be more robust against real-world adversities [45, p.35]. Figure 2.9 shows the fitness simulation-based evaluation process of a population.

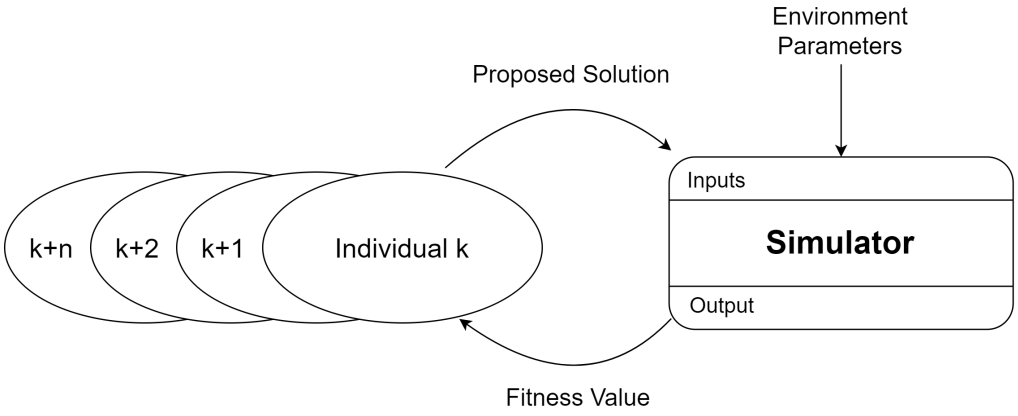


Figure 2.9: Simulation-based Fitness Evaluation Process

Another relevant challenge in this type of fitness evaluation is the processing complexity involved in some simulations, leading to a drastic increase in the time taken for each individual to be evaluated. Considering that some problems demand a large number of individuals per generation, this can also increase the overall processing time, making some problems impractical to execute. The problem can be further aggravated if the evolutionary process cannot be easily automated because each individual must be manually evaluated. A concrete example where an algorithm is being used in simulation-based optimization can be seen in the context of Aerodynamic Shape Optimization. The fitness evaluation can only be done through computational fluid dynamics simulations, which are resource-intensive operations.

On the other hand, Meta-EA problems, described in Section 2.1.4 can also have a complex fitness evaluation process, depending on the complexity of the optimizable problem. Considering that a large number of generations of the optimizable problem, containing multiple individuals that must be evaluated just correspond to one generation of the Meta-EA problem, it is possible to understand that the complexity of such problems can be largely scaled if some step in the evolutionary process contains inefficiencies.

Two distinct solutions can be applied to these types of problems using task distribution. The first can be achieved by using parallel processing techniques on a local computer. The second, which is more complex yet superior, is to implement task distribution among

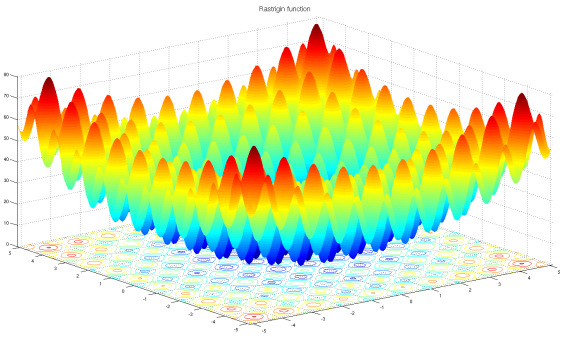
multiple computers. This technique is often mentioned in literature by the Master-Slave model or Distributed Evaluation. These two techniques are further explained and compared in Section 2.3.1.

2.2.3 Multimodal Problems

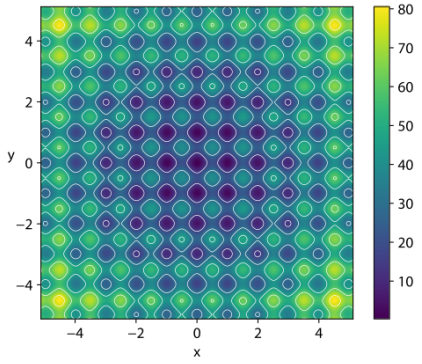
On the other hand, some problems have a fairly simple fitness function, that can be quickly calculated but have challenging attributes that make it difficult to achieve a solution. An example of such challenging attributes include functions with multiple local optima. This makes it challenging for optimization algorithms to find the global optimum [45, p.33,33].

Genetic Algorithms can be applied to minimize or maximize mathematical functions by treating its input variables as the genes of individuals within the population. The fitness is calculated by applying the genotype of each individual to the input variables of the function.

A popular function used in minimization problems for performance testing is the **Rastrigin function**. It is a non-linear multimodal function, first proposed in 1974 by Rastrigin [28] as a 2-dimensional function and has been generalized by Rudolph [32]. Finding its global minimum ($x = 0$ where $f(x) = 0$) poses a considerable challenge due to its vast search space and multitude of local minima. In Figure 2.10 it is possible to see the 3-dimensional and contour representations of the Rastrigin function and its multiple local minima.



(a) 3D Representation (Diego A. Alvarez, 2010)



(b) Contour Representation (Nico Schlömer, 2022)

Figure 2.10: Rastrigin Function Representation

In these types of problems, the objective is not related to computing performance, but to evolutionary efficiency. One solution to overcome the problem of local optima is the migration of individuals from other populations, running in parallel, with their own evolution. This introduces diversity into the population, which can help the algorithm explore a wider range of options, potentially escaping from local optima and discovering better global solutions. The concrete implementation for this method is often mentioned in the literature by Island Model, and it is better explained in Section 3.2.

2.3 Distributed Computation

Large companies must constantly manage an immensity of tasks that need to be done in a limited amount of time, to ensure progress and efficiency. Complex evolutionary problems also demand a substantial amount of tasks, depending on the considered granularity, that when performed correctly, reach a solution. The bridge between these two concepts is task distribution. While large companies need a vast number of workers, complex algorithms need computers, or even more precisely, processing units. These can be in the form of CPUs (Central Processing Units), GPUs (Graphics Processing Units), or even ASICs (Application-Specific Integrated Circuits). But in this context, the amount of resources is not directly related to efficiency. A company can have thousands of highly qualified workers, but they could do not work well together as a team. To distribute tasks efficiently and reliably, the participants must communicate effectively with each other, and a distribution algorithm must be conceived and followed by all. There are multiple variants of task distribution protocols for computational tasks, each with its characteristics. The desirable outcome is a group of processing units that work perfectly together to solve a common set of tasks efficiently [43, p.17,18].

This section explains some key concepts on distributed computation.

2.3.1 Parallel vs. Distributed Processing

In parallel processing, tasks are executed simultaneously on multiple processors, cores, or threads within a single computing device. Although all types of parallel distribution seem similar, it is important to define a line between multithreading and multicore processing. While multithreading techniques focus on task distribution within a single processor core by switching between executing instructions from different threads rapidly, not being

truly considered parallel processing, multicore techniques focus on task distribution in distinct physically separated cores, making it far more efficient. Modern computers equipped with multi-core CPUs often combine both techniques to maximize performance [39].

In distributed processing, tasks are executed on multiple devices that are interconnected via a network and together form a distributed system. These systems are often geographically dispersed and share a common algorithm that enables cooperation in solving specific tasks. There are some factors that can limit performance, and since all modules must communicate with each other at some point, this can lead to unwanted overheads. In parallel processing, communication between processing units is usually very fast and involves low overhead since they share memory and are often tightly coupled. In contrast, communication between distributed components involves network message exchange, which incurs higher latency and overhead. This is due to factors such as data transmission over the network, potential failures, and network congestion.

In the context of scalability, parallel processing is constrained by the resources available within a single computing device. Adding more processors or cores can improve performance up to a certain point, but scalability beyond that may be limited by factors such as memory bandwidth and contention for shared resources. On the other hand, distributed processing offers greater scalability by allowing additional computing devices to be added to the network. This enables distributed systems to handle larger workloads and accommodate growing demands by leveraging resources across multiple devices. Although this does not mean that scalability in distributed systems is infinite, factors such as network load and communication overhead are aggravated with the addition of new machines, and this can limit the system's performance.

Both methods have their strengths and weaknesses and are suitable for distinct environments and applications.

2.3.2 Client-Server Architecture

In the conception process of any distributed system, one of the main components to design is the protocol. It defines how the system works, the interaction between nodes, and the roles of each node in the system. By combining the protocol demands and other external conditions, it is possible to establish an architecture for the distributed system.

The Client-Server architecture is a well-known and established architectural model that divides tasks or workloads between the providers of a resource or service, called servers, and service requesters, called clients [43, p.37]. Figure 2.11 illustrates the client-server architecture.

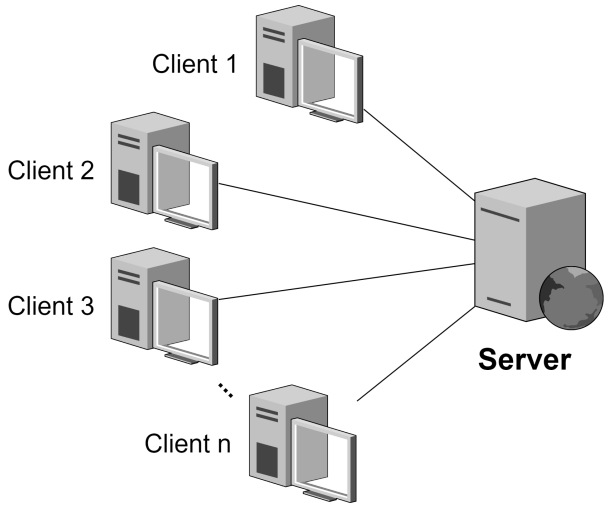


Figure 2.11: Client-Server Architecture

The server can be idealized by a central element that processes requests from clients and sends back the responses. The way the server manages client sessions can be further classified. A *Stateful server* retains client data, also known as state, across consecutive requests, storing session information. Additionally, it manages details such as accessed files, current read and write positions within files, file locking statuses by clients, and more. A *Stateless server* does not store any session state, which means that every client request is treated independently and is not a part of a new or existing session. In order to simultaneously process requests from multiple clients, they are often distributed across multiple servers to improve responsiveness, avoid traffic cohesion, and maintain the system's availability.

The client is the initiator node that is responsible for creating valid requests, sending them to the server, and waiting for a response. It is then crucial for this node to know the address of the server. The workload division between client and server can vary depending on the application demands, ranging from a fat-server to a thin-client, where most heavy processing is done on the server, to the other way around.

This architecture is used in many distributed applications, such as web servers, where the browser acts as the client, requesting web pages, while the web server fulfills these requests by sending the requested content [43, p.39]. In database systems, clients query and manipulate data while the server handles data storage and processing. Figure 2.11 illustrates the operation of both examples mentioned before.

Centralization is the key feature of this architecture, but the polarity of its effect depends on the specific application. Considering a cloud-based document management system, where all documents are stored on a central server. If this server experiences a technical issue or undergoes maintenance, all users are unable to access their documents until the server is restored. This example shows that the central server could become a single point of failure in some scenarios. In contrast, centralization can contribute to consistent data management, security, and control.

Another susceptible point of client-server architectures that is often exploited by malicious users is traffic congestion. Each server can handle a maximum number of simultaneous requests until it crashes or slows down its operation. The number of requests can grow organically, caused by an increase in demand, or artificially, by a DDoS (distributed denial-of-service) attack. The objective of such attacks is to generate an enormous amount of artificial requests using multiple machines for a specific service in order to overload its capacity, cause the servers to crash, and ultimately bring the service down. In this situation the objective is to mainly filter the authentic client requests from the artificial ones. However, this may pose a challenge. Figure 2.12 shows an example of a DDoS attack.

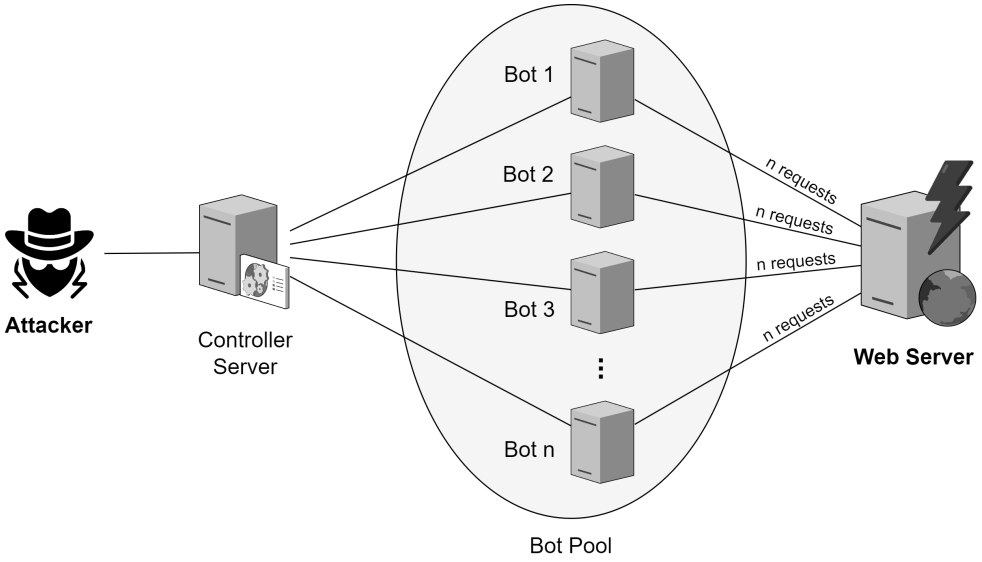


Figure 2.12: DDoS Atack Example

2.3.3 Remote Method Invocation

Communication is a key factor in distributed systems, allowing participants to exchange information and initiate actions. This is done through network communication protocols [43, p.120,121] such as UDP (User Datagram Protocol) and TCP (Transmission Control Protocol). While TCP is a connection-oriented protocol, meaning it establishes a connection between the sender and receiver before data transfer begins, UDP, on the other hand, is connectionless, as it does not establish a connection before sending data. This affects reliability, where TCP provides reliable delivery of data in the correct order but with a larger overhead compared with UDP, which does not provide any of the features mentioned before, but has a greater transmission speed.

In *Object Oriented* distributed applications, this communication is done between objects that can be geographically distributed across multiple machines in the form of **Remote Method Invocation (RMI)**, based on the general concept of **Remote Procedure Call (RPC)** [43, p.125,126]. In this communication protocol, the client-server architecture (Section 2.3.2) is used, where the client calls an object's method implemented in the server in the form of a request and the respective return value is sent back as a response. The widely used approach on implementing the communication channel is realized by using stubs and skeletons. They provide a communication abstraction between

both participants, as seen in Figure 2.13.

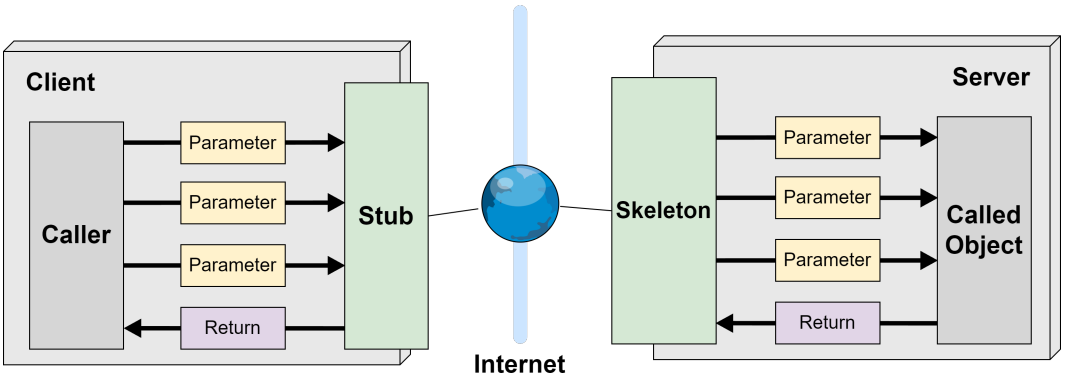


Figure 2.13: Remote Method Invocation

A **Stub** is the object that handles the communication protocol on the client machine. When a method is invoked, the stub initiates communication with the server skeleton, encodes the parameters into a data format suitable for network transmission (marshalling), informs the skeleton that the call should be invoked, and passes the encoded arguments to the skeleton over the network. When a response is received, the stub decodes it to environment-specific objects (unmarshalling) and finishes the process [43, p.128,129].

A **Skeleton** on the other hand, is the object that handles the communication protocol on the server machine. When a request is received, the skeleton translates incoming data from the stub to the correct up-calls to server objects, unmarshals the invocation arguments, and passes them to the object. When the local invocation returns, the skeleton marshals the result and passes it back to the client stub over the network.

In Java, this model is implemented through **Java Remote Method Invocation** [43, p. 461-463], facilitating distributed object communication among Java Virtual Machines (JVMs). A reliable, ordered, and error-checked communication is achieved by using TCP/IP as the primary network communication protocol. In practical terms, three components are required to use this protocol. A **Remote Interface**, where the methods available on the server are listed, including their parameters and return value. A **Remote Object**, where all methods listed in the remote interface are implemented. Finally, a **Remote Reference**, which is a reference to a remote object. This reference is provided by the RMI registry (Fig. 2.14), which provides a mechanism for locating and accessing remote objects. First, the server registers a remote object under a unique name, and

then the client can search in the registry for that name and get the respective remote reference to that object for later invocations.

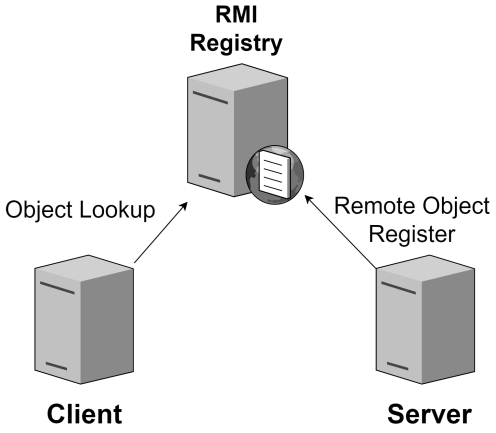


Figure 2.14: Java RMI Registry

2.3.4 Master-Slave Model

Several models and algorithms were proposed to efficiently distribute tasks among a group of computers, and their effectiveness is heavily dependent on the application. The Master-Slave is an asymmetric model that classifies each participant of the system in two distinct roles, master and slave, with different tasks and responsibilities [9]. Figure 2.15 illustrates the model's architecture.

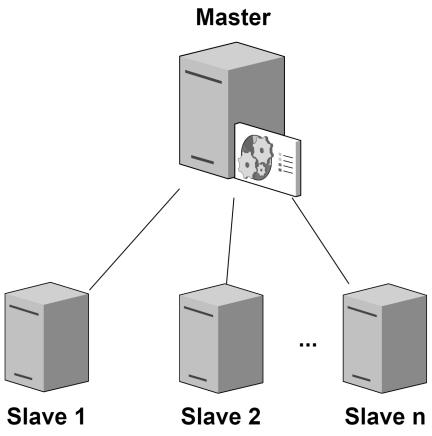
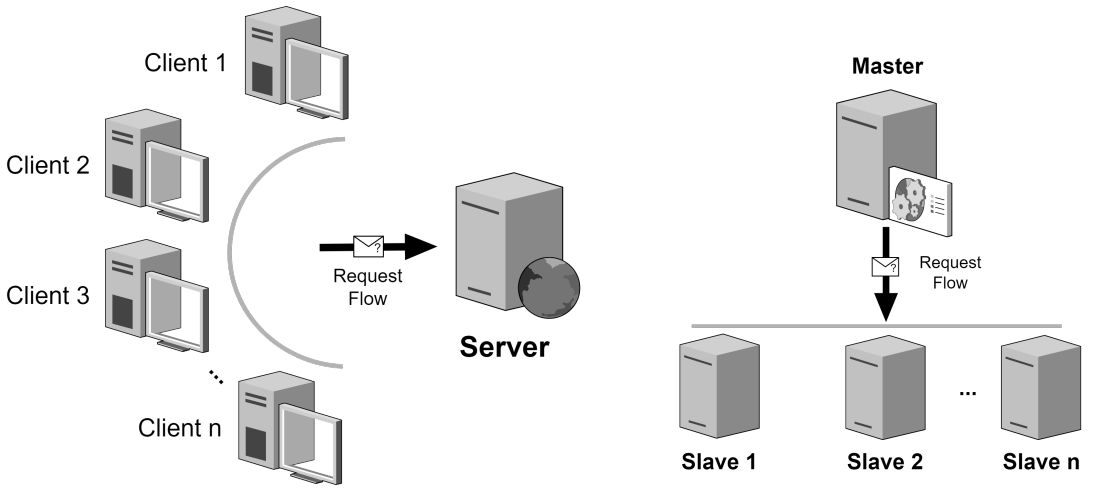


Figure 2.15: Master Slave Model

The **Master** is responsible for dividing the main task into smaller subtasks, assigning them to the slaves, and collecting the results. There can only be one master in the system, known to all slaves. The **Slaves** perform the specific tasks assigned to them by the master and report the results back upon completion. Each slave works independently and does not communicate with each other, which makes them loosely coupled and maintains the system's functioning even when errors occur. In the case of a crashed slave, the master can reassign the subtask to another slave and proceed normally with the algorithm.

The architecture used on this model can be compared to the *Client-Server* (Section 2.3.2) due to its similarities, having only one master/server and multiple slaves/clients. The primary difference lies in the tasks performed by each participant. The central entity (the server) in the client-server architecture only processes requests sent by clients. In contrast, the central entity (the master) in the master-slave model is the one that sends requests for slaves to process. Figure 2.16 establishes a comparison between the flow of requests in both Client-Server architecture and Master-Slave model.



(a) Client-Server (b) Master-Slave

Figure 2.16: Requests Flow comparison between Client-Server and Master-Slave

The Master-Slave model can also be implemented in the distributed processing of evolutionary algorithms. Section 3.1.1 explains how this model can efficiently distribute complex tasks in these algorithms.

The number of slaves in the system can positively affect the processing time of a large and complex task, but its logistical overhead can lower the performance of short and simple tasks. The impact of the number of slaves in a distributed evolutionary processing environment will be further analyzed in Section 6.

2.3.5 Communication Protocol

Humans developed several systems to exchange information between two physically separated individuals. One of these systems is the postal service, where it is possible to exchange letters containing written text from individuals around the world. As with any communication system, postal services implement a protocol, which defines a standard set of rules that everyone who wants to use the service needs to follow in order to maintain the system's operation. In distributed systems, the same concepts apply since all communications between nodes are made through network message passing. The main component of inter-computer communication is the protocol, that must be followed by all participating nodes to successfully exchange information.

When designing a communication protocol, there are some basic aspects that must be covered. The **structure and format** of data packets must be well defined in order to ensure consistent interpretation by all nodes. Each node must be able to decompose a binary-encoded packet into the different pieces of information it contains. Communication over the network is not always flawless, and errors can occur during the transmission of messages between nodes. Therefore, **detecting and correcting errors** in data transmission is a crucial aspect to consider. The timing and **synchronization** of message exchange must be coordinated to maintain the order and consistency of information.

A robust communication protocol can also detect attackers and non-legitimate participants by analyzing the behavior of every node. If the protocol is not being followed recurrently by a specific node, there is something wrong. This can happen due to a faulty participant or an attacker who is trying to learn how the protocol works. In both scenarios, that participant must be excluded from the group and their actions ignored until it reaches normal behavior again.

2.4 Technology

In this section, the technologies used throughout this thesis are introduced and briefly explained with simple examples. First, the Java-based Evolutionary Computation Research System (ECJ), used in the implementation of the Evolutionary Engine (Section 5.1), and then the Vaadin framework, used in the implementation of the web application (Section 5.2).

2.4.1 ECJ

ECJ is a Java-based evolutionary computation framework designed to meet large-scale, heavyweight experimental requirements. It offers tools that support numerous popular evolutionary computation algorithms and conventions, with a special focus on genetic programming. ECJ is licensed under the Creative Commons Attribution-No Derivative Works 3.0. It has been in existence for over fifteen years and has proven to be a mature and stable framework. Its design has been flexible enough to incorporate numerous enhancements, including multiobjective optimization algorithms, island models, master/slave evaluation, coevolution, steady-state methods, evolution strategies, parsimony pressure techniques, and various new individual representations such as rule sets. The framework is widely used within the evolutionary computation community and is one of the few implemented in Java at its scale. However, since ECJ was mainly designed for big projects and to provide many facilities, it comes with a relatively steep learning curve, both for usage and source-code contributions. One vision of the proposed system is to provide a straightforward and easy-to-use platform that simplifies some complex procedures in ECJ and allow for less-experienced users to take advantage of the framework's advanced features.

The basic functioning of ECJ starts with the construction of a *Parameter Database* object, which contains the necessary parameters to configure the evolutionary process and is based on an existing *Parameter File* (Section 5.1.1). Then the evolution can happen in two different ways. It can be triggered once and returned when it is finished, blocking the thread on which it is being executed, or instead triggered inside a cycle, where each iteration evolves one generation and the results and inference metrics are updated regularly. Both methods have minor performance impacts.

2.4.2 Vaadin

Vaadin is an open-source web application development framework for Java, licensed under the *Apache License 2.0*. This framework enables the development of modern web applications using Java code only, instead of web programming languages like HTML, CSS, JavaScript, or TypeScript, containing a set of web components for easy and efficient use. Featuring a server-side architecture, which means that most of the UI logic runs securely on the server, reduces the exposure to attackers and other threats. The client-server communication is automatically handled through WebSocket or HTTP with light JSON messages that update both the UI in the browser and the UI state on the server. Figure 2.1 shows a simple *Hello World Java class* that represents a *Vaadin view*, excluding the necessary imports and the respective output in Figure 2.17.

Listing 2.1: Vaadin "Hello World" Code

```
@PageTitle("Hello World")
@Route(value = "", layout = MainLayout.class)
@RouteAlias(value = "", layout = MainLayout.class)
public class HelloWorldView extends HorizontalLayout {
        private TextField name;
        private Button sayHello;
        public HelloWorldView() {
                name = new TextField("Your name");
                sayHello = new Button("Say hello");
                sayHello.addClickListener(e -> {
                        Notification.show("Hello " + name.getValue());
                });
                sayHello.addClickShortcut(Key.ENTER);
                setMargin(true);
                setVerticalComponentAlignment(Alignment.END, name, sayHello);
                add(name, sayHello);
        }
}
```

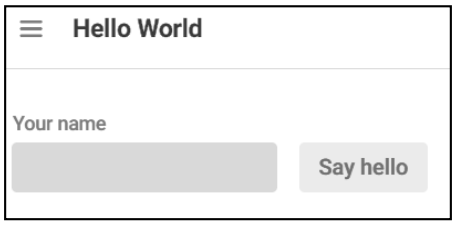


Figure 2.17: Vaadin "Hello World" Output

By analyzing the code, it is possible to understand that Vaadin web components such as *Buttons* or *Text Fields* are implemented as objects, and their attributes can be accessed in a typical Object-Oriented way. This creates a seamless integration with Java and the server backend code.

2.5 Graphical User Interface

A crucial step in the design phase of any system is to understand the target group that would be interested in using it. This influences the existence of some components and the way they are developed. In a context where a system will be only used by specialized users, it might not be necessary to have a user interface at all. This scenario changes when the system is intended for non-specialized users who don't have the technical knowledge to directly interact with a console application or other technical interfaces.

The next crucial decision is determining which type of GUI best suits the system's properties. While native applications can provide better performance with deeper customization and integration with the operating system and other local applications, they require an installation process where errors can happen, compromising their effectiveness, updates on each machine may be required and they are often operating system-dependent, compromising their compatibility. On the other hand, web applications are pieces of software stored on remote servers and delivered over the network through a browser interface, just like websites. In contrast with native applications, web apps can be accessed from any device that can reach the provider machine (server), they are platform-independent, eliminating compatibility issues, the updates are centralized on the server, which means that all users have access to the latest version with no required action and do not need to be locally installed. Since no solution is perfect, it can be limited by the capabilities of the web browser and may not provide the same level of responsiveness and interactivity as a well-optimized native application.

In the development of web applications, there is an important decision that should be made regarding the complexity and roles of both client and server nodes (Figure 2.18). This can go from one extreme with a thin client, which only partially processes the user interface, and a fat server, which takes care of all other operations and the preprocessing of the user interface, to the other extreme, where a fat client takes care of the user interface, application, data processing and a thin server that only sends raw data to the client.

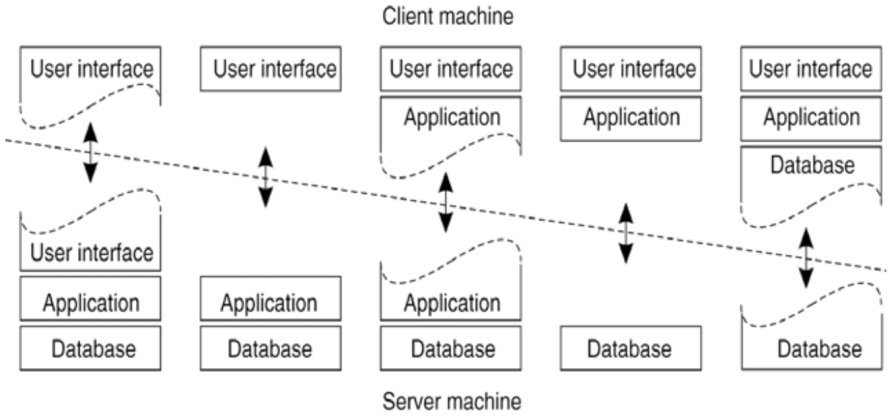
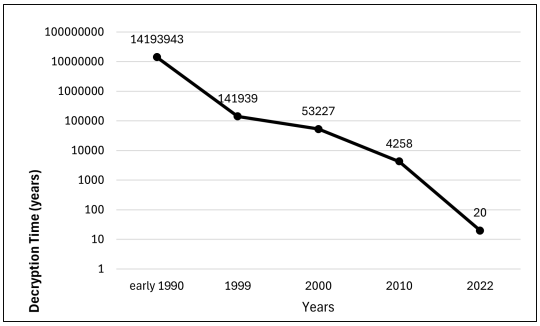


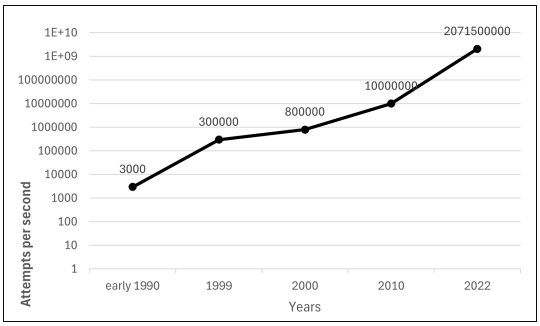
Figure 2.18: Client-Server Complexity Distribution [43, p.41]

2.6 Software Security

With the evolution of technology and computational power, an increased motivation for building safer software arises. For a long time, developers and IT security experts relied on difficult encryption techniques combined with low computational power, which made the decryption process impractically long and resource-intensive. At the time, this was seen as "impossible" within an acceptable time frame. To better understand this concept, Figure 2.19 shows a comparison between the maximum decryption time taken to brute force the password "3a4}71S:6qeE", hashed with the MD5 function [31] and computational power, expressed in the number of password attempts a piece of hardware can make per second within the past years.



(a) Decryption time over the years



(b) Attempts per second over the years

Figure 2.19: Brute Force Attack Statistics

In the early 1990s, a computer would need a maximum of 14193943 years to decrypt the password, which can be considered impractical in this context. The most interesting conclusion from both graphics is the computational evolution over approximately 34 years. Figure 2.19b indicates a staggering 690,499,333.33% increase between the number of attempts per second that could be performed in the early 1990s and recent years. Different hashing functions and encryption methods were created in order to adapt to the technological evolution, but encryption methods are only a small piece of the IT security field.

In this section several security techniques and vulnerabilities are analyzed.

2.6.1 Communication Security

The exchange of information between two nodes over the internet and Ethernet is transparent and visible to anyone. Network packets can be intercepted and their content analyzed, including the sender, receiver, protocol-specific information, and concrete content. This poses a challenge when dealing with sensitive and confidential information that should not be publicly known. The concept behind the solution for this problem is to create a new "language" that only the sender and receiver understand, so that it is possible to establish a public conversation between them with several other people listening, but no one, except them, understands the content. This is called encryption [43, p.389].

Hypertext Transfer Protocol Secure (HTTPS) is an extension of the Hypertext Transfer Protocol (HTTP), which uses TLS (Transport Layer Security) in order to provide encrypted, secure communication between client and server. This protocol is often used on websites and web applications in order to safely exchange sensitive information over the internet. It is also a form of server authentication that uses certificates to verify the integrity of the accessed website.

The exchange of unencrypted information between any two nodes in a network makes the system vulnerable to a variety of attacks. Probably the most known attack that exploits this vulnerability is the man-in-the-middle (Fig. 2.20), intercepting the exchanged packets and retrieving the information, which can be used directly or in other attacks, such as identity fraud.

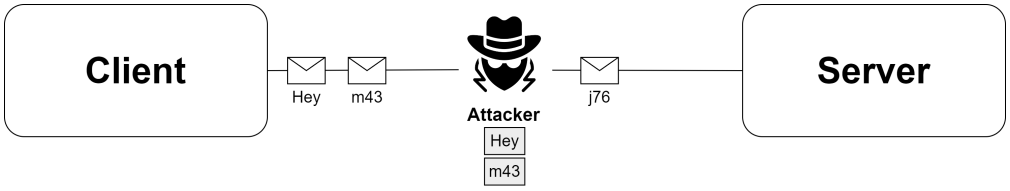


Figure 2.20: Man-in-the-middle Attack Example

2.6.2 Data Integrity

The integrity of data and entities is a critical aspect of information security and data management, ensuring data remains unaltered and trustworthy from its creation and entities follow a secure procedure. Attackers can intercept and modify sensitive information to exploit a system that lacks security verification [43, p.405-407].

In a web application, the attacker can focus on the manipulation of the client-side scripts in order to trigger actions on the server that were not supposed to happen in a regular case. A practical example of this scenario is the synchronization of the application state implemented in Vaadin (Section 2.4.2). The server is always aware of each client application state, which means that it will deny actions from components that aren't currently visible or active on the client screen. Considering an application containing a disabled button, this state is registered on both the client and server. The attacker can circumvent this since he has full control over anything in the browser. However, the server detects that the button is disabled and blocks any interaction with it, registering the event on the server logs for further investigation. This mechanism detects and prevents a fraudulent client from affecting the normal functioning of the web application.

2.6.3 Web Security

Attackers often explore the client components in the search for vulnerabilities. This technique has proved to be effective, and the result is creative and complex attacks that can cause damage to the server or even other innocent clients.

The **Cross-Site Request Forgery (CSRF)** is an attack that explores the open sessions of users with multiple web servers. With the emergence of cookies in modern web browsers, the user can now establish persistence connections, often called sessions with

multiple servers, through an identifier stored in a cookie. These sessions enable the ability to maintain the client authenticated on a web server for a limited period of time. This also means that every request made by the client's browser skips any sort of manual authentication if a session is still open. In CSRF attacks, the objective is to perform an authenticated action through the client's opened session without its consent. This can be done by crafting a special URL that performs the attacker's action and hiding it on another website where the client can be influenced to access it. Vaadin has a built-in mechanism that prevents CSRF attacks. When a client session is initialized, a user-session-specific CSRF token is generated and sent to the client's browser. This process is only repeated when the browser tab is refreshed. With every request between client and server, a user-session-specific CSRF token is sent and verified on the server. If it matches the initially assigned one for that session, the request is legitimate and processed. If it doesn't match or is not present, the request is ignored and the connection is closed. With the implementation of this protocol, the CSRF requests would not be approved because they wouldn't include the CSRF token.

There is also another type of attack that aims to execute a malicious script in the client browser. Here, the attacker exploits the lack of data validation in the HTML elements. Consider a webpage that contains an input field and uses the following JavaScript (Code Listing 2.2) to read its value and write it to an element within the HTML. Then the attacker crafts a special URL, containing a malicious script embedded in the input element value (Code Listing 2.3), and similarly to the CSRF attack, influences the victim to execute this URL. When the browser tries to render the value, the malicious script is executed in the victim's context.

Listing 2.2: JavaScript Code

```
var search = document.getElementById('search').value;
var results = document.getElementById('results');
results.innerHTML = 'You searched for: ' + search;
```

Listing 2.3: Input Value

```
You searched for: <img src=1 onerror='/* Bad stuff here... */'>
```

Vaadin has built-in protection against cross-site scripting attacks that use browser APIs that render content as text instead of HTML, using `innerText` instead of `innerHTML`. When this mechanism is applied to the previously described attack example, the malicious script will be rendered as text, exposing it instead of being interpreted as HTML.

2.7 Software Testing

A crucial part of the development of high-performance software is the evaluation of its behavior under different conditions and configurations. This ensures reliability and standard performance output. In complex applications that involve user interaction, such as web apps, manually testing all possible actions and ensuring that they are handled correctly by the system can be a challenging and inefficient task. To make things worse, web applications can be used by several users at the same time, which often raises numerous problems when developers don't use automated tests for this scenario.

2.7.1 End-to-End Testing

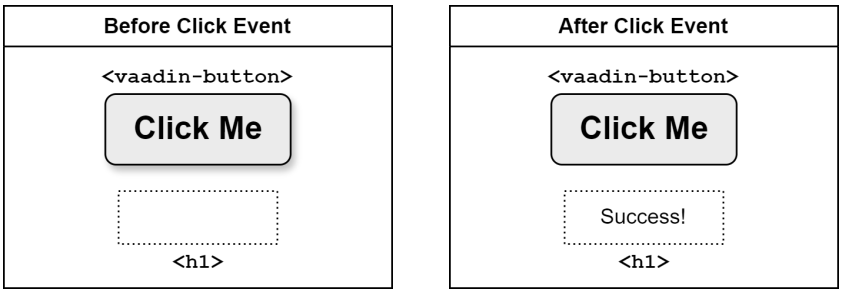
End-to-end (E2E) testing is a methodology used in the software development lifecycle that tests the functionality of an application from start to finish under real-life circumstances. The goal is to simulate what a real user scenario looks like by recreating a precise client-side environment and automating user interactions to form a complete use case. The completion of these tests does not only validate the system's integrity but also the sub-system's components [44].

In a web application, the client-side environment is a web browser running the application, where all automated interactions are performed by interacting with page elements and comparing the expected final state with the actual state. The definition of a successful system state can be composed of a specific set of verifiable conditions in the web application and in the internal system environment, including all sub-system components. One challenge in this type of testing in web applications is the identification and selection of specific elements in the web browser, which are often managed by frameworks that can make this process harder, as in the case of Vaadin web applications. Compared with unit testing, end-to-end tests are often more complex and time-consuming to execute due to their internal operations in recreating a precise client-side environment.

2.7.2 Playwright

To implement a seamless, integrated end-to-end testing solution for the proposed system, the **Playwright** automation library was used. This is an open-source library developed by Microsoft and provides the ability to automate browser tasks in Chromium, Firefox,

and WebKit with a single API. Together with **JUnit**, a test automation framework for Java, a complete end-to-end testing pipeline was implemented and further analyzed in Section 5.6. Figure 2.21 illustrates a simple example web page where an end-to-end test is implemented in Code Listing 2.4, using Playwright and JUnit.



(a) State before Click Event (b) State after Click Event

Figure 2.21: Example Web Page

Listing 2.4: End-to-End Example Test

```
1  public class PlaywrightTest {
2          static Playwright playwright = Playwright.create();
3          @Test
4          public void buttonClickTest() {
5                  Browser browser = playwright.chromium().launch();
6                  Page page = browser.newPage();
7                  page.navigate("http://localhost:8080/");
8
9                  assertThat(page.getByText("Click␣Me")).isVisible();
10                 page.locator("//vaadin-button[contains(text(),'Click␣Me')]").
                       click();
11                 assertThat(page.locator("h1")).containsText("Success!");
12         }
13 }
```

In this example, the automation library is used to click on the button element by first verifying if it exists and confirming that an <h1> element contains the text "Success!". Line 2 initializes the Playwright environment and only needs to be executed once. Lines 5, 6, and 7 launch a new Chromium browser, open a page and navigate to the local port where Spring Boot started the application. Afterwards, in line 9, it is verified that the button exists and is visible by first automatically waiting for the element to be loaded, as implemented in Playwright's assert methods. The button is then selected on line 10

using an XPath selector and clicked. Finally, the success message is confirmed to exist on line 11.

3 State of the Art

In this chapter, the current state of research is presented and explored. Section 3.1 establishes a knowledge base in the context of Evolutionary Distribution. This thesis focuses especially on the *Distributed Evaluation* and *Island Model* evolutionary distribution methods, in Sections 3.1.1 and 3.1.2, respectively. Finally, an overview of the current state-of-the-art evolutionary frameworks is established in Section 3.2.

3.1 Evolutionary Distribution

Evolutionary algorithms can, in some cases, be computationally difficult to process. This is due to some particular features, further explained in Section 2.2, among others. One solution to increase efficiency in the processing of these algorithms is to decompose and distribute the main problem.

Decomposing a complex algorithm into a series of independent sub-tasks can be a challenging process. However, the nature of EAs enables such division at different levels. Rivera [30] categorized four possible strategies to distribute EAs. Global parallelization, fine-grained parallelization, coarse-grained parallelization, and hybrid parallelization. In **Global Parallelization**, the fitness evaluation of individuals is distributed across multiple processing nodes. This technique is called *Distributed Evaluation*, and it is further explained in Section 3.1.1. **Fine-grained Parallelization** is frequently implemented on high-dimensional distributed systems, where each processing node is assigned a single individual, and interactions are limited to specific neighborhoods. This technique is particularly suitable for problems involving individuals with large and complex representations, where all operations within each individual are resource-intensive to process. In **Course-grained Parallelization**, several individual evolution processes occur concurrently on distinct processing nodes, enabling the exchange of individuals through migration between them. This strategy is often implemented in the form of the *Island Model*,

further explained in Section 3.2. Cantú-Paz [3] stated that understanding coarse-grained parallel evolutionary algorithms can be challenging, as the full effects of migration remain not entirely understood. However, this technique showed that it can be effective when dealing with multimodal problems (Section 2.2.3). Finally, in **Hybrid Parallelization**, several approaches are combined, and the level of hybridization in the algorithm defines the distribution complexity.

Gong et al [14] establish a comparison between the different distribution models and their features.

3.1.1 Distributed Evaluation

The evaluation phase can be the most resource-consuming process in the evolutionary cycle (Section 2.1.1). This is frequently seen in problems with complex fitness functions (Section 2.2.2), which are difficult to calculate, and large populations containing a large number of individuals.

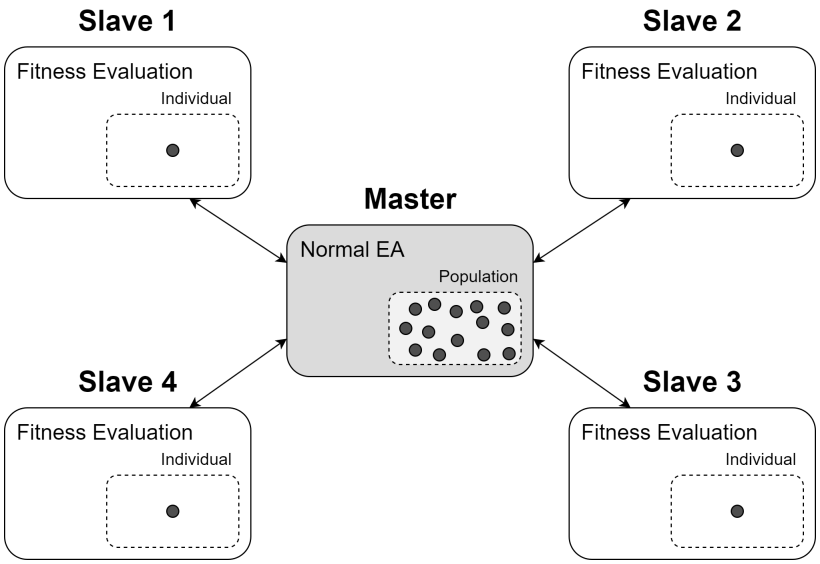


Figure 3.1: Distributed Evaluation [45, p.37]

One solution for these types of problems involves the distribution of the fitness evaluation process within several processing nodes to speed up the process and make the global evolution faster, as described in Figure 3.1. This is possible due to the independence

of the fitness evaluation task, which does not require a certain order or communication among slaves. It can be implemented following the *Master-Slave* model (Section 2.3.4) by having the master node execute the usual evolutionary cycle and its operations, except the fitness evaluation phase, which is then distributed among the slaves. Usually, this process happens synchronously, requiring the master to stop and wait until all fitness values are received from the slaves before proceeding to the next generation. There are also asynchronous variations, where the master node performs the selection operations on a fraction of the population only [33], [34].

Regarding scalability, distributed evaluation stands out when considering that by adding a new processing node to the processing network, a new CPU with, hopefully, multiple cores and its own ability to perform internal parallel processing by multi-threading is added. Since there are no perfect solutions, the limiting factor of this technique is the number and complexity of the messages exchanged by the nodes in the processing network and their respective communication overhead [4]. Dubreuil et al. [9] demonstrate that this distribution technique performs effectively when individual evaluation time significantly exceeds the message transmission time. Their experiment showed that solving a problem with a 0.25-second evaluation time resulted in 82% efficiency. When the evaluation time increased to 1 second, with the communication overhead unchanged, efficiency improved to 95%.

Fault-tolerance and population diversification can also be combined, as seen in [41], where a fault-tolerant distributed evaluation algorithm based on the master-slave model is proposed. When a certain evaluation fails and the respective fitness value is not returned in an acceptable time, its individual is replaced with a new random one, improving population diversity and solving the problem.

3.1.2 Island Model

The Island Distribution Model consists of a different approach to distributed computing. With this model, the objective is not to achieve better computing performance, but a more efficient evolution process. Comparing it with the *Distributed Evaluation* method, which aims to reduce the time taken for an evolution process to reach a certain number of generations or perform a certain number of evaluations, the *Island Model* aims to reduce the number of generations needed to achieve a global solution while escaping possible local optima. This is achieved by migrating individuals from different isolated

evolutionary processes, which are called islands.

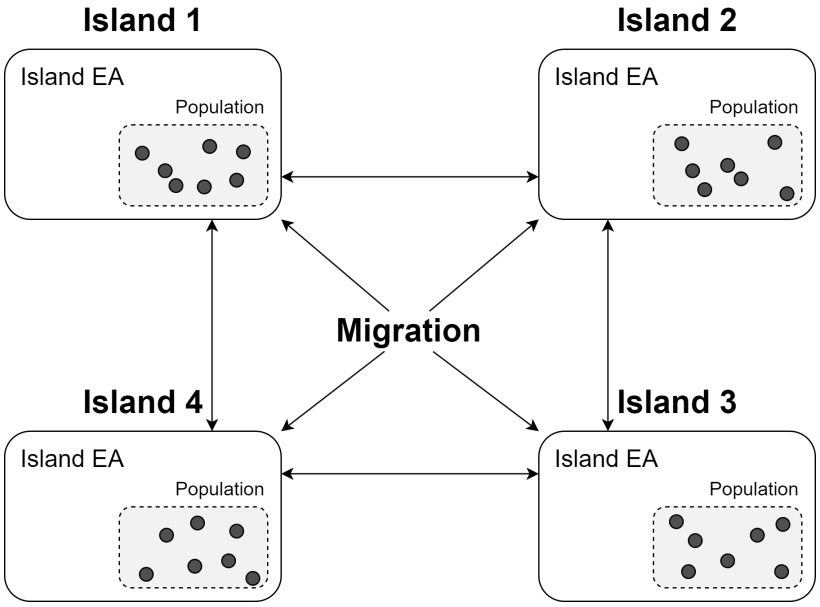


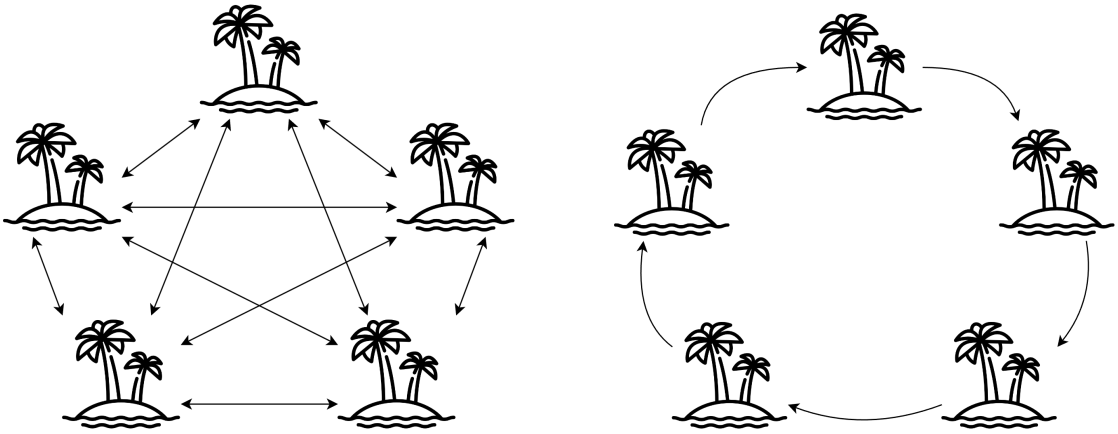
Figure 3.2: Island Model [45, p.37]

In practical terms, this is done by having each processing node on the network running an isolated evolution process (Island) and occasionally sending the best N individuals to several "islands" running on different nodes, as seen in Figure 3.2. When individuals from one island migrate to another, they are combined with the local population. This increased diversity can enhance the evolutionary process, leading to better and more efficient solutions.

These migrations can be strategically configured to follow a specific topology and order by changing their destination, frequency, and start generation. The parameter which defines the first generation when an island starts the migration process and its respective frequency sets the synchronicity of the migrations, where they can be synchronous, as seen in the literature [24], and [26], or asynchronous, in [23]. This affects not only the overall migration efficiency but also the network load. In synchronous migrations, the network load and processing time for sending and receiving individuals can be considerably high due to their peak usage. This can delay message transmission and negatively impact the performance of the global evolution process. Asynchronous migrations, on the other hand, can distribute network usage more evenly and minimize message overhead, thereby

alleviating network congestion. Hidalgo and Fernandez [15] suggest that the effectiveness of island-based dEAs is greatly influenced by the number of islands used and the resulting granularity. This could potentially limit the scalability of the island model.

The choice of migration topology should be carefully considered because it can have different impacts on the diversity and, consequently, the evolution efficiency of each island. In [18], multiple migration topologies are tested and compared. Figure 3.3 illustrates two possible migration topologies using five islands.



(a) 5 Islands Full Topology (b) 5 Islands Ring Topology

Figure 3.3: Island Migration Topologies

Full Topology (Fig. 3.3a) implements a logical structure where all islands are connected. This implies that the migration flow happens in both ways, and each island sends a migration packet to all other islands in the processing network, which allows for greater diversity in the global evolution process. However, when this migration topology is combined with synchronized migrations across all islands, it is important to prioritize the local individuals over the immigrants in the recombination process in order to avoid global convergence. This is due to the fact that all islands receive the same individuals when using this configuration, considering that each island selects its best N individuals for migration and recombination. Considering a processing network with N islands, structured following a *Full Topology* and synchronized migrations, on each migration there should be at least $N * (N - 1)$ migration packets in the network, the product of participating islands by the number of messages each island should send.

This poses a scalability challenge that should be considered when deciding on which topology to adopt.

Ring Topology (Fig. 3.3b) implements a typical logical structure where islands are connected as a ring. This means that each island is only connected to its direct neighbors, and communication typically follows a defined direction, often proceeding in a clockwise manner. Since each island only receives individuals from its direct neighbors, this implies a lower but more refined diversity in the global evolution process. The lack of diversity can prevent some islands from ever leaving local optima after entering one. Assuming that one island only receives individuals from its previous neighbor, if both islands are stuck in the same local optima, they won't be able to leave it and reach a solution because the immigrants won't contain the necessary diversity to evolve. Considering a processing network with N islands, structured following a *Ring Topology* and synchronized migrations, on each migration there should be at least N migration packets. When compared to the *Full Topology* (Fig. 3.3a), it's evident that the *Ring Topology* offers better scalability.

This model can also be combined with the *Distributed Evaluation* method by distributing each island individual's evaluation by other processing nodes in the network, as seen in figure 3.4.

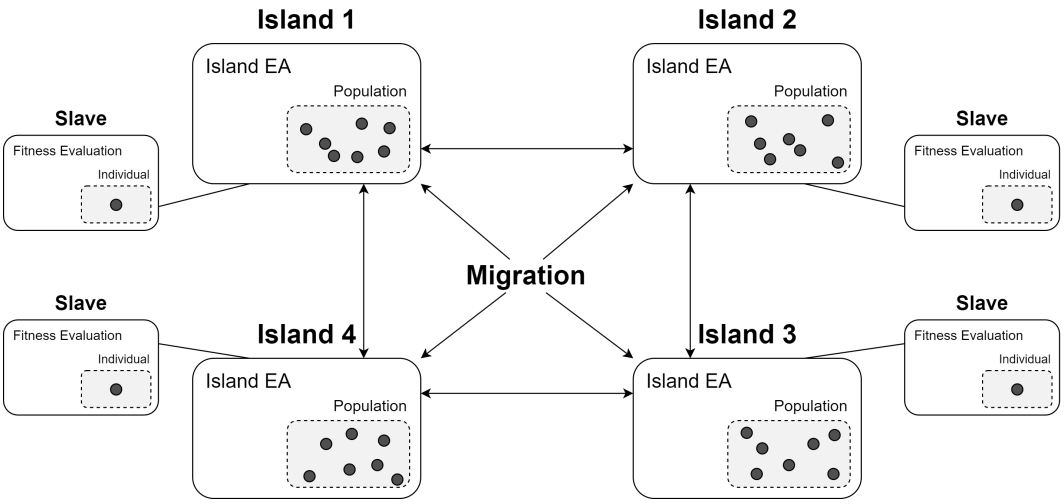


Figure 3.4: Combination of the Island Model and Distributed Evaluation

3.2 Evolutionary Frameworks

Several implementations of the previously described distribution strategies and algorithms have been proposed to efficiently distribute the evolutionary process and hide the intricate complexities of evolutionary algorithms. These often try to make every step of the process as explicit, easy to read, and understandable as possible, following distinct approaches.

BEAGLE (C++ Evolutionary Computation Framework) [13] follows a strong principle of object-oriented programming, where abstractions are represented by loosely coupled objects, making it easy to reuse code. However, only the Master-Slave model is supported for parallel fitness evaluation.

EO (Evolving Objects) [19] is a C++ framework that follows a component-based approach, where algorithm design consists of choosing components that serve its specific needs, "like building a structure with Lego blocks", as its creators describe it. These features enable support for more exotic problems, where the "building blocks" must be implemented from scratch. This framework only supports a more complex version of the Master-Slave model.

JDEAL (Java Distributed Evolutionary Algorithms Library) [5] implements the master-slave architecture in Java.

Paladin-Dec [42], also implemented in Java, supports genetic algorithms, genetic programming, and evolution strategies, featuring dynamic load balancing and fault tolerance. However, communication between nodes in the distributed architecture remains a challenge.

DREAM [1] uses the island model architecture on peer-to-peer connections.

ParadisEO [2] combines both island-model and master-slave architectures.

In a different approach, **DEAP** (Distributed Evolutionary Algorithms in Python) [12] features rapid prototyping and testing of ideas while seeking to make algorithms explicit and data structures transparent. Although it doesn't support distribution between multiple machines in the network, it works in perfect harmony with parallelization mechanisms such as multithreading and SCOOP [16].

Finally, **ECJ** (Java-based Evolutionary Computation Research System) [37] is a Java-based framework that stands out for its versatility, features, and well-representative documentation. It also includes multiple distributed computation features like the *Island Model* (Section 3.2), Distributed Evaluation (Section 3.1.1) over multiple processors using the Master-Slave model (Section 2.3.4), and support for generational, asynchronous steady-state, and coevolutionary distribution. It was the chosen framework for the DECS system evolutionary engine, as further explained in Section 2.4.1.

4 System Design

Based on the research question and its objectives, this chapter describes the design and architecture of the proposed system. It begins with a broader scope by analyzing the global architecture, including the external participants and their interactions. Afterwards the internal architecture of the main system roles, *Coordinator* and *Processing Node*, is described and explained. Finally, the vulnerabilities and security considerations are further analyzed.

4.1 Global Architecture

The DECS system is composed of two main architectural concepts that fulfill distinct requirements. On one side, the system should be able to provide a user interaction platform that is used to manage and control its functions in the form of a web application and follows a *Client-Server* architecture (Section 2.3.2). On the other side, it is necessary to process incoming evolutionary jobs initiated by the client through the web application and, if necessary, decompose them according to the chosen distribution model, managing the distributed evolutionary process in the form of a processing cluster, following a *Master-Slave* architecture (Section 2.3.4). A bridge must then be established between these two complex components in order to form a robust system, and this task is done by a central node called *Coordinator* (Section 4.2.1). The decision to host both components on the same node relates to the expected low usage of the web application due to the system's accessibility to only local networks. However, if justified, these can be easily separated and hosted on distinct machines, increasing their individual processing capabilities. Figure 4.1 represents the global architecture of the DECS system.

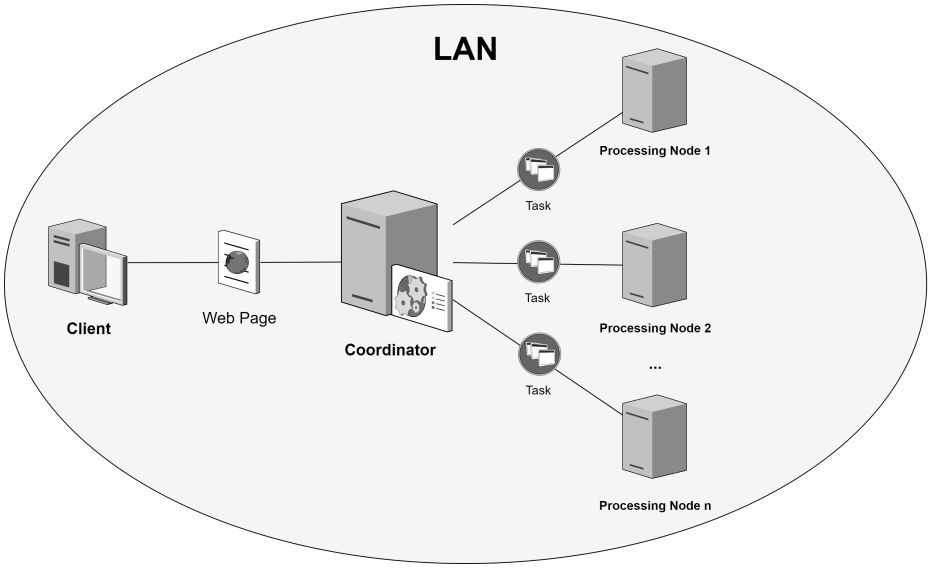


Figure 4.1: Global Architecture

This system has three types of participants, each with different roles and responsibilities. The **Coordinator**, or "Server", according to some literature, represents the system's central node. It is responsible for handling the web application engine, providing a web interface for each client that establishes a connection, and managing distributed evolutionary processing. The **Client** connects to the *Coordinator* with an HTTP request through a web browser and receives a functional web interface, further analyzed in Section 5.2. Finally, each **Processing Node** connects to the coordinator and processes incoming tasks. There are no general limitations regarding the number of processing nodes required for the system to function. In fact, DECS supports local evolutionary problems, without any distribution at all, but some distribution models can specify a minimum number of processing nodes required to process them. This is a common practice in the island distribution, where the specified number of islands must be respected and the problem cannot be processed if the number of connected DECS-Slave instances is lower than the limit.

The system was originally designed to be available only within the scope of a local network, since making it available over the Internet would require additional deployment and security considerations, further analyzed in Sections 5.5 and 2.6, respectively, both of which are not the main focus of this work. However, the system can be easily expanded to fulfill these features.

The idea of centralizing task distribution management on a single node aims to free up resources on the processing nodes, thereby maximizing their computational power and availability. This also opens the possibility of hosting multiple DECS-Slave instances on each Slave computer, increasing processing capabilities.

This design was specially conceived for distribution algorithms that rely on a central node to manage and coordinate the distribution process. In the case of the distributed evaluation method, this translates into a perfect implementation of the diagram presented in Figure 3.1, with a central node distributing tasks to several processing nodes. On the other hand, according to the island model diagram (Figure 3.2), there is no logical central entity, and each participant runs its own evolutionary cycle. However, in this distribution model, a central entity must coordinate some aspects, such as synchronicity between each island evolution cycle in the case of synchronous evolution, migration configurations, or even act as the initiator, providing each participant with the required information before the evolutionary process can actually start. The conceived system design proves to be compatible with both distribution models, allowing for effective implementation and achievement of DECS objectives.

The scalability of this system is influenced by both architectural models implemented, the *Client-Server* in the web application context and the *Master-Slave* in the distributed evolutionary processing. However, both models share a common scalability weakness in the central node, or in this case, the *Coordinator*. Both clients and processing nodes are dependent on the coordinator to properly function and assure global system efficiency. The limitations of the coordinator should be carefully tested and analyzed in order to understand at which point the system starts to lose its effectiveness with the increase in clients and processing nodes.

This dependency also creates a typical, but dangerous, if not taken into account, feature of these architectural models that is known as the single point of failure. Considering a scenario where a client suddenly fails and disconnects, the system will be able to continue its normal processing and even continue serving other clients. In another scenario, where a processing node fails, the coordinator redirects its tasks to other available processing nodes, or if any are available, the coordinator itself continues the processing. The same doesn't happen when the coordinator fails. In this case, the system collapses, and no functionalities are assured. To prevent this scenario, consistent maintenance and supervision of the coordinator instance are essential.

4.2 Internal Architecture

4.2.1 Coordinator

The *Coordinator* has a complex set of tasks due to its multiple responsibilities and management duties. It is composed of four modules with distinct roles, as represented in Figure 4.2.

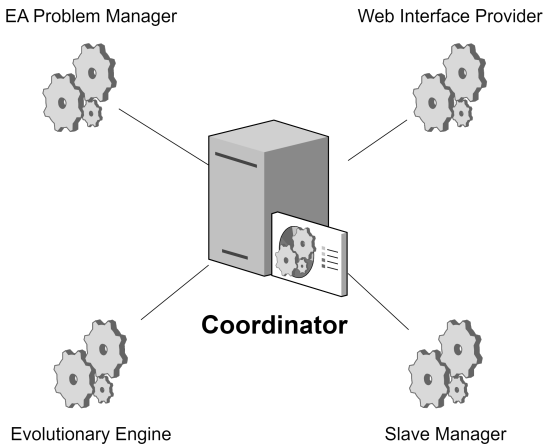


Figure 4.2: Coordinator Internal Architecture

Starting with the **EA Problem Manager** module, which is responsible for managing the system's evolutionary problems repository. Within the context of DECS, a *Problem* is defined as a group of parameter files, further explained in Section 5.1.1, that configure a specific evolutionary process, including its distribution parameters, if any. It also includes a special configuration file with the extension ".conf" which contains serialized meta information regarding the problem characteristics. This module hides the complexities behind creating and reading problems in the local problem repository and provides a high-level interface for handling these operations.

The **Web Interface Provider** module is an HTTP web server that processes requests from clients and provides them with a stateful dynamic web application, further explained in Section 5.2 and allows easy user interaction with the coordinator, hiding the system's complexity. A session is maintained for each client, even if the browser is closed, in order to enable some enhanced features, like authentication recall, which prevents the user from being requested authentication every time it accesses the web server or storing results

from past jobs. This session is maintained for a limited time when there is no user activity, which positively impacts resource management. However, handling multiple sessions with the same resources can raise numerous problems, which are covered in Section 5.3.

The Coordinator's core is the **Evolutionary Engine**, which is responsible for handling the evolutionary process and its respective distribution while returning regular feedback on the evolution and ultimately a solution. It is also responsible for performance metrics collection, such as Wall-Clock time, which refers to the real elapsed time, including all waiting operations, and CPU time, which measures the time the CPU spent processing the required instructions, excluding all waiting times, for the respective evolution. These measurements and other other operations made in the evolutionary cycle can affect its performance, so it is important to take some considerations in order to maintain efficiency and speed. The internal functioning of this module is further explained in Section 5.1.

Finally, the **Slave Manager** is the module that handles the three phases of a Slave lifetime, also called processing node. The first phase is the initial connection to the coordinator. This is done by a submodule that continuously listens for incoming registration messages from processing nodes and adds them to the processing pool (Section 5.4.1). In the second phase, tasks are assigned to registered processing nodes. This is done by another submodule that handles the necessary communication procedures in order to exchange information and trigger actions on the respective processing nodes which provides a high-level interface that hides the complexity of these protocols, as further explained in Section 5.4.2. Finally, the third phase is triggered when a processing node disconnects from the processing pool, either by failure or intentionally. In order to detect both scenarios and prevent unnecessary ghost management, which means that the *slave manager* is trying to manage a node that does not exist anymore, consuming valuable resources in the meantime, an *Heartbeat System* was developed and further explained in Section 5.4.3.

4.2.2 Processing Node

The **Processing Node** is a vital part of the system, contributing processing power to the pool and hopefully increasing the efficiency and overall speed of each evolutionary process. It must be carefully designed in order to maximize the computational resources available for the processing of the incoming tasks. This implies a simple architecture

with limited unnecessary functionalities and the lowest number of internal concurrent tasks. Figure 4.3 represents the internal architecture of each *processing node*.

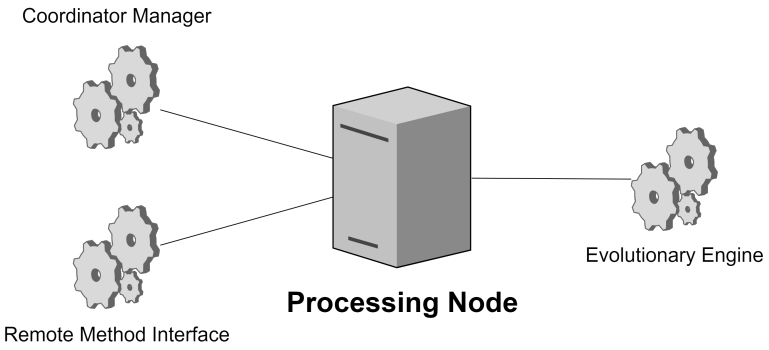


Figure 4.3: Processing Node Internal Architecture

The **Coordinator Manager** is a simple network communication module that interacts with the coordinator's slave manager in order to register the respective Slave in the processing pool (Section 4.2.1).

The **Remote Method Interface** module handles the RMI (Remote Method Invocation) components in order to provide a functional interface for the coordinator to remotely invoke the necessary methods in each processing node. The detailed process is further explained in Section 2.3.3.

Finally, one of the most important and resource-consuming components of the processing node is the **Evolutionary Engine**. This module handles the evolutionary processing remotely triggered by the coordinator and the specific distribution communications.

Each *Processing Node* can host several DECS-Slave instances concurrently, making it possible to accept multiple tasks at the same time and maximize efficiency. However, since this technique requires the use of internal multithreading, when used excessively, it can negatively impact performance. This underscores the importance of analyzing the number of DECS-Slave instances that a machine, with a set of hardware features can handle before the performance is affected. This analysis is performed in Section 6.

4.3 Security Considerations

In this section, several security measures and vulnerabilities, previously explained in Section 2.6 are analyzed in the context of the DECS system and future improvements are formulated and suggested.

Communication Security

Private communication is an important security feature of distributed systems, as further explained in Section 2.6.1. In the context of DECS, the importance of supporting HTTPS in the web-application connection is defined by two factors. The confidentiality of the information visible to the user and the availability of the system. Considering that the information present in the web application needs to be transported from the server to the client's web browser and that this communication can be intercepted, it is important to evaluate its confidentiality. This can be easily escalated if the system can be accessed outside a controlled local network.

It is also important to decide if the communication between *Coordinator* and *Processing Nodes* should be encrypted. The decision should follow the same factors, as mentioned before, but with slight differences. While the evolutionary information sent to the client is limited, in the internal evolutionary distribution communication, concrete individuals are exchanged, possibly containing sensitive information. The location of the processing nodes should also be considered, while different measures are applied to Ethernet and Internet communications.

Since DECS is designed as an academic research tool, deployable in secure local networks, which is mainly focused on the distribution of evolutionary algorithms, these vulnerabilities must be taken into consideration and security measures should be implemented, posing a suggestion for future improvements.

Access Control

Private resources available within a local network or even in the Internet often need to be protected with some sort of authentication. This tries to assure that only the group of authorized users can access a resource. In the context of the DECS web-application, there are two main mechanisms which assure this condition.

One of the most popular authentication mechanisms is the **User Login**, which enables per-user access control. This can be useful when there is an access distinction between individual users, often called as roles. Considering a database that should be accessed by regular technicians that can only insert new information and administrators who have privileged access and can perform all actions. A simple and direct way to implement these features is through a login system that distinguishes the allowed actions according to the type of user logged in. In DECS, the same concept can be implemented in the form of *researchers*, which have full access to the web application and *guests*, which can only visualize the results of executed jobs and explore the system without making any change or action.

Data Integrity

The registration process of processing nodes can also present some vulnerabilities, as seen in Section 2.6.2. While DECS does not implement any integrity verification, any attacker can intercept the exchanged information in a regular registration process, reproduce the protocol in order to register a modified Slave instance and latter exploit the system. A possible solution for this vulnerability is to implement an integrity validation system using cryptographic key pairs [43, p.394,395] issued and managed by the system administrator and an additional random value in the registration protocol, called nonce. A key pair consisting of both private and public keys is generated for this purpose, ensuring that only the intended receiver can decrypt the message and view its content. The private key is stored and protected in the coordinator, and the public key is distributed for each processing node. The nonce represents a random value that must be different for every registration process and prevents an attacker from using a previously intercepted registration request to register the modified Slave instance. A possible registration protocol, where **`PN`** is a processing node and **`C`** is the coordinator, can be implemented using the following steps:

1. **PN** starts the registration process by sending a request packet to **C**, encrypted with **C's** public key, containing a random nonce value.
2. **C** decrypts the request with his private key and verifies if the nonce has already been used by another registration request. If the nonce is proved to be unique, it is stored in the database and the registration is successful. If not, the request is considered suspicious, the incident is reported and the registration fails.

With the implementation of this protocol, an attacker can intercept an encrypted registration request from a processing node. However, it cannot view the content since it doesn't possess C's private key, nor can it replicate the request because the coordinator will detect that the nonce was previously used in another request. This protocol also ensures that changing the nonce is not feasible, as any modification to the encrypted request would result in invalid data upon decryption, easily detected by the coordinator.

5 Implementation

This chapter outlines some crucial components and aspects of the proposed system (DECS). It begins with a preliminary explanation of the system's core components, starting with the *Evolutionary Engine* (Section 5.1) and *Web Application* (Section 5.2), followed by the analysis of the designed solution to manage multiple client sessions (Section 5.3). Afterwards, the conceived communication protocol is explained, including its phases and sub-mechanisms (Section 2.3.5). Finally, a brief introduction to possible deployment scenarios for the system (Section 5.5) is made and the implemented software tests are presented and explained (Section 5.6). The source code of the DECS system can be accessed in Appendix E.

5.1 Evolutionary Engine

The evolutionary engine is one of the main components of DECS. It is responsible for handling all evolutionary-related actions, including evolutionary distribution and the implementation of the evolutionary cycle (Section 2.1.1). The engine receives a **Problem** (Section 5.1.2) to process, which is first analyzed and the respective **Parameter Files** (Section 5.1.1) passed as an input to the **ECJ** framework (Section 2.4.1) that evolves the problem and returns a solution. This section briefly explores some components of the evolutionary engine.

DECS uses a custom compiled distribution of ECJ 27, which was carefully adapted to establish a robust interaction between both systems and solve some minor bugs in the island model distribution related to non-closing network ports and Input/Output (IO) errors. Each distribution contains several evolutionary problems with both behavioral implementations and parameter files, but it is also possible to implement new types of problems. The implementation process is

algorithmic-dependent, but there are some common requirements for all, such as how the population is initiated and how fitness is calculated. However, in order to be able to use them in DECS, it is necessary to include them in the ECJ source code, compile the framework, and include the new JAR in DECS.

5.1.1 Parameter Files

Evolutionary engines usually provide an extensive list of parameters that define the functioning of every component in the evolutionary cycle. Each framework implements this configuration system differently, differing in the parameter source and level of customization. Figure 5.1 establishes a relation between the configuration system and the user learning curve.

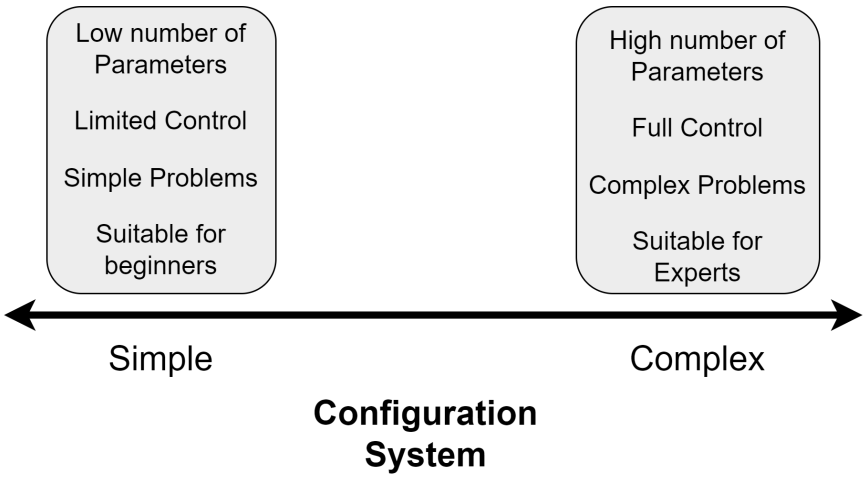


Figure 5.1: Configuration System Complexity

With the increase in evolutionary configuration complexity, more parameters need to be defined for each execution, which implies that the user knows the meaning and influence that each parameter has on the evolutionary process. For less experienced users, this can be overwhelming and overly complex in some contexts. On the other hand, low-depth configuration systems focus on the most important and relevant parameters, which limits control over the evolutionary process but makes the configuration process faster and simpler. ECJ has a steep learning curve, as

mentioned in Section 2.4.1, partly due to the high level of customization, expressed by the vast amount of parameters for each component of the evolutionary cycle. This is useful for research purposes, where advanced customization is needed and experts in EAs are operating the framework.

As mentioned in Section 2.4.1, each evolutionary execution must be configured in one, or more **Parameter Files**, which contain all necessary parameters for the components that are being used for that specific execution. They have a specific syntax and set of rules, that follow the concept of `<parameter> = <value>`, where the value must follow the specified type and range in the documentation. In the construction process of the Parameter Database, a parameter file undergoes syntax verification. Once verified, it is parsed, and the parameters are extracted and stored in the database. As part of this verification, it is ensured that the mandatory parameters are set and their values are valid. Code listing 5.1 shows an example of different syntax rules in ECJ parameter files.

Listing 5.1: Parameter File Example

```
# Example of an ECJ Parameter File
# Parameter 1 - Inheritance
parent.0 = fileB.params
# Parameter 2 - Class Name
pop = ec.Population
# Parameter 3 - Number
generations = 400
# Parameter 4 - Arbitrary String
crossover-type = two-point
# Parameter 5 - Multi-layer parameter
pop.subpop.0.size = 1000
```

Comments are defined using the symbol # at the start of each comment line. **Parameter 1** includes all parameters defined in the file "fileB.params". This is useful for complex configuration setups, where the separation improves readability and organization. Multiple files can be imported by adding similar lines but with the parent number incremented. **Parameters 2, 3, and 4** exemplify some of the different types of values a parameter can have. First, class names, including the respective package, then numbers contained in a range specified by the parameter, followed by arbitrary strings, which can select specific values of a

parameter. Finally, **Parameter 5** represents a multi-layer parameter, where it is possible to understand the structure of a parameter tree. This parameter derives from the "pop" parameter (*Parameter2*) and defines a value for the size of the first sub-population of the main population.

Parameter files are not natively secure. They are plain text files with the ".params" extension, which makes them easy to edit and read but inappropriate for secure contexts. In DECS, parameter files are neither encrypted nor protected, which makes them editable by everyone. Since they are the core of a Problem in the system, this can lead to security issues when dealing with sensitive information. To address this issue, multiple users with distinct roles can be created. Each user is then authenticated through the login system, and the privileges for accessing parameter files are managed based on their respective roles. For a more secure solution, each problem can be encrypted using a custom password in the creation process and decrypted for its execution and edition.

5.1.2 Problem

In the context of DECS, a **Problem** is a data structure object that contains the necessary information to configure the evolutionary engine for a specific scenario. A big part of this information is stored in parameter files (Section 5.1.1) that can be interpreted by ECJ. There are also other relevant problem-specific informations that must be permanently stored in the form of a configuration file with the extension ".conf", such as the problem code, full name, type, origin, and distribution type, among others. Each problem can be previously created and available out of the box with the system, having its origin defined as *factory*, or created from scratch on the fly by a user using the problem creator (Section 5.2.2), with the respective origin defined as *user*. Since each problem must have specific parameter file for the respective type of distribution, if any, this information is included in the characteristics of a problem, having as possible values both distributed evaluation, island model, and local if no distribution is applied.

The lifecycle of a **Problem** can be represented by figure 5.2.

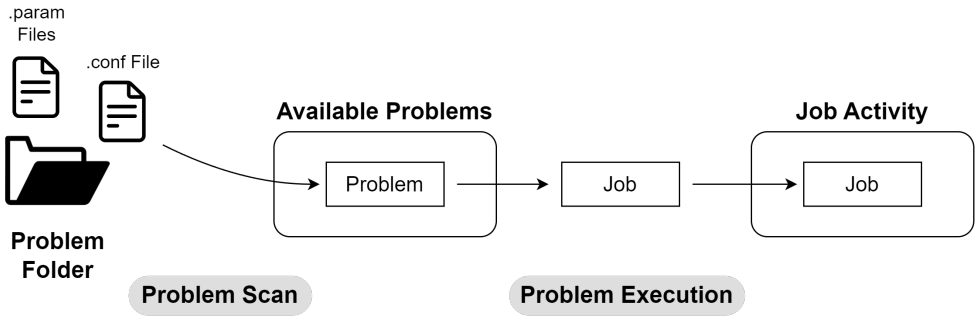


Figure 5.2: Problem Lifecycle Diagram

First, the problem must be imported from its permanent representation as a folder containing the respective parameter files and configuration file into a Java object. This is done through the deserialization of the configuration file, which suffered from the inverted process (serialization) when saved. When a problem needs to be executed, it first creates a new Job object using the required information in the respective problem object and passes it to the evolutionary engine to process the inference. After the job is executed, its object is then stored for future reference.

5.2 Web Application

Achieving a well conceived graphical user interface can be a challenging task and several aspects must be considered, further explained in Section 2.5. The DECS system is intended for non-specialized users with a fundamental understanding of evolutionary algorithms, from a user-side perspective. It is then essential to have a *Graphical User Interface (GUI)* for this target group. Following a simple and lightweight concept, a web application was developed with the Vaadin framework (Section 2.4.2). It implements a thin client and fat server, where the graphical user interface is pre-compiled on the server and sent to the client, where it just needs to be executed. This can be resource-demanding for the server, but it also poses some advantages, such as better responsiveness in the client's user interface, and an overall lightweight client.

The DECS web application is structured in three views with distinct functionalities, better explained in sections 5.2.1, 5.2.2 and 5.2.3.

5.2.1 Job Dashboard

This is the main view of the web application, where the primary functionalities of the system are located, as shown in figure 5.3.

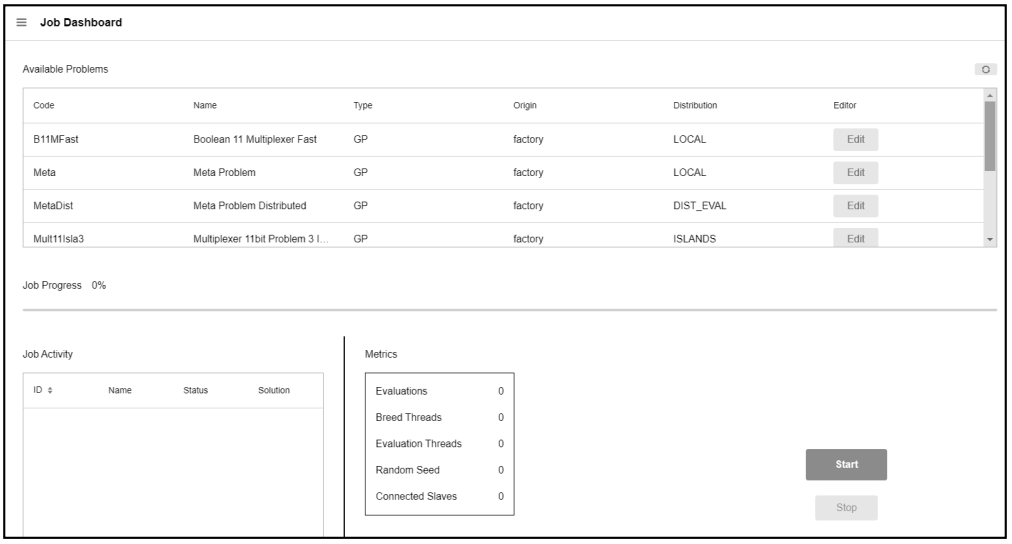


Figure 5.3: DECS Job Dashboard View

The upper panel displays a table of all available problems, offering basic information about each problem and the option to edit its *parameter files* (Section 5.1.1). To execute a problem, it must first be selected from this list. A job progress bar provides a visual representation of the current status and completion percentage of the ongoing job. This value is calculated by comparing the current generation of the evolution process with the respective stopping condition. Since it is impossible to precisely estimate in which generation a solution will be found, predicting when a job will terminate is not feasible, as a job concludes only when a solution is identified. This means that the progress bar does not need to reach 100% completion for a solution to be found. The bottom panel displays job-specific components, such as a job activity list, where running and finished jobs are listed, distinguished by their status value, and their respective solutions can be accessed. On the right side, the currently executing job real-time metrics are displayed and updated for each generation, as the start/stop control buttons.

5.2.2 Problem Creator

The process of creating a problem from scratch, including all the parameter files for the evolutionary configuration of ECJ (Section 5.1.1) can be complicated and time-consuming, especially for users unfamiliar with the framework. Addressing this issue, DECS provides a basic problem builder that maps the possible parameters into user-friendly components with data validation rules, eliminating possible errors. Since ECJ was originally conceived to be expandable, based on the supported features and algorithms, this creates a higher level of complexity in the parameter configuration, which gets close to a programming language. Creating a simple user interface that supports all the parameters and their combinational behaviors poses a huge challenge, which can be compared to the creation of a block-based user interface for a small programming language. One way to simplify this process is to allow the creation of new problems based on existing ones with the possibility of tweaking the most relevant parameters, with a special focus on the distribution configuration. Figure 5.4 shows the Koza tab of the problem editor. Each tab represents a group of parameters that a pre-built problem supports.

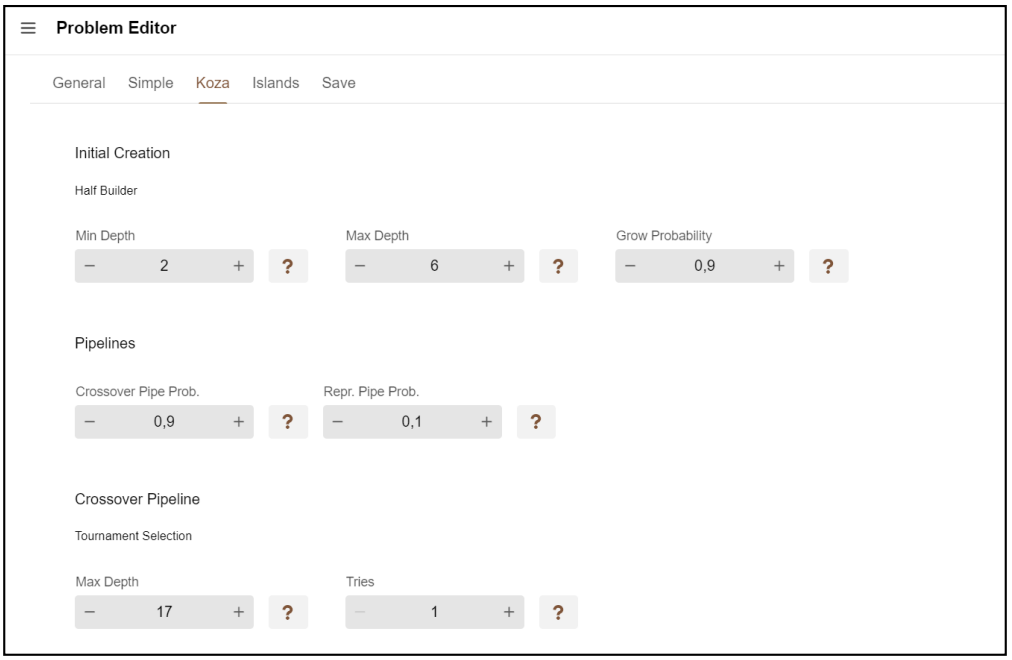


Figure 5.4: DECS Problem Creator View

Since the *Island Model* (Section 3.1.2) can be difficult to configure textually through property/value attributes, due to the complex migration customization possibilities, there is a special tab for this distribution technique, illustrated in Figure 5.5.

Figure 5.5: Island Model Tab

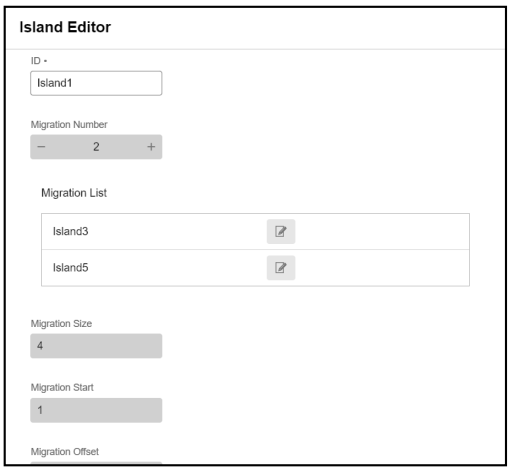


Figure 5.6: Island Migration Editor

The core components of this model are the islands, so a list is provided where these are displayed. Further individual island customization can be made in the *Island Editor* (Fig. 5.6), where it is possible to define the respective migration configurations.

5.2.3 Node Manager

The **Node Manager** (Fig. 5.7) is a view that provides information about the connected processing nodes. Each node can be further analyzed, retrieving real-time information about the hardware and characteristics of the machine in which it is being executed (Fig. 5.8). This information is often relevant when performing experiments with a specific cluster, where the hardware and software specifications of the processing nodes should be collected and included in the results.

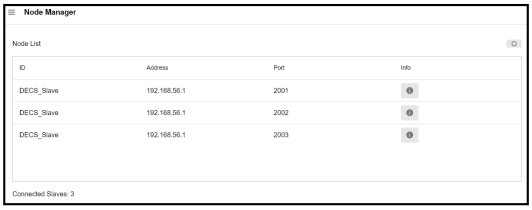


Figure 5.7: Node Manager View

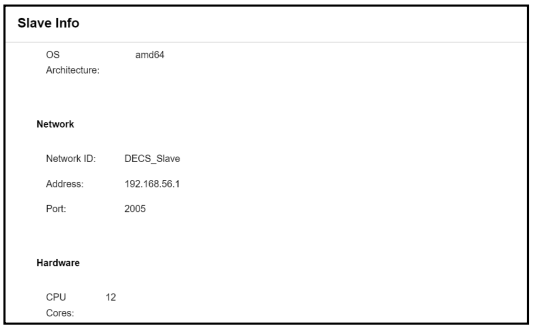


Figure 5.8: Node Information Popup

5.3 Multiple Client Sessions

Distributed systems must be widely available in order to fulfill clients needs while maintaining efficiency. This comes with a computational cost heavily influenced by the number of active users, which is often directly connected with the demand for concurrent tasks. Different types of distributed systems handle this in various ways. On simple websites, where client requests are usually fast and lightweight to process, it is easy to parallelize tasks and supply a large number of users without drastically reducing performance and availability. These systems can be implemented with good concurrency transparency, which means that multiple client requests can be processed concurrently without interfering with each other, giving them the sense that they are the only ones using the system. On the other hand, heavy web applications that provide time and resource-consuming services deal with several difficulties in request parallelization. Considering a popular type of web application that has emerged in recent years, due to advancements in generative artificial intelligence algorithms, which generate images and videos from textual descriptions. The service provided by these web applications is, at the moment, both time and resource-consuming which raises an efficiency problem when providing these types of services. This phenomenon can easily escalate with the number of active users on the web application and ultimately leaves two main solutions to tackle this issue.

The first solution involves the upgrade of existing hardware and the addition of new one, in order to fulfill the computational power needed to process all requests concurrently and, hopefully, maintain both efficiency, availability, and concurrency

transparency. This usually requires a substantial investment, which may not be a feasible option for every case. In the context of DECS, this could be implemented by increasing the number of processing nodes and the respective number of DECS-Slave instances running, analyzed in the *Technical Study* (Section 6) in order to increase evolutionary efficiency. Multiple *Coordinator* nodes (Section 4.2.1) can also be combined to increase task parallelization capabilities and consequently the number of active web application users while maintaining the expected functionalities.

The second solution involves a limiting factor, which controls the maximum number of concurrent tasks being processed by the system. This factor can be implemented in multiple ways, and its value is highly dependent on the system's computational capabilities since the objective is to maintain efficiency. Every parallelization and distribution method has an efficiency limit, where adding more tasks will negatively affect the efficiency of the global process. This concept is the main motivation for the limiting factor technique, where a system accepts just the right amount of tasks it can efficiently process before global efficiency starts to deteriorate. One practical example of this technique is the implementation of a service queue, where the system only accepts a certain number of requests and the others stay in a waiting queue before being processed sequentially. This respects the efficiency limit of the system's configuration and maintains a sequential task flow. However, the solution cannot assure concurrency transparency, because client's actions interfere with each other, which can lead to a worse user experience by possibly making users wait for long periods to get their tasks processed. DECS implements a simpler technique, which blocks all user sessions from executing new evolutionary processes when one is being processed. This can be scaled in the future to a waiting queue system, as illustrated in Figure 5.9 or an algorithm that efficiently distributes different evolutionary processes among Slave processes. Currently, the focus of this work is distribution techniques among one evolutionary process.

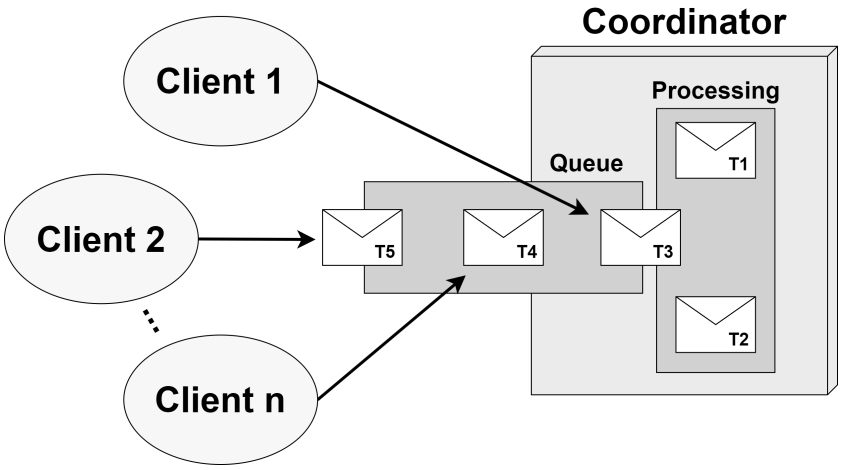


Figure 5.9: Queue System

The decision on which solution to adopt is highly context-dependent. If upgrading the system's computational resources is not an option, limiting the number of concurrent tasks poses advantages in multiple scenarios. A balance must then be achieved between user experience and system efficiency.

5.4 Two-Phase Protocol

Based on the concepts explained in Section 2.3.5, DECS implements a two-phase communication protocol involving both the coordinator and processing nodes. In phase 1 (Section 5.4.1), the processing node **(PN)** registers with the coordinator **(C)**, followed by phase 2 (Section 5.4.2), where remote method invocations can be initiated by the coordinator. Concurrently, the Heartbeat system (Section 5.4.3) verifies the status of each processing node, inferring their liveliness and detecting any errors. Figure 5.10 shows the communication protocol diagram.

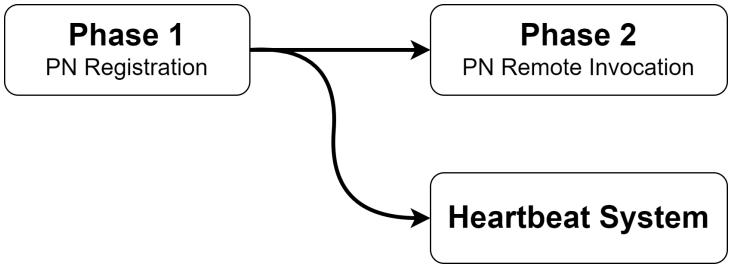


Figure 5.10: Communication Protocol Diagram

5.4.1 Phase 1 - Processing Node Registration

In the master-slave model, the coordinator usually doesn't have any pre-defined information about the processing nodes since they can vary in number and specific characteristics, such as their network address. The solution is to include processing nodes dynamically through a registering process, denoted by **Phase 1** of the communication protocol. For this process to work, each processing node must possess information about the coordinator, specifically its address, in order to establish an initial communication. Figure 5.11 illustrates the processing node registration sequence diagram.

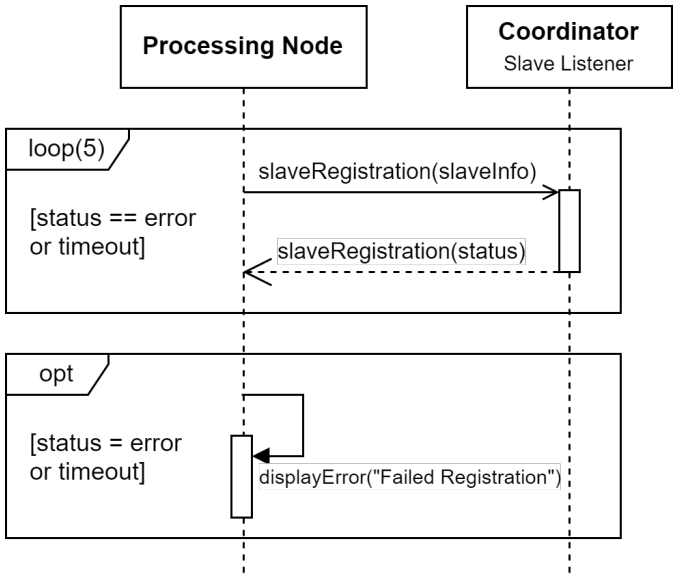


Figure 5.11: Slave Registration Sequence Diagram

The processing node is the process initiator, which initially prepares a datagram packet containing a serialized object that represents its information, such as identification and address details, to be further contacted. This packet is then sent to the coordinator, and a confirmation reply is expected within a limited amount of time. If this reply is negative, in the case of any error or a timeout being reached, the processing node repeats this process four more times, separated by a defined time interval to prevent overloading the coordinator. After five registration attempts, the process is considered unsuccessful and must be aborted. There can be two different causes for this event. If the coordinator responded with an error message, it means that it is running and can receive messages, but for some reason, it couldn't finish the registration process. On the other hand, if the initial request timed out and no response was received, it could indicate that the coordinator is either in a crashed or inactive state or that its address information is incorrect. In all scenarios, the user must be informed of the respective registration status.

The coordinator implements a *Slave Listener* concurrent service that continuously listens on the registration port for incoming datagram packets from processing nodes. These packets are then deserialized, and their information is used to create a remote interface for the processing node to be used on *Phase 2*. Problems in the message transmission process can lead to the receipt of corrupted data by the coordinator and create recurring problems in *Phase 2*. This event should be carefully detected if the information contained in the datagram is critical and must be verified. Finally, the processing node is added to the processing network, starting *Phase 2* (Section 5.4.2) and the *Heartbeat System* (Section 5.4.3), according to the communication protocol diagram (Fig. 5.10).

5.4.2 Phase 2 - Processing Node Remote Invocation

After a successful registration process, each processing node can now be remotely invoked by the coordinator through a remote interface created during the previous phase. *Phase 2* starts with the heartbeat system (Section 5.4.3), and it represents the interactions between the coordinator and processing node.

In this phase, there are four primary procedures outlined in the remote interface, each referenced accordingly with methods that can be remotely invoked, similarly

to a local invocation context. Figure 5.12 illustrates the remote interface class diagram.

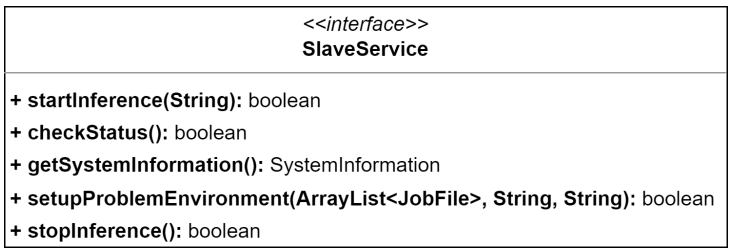


Figure 5.12: Remote Interface Class Diagram

The **Start Inference** procedure (Fig. 5.13) is responsible for executing a distributed evolutionary process in a processing node. This is a two-step process that includes an initial environment setup, implemented by the `setupProblemEnvironment(ArrayList<JobFile>, String, String)` remote method, where a list of the problem's parameter files, problem code, and type of distribution are passed as arguments, and the concrete initialization of the evolutionary process on the processing node, implemented by the `startInference(String)` remote method, where the problem code is passed as argument. In the environment setup process, the problem folder and the respective parameter files passed as arguments are created on the processing node machine, and distribution-specific actions are performed so that the node is ready to process the problem. Finally, the evolutionary engine is initialized and the distributed evolutionary cycle starts.

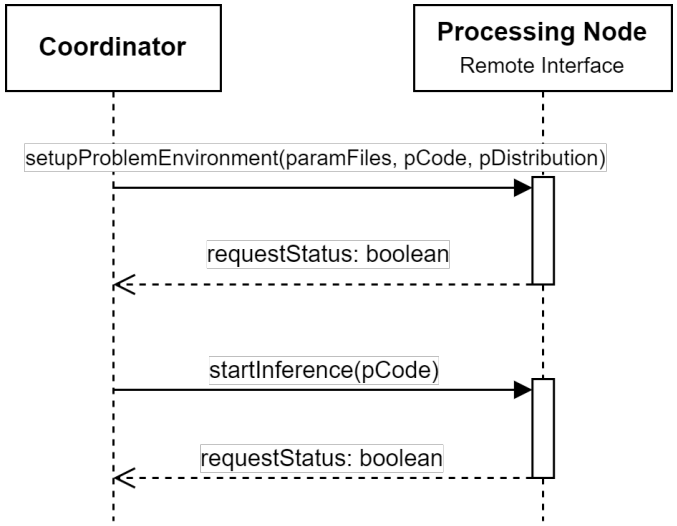


Figure 5.13: Start Inference Sequence Diagram

The **Stop Inference** procedure (Fig. 5.14) safely interrupts any executing evolutionary process on the processing node, making it available for other executions. This procedure can be automatically initiated by the coordinator in the case of a central error that demands all nodes to stop their current executions or be ordered by the user. It is directly implemented by the `stopInference()` remote method.

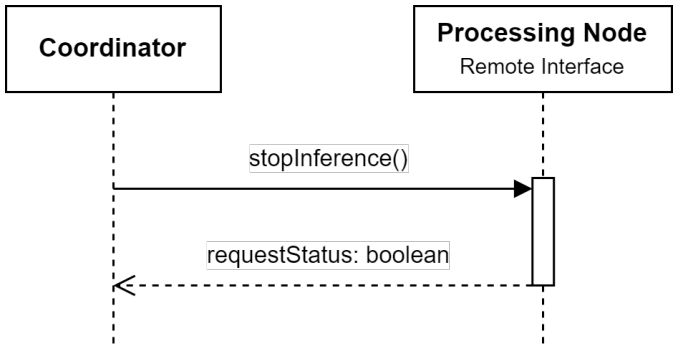
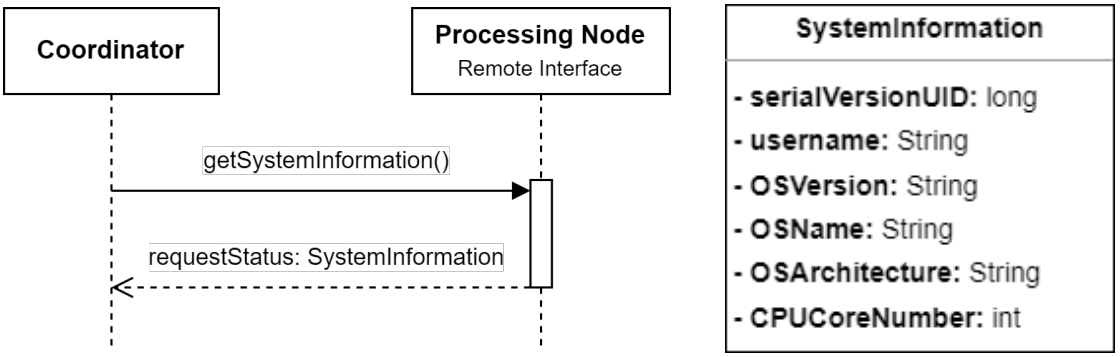


Figure 5.14: Stop Inference Sequence Diagram

The **System Information** procedure (Fig. 5.15a) retrieves machine-specific specifications of the processing node. It is directly implemented by the

`getSystemInformation()` remote method, which returns an object represented by figure 5.15b.



(a) Sequence Diagram (b) Class Diagram

Figure 5.15: System Information Procedure

The **Check Status** procedure (Fig. 5.16) verifies the liveliness of registered nodes. This is directly implemented by the `checkStatus()` remote method and simply returns a boolean representing its status. `True` in a normal execution state and `false` in the case of any error. The *Heartbeat System* (Section 5.4.3) uses this remote method as its verification tool.

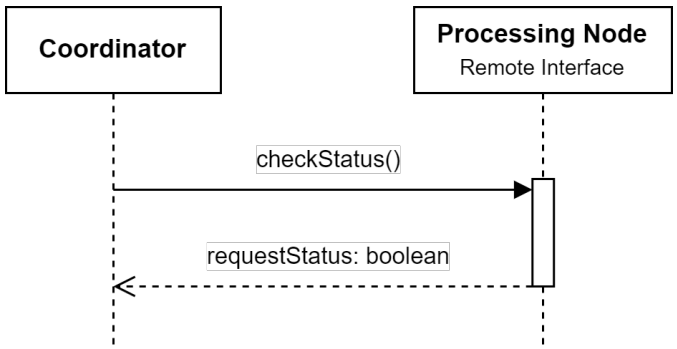


Figure 5.16: Check Status Procedure Sequence Diagram

5.4.3 Heartbeat System

Handling intentional node disconnections or those originated by internal errors can be a challenge in distributed systems. In both scenarios, the node must be excluded from the system, and the administrators notified in the event of an error. If a node intentionally disconnects from the system, it must send a message containing its intention. The scope of this message is defined by the system's structure, being global if all nodes must have a global snapshot of the system or a single message if the system has a coordinator. Some difficulties arise with faulty node detection, since there are no guarantees that the node will notify the system about the error. There is a simple phenomenon that faulty nodes share in common and can be used for error detection systems. When a faulty node is requested to take some action, like a simple message acknowledgment or a task solution, it must respond within a suitable time period for the respective task. If no response arrives to the requester within the given time limit, the node timed out, which indicates an error. This error can have multiple origins, such as network issues not related to a specific system node, so it is important to establish a threshold value for faulty node detection. The time taken to detect a faulty node in a distributed system is directly related to the message frequency within each node. Increasing the message frequency will detect faulty nodes faster but with higher network overhead, while decreasing it will detect faulty nodes slower but with lower network overhead. It is important to analyze the impacts of a faulty node in the global system to find a balance for this factor.

In DECS, after a processing node completes registration phase 1 and is accepted by the coordinator for latter requests, there is a constant need to confirm that the node is still running and ready for request processing. In the *Distributed Evaluation* method, previously explained in Section 3.1.1, faulty nodes create unnecessary processing overhead in the coordinator. This occurs when requests are sent to non-responsive nodes. Importantly, despite this issue, the overall functionality of the system remains unaffected. However, in the *Island Model* distribution (Section 3.1.2), there is a minimum number of connected and functioning processing nodes in order to execute the evolutionary process. Therefore, a faulty node must be immediately detected in order to maintain the system's integrity. As mentioned

before, a high message frequency is necessary to lower the fault detection time, so the **Heartbeat System** was implemented.

This is a simple system implemented in the coordinator that regularly calls the `checkStatus()` remote method on all registered nodes, who must return an acknowledgment in the form of a `true` boolean within a time limit. If a node does not comply with this protocol by not responding, exceeding the time limit, or raising a remote exception, it is considered faulty or disconnected and removed from the registered node list. The frequency of the heartbeat messages can be adjusted depending on the system's needs, but it is important to keep in mind that an unnecessarily high message frequency creates additional network overhead. Figure 5.17 contains a diagram sequence of the Heartbeat system protocol.

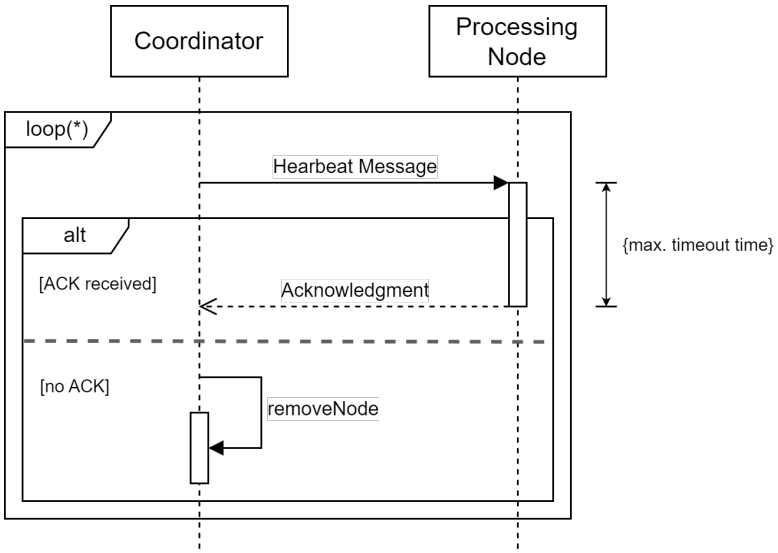


Figure 5.17: Heartbeat System Sequence Diagram

5.5 Deployment

When a software development project reaches a mature and stable phase, it can transition to production stage. This often implies the release and deployment of a stable version to be used by clients. On web applications, deploying to a production server involves compiling and packaging the application to be suitable and optimized for the server. This can be a complex process that involves important decisions regarding the structural requirements for the deployment to be successful. This includes service demand, which tries to understand and estimate the peak user count and their influence on the system workload. These values help to design the optimal hardware specifications, as confirmed by stress tests, which assure that a specific deployment configuration can handle the load estimations.

The deployment of DECS consists on two separate processes. The coordinator, which contains the web application that must be deployed on a *Servlet Container*, *Spring Boot*, or even *Cloud Providers*, and the processing nodes, which can be executed as a Java application on a standard terminal environment.

Initially, a production build of the web application must be compiled. This can be accomplished by executing a simple command (`mvn clean package -Pproduction`), which builds a JAR or WAR file, defined on the Project Object Model (POM) file in the `<packaging>` option, with all of the dependencies and bundled frontend resources, ready to be deployed. Afterwards, the web application can be deployed in a *Servlet Container* using a WAR build file, such as Tomcat, Jetty, or any Java/Jakarta EE server. Deploying Spring Boot applications is a slightly different process from traditional Java Web applications (or Jakarta EE applications) in the way that there is a need to create a JAR file with an embedded server rather than a traditional WAR file that is deployed on a standalone Servlet container or fully featured Java EE server. The decision between deploying the web application on a Servlet Container, using a WAR build, or using a JAR build for Spring Boot applications should take into account the following aspects (Table 5.1).

Table 5.1: Build Type Comparison

Build Type	Server Binary	Package Size
JAR	Not needed	Bigger
WAR	Java Servlet	Smaller

When using a **JAR** build file, there is no need for a server binary install, given that a Java runtime is the only requirement. This comes with the negative aspect of having a larger package size. On the other hand, when using a **WAR** build file, a separate Java Servlet container is needed, with a free choice of servers (Jetty, Tomcat, etc.) but with the advantage of a smaller package size, which can make a difference for extensive and complex applications.

Vaadin applications are typical Java web applications and can therefore be deployed on most cloud services that support Java. Each provider has its own deployment procedure and specifications. While some require a pre-built JAR or WAR file, others allow direct deployments, for example, from GitHub. Cloud providers offer a wide range of advantages and access to robust infrastructures which provide performance transparency to the deployed application and assures an efficient production release. Some of these advantages are high availability, minimizing access time and downtime due to failures, robust security measures with constant maintenance, and scalability possibilities, allowing the application to handle increased traffic and workload seamlessly based on demand, among others.

5.6 Software Test Suite

With the objective of implementing end-to-end tests, further explained in Section 2.7.1, a test suite divided into four test groups was conceived and implemented using the Playwright (Section 2.7.2) framework to simulate typical use cases in the different sections of the web application, as illustrated in Table 5.2. This approach allows for an easy assessment of the system's overall success and the performance of its individual components. Each test group includes specific scenarios for testing

the functionalities and use cases of each section of the web application. A simple set of tests was implemented to verify the system's basic use cases.

Table 5.2: End-to-end Test Groups

Code	Name
JDT	Job Dashboard Tests
NMT	Node Manager Tests
PET	Problem Editor Tests
ST	Stress Tests

Test group **JDT** implements tests for the *Job Dashboard View* (Section 5.2.1). First, all view elements are verified to be present and visible on the application. Afterwards, the job execution use case is simulated and then verified. This is done by first starting a new local job, and verifying if it is included on the job activity list with the correct name and finished status. Finally, the elements on the solutions tab are verified, and the test concludes with the respective result. Test group **NMT** implements tests for the *Node Manager View* (Section 5.2.3), which includes the visibility verification of all elements on the view. This test group can be further expanded to include the execution of DECS-Slave processes and verify their registration on the node list grid. The **PET** test group implements tests for the *Problem Editor View* (Section 5.2.2). Identical to the previous test groups, the visibility of all elements on the view is verified, followed by the problem creation use case testing. On this test, the 11-Bit Multiplexer problem is selected with a local execution, all parameters from all tabs are changed to a specific value, and the problem is saved. Finally, the created problem is loaded, each defined parameter's value in the problem creation process is verified, and the use case success is determined. The **ST** test group implements stress tests [8, p.223] in order to understand how the system behaves in demanding situations and assure that a consistent and robust state is maintained for all clients. One of the implemented tests simulates the simultaneous access of N clients, where N is preferably a large number. The test's success is defined by a set of verifications of the system's state for each client. The browser must be connected, and all web application sections must be accessible on each client. Since the result of this test can be

influenced by the machine's hardware, it can be performed on the deployed system to accurately define access limits for that specific setup. Another implemented test simulates a scenario where N clients access the web application and one of them initiates a job execution. According to the described behavior in Section 5.3 regarding this specific scenario, all other clients apart from the initiator must be blocked from initiating another job. This condition is verified by the state of the start button in the job dashboard view, where it must be disabled during the job execution together with the informative notification and enabled right after the job is finished.

Test Pipeline Each Playwright test must follow a defined set of preparatory and termination procedures, which are essential for a successful test execution. The following Figure 5.18 illustrates the conceived test pipeline diagram.

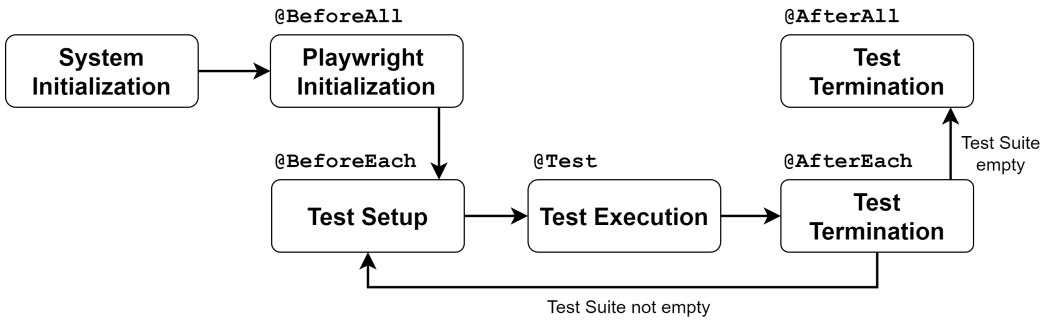


Figure 5.18: Testing Pipeline

In the **System Initialization** phase, DECS is built and started in a new isolated instance, including the web server. Afterwards, the JUnit testing cycle starts with the **Playwright Initialization** phase, marked with the `@BeforeAll` JUnit annotation, which means that the annotated method will be executed only once at the beginning of the cycle. In this phase, the Playwright environment is initialized and configured, similarly to the previous example in the Code Listing 2.4. The `headless` property defines if the testing browser is shown when tests are being performed. This property set to `false` can be useful when developing new tests since there is visual feedback of which actions are being taken in real-time, but it is sometimes unnecessary after the development stage. The **Test Setup** phase starts the test suite cycle. First, a new browser instance is initialized with a

new page accessing the DECS web application. The method that implements this phase is marked with the `@BeforeEach` JUnit annotation, which will be executed before each test in the *Test Suite* (Section 5.6). Since DECS requires an initial authentication process before accessing the dashboard, the login process must be executed before each test and included in this phase. A special class was created to handle this procedure by filling out the authentication form and submitting it. Then, a concrete test is performed in the **Test Execution** phase, marked with the `@Test` JUnit annotation. When the test concludes its execution, the **Test Termination** phase starts, where the browser is terminated, including all opened pages used in the test. The method that implements this phase is marked with the `@AfterEach` JUnit annotation, which is executed right after each test. With the conclusion of this phase, the test suite cycle restarts from the *Test Setup* phase, if there are more unexecuted tests on the test suite or the **Playwright Termination** phase starts, if all tests in the test suite were already performed. In the latter case, the method implementing the termination phase is marked with the `@AfterAll` JUnit annotation, which means it will be executed only once, at the end of the testing process. This phase is used for terminating the Playwright environment, including the browser, representing the conclusion of the testing procedure.

6 Technical Study

The purpose of DECS is to achieve high processing performance and optimal results when computing complex evolutionary problems by applying distribution models in the evolution process. Besides the functional testing phase, explained in Section 5.6, a technical study was conceived to analyze each distribution model and the different use cases they can be applied to, in order to evaluate the system and the fulfillment of its core objectives. This chapter describes the mentioned study by first characterizing the *Experimental Environment* (Section 6.1), followed by the collected *Metrics* for each test group in Section 6.2, which are presented in detail. The evolutionary problems used in this study are defined in Section 6.3, and each test group is further described in Sections 6.4, 6.5, and 6.6, including the respective results. All collected and processed data related to this study can be accessed in the supplementary file A.3 of Appendix A.

6.1 Experimental Environment

Experimental studies in general are scientific procedures that have some strict requirements in order to maintain their scientific validity and credibility. Imprecise experimental methodologies can invalidate the results and respective conclusions, which is why a robust design phase must precede the actual execution. In order to assure consistency among all results, the experiments were performed with the same exact hardware, using identical computers and network conditions for all executions, with the same structure and physical topology. These measures try to prevent external factors from influencing the results, which can hinder their comparison and possibly lead to incorrect conclusions. The concrete specifications of all computers used in the technical study are described in Table 6.1.

Table 6.1: Machine Hardware Specifications

Specification	Value	Additional Information
Brand	Dell	OptiPlex 7000
OS	Ubuntu	x86_64
CPU	Intel i5 12th Gen	12 threads (6 cores), 3Gz
RAM	16 GB	

Computers are very complex machines that perform multiple tasks at the same time. This can negatively impact the precision of some collected metrics, such as memory consumption and elapsed time, which can be easily influenced by other processes running concurrently. Therefore, it is crucial to maximize the efforts to isolate these measurements within the evolution-related computations. Taking into account that such perfect isolation is not possible and each evolution has a randomness factor that can influence multiple factors, each test counts with five executions in the same conditions. This makes it possible to create average results for each metric in one specific test, minimizing the impact of outliers and increasing the reliability of the results.

The following experimental procedure was followed for all tests, ensuring consistency among results.

1. Machine Cleaning
2. Software Execution
3. Results Extraction

In the **Machine Cleaning** procedure, all the non-critical unnecessary processes are terminated in order to free hardware resources for the testing phase and avoid external influences. The **Software Execution** procedure involves the system setup across all involved machines and the concrete test execution. Finally, in the **Results Extraction** procedure, the system is terminated and the execution results are extracted, followed by a cooldown waiting time where the operating system of each machine gradually stops all system components and restores its initial state.

6.2 Metrics

To evaluate the effectiveness and impact of the DECS system, several metrics were collected on the coordinator machine in the experimental procedure, defined in Section 6.1. On each test execution, a profiler was used to extract and record each machine's behavior for further post-processing. This technique adds additional processing overhead on top of the system execution, but it provides a highly detailed and granular description of all events and low-level actions performed on the machine, which opens the possibility to re-analyze and enhance this study in the future with different metrics and conclusions. In order to ensure data consistency, all tests were profiled using the same software tool with the same configurations across all involved machines, with the objective of ensuring a constant overhead value throughout the test set.

Table 6.2 contains the metrics collected for *Distributed Evaluation Test Group* and *Island Distribution Test Group*, while Tables 6.2 and 6.2 contain each group-specific metrics. The distinction in some collected metrics in both test groups is due to their divergent evaluation objectives and evolutionary nature. While the *Distributed Evaluation Test Group* metrics are focused on processing performance, the *Island Distribution Test Group* metrics are focused on evolutionary efficiency, as further explained in Section 3.1.

Metric	Unit
Wall-Clock Time	ms
CPU Time	ms
Network Data	KiB
Socket Read	KiB
Socket Write	KiB
Used Memory	MiB
Heap Memory AVG	MiB
Non-Heap Memory AVG	MiB

Metric	Unit
Fitness	

Metric	Unit
Island #	
Generation	

Table 6.2 is divided into three groups of variables, focused on processing efficiency.

The first group measures the elapsed time of the evolutionary process in two different ways. On one hand, the **Wall-Clock Time** reflects the real-world time elapsed from the start of the evolution until the moment it is terminated in milliseconds. On the other hand, the **CPU Time** reflects the time actually spent by the CPU executing instructions of the evolution process in milliseconds. While wall-clock time influences the user experience, representing the time a user needs to wait for a problem to be processed, CPU time represents the low-level efficiency of the evolutionary process.

The second group measures the amount of data that is exchanged in the processing network, expressed in Kibibytes. It is directly represented by the **Network Data** metric, which is the sum of both **Socket Read** and **Socket Write** metrics. These two metrics measure the amount of data that is read and written by the coordinator's machine sockets, respectively. In the context of computer networks, a socket is a software component used to establish network communications within a network. The network data metric has distinct meanings for each test group. While in the distributed evaluation test group, it represents the exchanged data in the processing network, since processing nodes do not inter-communicate and all exchanged data must pass through the coordinator sockets, in the island distribution test group, this metric only measures the exchanged data per island.

Finally, the third group measures the amount of Random-access memory (RAM) used in the evolution process, expressed in Mebibytes. It is directly represented by the **Used Memory** metric, which is the sum of both **Heap Memory AVG** and **Non-Heap Memory AVG** metrics. These two metrics represent distinct parts of the RAM, and their readings must be isolated, especially in Java programs. *Heap memory* is a portion of memory allocated to a program for the dynamic allocation of objects, variables, and even arrays. It allows objects to be created and destroyed dynamically during the program's execution due to the garbage collector, which reclaims memory occupied by objects that are no longer in use, preventing memory leaks. *Non-heap memory* includes all other memory structures besides the heap, primarily the method area and the memory allocated for the internal processes of the Java Virtual Machine, in the context of Java. It is used to store class metadata,

Just-In-Time compiled code (JIT), and static variables, among others. The non-heap memory is also subject to a garbage collector, but it is far less active than the heap memory. The differences between the garbage collectors in both memory sections make the non-heap memory more stable across the overall test execution and a more reliable metric to compare within different test executions.

Table 6.2 delineates the specific metric related to the distributed evaluation test group, which solely pertains to the **Fitness** value. This metric helps to understand how close an evolutionary cycle is to reaching a solution.

Finally, table 6.2 describes the *Island Distribution* specific metrics. All three metrics characterize the first solution for the respective problem, including the number of the island who found it with the **Island #** metric, followed by the generation when it was found with the **Generation** metric, and the number of evaluations performed with the **Evaluations** metric.

6.3 Problem Definition

In order to test both distribution techniques supported by DECS, two problems were carefully chosen with different properties suitable for each test group.

For the *Distributed Evaluation Test Group* (Section 6.5), a *Meta Evolutionary Algorithm* optimizing the *Rastrigin* function was conceived. This problem is highly suitable for this distribution method due to its resource-intensive fitness evaluation, further explained in Section 2.1.4, which can be distributed across several machines, hopefully increasing the evolution performance. On the other hand, for the *Island Distribution Test Group* (Section 6.6), a *11-Bit Multiplexer* problem was conceived. This is a genetic programming problem further explained in Section 2.1.3.

To implement specific problems in DECS and consecutively in ECJ, it is necessary to create *Parameter Files* (Section 5.1.1). A problem can be defined using one condensed and often big parameter file, containing parameters for all components and layers, which can affect readability and isolation, or several parameter files that logically separate parameters from different contexts, improving organization and parameter isolation. These files often inherit from each other, using

the parameter `parent.N = ...` in a contextual hierarchy defined by the creator, possibly overriding previously defined parameters. The Meta-EA and 11-Bit Multiplexer problems conceived for both test groups are implemented using the following parameter file structures (Figure 6.1, 6.2).

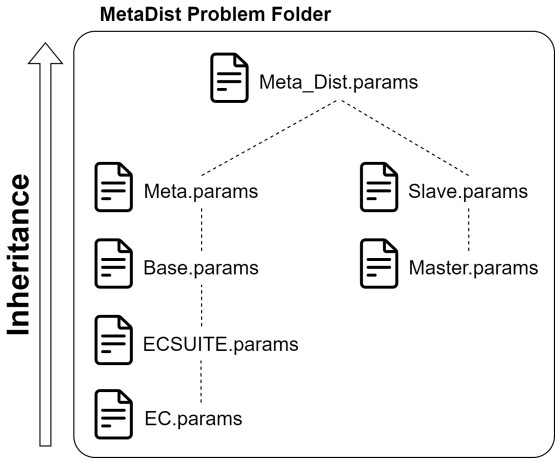


Figure 6.1: Meta-EA Parameters Structure

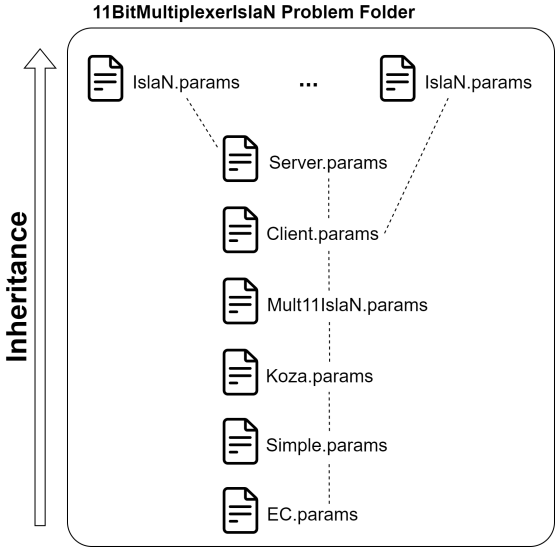


Figure 6.2: 11-Bit Multiplexer Parameters Structure

Figure 6.1 represents the **MetaDist** problem folder, containing the required parameter files, which are consecutively inherited from each other following a hierar-

chy from bottom to top. This means that `Meta.params` inherits the parameters from `Base.params` which inherits the parameters from `ECSUITE.params` and so on. The main problem parameter file `Meta_Dist.params` inherits parameters from both tree branches. The branch on the left represents the evolutionary configuration. On its base, general parameters are defined in the `EC.params` file, which are common to all algorithms. Then, the optimized problem is configured, in this case, the Rastrigin function, with the `ECSUITE.params` and `Base.params` files. On top of the branch, the Meta-EA algorithm is configured in the `Meta.params` file. The branch on the right represents the distribution configuration, in this case distributed evaluation, with the respective master and slave parameters in files `Master.params` and `Slave.params`, respectively.

On the other hand, Figure 6.2 illustrates a generic representation of the **11Bit-MultiplexerIslaN** problem folder, following the previously explained inheritance logic. On the left branch of the tree, the evolutionary configuration is defined for the specific 11-Bit Multiplexer problem. On its base, generic parameters are defined in the `EC.params` file. General *Genetic Programming* (Section 2.1.3) parameters are defined in `Simple.params` and `Koza.params` files, followed by problem-specific parameters defined in the `Mult11IslaN.params` file. Then, the distribution parameters are defined for the coordinator island and regular islands in `Server.params` and `Client.params`, respectively. Coordinator islands are regular islands with the additional task of coordinating the migrations, which are defined in the `Server.params` file. On DECS, this special island is always executed on the coordinator machine. Finally, each island inherits parameters from the respective file according to its role, having only one coordinator island and several regular islands. Since each island has to be configured separately, with an individual parameter file, the same problem with different numbers of islands must also be separately defined.

6.4 Local Test Group

Every experimental study should have a well-defined control group in order to establish concrete conclusions through data comparison.

A substantial part of this technical study revolves around exploring various distribution models, settings, and their potential impacts on the overall evolutionary process. Therefore, the primary focus lies in comparing the local and distributed evolution processes, which serves as a fundamental aspect of the analysis. This test suite addresses specifically the execution of local evolution processes on the same computer, without any cross-computer distribution models or algorithms, except for the use of multithreading. The latter technique does not compromise the validity of the collected data, as multithreading is employed by each computer within the processing network for both distribution models outlined in this study. Since these are tested using distinct evolutionary problems (Meta 2.1.4 and 11-bit Multiplexer 2.1.3), they are executed individually, allowing a valid comparison.

6.4.1 Experimental Questions

EQ1 How does the fitness evaluation distribution affects performance?

EQ2 How does the island distribution model affects the evolutionary efficiency?

6.4.2 Test Suite

The local test group counts with two main tests, **L1** and **L2**, in order to answer both experimental questions, respectively. Table 6.2 represents the Local test suite.

Table 6.2: Local Test Suite

Code	Problem	Generations	Evaluations
L1	Meta-EA	100	5,000
L2	11 Bit Multiplexer	500	512,000

Test **L1** represents the local execution of the Meta Problem (Section 2.1.4), optimizing the Rastrigin Function (Section 2.2.3), which aims to answer experimental question EQ1, when compared with the Distributed Evaluation study (Section 6.5) results. For each execution, 100 generations representing 5,000 evaluations were performed, also applied in the distributed evaluation test group.

Test **L2** represents the local execution of the 11-Bit Multiplexer problem (Section 2.1.3), which aims to answer question EQ2, when compared with the Island Model study (Section 6.6) results. Each execution was limited to 500 generations, representing 512,000 evaluations, also applied in the island distribution test group. If a solution was not found within this limit, the execution was considered unsuccessful.

6.4.3 Results

This section presents and describes the results of the *Local Test Group* for each test in the test suite, aiming to answer the respective experimental questions.

Experimental Question EQ1 establishes a relation between local executions, without using DECS-Slave processes or any inter-computer distribution technique, and distributed executions, specifically using the distributed evaluation method. In order to establish this relation and compare both results from the local execution test (**L1**) and the distributed evaluation test with 50 Slave processes equally distributed by five computers (**DE4.1**), they were collected, normalized, and put together in a bar plot illustrated in Figure 6.3. The distributed evaluation test was selected considering the best overall test from the *Distributed Evaluation* test suite. Since the metric most affected by this distribution technique is the *Wall-Clock Time*, it was given a bigger weight, and the selected test presents the lowest value from the test suite. Table 6.3 contains the real values from both tests.

Table 6.3: Local L1 vs. Distributed Evaluation Table

Test / Metric	Local (L1)	Distributed Eval (DE4.1*)	Difference
Fitness	-1.39	-1.39	**0%**
Wall-Clock Time (ms)	48803	18720	↓ **61.64%**
CPU Time (ms)	152	288	↑ **89.47%**
Network Data (KiB)	36.34	1020	↑ **2706.82%**
Socket Read (KiB)	6.72	104	↑ **1447.62%**
Socket Write (KiB)	29.61	916	↑ **2993.55%**
Used Memory (MiB)	413.44	263	↓ **36.38%**
Heap AVG (MiB)	233.49	79	↓ **66.16%**
Non-Heap AVG (MiB)	179.95	184	↑ **2.25%**

* 50 Slave / five computers test.

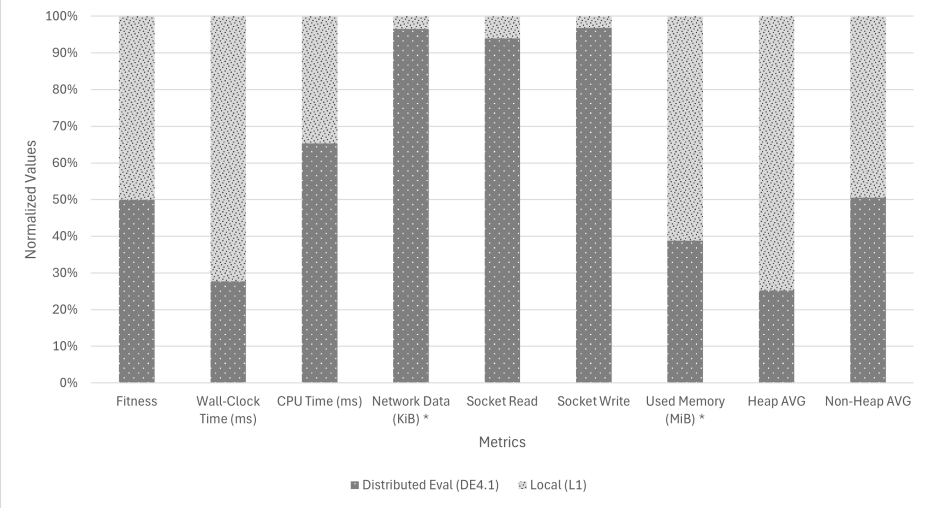


Figure 6.3: Local vs. Distributed Evaluation Graph

Since each collected metric has its own range of values with different magnitudes, it becomes impractical and challenging to compare them without any normalization procedure. In this case, for each metric, the respective value from the local and distributed tests was divided by the biggest value of both. This normalization makes the comparison between series easier to read in one single graphic, illustrated in Figure 6.3. When a series has a value inferior to 50%, it means that its real

value for the specific metric is smaller than the other series, which also verifies in the opposite scenario. This graphic shows an overall distinction between both series, having smaller values on the majority of the metrics in the distributed execution, with the exception of the CPU Time and Non-Heap Memory average metrics, which present a bigger value than the local execution.

With the analysis of Table 6.3, it is possible to further conclude that the fitness values of both tests are approximately the same. The difference values for each metric in the table represent the percentage and were calculated following the equation $Difference = \frac{|x_2 - x_1|}{x_1} * 100$, where x_1 and x_2 are the local and distributed values, respectively. A reduction of approximately 61.64% is seen in the Wall-Clock Time metric in favor of the distributed evaluation test. However, there is an increase of approximately 89.47% regarding the CPU Time metric in favor of the local execution test. In the context of Network Data, there was an overall significant increase between the local and distributed executions of approximately 2706.82% for that metric, in favor of the local execution, 1447.62% and 2993.55% for the Socket Read and Write metrics, respectively. On the other hand, in the context of Used Memory, there was a decrease of approximately 36.38% on that metric and 66.16% on the average use of Heap Memory. However, a slight increase of approximately 2.25% is seen in the use of Non-Heap Memory.

Experimental Question EQ2 establishes a similar relation to the previously described, but specifically using the island distribution model. In order to establish this relation and compare both results from the local execution test (**L2**) and the island model test with 5 islands and high frequency low populated migrations (**IFS2.3**) they were collected, normalized using the same procedure as L1, and put together in a bar plot illustrated in Figure 6.4. The island model test was selected considering the best overall results from the *Island Model* test suite. Since the metric most affected by this distribution model is the number of *Generations* required to reach a solution, it was given a bigger weight, and the selected test presents the lowest value from the test suite. Table 6.4 contains the real values from both tests.

Table 6.4: Local L2 vs. Island Model Table

Test / Test	Local (L2)	Island Model (IFS2.3)	Difference
Generation	307	29	↓ 90.55%
Wall-Clock Time (ms)	14660	10820	↓ 26.19%
CPU Time (ms)	13870	795	↓ 94.26%
Network Data (KiB)	24.53	138.92	↑ 466.32%
Socket Read (KiB)	5.61	31.97	↑ 469.87%
Socket Write (KiB)	18.92	106.95	↑ 465.27%
Used Memory (MiB)	365.58	262.10	↓ 28.30%
Heap AVG (MiB)	191.66	81.38	↓ 57.53%
Non-Heap AVG (MiB)	173.92	180.72	↑ 3.90%

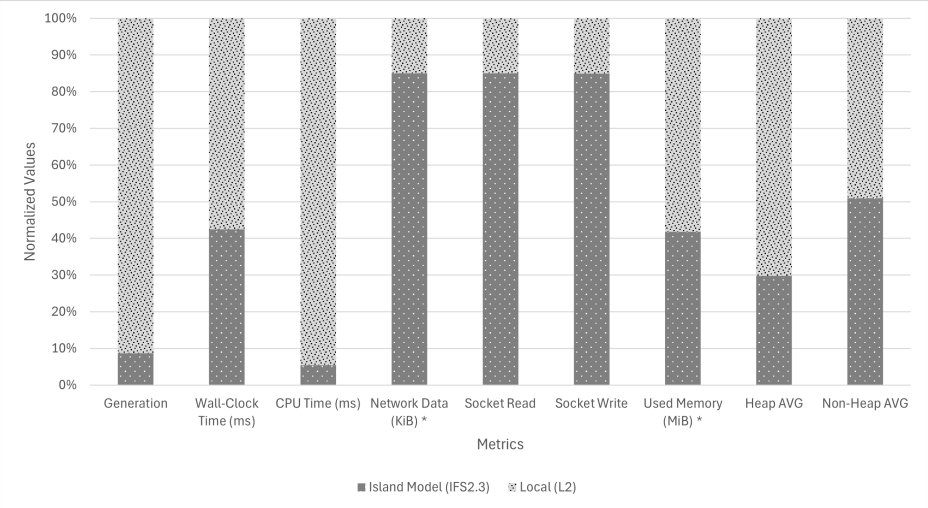


Figure 6.4: Local vs. Island Model Graph

The analysis of Figure 6.4 shows a clear reduction of approximately 91% from 307 to just 29 generations required for the executions using the island model to achieve a solution compared with the local ones. Regarding the wall-clock time, there was a slight decrease of approximately 26% in the island model test from 14660 to 10820 milliseconds and a drastic decrease in the CPU time of approximately 94% from 13870 to 795 milliseconds. As previously seen in test L1, the network data metric, including both of its sub-metrics, socket read and socket write, suffers

a considerable increase of approximately 466% from 24.53 to 138.92 KiB in the island model test when compared to the local one. Finally, when analyzing the used memory metric, it is possible to visualize a slight decrease of approximately 28% from 365.58 to 262.1 MiB in the island model and local tests, respectively. The memory analysis can be further extended to its sub-metrics and conclude that there was a decrease of approximately 58% from 191.66 to 81.38 MiB in the heap memory usage and a slight increase in the non-heap memory usage of 4% from 173.92 to 180.72 when comparing the local and island distribution tests.

6.5 Distributed Evaluation Test Group

In order to evaluate the *Distributed Evaluation* method (Section 3.1.1) on DECS and further understand its behavior in different scenarios, a test group was conceived with a robust set of experimental environments that aim to answer the experimental questions.

6.5.1 Experimental Questions

EQ3 How does the number of Slave processes in the processing network affect performance?

EQ4 How does the number of Slave computers in the processing network affect performance?

EQ5 How does the number of Slave processes running on the same computer affect performance?

6.5.2 Test Suite

The distributed evaluation test suite consists of four sets of selected tests that aim to answer the group's experimental questions. The distributed evaluation test suite can be mapped using the following Tables 6.5 and 6.6.

Table 6.5: Distributed Evaluation DE Test Suite

ID	# Slaves Network	S. Machine 1	S. Machine 2	S. Machine 3	S. Machine 4	S. Machine 5
DE1.1.1	1	1				
DE1.1.2	2	1	1			
DE1.1.3	3	1	1	1		
DE1.1.4	4	1	1	1	1	
DE1.1.5	5	1	1	1	1	1
DE1.2	5	5				
DE2.1	10	2	2	2	2	2
DE2.2	10	5	5			
DE2.3	10	10				
DE3.1	25	5	5	5	5	5
DE3.2	25	25				
DE4.1	50	10	10	10	10	10
DE4.2	50	25	25			
DE4.3	50	50				

Table 6.6: Distributed Evaluation Global Mapping

# Slaves	0 S. Machines	1 S. Machines	2 S. Machines	3 S. Machines	4 S. Machines	5 S. Machines	Total
1 Slave	Control	DE1.1.1	DE1.1.2	DE1.1.3	DE1.1.4	DE1.1.5	30
2 Slaves						DE2.1	10
3 Slaves							5
4 Slaves							5
5 Slaves		DE1.2	DE2.2			DE3.1	20
10 Slaves		DE2.3				DE4.1	15
25 Slaves		DE3.2	DE4.2				15
50 Slaves		DE4.3					10
Total	40	25	15	5	5	20	**110**

Table 6.5 represents the selected tests, identified by the respective **ID**, that can reach important conclusions and answer the experimental questions. # **Slaves Network** represents the number of Slave processes in the network for the respective test, and **S. Machine N** identifies the number of Slave processes running on each computer in the processing network, excluding the Coordinator.

Table 6.6 represents a complete view of the test suite mapping, where a relation is made between the number of Slave processes and Slave machines in the processing network. An additional control group is included where both coordinator

and Slave processes are executed on the same computer, represented by the **0 S. Machines**. A total of 110 executions were performed in this test group, since each cell represents five executions.

The **DE1.1** test group aims to collect data in order to answer experimental questions EQ1 and EQ2, by testing the performance of the evolution process while adding computers and Slave processes to the processing network while maintaining the number of running Slaves on each computer to one. However, a theory *TH1* can be formed with the following question. Since the number of Slave processes in the processing network increases by one with each test and each process is run on a different computer, the performance metrics are not directly influenced by multiple Slave processes sharing the same CPU with multi-threading. Therefore, are the performance metrics at DE 1.1.2, 1.1.3, 1.1.4, and 1.1.5 represented by the metrics collected at DE 1.1.1, divided by the number of computers in the processing network? The **DE1.2** test aims to answer questions EQ1 and EQ3 when comparing the results with **DE1.1.1** and question EQ2 when comparing with **DE1.1.5**.

The **DE 2.1**, **3.1**, and **4.1** tests aim to answer question EQ2 when compared with **DE 2.2**, **3.2**, and **4.2**, respectively, changing the number of computers in the processing network but maintaining the number of Slave processes in the network (10, 25, 50). When comparing **DE 1.1.1**, **1.2**, **2.3**, **3.2**, and **4.3**, it is possible to clearly answer question EQ3, with only one computer in the processing network but changing the number of Slave processes on the same computer.

6.5.3 Results

This section presents and describes the results of the *Distributed Evaluation Test Group* for each test in the test suite, aiming to answer the respective experimental questions. Tables 6.7, 6.8, 6.9, 6.10, and 6.11 were built from Table A.1 of Appendix A, which contain the raw data represented in this section graphics.

Table 6.7: Fitness Results

ID	#Slaves	0 Mach.	1 Mach.	2 Mach.	3 Mach.	4 Mach.	5 Mach.
DE1.1.1	1	-1.3866	-1.4019				
DE1.1.2	2	-1.3949		-1.4050			
DE1.1.3	3	-1.3813			-1.3631		
DE1.1.4	4	-1.3851				-1.3830	
DE1.1.5	5	-1.3871	-1.3949				-1.3521
DE1.2	10	-1.3914		-1.3780			-1.3844
DE2.1	20		-1.4005				
DE2.2	25	-1.3988	-1.3862				-1.3907
DE2.3	50		-1.3952	-1.3679			-1.3864

Table 6.8: Wall-Clock Time Results

ID	#Slaves	0 Mach.	1 Mach.	2 Mach.	3 Mach.	4 Mach.	5 Mach.
DE1.1.1	1	324351	307798				
DE1.1.2	2	166895		158173			
DE1.1.3	3	119140			108580		
DE1.1.4	4	97069				84123	
DE1.1.5	5	83506	82409				69364
DE1.2	10	55000		47395			41210
DE2.1	20		53736				
DE2.2	25	56829	51094				23651
DE2.3	50		75353	34372			18719

Table 6.9: CPU Time Results

ID	#Slaves	0 Mach.	1 Mach.	2 Mach.	3 Mach.	4 Mach.	5 Mach.
DE1.1.1	1	166	343				
DE1.1.2	2	159		323			
DE1.1.3	3	157			327		
DE1.1.4	4	159				321	
DE1.1.5	5	162	324				331
DE1.2	10	148		344			311
DE2.1	20		211				
DE2.2	25	177	323				317
DE2.3	50		307	313			288

Table 6.10: Network Data Results

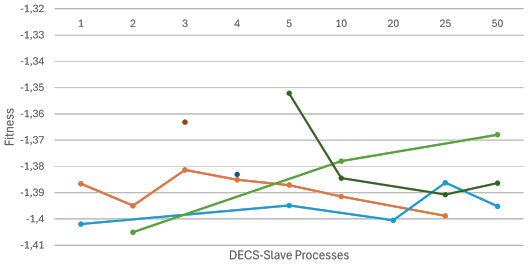
ID	#Slaves	0 Mach.	1 Mach.	2 Mach.	3 Mach.	4 Mach.	5 Mach.
DE1.1.1	1	2170	1473.4				
DE1.1.2	2	410.9		439.9			
DE1.1.3	3	962			428.4		
DE1.1.4	4	1859.5				477	
DE1.1.5	5	480.7	435.7				465.5
DE1.2	10	506.1		538.4			554.2
DE2.1	20		2491				
DE2.2	25	677.2	633.8				636.7
DE2.3	50		1531.7	992.1			1020.2

Table 6.11: Memory Usage Results

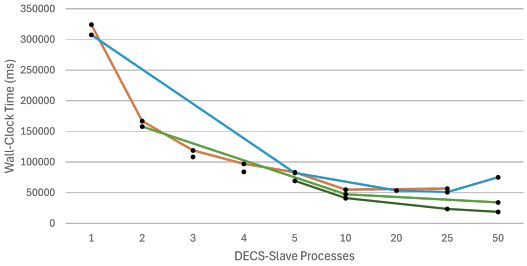
ID	#Slaves	0 Mach.	1 Mach.	2 Mach.	3 Mach.	4 Mach.	5 Mach.
DE1.1.1	1	252.9	258.2				
DE1.1.2	2	259.8		260.1			
DE1.1.3	3	259.1			248.7		
DE1.1.4	4	257.4				261.7	
DE1.1.5	5	267.9	262.7				270.3
DE1.2	10	262.3		255.4			263
DE2.1	20		257.8				
DE2.2	25	207	209.4				262.7
DE2.3	50		265.3	266.3			253.7

Experimental Question EQ3 aims to understand the evolution performance impact of the number of Slave processes in the processing network. In order to answer this question, the collected metrics, which can represent the distribution performance, must be compared while changing the number of DECS-Slave processes in the processing network. This comparison must also consider multiple distribution ratios for each computer in the network, meaning that for a given number of slave processes, these can be hosted on one single computer or distributed by x computers, which can also affect the overall performance. Since each metric has its own range of values, combining them in the same graphic for each distribution setting would create an overcrowded and confusing graphic. However, it is important to consider all collected metrics since they directly or indirectly represent an aspect

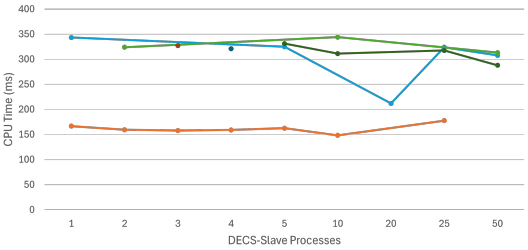
of the evolution's performance. To improve visualization, a line graphic was created for each metric and compiled on Figure 6.5, where it is possible to establish a relation between the number of Slave processes in the processing network and the evolution performance.



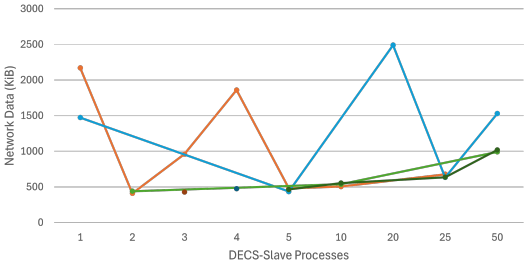
(a) EQ3 Fitness Graphic



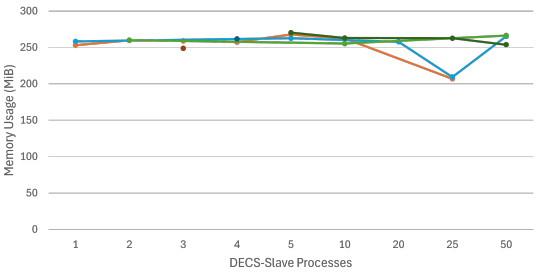
(b) EQ3 Wall-Clock Graphic



(c) EQ3 CPU Time Graphic



(d) EQ3 Network Data Graphic



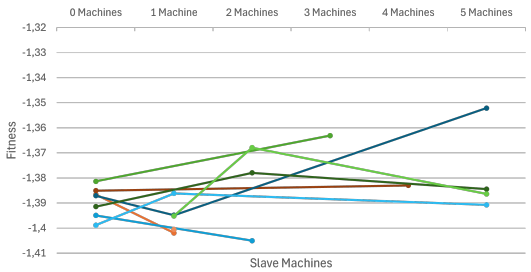
(e) EQ3 Memory Usage Graphic

(f) EQ3 Legend

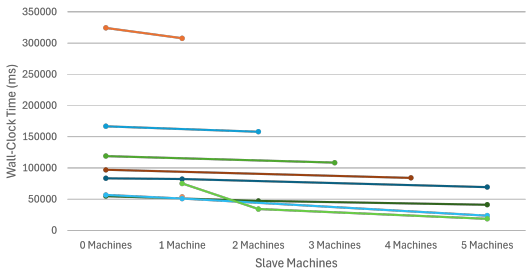
Figure 6.5: EQ3 - Plots Group

With the analysis of **Figure 6.5a**, the *Fitness* values on each execution seem to not follow a common pattern, with the exception of the *2 Machines* series, which accompanies the increase of the Slave processes. **Figure 6.5b** represents the *Wall Clock Time*, a relevant metric, which directly symbolizes the distribution performance. With the analysis of this graphic, a common decrease in its value with the increase of Slave processes is clearly visible across all distribution settings. However, an interesting event is seen at the very end of the *"1 Machine"* series, represented in blue, which increases its value from 25 to 50 slave processes in the processing network as an exception from the rest. **Figure 6.5c**, representing the *CPU Time* shows relatively stable values across all settings with a global tendency to decrease from 25 to 50 slave processes. However, there are two peculiar aspects to this graphic. Firstly, there is a relatively big difference in the CPU Time values between the *"0 Machines"* distribution setting and the rest. Secondly, there was a sudden decrease in the 20 slave processes with *"1 Machine"* distribution setting. Regarding *Network Data*, **Figure 6.5d** illustrates on the one hand, an unstable up-and-down pattern followed by both *0 and 1 Machines* distribution settings and a stable, patterned behavior with a constant increase with the increase of slave processes for the rest. Finally, **Figure 6.5e** represents the *Memory Usage*, where a constant behavior is followed by all distribution settings, with the exception of 25 slave processes, equally distributed across *"0 and 1 Machines"*, in which a one-off decrease is visible.

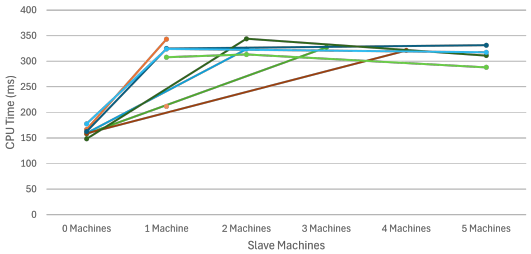
Experimental Question EQ4 establishes a relation between the number of computers hosting DECS-Slave processes and the distribution performance. In order to answer this question, the collected metrics must be compared while changing the number of Slave computers in the processing network. This comparison also takes into consideration the total number of Slave processes participating in the evolution. Similar to the previous experimental question, a line graphic was created for each metric and compiled on Figure 6.6, where it is possible to establish the required relation to answer the experimental question.



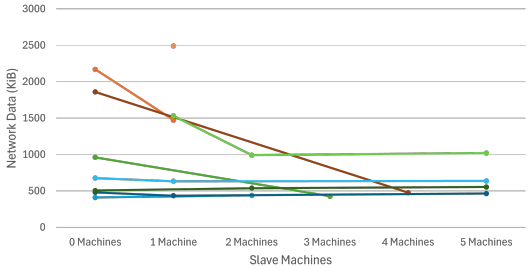
(a) EQ4 Fitness Graphic



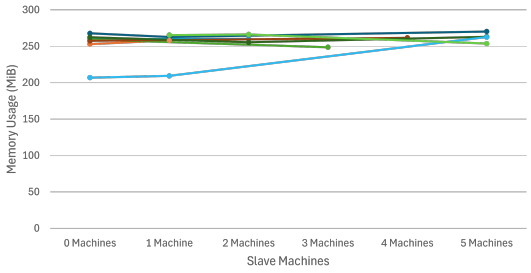
(b) EQ4 Wall-Clock Graphic



(c) EQ4 CPU Time Graphic



(d) EQ4 Network Data Graphic



(e) EQ4 Memory Usage Graphic

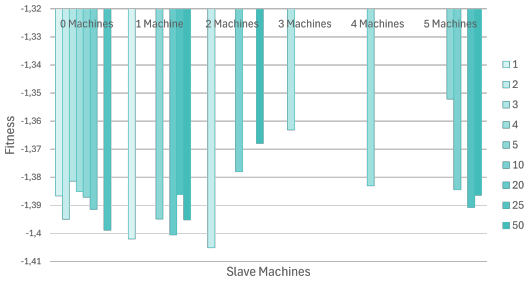
(f) EQ4 Legend

Figure 6.6: EQ4 - Plots Group

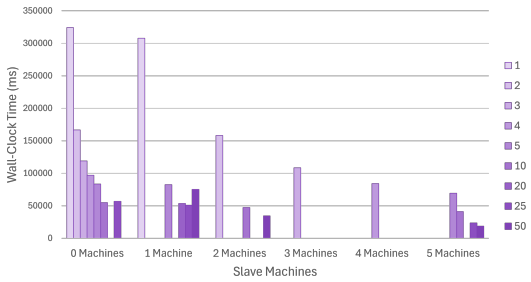
Starting with **Figure 6.6a**, representing the *Fitness* values for each distribution setting, similarly to the previous fitness results, there seems to not exist a common pattern. Regarding *Wall-Clock Time*, illustrated in **Figure 6.6b**, a common behavior is followed by all distribution settings where a general decrease in the wall-clock time values is seen with the increase of the number of Slave machines

in the processing network. In **Figure 6.6c**, representing the *CPU Time*, it is possible to visualize a general increase of approximately double the CPU Time values between executions with only 0 Slave Machines and the rest. A general slight decrease can be seen in the 5 Slave Machines setting compared to the rest. The *Network Data* metric, illustrated on **Figure 6.6d**, shares a split behavior where some series, representing different numbers of Slave processes in the network, follow a stable low increase behavior when increasing the number of Slave computers and other series, which start with a higher network data value, tend to decrease when increasing the number of Slave computers. Finally, **Figure 6.6e** represents the *Memory Usage* metric, where a stable behavior is followed across all distribution settings, both the number of Slave processes and Slave computers in the processing network. A slightly different variation can be seen in the 25 Slave processes series, where it starts with a lower value, compared with the rest and increases with the increase of Slave computers.

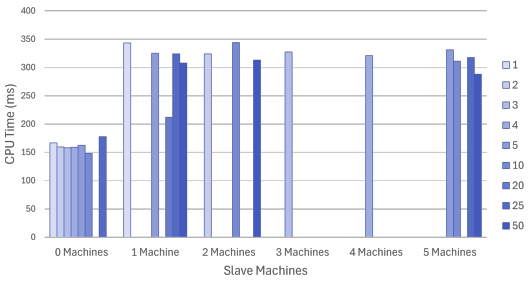
Experimental Question EQ5 is closely related to EQ4 and specifically aims to understand how the distribution performance is influenced by the number of hosted Slave processes on each computer. To establish this relation, the data from Tables 6.7, 6.8, 6.9, 6.10 and 6.11 was arranged in clustered column graphics, which help in the comparison of the values for each number of Slave computers and the respective number of Slave processes in the processing network. Since Slave processes are equally distributed by the number of participating computers on all tests, it is possible to affirm that when increasing the number of Slave processes, it is also being raised equally on all computers, which is the key point on experimental question EQ5. Each collect metric is represented by a different color and its variations, the total number of Slave processes, as displayed in Figure 6.7.



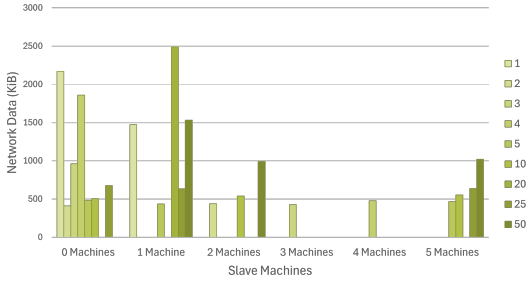
(a) EQ5 Fitness Graphic



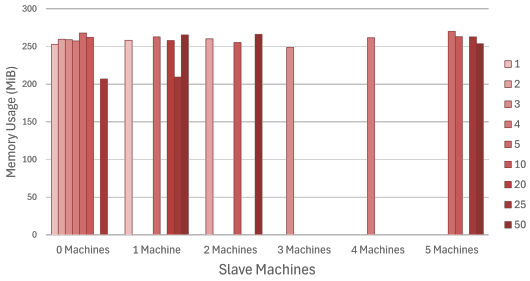
(b) EQ5 Wall-Clock Graphic



(c) EQ5 CPU Time Graphic



(d) EQ5 Network Data Graphic



(e) EQ5 Memory Usage Graphic

Figure 6.7: EQ5 - Plots Group

With the analysis of **Figure 6.7a**, representing the *Fitness* values, it seems that no pattern is followed in the context of question EQ5. In **Figure 6.7b**, which represents the *Wall-Clock Time*, it is possible to establish a down-hill pattern while increasing the number of Slave processes on each computer. This means that the wall-clock time variable is inversely related to the number of Slave processes hosted on each computer. The magnitude of the values generally decreases with the increase in participating Slave computers. Regarding *CPU Time*, illustrated

on **Figure 6.7c**, it is possible to visualize a general stability on all configurations. However, the average value range from the *"0 Machines"* distribution setting is approximately half than the rest. **Figure 6.7d** represents the *Network Data* metric, where it is possible to recognize an up-hill pattern followed by tests with one, two, and five participating Slave computers. This means that the network data variable is directly related to the number of Slave processes hosted on each participating Slave computer. In the *"0 and 1 Machines"* settings, the previously described pattern can be visualized with some exceptional peaks in between. Finally, the *Memory Usage* metric is represented by **Figure 6.7e**, where it is possible to visualize an subtle up-hill pattern on the *"0 Machines"* setting and, in contrast, a subtle down-hill pattern in the *"5 Machines"* setting. A relatively general stability is established for the values in all settings.

Theory **TH1** presented in Section 6.5.2, raises the following question:

TH1: "Are the performance metrics at DE 1.1.2, 1.1.3, 1.1.4, and 1.1.5 represented by the metrics collected at DE 1.1.1, divided by the number of computers in the processing network?"

To answer this question, Table 6.12 was created with the expected values, according to the previously mentioned theory, experimental values from Tables 6.8, 6.9, 6.10, 6.11, and the percent error between them.

Table 6.12: TH1 Result Comparison

	Wall-Clock Time (ms)			**CPU Time (ms)**			**Network Data (KiB)**			**Used Memory (MiB)**		
# Machines	Actual	Expt.	Error	Actual	Expt.	Error	Actual	Expt.	Error	Actual	Expt.	Error
1	307798	307798	0%	343	343	0%	1473	1473	0%	258	258	0%
2	158173	153899	3%	323	172	89%	439.9	737	40%	260.1	129	101%
3	108580	102599	6%	327	114	186%	428.4	491	13%	248.7	86	189%
4	84123	76950	9%	321	86	274%	477	368	30%	261.7	65	305%
5	69364	61560	13%	331	69	383%	465.5	295	58%	270.3	52	423%

Actual : Experimental value
Expt : Expected value according to TH1
Error (%) : Percent error

With the analysis of the percent error column in Table 6.12, it is possible to visualize a general increase in the error values with the increase of computers in the processing network, but only the Wall-Clock time and Network Data metrics stay below the 50% error range, with a slight exception in the five machines execution for the network data metric. The other metrics present percent error values up to 423%.

Further conclusions and interpretations on the results are made in Section 8.

6.6 Island Model Test Group

Proceeding with the evaluation of the *Island Model* method (Section 3.1.2) on DECS, a test group was conceived with different experimental environments that aim to better understand the method's behavior and answer the experimental questions.

6.6.1 Experimental Questions

EQ6 How does the number of islands in the processing network affect the evolution efficiency?

EQ7 How does the frequency of the migrations between islands affect the evolution efficiency?

EQ8 How does the size of the migrations between islands affect the evolution efficiency?

EQ9 How does the topology of the migrations between islands affect the evolution efficiency?

EQ10 How does the migration synchronization between islands affect the evolution efficiency?

6.6.2 Test Suite

The island distribution test suite is composed of two test groups, identified as **IFS** and **IT**, which explore different parameters, aiming to answer the group's experimental questions. The following Table 6.13 describes the **IFS** test suite.

Table 6.13: Frequency & Size IFS Test Suite

Frequency & Size

ID	# Islands	Frequency	Size
IFS1.1	3	Low	Low
IFS1.2		Low	High
IFS1.3		High	Low
IFS1.4		High	High
IFS2.1	5	Low	Low
IFS2.2		Low	High
IFS2.3		High	Low
IFS2.4		High	High

IFS tests aim to answer questions EQ6, EQ7, and EQ8 by modifying the number of islands in the processing network, the migration frequency, and the size of the migrations. Migration frequency and size values were divided into two settings, *Low* which means infrequent and low-density migrations (fewer individuals exchanged between islands), and *High*, which means frequent and highly populated migrations, considering frequency and size settings, respectively.

Table 6.14: Topology IT Test Suite

Topology

ID	# Islands	Full	Ring
IT1.1	3	Sync	
IT1.2		Async	
IT1.3			Sync
IT1.4			Async
IT2.1	5	Sync	
IT2.2		Async	
IT2.3			Sync
IT2.4			Async

The previous Table 6.14 describes the **IT** test suite. **IT** tests aim to answer questions EQ6, EQ9, and EQ10 by exploring two distinct migration topologies, *Full* and *Ring* (Fig. 3.3a, Fig. 3.3b), inspired by conventional network structures. The synchronization of the migrations between all islands is an important aspect to take into consideration. In order to better understand its influence on the overall evolution process, two synchronization settings were defined and tested with each topology. The **Sync** (Synchronized) setting means that all migrations between islands in the processing network happen at the same time, starting in the same generation and having the same interval between them. This setting provides each island with more diversity on each migration, receiving individuals from all other islands at the same time but with fewer frequent migrations, compared with the Async setting. Regarding network load, this setting is not ideal due to its intensive peak usage, overloading the network with multiple and possibly heavy messages sent simultaneously. These messages contain several individuals that, depending on their representation and specificities, can be complex and large, creating an overhead delay that negatively impacts the evolution process performance and scalability of the system. The **Async** (Asynchronous) setting means that all migrations between islands in the processing network happen separately. This can be achieved by defining different migration start values and the same frequency for each island, which provides less diversity on each migration but more frequent migrations while maintaining migration frequency. Regarding network load, this setting is preferable compared to the Sync setting due to its distributed network

usage. Instead of processing multiple migrations at the same time, in this setting, the migrations are more frequent but separate in time, lowering the network communication overhead delay and allowing for better system scalability.

Both topologies, previously explained in Section 3.2, can be further combined with each synchronization setting, which could lead to different outcomes.

When the **Full Topology** is combined with the Sync setting, all islands should exchange migration packets bidirectionally simultaneously. On the other hand, when used together with the Async setting, islands are chosen alternately, always in the same sequence, and send their migration packets to all other islands, one at a time.

In the case of the **Ring Topology**, when combined with the Sync setting, each island should send a migration packet to its direct neighbor and receive one from its previous sibling simultaneously. On the other hand, when used together with the Async setting, islands are chosen alternately, always in the same sequence, following the clockwise direction, to send their migration packet to the next ring neighbor and receive a migration packet from the previous ring neighbor.

6.6.3 Results

In this section, the results of the *Island Model Test Group* are presented and described for each test in the test suite, aiming to answer the respective experimental questions. Tables 6.15 and 6.16 were built from Table A.2 of Appendix A, which contain the data represented in this section graphics.

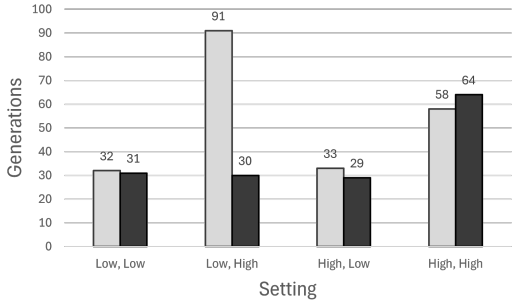
Table 6.15: IFS Results Table

Test ID	# Islands	Generations	Network Data (KiB)	Used Memory (MiB)
1.1	3	32	82,02	267,65
1.2		91	12331,55	276,96
1.3		33	127,64	259,00
1.4		58	8741,71	264,76
2.1	5	31	77,97	259,51
2.2		30	1978,63	259,60
2.3		29	138,92	262,10
2.4		64	10039,68	266,44

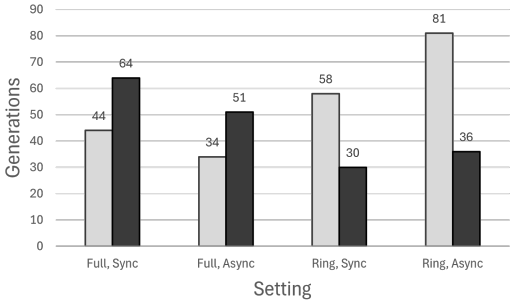
Table 6.16: IT Results Table

Test ID	# Islands	Generations	Network Data (KiB)	Used Memory (MiB)
1.1	3	44	132,28	264,62
1.2		34	100,22	265,79
1.3		58	146,11	263,99
1.4		81	276,86	280,95
2.1	5	64	296,18	263,80
2.2		51	182,73	261,83
2.3		30	169,32	260,25
2.4		36	99,03	263,65

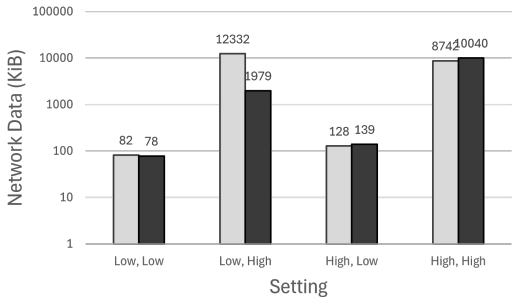
Experimental Question EQ6 aims to understand the impact that the number of islands in the processing network has on the evolution efficiency. To answer this question, a collection of metrics was selected to represent the classification, and the different tested settings on each test suite (IFS and IT) were compared when executed with three and five islands. The metric that best represents evolution efficiency is the number of generations needed to find a solution, which in most cases is directly connected with the elapsed time since, for each generation, it is necessary to process the same number of evaluations with the exception of an early algorithm termination due to the finding of a solution. To better illustrate the results, two bar graphics representing each test suite for each selected metric were compiled in Figure 6.8.



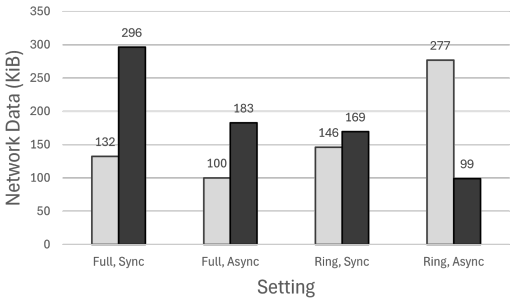
(a) Generations IFS



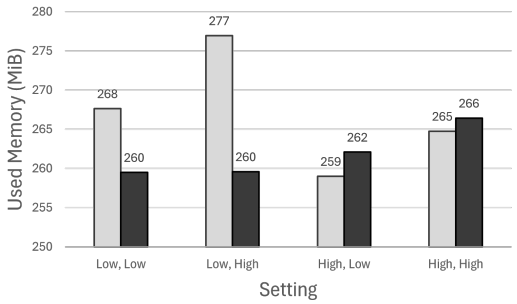
(b) Generations IT



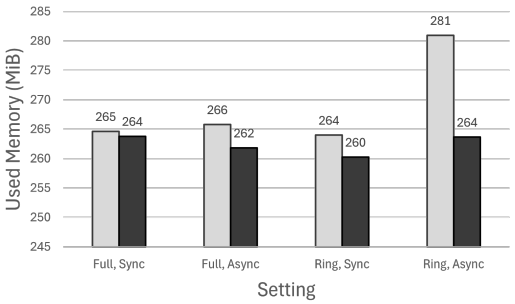
(c) Network Data IFS



(d) Network Data IT



(e) Memory Usage IFS



(f) Memory Usage IT

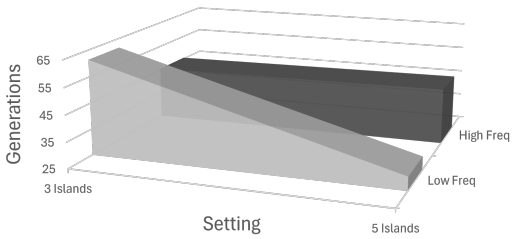
(g) Legend

Figure 6.8: EQ6 - Island Model Results

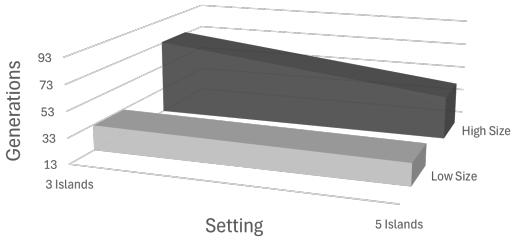
With the analysis of **Figure 6.8a**, which represents the number of *Generations* metric, it is possible to conclude that in the IFS test, which tests the migration

frequency and size, the five island setup is the most favorable, having migration values below the three island setup on all settings, with the exception of the highly frequent and big migrations setting. A low migration number indicates a highly efficient evolution, where a solution was found in an early stage of the evolution. In the IT test (**Figure 6.8b**), which tests the migration topology and synchronization, an interesting pattern can be visualized. When the full topology is used, independently of the synchronization setting, the three islands setup is the most favorable. However, when the ring topology is used, the five islands setup becomes the most efficient with both synchronization settings. Figures **6.8c** and **6.8d**, which represent the *Network Data* exchanged in both reading and writing socket operations, show that it is possible to identify the same pattern in both IFS and IT graphics as the previously analyzed graphics, representing the number of generations, with the exception of IFS and IT high frequency, low size migration, and ring, synchronized settings, respectively, where a slight advantage is given to the three island setup. Lastly, **Figure 6.8e** represents the *Used Memory* metric, where the same pattern as the previous graphics is repeated in the IFS test. However, in the IT test (**Figure 6.8f**) the scenario is different, indicating a clear advantage to the five island setup.

Experimental Questions EQ7 and EQ8 establishes a relation between the frequency and size of the migrations between islands and the evolution efficiency. To answer these questions, the collected metrics must be compared while changing the frequency and size of the migrations in order to understand how these settings influence the evolution efficiency, as performed in the IFS test suite. Figures 6.9, 6.10 and 6.11, compile two graphics, one for each changed setting (migration frequency and size) for each selected metric.

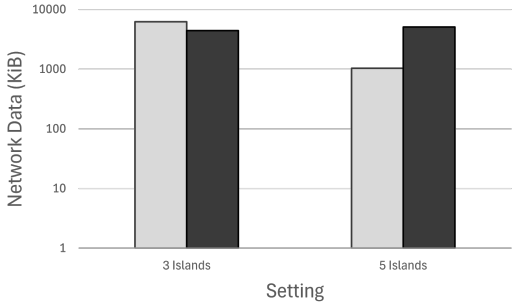


(a) EQ7 - Generations

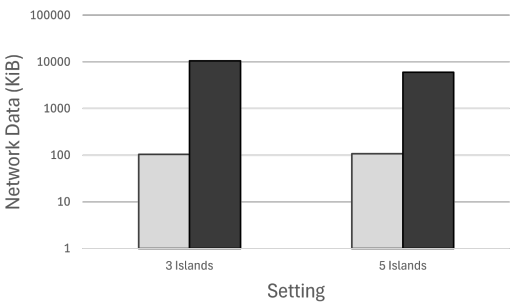


(b) EQ8 - Generations

Figure 6.9: IFS Generations Plots



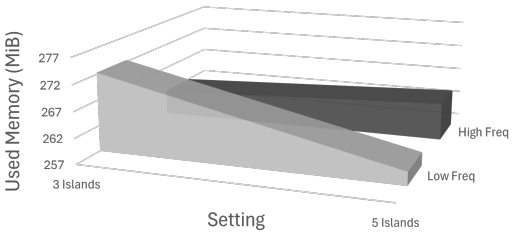
(a) EQ7 - Network



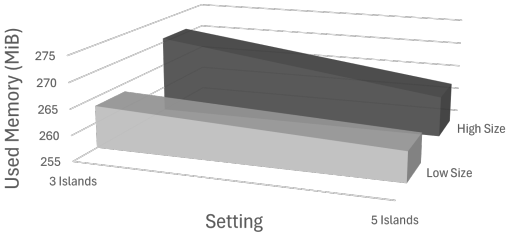
(b) EQ8 - Network

(c) Plot Legend

Figure 6.10: IFS Network Plots



(a) EQ7 - Memory Usage

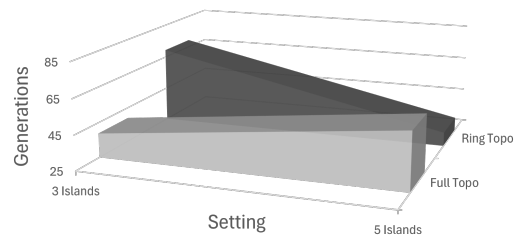


(b) EQ8 - Memory Usage

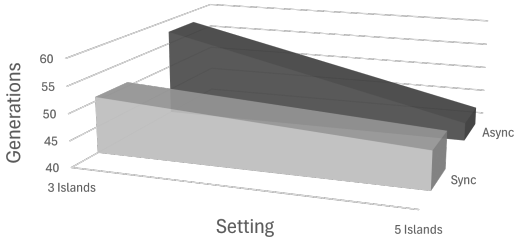
Figure 6.11: IFS Memory Plots

Starting with **Figure 6.9a**, representing the number of generations needed to find a solution, it is possible to conclude that an increase in the number of islands leads to a decrease in the number of generations required to find a solution when using low-frequency migrations. When using high-frequent migrations, a slight increase in the number of generations is seen when increasing the number of islands. Regarding the migration size, illustrated in **Figure 6.9b**, a slight decrease in the number of generations is seen when increasing the number of islands when using low-size migrations, together with a more pronounced decrease when using large-size migrations. When considering the network data exchanged in the evolution process, illustrated in **Figure 6.10a**, a lower value is seen in highly frequent migrations when using a three island setup and the opposite in a five islands setup. Regarding the migration size (**Figure 6.10b**), a clear advantage is seen for low-size migrations both in three and five islands setup. Finally, when considering the used memory metric, illustrated in **Figure 6.11a**, a clear decrease is seen when increasing the number of islands when using highly frequent migrations and the opposite when using less frequent migrations. A decreasing scenario together with an increase in the number of islands is seen when using low-size migrations, which is similar when using large-size migrations, as seen in **Figure 6.11b**.

Experimental Questions EQ9 and EQ10 aim to understand the influence of both migration topology and synchronization on the evolution efficiency. To answer these questions, the selected metrics must be compared while changing the topology and synchronization of the migrations in order to understand how these settings influence the evolution efficiency, as performed in the IT test suite. Figures 6.12, 6.13 and 6.14, compile two graphics, one for each changed setting for each selected metric.

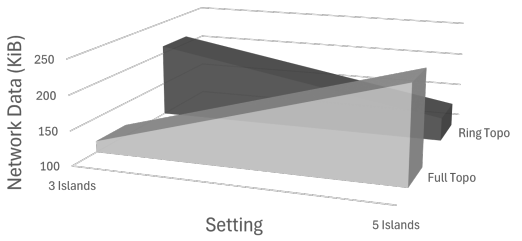


(a) EQ9 - Generations

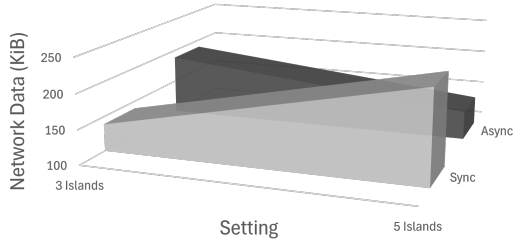


(b) EQ10 - Generations

Figure 6.12: IT Generations Plots

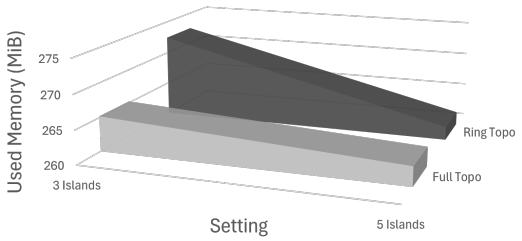


(a) EQ9 - Network

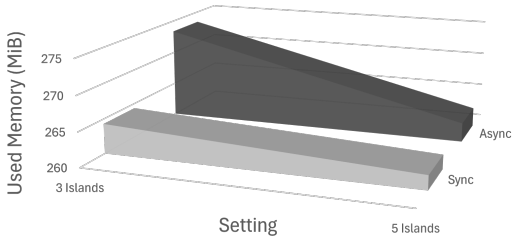


(b) EQ10 - Network

Figure 6.13: IT Network Plots



(a) EQ9 - Memory



(b) EQ10 - Memory

Figure 6.14: IT Memory Plots

Regarding the number of generations, illustrated in **Figure 6.12a** it is possible to see an inverted behavior between the full and ring topologies when changing the number of islands, with the full topology being more favorable with a three island setup and the inverse for a five island setup. **Figure 6.12b** illustrates a common decreasing behavior of both sync and async settings when increasing the number of islands. Considering the network data metric, illustrated in Figures **6.13a** and **6.13b**, a common inverse behavior can be seen in both topology and synchronization graphics, with an increasing curve for the full topology and sync settings and a decreasing curve for the ring topology and async settings when increasing the number of islands. Finally, the used memory is analyzed in Figures **6.14a** and **6.14b**, following a similar decreasing curve on both settings.

7 User Experience Study

The conception of software for human interaction can be a challenging process due to the irregular behavior each user can have. This unpredictability raises a special importance to ensure that the user interface (UI) is robust and correctly responds to all possible user actions. In large, complex UIs, the task of predicting the universe of possible actions a user can take is difficult and sometimes undoable. This can be experienced in input forms, where a user must fill custom and unpredictable values on visual components, which are not always within an acceptable range, and a response must be returned from the UI in the form of error messages. One solution is to group a set of actions and create a general response for that group, which effectively decreases the number of possible responses a user interface must consider. However, the granularity of responses must be balanced between too generalized responses, which are the result of many possible actions combined in a small number of groups, and too specific responses, which are the result of a large number of groups containing a small number of possible actions.

An important aspect that must be evaluated, which combines all aspects of the UI with the system's effectiveness, is the **User Experience (UX)** [8, p.224]. This evaluation can occur in a mature state of the development phase, with the majority of the use cases implemented, but it can also be performed in a production state system that has undergone some UI changes that could affect the user experience. This metric combines the system usability and effectiveness of its use cases, the different UI aspects, including structure, coloring, visual aspect, responsiveness, and overall user satisfaction after using the different UI components. The evaluation participants must be carefully selected to reflect the target user group the system was conceived for. An incorrect participant selection can negatively influence the results, caused by a vast number of factors. One of these causes can be the level of technical knowledge required in order to correctly use the system,

which can generate bad UX results if the participants don't have the minimum technical knowledge.

In the context of DECS, a user experience evaluation was performed on the web application and further described in this chapter.

7.1 Methodology

The user experience evaluation of DECS is composed of two main components. The **Task Sheet**, further analyzed in Section 7.2, which is a document where all tasks are listed and explained for easy understanding by the participant, and the **Survey**, which contains a set of specific questions related to the execution of the tasks and general system usage, described in Section 7.3.

This evaluation was performed in the same technical environment as the *Technical Study* (Section 6), including the same exact hardware components, specifically the computers and network infrastructure. This aspect helps to reduce the hardware influence between the results of this thesis and maintain a comparable set of collected data. The hardware specifications of the computers used in both evaluations can be found in Table 6.1.

An introductory presentation of the system and the contextual background of the tasks was conducted, including the basic sections and functioning of DECS, and the evolutionary problems executed during the activity. No further detailed instructions were given since the objective of the activity is also to evaluate the success rate of achieving a certain state within a system the participants are not familiar with, which determines the system's usability and ease of use.

Evaluation Workflow

In order to achieve isolated responses for each task and section of the task sheet, an intended workflow was conceived. There are two main ways to conduct user experience activities. If the survey does not have any task-specific questions and no isolation between each task is needed, it can be applied at the end, after the participant completes all planned tasks. Otherwise, both components must be

applied concurrently and interspersed by explicitly defining points where the participant must complete a certain task or answer a section of the survey. In DECS, the latter approach is used, as illustrated in Figure 7.1.

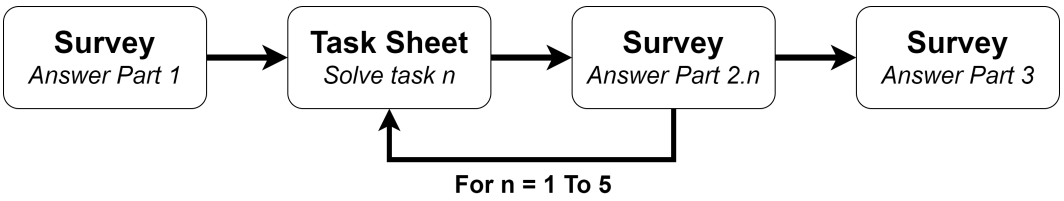


Figure 7.1: Workflow Diagram

First, the participants must answer the introductory questions of the survey (**Survey Part 1**). Afterwards, it is intended to solve task n, followed by the survey's answer to the respective task (**Survey Part 2.**n), and repeat this process for all tasks. Finally, the last part of the survey must be answered (**Survey Part 3**) after completing all tasks.

7.2 Task Sheet

One of the main components of this activity is the **Task Sheet**. This document explains each task that must be carried out by the participants. The level of detail in the instructions must be limited to make the participants think by themselves and figure out how to solve each abstract task and reach the desired state. A highly detailed task description changes the participant's behavior from actively exploring the system resources, enhancing their sense of freedom, to just following specific and atomic tasks without really understanding the scope of the actions or what they are trying to achieve. This phenomenon can give the participant a false sense of achievement due to their straightforward success without ever failing or having to think about how to achieve a certain objective, which will actively influence their experience with the system and create highly favorable invalid results. In conclusion, the task sheet must be well articulated with a previous introductory presentation in order to find a balance between the task's level of detail and ensure

a successful evaluation procedure. Table 7.1 contains the designation of each task in the task sheet.

Table 7.1: Task Designation

ID	Name
T1	Local Problem Execution
T2	Distributed Evaluation
T3	Result Analysis
T4	Know more about your cluster
T5	Island Distribution

DECS **UX Task Sheet**, listed in Appendix B is composed of five tasks, exploring different use cases of the system. After each task, the participant must answer a group of task-specific questions, as explained in Section 7.1. The task sequence was specifically designed to follow an incremental approach at each difficulty level. Thus, as candidates become more comfortable with the system, they become increasingly capable of tackling more challenging tasks. Starting with **Task 1**, which explores a local problem execution without any inter-computer distribution methods, the participant is simply asked to run the pre-configured specified problem. **Task 2** focuses on the *Distributed Evaluation* method and asks the participant to edit a pre-configured problem according to the given specifications. In **Task 3**, the participant must explore and compare the results of the previous tasks and draw conclusions. This task also aims to create a better understanding of what is being done and why distribution methods should be used in the processing of evolutionary algorithms. **Task 4** focuses on the system's exploration, specifically the connected DECS-Slave processes and their characteristics. Finally, the intricate **Task 5**, which requires the participant to create a new problem from scratch, using the *Island Distribution* model. A set of evolutionary parameters is required to be incorporated into the problem, and the island model fully configured according to the given diagram, including island definition and migration mapping.

7.3 Survey

Another crucial component of this evaluation is the data collector, implemented in the form of a digital survey where all participants must express their opinions, evaluations, and suggestions about the system within the scope of the activity. Each question aims to understand a specific section of the participant's experience and ultimately formulate a conclusion on the topic, based on the global activity results. A compiled version of the survey used in this evaluation can be found in Appendix C.

The survey is divided into three sections. The **First Section** contains a set of introductory questions aimed at characterizing the participant's technical background, further described in Section 7.5.1. In the **Second Section**, each task is represented by a sub-section containing similar and task-specific questions, with the aim of obtaining more precise results and understanding the context of each answer. A question related to the overall experience with the system is repeated at the beginning of each sub-section in order to understand the general user experience evolution during task execution. Three rating questions related to task difficulty, individual success, and the system's impact on task execution are also included for each sub-section, followed by a task-specific verification question to confirm that the participant has achieved the desired state upon completing the task. Finally, the **Third Section** includes questions with a global scope, in order to understand the overall user experience after all tasks have been completed. These include the topics of usability, features, interactivity, future improvements, suggestions, and activity evaluation. An in-depth analysis of all questions and results is provided in Section 7.5.2.

7.4 Experimental Observations

In this type of activity, where several participants are involved, it is often important to collect relevant observations or events that occur during its execution. These aspects must be taken into consideration in the result analysis phase, due to external influences caused by these factors. Transparency is also a key factor in all research experiments in order to assure precise and valid results and conclusions,

so every event must be documented, regardless of its influence on the activity, positive or negative.

During the execution of the activity, two adverse events occur, which may influence some specific sections of the results. The first event happened as a consequence of a critical system error at the beginning of the activity, which required a complete system restart. Unfortunately, this procedure was neglected, and neither all intended DECS-Slave instances were started nor all Slave computers were used, which produced a functional, yet highly limited system. As previously analyzed in the *Technical Study* chapter, a low number of DECS-Slave instances combined with a low number of Slave computers can negatively influence the execution time of evolutionary problems using distribution techniques and even make them slower when compared to local executions, due to their processing overhead. This particular event is expected to directly influence Task 3, which aims to compare local and distributed executions, where the intended conclusion would be a faster execution using distribution methods. However, the event may cause the opposite conclusion to be made. During the activity, a small number of students asked about this unexpected phenomenon.

The second event was caused by a missing crucial folder, which stores all user-created problems, and resulted in the system's inability to store any user-created new problems. This event directly affected the last part of *Task 5*, since the participants were not able to store and execute the newly created problem. Fortunately, they were able to carry out most of the problem creation procedure, including the island distribution configurations.

It is important to notice that the occurrence of these events may have falsely changed the overall participant's perspective and experience with the system by creating obstacles and vulnerabilities caused by deployment negligence and ultimately influencing the overall evaluation results, presented in Section 7.5. Further conclusions and interpretations of the results are presented in Section 8.

7.5 Results

This section presents the results of the user experience study, summarizing the survey responses in a set of graphics that enhance data visualization. All collected and processed data from this study are available in Appendix D. Initially, a general characterization of the participants is made in Section 7.5.1, followed by the compilation of the results, in Section 7.5.2. A further discussion on this study is made in Section 8.2.

7.5.1 Participants Characterization

A group of 11 HAW informatics students with distinct technical backgrounds and knowledge bases were selected to participate in this evaluation. The only requirement is to have a practical basic understanding of evolutionary algorithms in order to be able to use the system since it requires some base knowledge on that matter to be operated. One of the core objectives of DECS is to create an abstraction layer between the user and the distributed computing operations running internally, which means that no knowledge of this matter is required to operate the system. Table 7.2 and Figure 7.2 represent the participant's distribution by the different informatics courses of HAW.

Table 7.2: Q1 - Participant Distribution by Course

Course	Responses	Percentage (%)
AI	8	72.7
ITS	2	18.2
ECS	0	0
WI	0	0
Other*	1	9.1

* Exchange student.

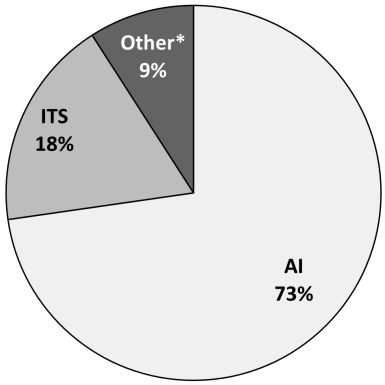


Figure 7.2: Q1 - Course Distribution Representation

It is possible to conclude that the majority of the participants are "Angewandte Informatik" students, followed by the "Informatik Technischer Systeme" course, which translates to English as "Applied Computer Science" and "Computer Science of Technical Systems", respectively. There was also one exchange student participating in the evaluation. It is also important to understand the participant's level of technical experience, which can be superficially represented by the stage of the respective ongoing course. This metric is represented by the semester number the participant is currently enrolled in, as illustrated in Figure 7.3, based on Table 7.3, considering that all informatics bachelor courses in HAW consist of a total of 6 semesters.

Table 7.3: Q2 - Participant Distribution by Semester

Semester	Responses	Percentage (%)
S4	1	9.1
S5	3	27.3
S6	7	63.6

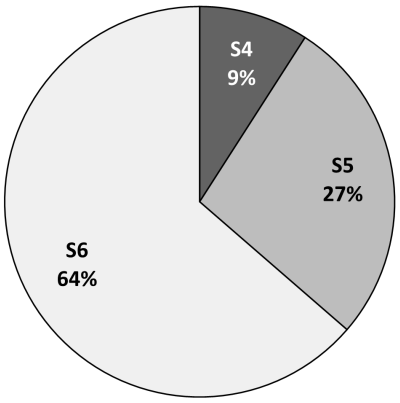
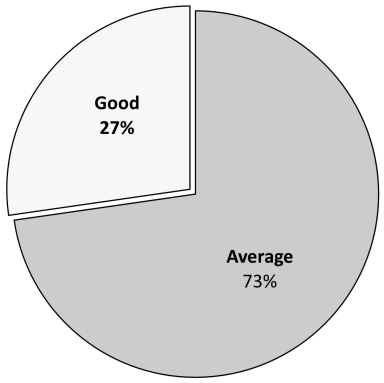


Figure 7.3: Q2 - Semester Distribution Representation

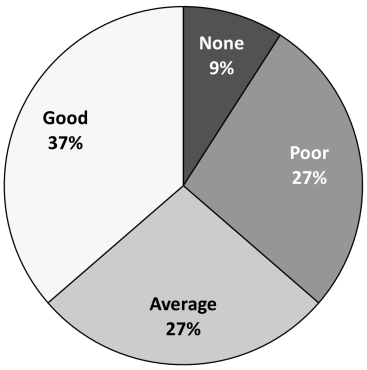
The majority of the participants are in a mature stage of their bachelor studies, largely distributed by the two last semesters of their bachelor's degree (S5 and S6), and only one student from semester four. This indicates that the participants may have a consolidated knowledge base in the context of computer science and its sub-fields. Finally, the participant's concrete technical skills in Evolutionary Algorithms and Distributed Computation must be determined. This was made through a five-level scale, ranging from no skills on the matter, categorized as "None" and expert skills, categorized as "Excellent". Figure 7.4 illustrates the skill level distribution among participants, based on the contents of Table 7.4.

Table 7.4: Q3 - Participant Skill Level

	Evolutionary Algorithms		Distributed Computation	
Descriptor	Resp.	%	Resp.	%
None	0	0	1	9.1
Poor	0	0	3	27.3
Average	8	72.7	3	27.3
Good	3	27.3	4	36.4
Excellent	0	0	0	0



(a) Q3a - Evolutionary Algorithms



(b) Q3b - Distributed Computing

Figure 7.4: Q3 - Participant Distribution by Skill Level

It is possible to conclude that the majority of participants have an intermediate understanding of evolutionary algorithms. This is evidenced by the fact that most participants rated their skills as "Average" with a total of eight responses or "Good" with three responses. There were no ratings for "None", "Poor", or "Excellent". In the context of distributed computation, there is a more diverse set of responses. Participants generally have a moderate skill level, with most feeling confident enough to rate themselves as "Average" or "Good". However, there is also a group of participants who rated their technical skills in the context of distributed computation as "Poor" or "None" which is beneficial for the evaluation, since there should not be any requirements on this matter to operate the system.

7.5.2 Survey Results

Starting with the overall task overview, common questions were asked for each task regarding the overall experience with DECS and its impact on task resolution, difficulty, success, and completion. With the analysis of the verification questions, the number of invalid answers was calculated. Table 7.5 contains the previously described metrics, expressed in percentage, supported by Figures 7.5 and 7.6.

Table 7.5: Overall Task Overview Data

Task	Evaluation	Difficulty	Success	DECS Impact	Completion	Invalid Answers
T1	51%	45%	80%	60%	100%	9%
T2	62%	44%	89%	71%	91%	9%
T3	68%	38%	87%	82%	91%	9%
T4	72%	42%	89%	80%	100%	9%
T5	52%	51%	36%	47%	0%	0%
Final	62%					

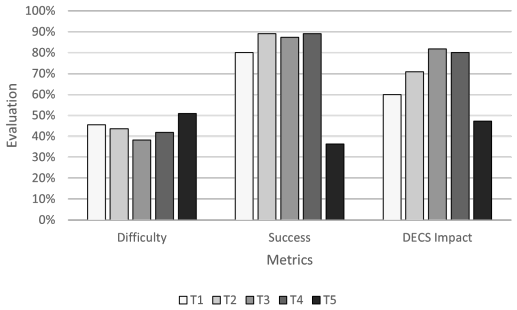


Figure 7.5: Overall Task Overview Plot

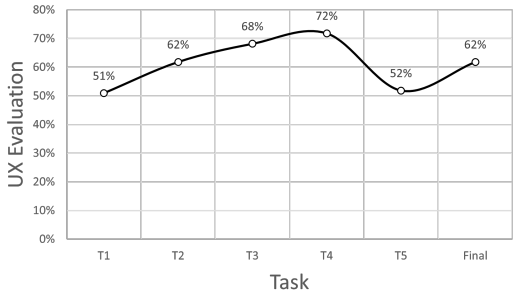


Figure 7.6: Participant Experience Evolution

Figure 7.5 illustrates the three metrics that characterize each task. The *Difficulty* metric expresses how difficult the participants rated each task, where a high percentage means a demanding task and the opposite means a straightforward one. All tasks were classified by the participants, in terms of difficulty, as moderate tasks, with all classifications below 50% with the exception of task 5, which received an average of 51%. The easiest task in this activity, according to the results, is the third, where the results from one and two are compared. Regarding

the *Success* metric, it is possible to conclude that every task, except for T5, had a positive success rate, ranging from 80% to 89%. In the case of T5, the majority of the participants reported that they could not finish the task due to the event further explained in Section 7.4. Finally, a positive *DECS Impact* was characterized by the participants to be stronger in T2, T3, and T4 with 71%, 82%, and 80%, respectively, where distributed techniques were applied. Similar to the previously described observation, T5 was characterized as being slightly negatively impacted by DECS, possibly due to the same event. Figure 7.6 illustrates the evolution of the participant's experience with DECS throughout this activity. This graphic starts with a neutral experience characterization of 50% and sequentially increases on each task until task 4, where it reaches its maximum value of 72%, before dropping considerably to 52% due to the previously mentioned event. When analyzing the participant's final experience evaluation, it is likely that task 5 influenced the participant's overall experience with the system and made this metric lower from 72% in task 4 to 62% in the final evaluation.

The analysis of task-specific questions like *Q17* and *Q21* of the task sheet, related to the conclusions of Tasks 3 and 4, respectively, makes it possible to both understand what were the conclusions reached by the participants and also a problem indicator, if something unexpected happened. Tables 7.6 and 7.7 compile and illustrate the results of *Q17* and *Q21*, respectively.

Table 7.6: Q17 - Results

Type	Responses	Percentage (%)
Local	8	80
Distributed	2	20

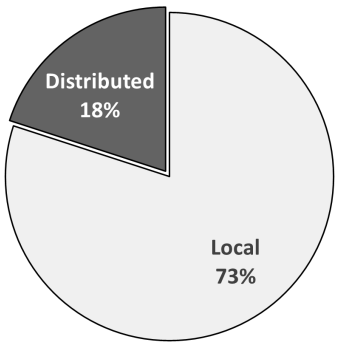


Figure 7.7: Q17 - Answer Representation

Figure 7.7, which represents the response distribution of question *Q17*, where the participants were asked which was the most efficient execution. A majority of 73%

stated that the local execution of Task 1 was the most efficient, while the rest stated it was the distributed evaluation execution of Task 2.

Table 7.7: Q21 - Results

Cores	Responses	Percentage (%)
4	1	9.1
12	5	45.5
48	5	45.5

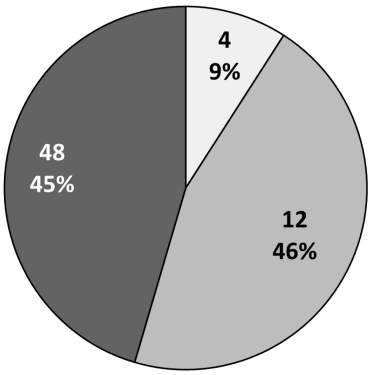


Figure 7.8: Q21 - Answer Representation

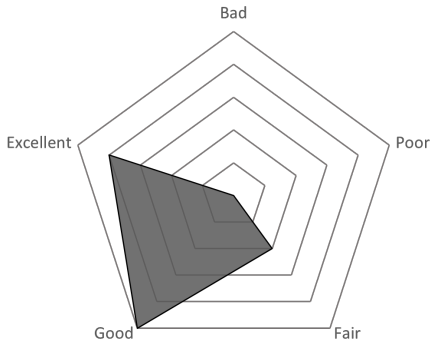
Figure 7.8, representing the results of *Q21*, shows an interesting divergence in responses. There were two Slave computers connected to the DECS system, each with a 12-core CPU. Interestingly, five participants answered both 12 and 48 cores. There was also one answer with the value 4.

Following the task-specific questions, the final part of the survey is then analyzed. Question *Q27* aims to generally rate DECS in different domains. Table 7.8 contains the responses to this question.

Table 7.8: Q27 - DECS Evaluation Results

	Visual Aspect		Structure		Easy Interaction		Usability	
Descriptor	Resp.	%	Resp.	%	Resp.	%	Resp.	%
Bad	0	0	0	0	0	0	0	0
Poor	0	0	0	0	4	36.4	3	27.3
Fair	2	18.2	5	45.5	1	9.1	4	36.4
Good	5	45.5	6	54.5	6	54.5	4	36.4
Excellent	4	36.4	0	0	0	0	0	0

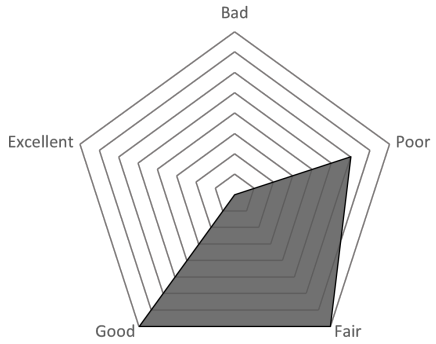
Resp. : Number of responses
% : Percentage



(a) Visual Aspect

(b) Structure

(c) Easy Interaction



(d) Usability

Figure 7.9: Q27 - DECS Evaluation Representation

Figure 7.9a represents the participant's evaluation of the web application *Visual Aspect*, which shows outstanding results, with most of the answers rating this metric as "Good" and "Excellent". Regarding the *Structure* metric (Figure 7.9b), the responses are mainly categorized as "Fair" and "Good", with a slightly greater emphasis on the latest. The characterization of the *Easy Interaction* metric (Figure 7.9c) was primarily identified as "Poor" and "Good" with one answer as "Fair". Finally, regarding the *Usability* metric (Figure 7.9d), the results are distributed within "Poor", "Fair" and "Good", with a stronger emphasis on both latest.

In order to better understand the system's usability, combined with the overall satisfaction, question *Q29* asks participants to rate how likely they would be to use DECS in the future.

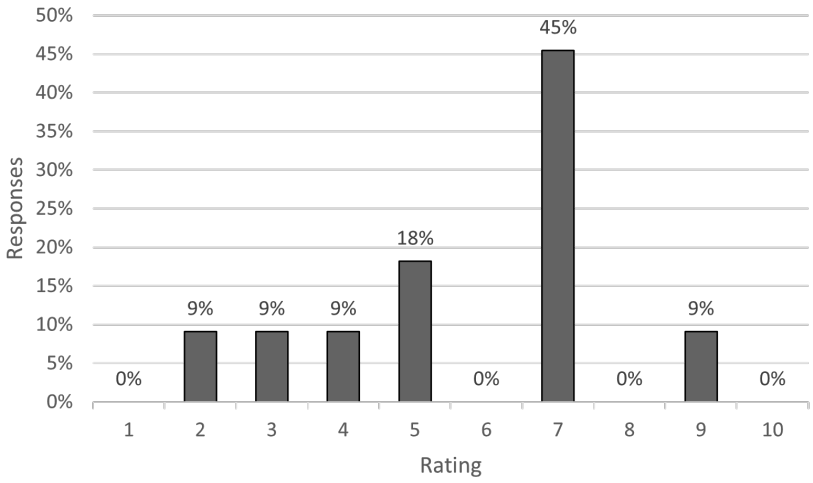


Figure 7.10: Q29 - Answer Distribution

Figure 7.10 represents the distribution of the *Q29* responses, indicating that approximately 54% of the participants would effectively use the system in the future, 18% expressed a neutral opinion, and only 27% stated that they would not be interested in using the system.

When asked on which area of DECS should be improved, in question *Q31*, the majority of the participants stated the "Usability" and "Internal Operation" domains,

as represented in Table 7.9 and Figure 7.11. It is important to take into consideration that the adverse events explained in Section 7.4 could have influenced some of the participant's answers.

Table 7.9: Q31 - DECS Improvements Answers

Section	Responses
User Interface	2
Internal Operation	5
Availability	3
Usability	7
None	1

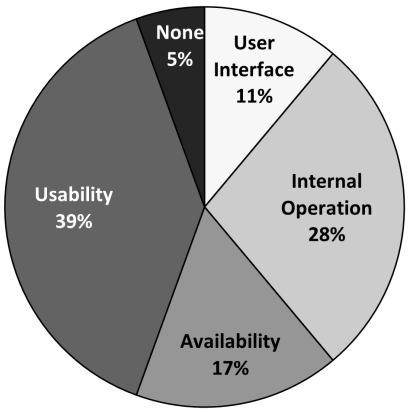


Figure 7.11: Q31 - Answer Distribution

Tables 7.10 and 7.11 represent the relation between the participant's skills in the Evolutionary Algorithms and Distributed Computing contexts, respectively, and their responses in question Q31. The results indicate that the majority of the students who rated DECS easy interaction as "Poor", rated their skills in EAs and DC as "Average" and "Bad", respectively. On the other hand, the majority of the students who rated DECS easy interaction as "Good", rated their skills in EAs and DC as "Average" and "Good".

Table 7.10: Relation between Participant EAs skills and Q31 response

EAs \ EI	Poor	Good
Bad	0	0
Average	75%	66.7%
Good	25%	33%

EI : Easy Interaction
Bad : None + Poor

Table 7.11: Relation between Participant DC skills and Q31 response

DC \ EI	Poor	Good
Bad	50%	33.3%
Average	25%	16.7%
Good	25%	50%

EI : Easy Interaction
DC : Distributed Computation
Bad : None + Poor

Question *Q32* aims to understand if the participants felt any improvement when using distribution methods for evolutionary algorithms. The answers were compiled in Table 7.12 and illustrated in Figure 7.12. The majority of the participants (55%) stated that there was no feeling of improvement. However, 45% of the participants stated that improvements were felt.

Table 7.12: Q32 - Distribution improvements answers

Option	Responses	Percentage (%)
Yes	5	45
No	6	54

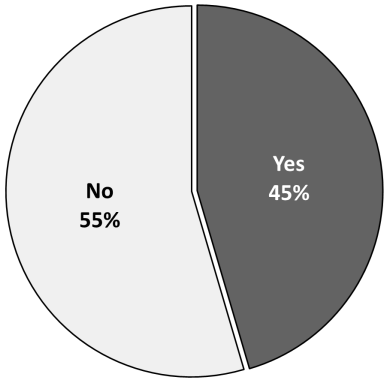


Figure 7.12: Q32 - Answer Distribution

Finally, question *Q33* asks the participants to rate what they have learned with this activity. Figure 7.13 illustrates the data represented in Table 7.13. A total of 63% of the participants positively rated the insights gained from this activity,

while 18% negatively rated them, and another 18% expressed a neutral opinion.

Table 7.13: Q33 - Learning rate answers

Rating	Responses	Percentage (%)
Negative	2	18
Neutral	2	18
Positive	7	63

Negative : Rating range 1-4
Neutral : Rating 5
Positive : Rating range 6-10

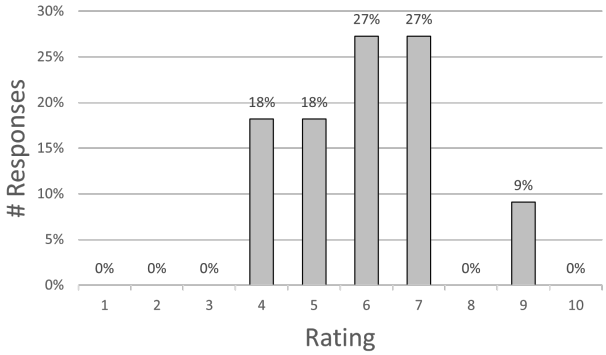


Figure 7.13: Q33 - Answer Distribution

8 Discussion

In this chapter, the results of both experimental studies previously described in this work are discussed and interpreted. First, the *Technical Study* (Section 6) results are interpreted with the aim of understanding how different distribution models impact the efficiency of evolutionary algorithms and ultimately answering the main research question. Afterwards, the *User Experience Study* (Section 7) results are interpreted in order to establish a user-side evaluation of the proposed DECS system. Finally, the limitations of both studies results are described.

8.1 Distribution Methods Evaluation

Local Group Evaluation

When comparing the results of both problems used on each test suite of the technical study, contained in Tables 6.3 and 6.4, specifically the Meta-EA problem optimizing the Rastrigin function and the 11-Bit Multiplexer problem, it is possible to establish some interesting conclusions. The Meta-EA problem tested in L1 has a very low CPU time of just 152 milliseconds compared with the L2 value of 14660 milliseconds. However, this does not make much sense because Meta-EA algorithms are much more complex and resource-intensive than normal EA problems due to their double evolution process, as explained in Section 2.1.4. This drastic difference can be explained based on a limitation further clarified in Section 8.3, which specifies how this metric is collected during the test execution. In the case of test L1, the CPU time metric only represents the amount of time the CPU spends on executing instructions related to the Meta optimizer and its own evolution, excluding the Rastrigin function evolution, which is executed in another thread and therefore not considered by this metric. In conclusion, the comparison

of the CPU time between tests L1 and L2 cannot be directly made without a close understanding of their different representations on each test.

An important interpretation can also be taken from this test suite regarding the network data metric. Since there is no distribution method applied in both executions, this metric represents the amount of data exchanged on the network by the web application. These values serve as a reference for understanding the impact of this component on the metric across other test suites.

Finally, when comparing the used memory values for each test (L1 and L2), it is possible to conclude that the amount of non-heap memory consumed is similar on both tests, but divergences can be seen when comparing the heap memory usage. While test L1 consumes an average of 233.49 MiB, test L2 only consumes an average of 191.66 MiB, representing a decrease. This is because heap memory is primarily used for object allocations. Test L1 requires object allocations for two evolution processes, which adds complexity. In contrast, test L2 involves only one evolution process, resulting in fewer overall object allocations.

Distributed Evaluation

Generally, for the same number of computers, the wall-clock time decreases with the increase of processing units in the form of DECS-Slave processes, with a limit. Concentrating a large number of slave processes on the same computer decreases the processing efficiency by raising the wall-clock time. This can be seen in Table 6.8 and illustrated in Figure 6.5b, where the wall-clock time increases and counteracts the decreasing behavior when running 25 slave processes on the same machine, both with zero and one Slave computer tests and increases even more when 50 slave processes are executed on the same computer. Considering the relevance of this metric in the overall efficiency of the evolutionary algorithm, these findings support the theory that concentrating more than 25 DECS-Slave processes on each computer negatively impacts processing efficiency. A concrete example of this finding is the cancellation of test DE2.3 with zero Slave machines, since multiple tries were made to execute the coordinator and 50 slave processes on the same computer. Consecutive OS crashes forced the abortion of the test and ultimately removing it from the test suite. On the other hand, the results show

that the structure and configuration of the system can heavily impact processing efficiency.

In Figure 6.5c, an interesting aspect stands out regarding the CPU time value range for the zero Slave machines series, which is considerably lower than the rest. There is no explanation for this difference supported by concrete evidence. Regarding the network data metric, represented in Figure 6.5d, the extreme variations seen in the zero machines and one machine do not have any evidence-supported explanation, however, they can be related to the study limitations (Section 8.3) regarding the collection method for this metric.

With a general analysis of Table 6.8, it is possible to establish a comparison between the longest execution, characterized by zero Machines with one slave process, which took 324.35 seconds to execute, and the fastest, characterized by five Machines with a total of 50 slave processes in the processing network, which only took 18.72 seconds to execute, totaling a difference of 305.63 seconds saved by using this distribution method.

When interpreting the results of the *TH1* theory described in Section 6.5.2, which tries to understand if the performance metrics can be predicted by the division of the ones obtained using one computer by the number of computers in the processing network, it is possible to conclude that this theory is true, but it is only verified in some metrics, as represented in Table 6.12. With a close interpretation of each metric percent error, the Wall-Clock Time and Network Data metrics stand out for their lower error values, indicating that this theory can be applied in real-life applications for predicting their values for a specific number of computers. However, the increase in error with the increase in computers indicates that this theory only Predicts relatively accurate values until the number of computers in the processing network reaches a certain threshold. This value cannot be defined within this study due to the limited number of computers tested.

The results of the *Local Test Group* (Section 6.4) are obtained from an important comparison between the execution of an evolutionary problem locally, without any distribution methods applied, and the overall fastest execution using the distributed evaluation method. As expected, the distributed execution is faster, with a total of 30.08 seconds saved. However, the internal complex tasks that are required to implement the distribution consume system resources, such as the

coordinator's CPU, which represents an increase of 136 milliseconds in CPU Time compared with the local execution. A considerable increase in the network data exchanged by the coordinator and slave processes can also be seen, which suffered an increase from 36.34 KiB and 1020.22 KiB in the local and distributed executions, respectively, totaling a 983.66 KiB increase, caused by the large number of Slave processes and computers in the processing network, which demand a large amount of information to be exchanged by the coordinator and Slave computers in order to accomplish the task distribution among all processing nodes. An interesting aspect can be noticed regarding the amount of used memory, which shows a lower value in the distributed execution. Since the non-heap memory usage only suffers a small increase of approximately 2.25%, it does not significantly impact the overall value. However, when analyzing the substantial decrease of 66.16% in the heap memory usage, an explanation can be formulated. Since the heap memory is typically used in object allocations, this decrease can be related to a possible object allocation distribution, which implies that instead of allocating all objects in the coordinator and sending them through the network, they could be allocated on each processing node, freeing up resources in the coordinator, such as its heap memory, as seen in the results.

In conclusion, the *Distributed Evaluation* method has a positive impact on some metrics, such as Wall-Clock Time and Used Memory, but it can increase the consumption of some resources, such as CPU time and network usage. Since all metrics have an indirect relation and influence the total time taken to execute the test, represented by the Wall-Clock time metric, this is considered the most relevant to the study and concludes that the distributed evaluation method has a generally positive impact on the processing of evolutionary algorithms.

Island Model

With the interpretation of the *Island Model* test suite results, it is possible to identify some patterns and establish relevant conclusions related to the configurations of this model. Generally, highly frequent big migrations increase the number of generations required for a solution to be found, which negatively impacts evolution efficiency. This can be explained by the fact that too frequent and large migrations can cause homogenization of the gene pool across islands, which reduces the

overall genetic diversity and can cause disruption of the local evolution and its gene adaptations by excessively introducing foreign genes. Migrations containing a large number of individuals increase the amount of data exchanged in the network, which increases the overall processing overhead. Interestingly, when using three islands, the migration frequency has an inverse relation with memory usage, which is raised on infrequent migrations and inverted on frequent ones. The migration size has a direct relation with memory usage. However, when using five islands, a direct relation is seen in both settings.

Generally, the full migration topology is more efficient with five islands, while the ring migration topology is the most efficient with three islands. Regarding the synchronicity of migrations, it is possible to conclude that asynchronous migrations provide better overall performance with the full topology when compared with synchronous ones. On the other hand, synchronous migration provides better network performance with three islands, however, better results can be achieved when combining five islands with asynchronous migrations. The best overall setting for network usage is three islands with full topology migrations, which is not fully expected due to the ring topology being theoretically the most efficient in the context of network usage. Interestingly, migration topology and synchronicity do not seem to have any influence on the used memory. In fact, regarding this metric, using five islands proved to be the most efficient solution in all tests, except for the frequency of the migrations, which creates a memory overhead and increases its usage. There seems to be an outlier in the IT test when using three islands with a ring and asynchronous migrations since its values on all metrics show a considerable increase, which does not seem to be connected with either setting. The setting that proved to be the worst in the IT test suite is the five island full topology synchronized migrations, which represent a higher value for all metrics, with the exception of memory usage.

The results presented in Table 6.4, in the *Local Test Group* regarding the test L2, where local executions of the problem are compared with a selected test using the island model distribution, allow some interesting conclusions to be drawn. A drastic reduction in the number of generations required to achieve a solution when using the island model represents a great improvement in the evolutionary efficiency and effectiveness of the model. A general decrease in the wall-clock time and a more drastic one in the CPU time when using the island model proves that

the model also improved computing performance, ultimately making the algorithm take less time to reach a solution and consume fewer resources. As expected, the use of the island model increases the overall network usage. This is expected and generalized for all inter-computer distribution methods where communication between processing nodes must be done through the network. This increased traffic is due to the migrations between islands and is especially aggravated because the selected test (IFS2.3) is configured with highly frequent low-size migrations between five islands. A decrease in the overall amount of memory used when using the island model is mainly due to the decrease in heap usage. This supports the idea that the use of this model successfully reduced resource consumption and increased computing performance.

In conclusion, the *Island Model* had a significant positive impact on the overall algorithm efficiency, both in evolutionary efficiency in the context of the generations metric and computing performance, considering the rest. Since this model has as its main objective to improve evolutionary efficiency by reducing the number of required generations to find a solution, improvements in this area were expected and verified. However, this model also proved to be effective in computing performance, as discussed in this section. It improved wall-clock time, CPU time, and even memory usage, which was not initially expected.

8.2 User Experience Evaluation

In Figure 7.6, where the evolution of the participant's experience with DECS throughout the activity is illustrated, an interesting pattern can be seen. The overall experience with the system increases with each task, depending on its success rate. This conclusion is verified due to the drastic decrease in the success rate of task five, which has a clear influence on its experience evaluation and global value. This can also mean that the majority of the participants, intentionally or not, considered their individual task success rate as a big contributor to their experience.

In Figure 7.7, is possible to see the perfect example of a problem indicator question. Task 3 (T3) was conceived to make the participants realize that by applying the distributed evaluation technique to an evolutionary problem, their execution could

be faster. As explained and proved in the *Technical Study* (Section 6), there is a minimum structure size required to compensate for the processing overhead of distribution methods and ultimately represent a faster execution. Since these requirements weren't achieved due to the adverse event explained in Section 7.4, the distributed execution of task 2 was even slower than the local one of task 1. This is indicated by the majority of the participants, who characterized the local execution as more efficient than the distributed.

In question *Q21*, which verifies if the participants can retrieve and interpret information from the connected processing nodes, specifically the number of CPU cores. The goal of this question was for the participant to identify the number of connected slave machines and their respective CPU core counts. With the multiplication of these values, it is possible to achieve the total number of CPU cores in the processing network, which was the intended response. However, there was a crucial detail, that the participants should have noticed. In the DECS web application, there were four entries, one for each connected DECS-Slave instance, but they were distributed on only two computers. Since the objective of this task was to explore all connected slave processes, the participants must have identified that there were only two different IP addresses on all slave instances. The almost bipolar divergence in these responses suggests that participants may have interpreted the question in two different ways, completely ignoring the fact that there were only two computers connected to the processing network. The question is formulated in the following way:

"Q21 - Please enter the number of cores of all slave machines in your cluster."

Some participants may have interpreted the question as the number of CPU cores on each slave process, where the correct answer would be *12*, concentrating 46% of the responses. Others may have interpreted it in the almost intended way, considering the sum of all cores, with the correct answer being *48*, totaling 45% of the responses. However, none of the participants totally understood the question in the intended direction, which was only *24* cores, since there were two computers on the processing network, each with 12 cores. In any case, the objective of the task was fulfilled, and the participants could retrieve information from the web application.

In question *Q27*, participants expressed opposite opinions regarding the system's ease of use. While most found it user-friendly, a significant number found it challenging. This aspect can be influenced by each participant's personal experience with the system. A very interesting conclusion from a further analysis of the relation between this question and *EQ3*, which collects the participant skills regarding Evolutionary Algorithms and Distributed Computation is represented by Tables 7.10 and 7.11. It is possible to generally conclude that most of the participants who found the system challenging to interact with, classified their skills lower in both EAs and distributed computing when compared with the ones who considered the system easy to use. This relation verifies that the use of DECS may require a certain knowledge base in the mentioned areas, and this can impact the user experience. Nevertheless, this evaluation must be taken into consideration for future improvements to obtain more stable and positive results.

Regarding question *Q33*, almost half of the participants stated that they felt improvements when using distribution methods for evolutionary algorithms. Since the island distribution task (T5) could not be executed and it is unlikely to have influenced this question, when comparing this result with Table 7.6, referring to question *Q17*, where participants stated, with 73% representation that the distributed execution was not the most efficient, these results seem to contradict each other in a superficial technical analysis. Nevertheless, this question aims to understand the participant's personal experience and opinion, so its results should not be technically interpreted.

Finally, the majority of positive answers to question *Q37* represent the success of the user experience evaluation activity, which above all was aimed not only at an evaluation of a system but also at the learning of the participants by introducing other techniques, applications, problems, and tools for the distributed computing of evolutionary algorithms.

8.3 Limitations

The findings of the technical study have to be seen in light of some limitations, as these can have an influence on the results and conclusions. The first is related to the fact that only one evolutionary algorithm was tested for each distribution method. Therefore, with this study, it is not possible to understand the influence of these methods on other types of evolutionary algorithms. However, the current research work can be easily extended by testing different evolutionary problems using different types of EAs in order to tackle this limitation. The flexibility of DECS and its evolutionary engine, powered by ECJ, facilitate straightforward problem implementation. This gives users the freedom to tackle a wide range of problems with ease. Secondly, the defined value for executions per test can be considered small in the context of high-precision testing standards. This can be further aggravated by the characteristic randomness factor used in some stages of the evolutionary cycle, such as the creation of the initial population of some problems in the initialization phase, the crossover points in the recombination phase, or even in the fitness evaluation of some noisy environments and simulations, where there is some degree of randomness introduced. These factors would ultimately require more executions to achieve a precise average, as seen in some collected metrics where each run generates considerably distant values from each other. However, in the illustrative context of this work, five executions proved to be sufficiently accurate to achieve valid conclusions. Finally, the collection of network and memory usage metrics does not follow a strict isolation procedure, which means that these metric values may not derive solely from evolutionary-related operations, including task distribution. This can be explained by the way these metrics are collected. Every test execution is recorded by a profiler, which extensively collects all sorts of events and information related to the DECS coordinator process. However, due to the complexity of the coordinator, several threads are performing simultaneous tasks to assure the functioning of all system components, including the web application and evolutionary engine. Since both of these components exchange information over the network and use both heap and non-heap memory sections, their global usage is included in these metrics. To address this limitation, only the strictly necessary actions were performed in the web application to avoid unnecessary network and memory consumption. Some

aspects of the CPU time metric collection must also be considered when analyzing its values. Unlike the previously described metrics, which refer to all coordinator components, the CPU time metric only refers to the thread running the evolutionary engine and consecutively all the evolutionary-related instructions, removing any external influences from this metric. It is also important to notice that any evolutionary operation that is executed in another thread won't be considered by this metric. Nevertheless, these metrics must be carefully analyzed in light of these limitations.

Nonetheless, the user experience study results must also be interpreted with caution, and two main limitations should be kept in mind. The first issue is related to the small number of participants, which in this case was limited to 11. This can have negative effects on the results, such as limited generalizability, where a small group may not be representative of the larger target audience, a higher risk of individual biases and outliers affecting the overall results, increased variability originating from the participant's individual differences, making it difficult to identify consistent patterns, and several more. Secondly, the adverse events that occurred during the evaluation and further described in Section 7.4 may have had a considerable impact on the results, so these must be carefully analyzed together with the results in order to avoid inaccurate conclusions.

In conclusion, for both studies, it is important to note that DECS is an evolving project undergoing continuous improvements and corrections with each new release. Therefore, future interpretations of the results from this work must consider that version 1.0 was used for both studies.

9 Conclusion

The purpose of this thesis is to further understand and evaluate the impact of different distribution models on the efficiency of evolutionary algorithms. In order to fulfill this objective, a distributed system that allows the creation and execution of evolutionary problems using distribution methods was conceived and implemented, using ECJ as its evolutionary engine. Afterwards, a technical study was performed in order to answer the main research question. Based on the experimental procedures described and interpreted in the Technical Study (Section 6) and Discussion (Section 8), respectively, which focus on the *Distributed Evaluation* and *Island Model* distribution methods, it is possible to conclude that they have a generally positive impact on distinct aspects of evolutionary algorithms. While the distributed evaluation method stands out for its increase in computing performance, lowering the required time to process a certain number of generations and lowering the memory consumption on each computer by distributing it, it also poses a processing overhead due to distribution handling, which is represented by a general increase in CPU time. On the other hand, the island model effectively enhances evolutionary efficiency by reducing the number of required generations to find a solution. However, the results also demonstrate that the model improves computing performance, as evidenced by reductions in wall-clock time, CPU time, and memory usage, leading to faster and more resource-efficient executions. Nevertheless, both methods increase network usage, which is essential for inter-computer distribution and must be considered when implementing them. To evaluate the proposed system in a non-empirical context, a user experience study was conceived and applied to a group of participants. This study, composed of a set of tasks to be solved with the system and then individually evaluated by each participant, concludes that a positive overall experience was felt by the participants, who shared positive impressions regarding the visual aspect and structure of the web application, but some divided opinions concerning the system's ease

of use and usability. Nevertheless, the full successful conclusion of all tasks, with the exception of the last, which could only be partially finished due to an adverse event, by all participants, represents the success of the system's implementation in solving evolutionary problems using distribution methods and ultimately in the creation of an abstraction layer, which hides the complexities of the distribution operations, providing a simple user interface.

Future Work

The DECS system is an evolving project and can be extended in various ways.

More distribution models for evolutionary algorithms can be implemented and further experimentally analyzed.

The technical study can be extended by including different types of problems and EAs, which can behave differently when distribution methods are applied.

The model proposed in Figure 3.4, which results from the combination of both island model and distributed evaluation, can be implemented in DECS and included in the technical evaluation to establish a performance comparison with the isolated distribution methods and identify possible improvements and setbacks.

Bibliography

[1] Arenas, M. G. ; Collet, Pierre ; Eiben, A. E. ; Jelasity, Márk ; Merelo, J. J. ; Paechter, Ben ; Preuss, Mike ; Schoenauer, Marc: A Framework for Distributed Evolutionary Algorithms. In: Guervós, Juan Julián M. (Hrsg.) ; Adamidis, Panagiotis (Hrsg.) ; Beyer, Hans-Georg (Hrsg.) ; Schwefel, Hans-Paul (Hrsg.) ; Fernández-Villacañas, José-Luis (Hrsg.): *Parallel Problem Solving from Nature — PPSN VII* Bd. 2439. Springer Berlin Heidelberg, S. 665–675. – URL http://link.springer.com/10.1007/3-540-45712-7_64. – Zugriffsdatum: 2024-02-18. – Series Title: Lecture Notes in Computer Science. – ISBN 978-3-540-44139-7 978-3-540-45712-1

[2] Cahon, S. ; Melab, N. ; Talbi, E.: ParadisEO: A framework for the reusable design of parallel and distributed metaheuristics. In: *Journal of Heuristics* 10 (2004), Nr. 4, S. 357–380

[3] Cantú-Paz, E.: A survey of parallel Genetic Algorithms. In: *Calculateurs Paralleles, Reseaux et Systems Repartis* 10 (1998), S. 141–171

[4] Cantú-Paz, E.: *Efficient and Accurate Parallel Genetic Algorithms*. Bd. 1. Springer, 2000

[5] Costa, J. ; Lopes, N. ; Silva, P.: *JDEAL: The Java Distributed Evolutionary Algorithms Library*. http://laseeb.isr.ist.utl.pt/sw/jdeal/home.html

[6] Cramer, N. L.: A Representation for the Adaptive Generation of Simple Sequential Programs. In: *Proceedings of the International Conference on Genetic Algorithms and Their Applications*, 1985, S. 183–187

[7] Darwin, Charles: *On the Origin of Species*. London : John Murray, 1859

[8] Di Lucca, Giuseppe ; Fasolino, Anna: *Web Application Testing*. S. 219–260, 03 2006. – ISBN 978-3-540-28196-2

[9] Dubreuil, M. ; Gagné, C. ; Parizeau, M.: Analysis of a master-slave architecture for distributed evolutionary computations. In: *IEEE Transactions on Systems, Man, and Cybernetics Part B (Cybernetics)* 36 (2006), Nr. 1, S. 229–235

[10] Eiben, A. E. ; Smith, James E.: *Introduction to Evolutionary Computing*. 2nd. Springer Publishing Company, Incorporated, 2015. – ISBN 3662448734

[11] Forrest, Stephanie ; Mitchell, Melanie: What makes a problem hard for a genetic algorithm? Some anomalous results and their explanation. In: *Machine Learning* 13 (1993), Nov, Nr. 2, S. 285–319. – URL https://doi.org/10.1007/BF00993046. – ISSN 1573-0565

[12] Fortin, Félix-Antoine ; Rainville, François-Michel De ; Gardner, Marc-André ; Parizeau, Marc ; Gagné, Christian: DEAP: Evolutionary Algorithms Made Easy. In: *Journal of Machine Learning Research* 13 (2012), Nr. 70, S. 2171–2175. – URL http://jmlr.org/papers/v13/fortin12a.html

[13] Gagné, C. ; Parizeau, M.: Open BEAGLE: A New Versatile C++ framework for evolutionary computation. In: *Late Breaking Papers of GECCO 2002*, 2002

[14] Gong, Yue-Jiao ; Chen, Wei-Neng ; Zhan, Zhi-Hui ; Zhang, Jun ; Li, Yun ; Zhang, Qingfu ; Li, Jing-Jing: Distributed evolutionary algorithms and their models: A survey of the state-of-the-art. 34, S. 286–300. – URL https://www.sciencedirect.com/science/article/pii/S1568494615002987. – Zugriffsdatum: 2024-02-18. – ISSN 1568-4946

[15] Hidalgo, J.I. ; Fernández, F.: Balancing the computation effort in genetic algorithms. In: *IEEE Congress on Evolutionary Computation (CEC)*, 2005, S. 1645–1652

[16] Hold-Geoffroy, Yannick ; Gagnon, Olivier ; Parizeau, Marc: Once you SCOOP, no need to fork. In: *Proceedings of the 2014 Annual Conference on*

Extreme Science and Engineering Discovery Environment ACM (Veranst.), 2014, S. 60

[17] Holland, John H.: Genetic Algorithms. In: *Scientific American* 267 (1992), Nr. 1, S. 66–73. – URL http://www.jstor.org/stable/24939139. – Zugriffsdatum: 2024-05-09. – ISSN 00368733, 19467087

[18] Ishimizu, T. ; Tagawa, K.: A structured differential evolution for various network topologies. In: *International Journal of Computer Communications* 4 (2010), Nr. 1, S. 1–8

[19] Keijzer, Maarten ; Merelo, J. J. ; Romero, G. ; Schoenauer, M.: Evolving Objects: A General Purpose Evolutionary Computation Library ". In: *Artificial Evolution* 2310 (2002), S. 829–888. – URL http://www.lri.fr/~marc/EO/EO-EA01.ps.gz

[20] Koza, John R.: *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* USA : The MIT Press, 1992

[21] Koza, John R.: Genetic programming as a means for programming computers by natural selection. In: *Statistics and Computing* 4 (1994), Jun, Nr. 2, S. 87–112. – URL https://doi.org/10.1007/BF00175355. – ISSN 1573-1375

[22] Koza, John R.: *Genetic Programming II: Automatic Discovery of Reusable Programs.* USA : The MIT Press, 1994

[23] Manfrin, M. ; Birattari, M. ; Stützle, T. ; Dorigo, M.: Parallel ant colony optimization for the traveling salesman problem. In: *Ant Colony Optimization and Swarm Intelligence*, 2006, S. 224–234

[24] Michel, R. ; Middendorf, M.: An island model based ant system with lookahead for the shortest supersequence problem. In: *Parallel Problem Solving from Nature (PPSN)*, 1998, S. 692–701

[25] Myles, M. ; Callahan, A. ; Fogel, L.J. ; Naval Research, United States. O. of ; Foundation, Allan H.: *Biophysics and Cybernetic Systems: Proceedings of the Second Cybernetic Sciences Symposium.* URL https://books.google.de/books?id=pwNCAAAAIAAJ, 1965 (Cybernetic Sciences Symposium). – 56–131 S

[26] Piriyakumar, D.A.L. ; Levi, P.: A new approach to exploiting parallelism in ant colony optimization. In: *International Symposium on Micromechatronics and Human Science*, 2002, S. 237–243

[27] Poli, Riccardo ; Langdon, William B. ; McPhee, Nicholas F.: *A Field Guide to Genetic Programming*. Lulu Enterprises, UK Ltd, 2008. – 14,15 S. – ISBN 1409200736

[28] Rastrigin, L. A.: *Systems of Extremal Control*. Moscow : Mir, 1974

[29] Rechenberg, Ingo: Evolutionsstrategie : Optimierung technischer Systeme nach Prinzipien der biologischen Evolution, URL `https://api.semanticscholar.org/CorpusID:60975248`, 1973

[30] Rivera, W.: Scalable parallel genetic algorithms. In: *Artificial Intelligence Review* 16 (2001), S. 153–168

[31] Rivest, Ronald L.: The MD5 Message-Digest Algorithm. In: *RFC 1321* (1992)

[32] Rudolph, G.: *Globale Optimierung mit parallelen Evolutionsstrategien*, Department of Computer Science, University of Dortmund, Dissertation, July 1990

[33] Said, S.M. ; Nakamura, M.: Asynchronous strategy of parallel hybrid approach of GA and EDA for function optimization. In: *International Conference on Networking and Computing*, 2012, S. 420–428

[34] Said, S.M. ; Nakamura, M.: Parallel enhanced hybrid evolutionary algorithm for continuous function optimization. In: *International Conference on P2P, Parallel, Grid, Cloud and Internet Computing*, 2012, S. 125–131

[35] Schwefel, H.-P.: *Interdisciplinary systems research*. Bd. 26: *Numerische Optimierung von Computer Modellen mittels der Evolutionsstrategie*. Birkhäuser Verlag, 1977

[36] Schwefel, Hans-Paul: *Evolutionsstrategie und numerische Optimierung*, Dissertation, 01 1975

[37] Scott, Eric O. ; Luke, Sean: ECJ at 20: toward a general metaheuristics toolkit. In: *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. New York, NY, USA : Association for Computing Machinery, 2019 (GECCO '19), S. 1391–1398. – URL https://doi.org/10.1145/3319619.3326865

[38] Simon, Dan: Evolutionary optimization algorithms : biologically-Inspired and population-based approaches to computer intelligence, URL https://api.semanticscholar.org/CorpusID:60429433, 2013, S. 35,36,49,74,141–143

[39] Singh, Inderpal: Review on Parallel and Distributed Computing, URL https://api.semanticscholar.org/CorpusID:15048804

[40] Storn, R. ; Price, K.: Differential Evolution - a Simple and Efficient Adaptive Scheme for Global Optimization over Continuous Spaces / International Computer Science Institute, Berkley. 1995 (TR-95-012). – Technical Report

[41] Su, S. ; Chung, C. ; Wong, K. ; Fung, Y. ; Yeung, D.: Fault tolerant differential evolution based optimal reactive power flow. In: *International Conference on Machine Learning and Cybernetics*, 2006, S. 4083–4088

[42] Tan, K. C. ; Tay, A. ; Cai, J.: Design and implementation of a distributed evolutionary computing software. In: *IEEE Transactions on Systems, Man and Cybernetics, Part C* 33 (2003), August, Nr. 3, S. 325–338

[43] Tanenbaum, Andrew S. ; Steen, Maarten van: *Distributed Systems: Principles and Paradigms*. 2. Upper Saddle River, NJ : Pearson Prentice Hall, 2007. – 17,18,37,39,41,120,121,125,126,128,129,389,394,395,405–407,461–463 S. – ISBN 978-0-13-239227-3

[44] Tsai, Wei-Tek ; Bai, Xiaoying ; Paul, Raymond ; Shao, Weiguang ; Agarwal, Vishal: End-To-End Integration Testing Design., 01 2001, S. 166–171

[45] Ursem, Rasmus ; Dissertation, Phd ; And, System ; Optimization, Control ; Krink, Supervisors ; Mayoh, Brian ; L, Typesetting ; Xfig, Figures: Models for Evolutionary Algorithms and Their Applications in System Identification and Control Optimization. (2003), 05, S. 14–16,20–22,31–33,35

A Technical Study Data

A.1 Distributed Evaluation

Table A.1: Distributed Evaluation Test Suite Data

	# Slaves	Fit.	W.C. Time	CPU Time	N. Data	S. Read	S. Write	U. Memory	H. Memory	N.H. Memory
0 Slave Machines	**1**	−1,39	324 351	167	2170,09	70,65	2099,44	252,99	73,74	179,25
	2	−1,39	166 895	159	410,97	65,92	345,06	259,81	80,89	178,92
	3	−1,38	119 140	158	962,09	67,39	894,70	259,20	77,86	181,34
	4	−1,39	97 070	159	1859,53	72,31	1787,22	257,44	78,54	178,90
	5	−1,39	83 507	163	480,71	68,52	412,19	267,91	88,32	179,59
	10	−1,39	55 001	148	506,14	73,26	432,87	262,32	83,34	178,98
	25	−1,40	56 830	178	677,26	82,91	763,67	207,02	77,86	180,91
1 Slave Machine	**1**	−1,40	307 798	343	1473,41	69,82	1403,59	258,30	78,91	179,39
	5	−1,39	82 409	325	435,73	67,76	367,97	262,76	82,67	180,08
	10	−1,40	53 736	212	2491,09	74,77	2416,32	257,88	77,84	180,04
	25	−1,39	51 095	324	633,81	97,33	694,94	209,47	80,49	181,35
	50	−1,40	75 353	308	1531,77	265,19	1266,58	265,37	81,38	183,99
2 S. M.	**1**	−1,41	158 174	324	439,99	63,46	376,53	260,13	81,09	179,04
	5	−1,38	47 396	344	538,45	66,67	471,78	255,45	76,89	178,56
	25	−1,37	34 373	313	992,10	94,21	897,89	266,33	83,84	182,49
3 S. M.	**1**	−1,36	108 580	327	428,42	74,62	353,81	248,78	69,11	179,67
4 S. M.	**1**	−1,38	84 123	321	477,04	86,50	390,54	261,74	80,29	181,45
5 Slave Machines	**1**	−1,35	69 364	331	465,56	71,09	394,47	270,34	88,45	181,90
	2	−1,38	41 210	311	554,29	80,00	474,28	263,07	80,98	182,09
	5	−1,39	23 651	317	636,74	91,77	544,97	262,73	80,65	182,07
	10	−1,39	18 719	288	1020,22	104,32	915,90	262,57	78,90	183,67

 # Slaves : Number of DECS-Slave processes
Fit. : Fitness
W.C. Time (ms) : Wall-Clock Time in milliseconds
CPU Time (ms) : CPU Time in milliseconds
N. Data (KiB) : Network Data in kibibytes
S. Read (KiB) : Socket Read in kibibytes
S. Write (KiB) : Socket Write in kibibytes
U. Memory (MiB) : Used Memory in mebibytes
H. Memory (MiB) : Heap Memory in mebibytes
N.H. Memory (MiB) : Non-heap Memory in mebibytes

A.2 Island Model

Table A.2: Island Model Test Suite Data

	# Islands	ID	Gen.	W.C. Time	CPU Time	N. Data	S. Read	S. Write	U. Memory	H. Memory	N.H. Memory
IFS	3	1.1	32	9038	693	82	12	69	267	89	178
		1.2	91	23 999	4016	12 331	5715	6616	276	100	176
		1.3	33	7628	883	128	37	91	259	84	175
		1.4	58	15 639	742	8742	7548	1194	265	86	178
	5	2.1	31	9639	753	77	12	65	259	80	178
		2.2	30	11 959	861	1978	1376	602	259	78	180
		2.3	29	10 820	794	138	31	106	262	81	180
		2.4	64	16 926	2518	10 039	3152	6887	266	87	179
IT	3	1.1	44	8767	1052	132	46	85	264	88	176
		1.2	34	6277	920	100	23	76	265	88	177
		1.3	58	8616	1647	146	55	91	264	86	178
		1.4	81	13 538	2641	276	119	157	280	102	178
	5	2.1	64	14 914	1953	296	119	176	263	85	178
		2.2	51	12 345	1395	182	50	132	261	85	176
		2.3	30	9407	821	169	31	138	260	79	180
		2.4	36	10 015	951	99	18	80	263	83	180

 # Islands : Number of Islands
ID : Test ID
Gen. : Number of generations to find a solution
W.C. Time (ms) : Wall-Clock Time in milliseconds
CPU Time (ms) : CPU Time in milliseconds
N. Data (KiB) : Network Data in kibibytes
S. Read (KiB) : Socket Read in kibibytes
S. Write (KiB) : Socket Write in kibibytes
U. Memory (MiB) : Used Memory in mebibytes
H. Memory (MiB) : Heap Memory in mebibytes
N.H. Memory (MiB) : Non-heap Memory in mebibytes

A.3 Results Spreadsheet

Description:

This supplementary Excel spreadsheet contains all collected and processed data related to the Technical Study. Each sheet corresponds to a specific test group: "Local" represents the local test group (Section 6.4); **"Dist_Eval"** represents the distributed evaluation test group (Section 6.5) together with a data analysis in **"Dist_Eval_Visuals"**; **"Islands"** represents the island test group (Section 6.6), together with a data analysis in **"Islands_Visuals"**.

Filename:

`DECS_Technical_Study_Results.xlsx`

File Path:

`\BA_Digital_Appendix_Bruno_Guiomar\DECS_Evaluation\Technical_Study\`

B User Experience Task Sheet

Local Problem Execution In this task you will run the Meta Problem locally, without any type of distribution. The result of this execution can be used as a control group for the efficiency evaluation of the distribution techniques.

Problem ID	Meta
Job Name	Meta_team_number

1.1 Run the Meta Problem with the given specifications.

1.2 Please answer Section 1 of the survey.

Distributed Evaluation Now, lets try to use the system's full power to execute this problem using the distributed evaluation technique, but first we need to change some parameters on the problem configuration.

Problem ID	MetaDist_team_number
Job Name	MetaDist_team_number

master.params

Parameter	**Value**
eval.masterproblem.max-jobs-per-slave	2

Meta.params

Paramter	**Value**
generations	20
pop.subpop.0.size	30
pop.subpop.0.species.mutation-prob	0.25
pop.subpop.0.species.mutation-stdev	0.13

2.1 Edit the problem parameter files according to the specifications and save them. **Note: Please only edit your team's problem (Problem ID in the specifications)**

2.2 Run your team's MetaDist Problem with the specified job name.

2.3 Please answer Section 2 of the survey.

Result Analysis Did you noticed any differences between both executions? I will give you an hint! Maybe you didn't had to wait so long for the last execution, but we are scientists and we don't trust anyone without empirical evidences :). Fortunately DECS have a detailed logging system that provides you with multiple information about each Job, including both Wall-clock time and CPU time.

3.1 Explore the results from the previously executed jobs. **Note: Pay attention to the job name, jobs from other teams may show in your Job Activity List.**

3.2 Fill in the following tables with the results from both jobs.

3.3 Analyse and compare both tables. What conclusion do you draw from this analysis?

3.4 Please answer Section 3 of the survey.

Meta Job	
Wall-Clock Time	
CPU Time	

MetaDist Job	
Wall-Clock Time	
CPU Time	

Know more about your cluster In some cases, it is important to register some features of the machines running in our cluster and these should be accessible to the user. This can be particularly important when you are performing experimental studies.

4.1 Explore the information provided about the available Slave machines.

4.2 Please answer Section 5 of the survey.

Island Distribution Since there is no problem available using Island Distribution we need to first create one with the following specifications:

Problem ID	B11Isla_team_number
Job Name	B11Isla_team_number

Simple Tab

Parameter	Value
Generations	120

Koza Tab

Parameter	Value
Crossover Pipe Prob.	0.75
Repr. Pipe Prob.	0.3
Tournament Size	30

Islands Tab

Parameter	Value
# Islands	5
Island ID	[see B.1]
Migration Number	1
Migration Size	18
Migration Start	[see B.1]
Migration Offset	20
Mailbox Size	40

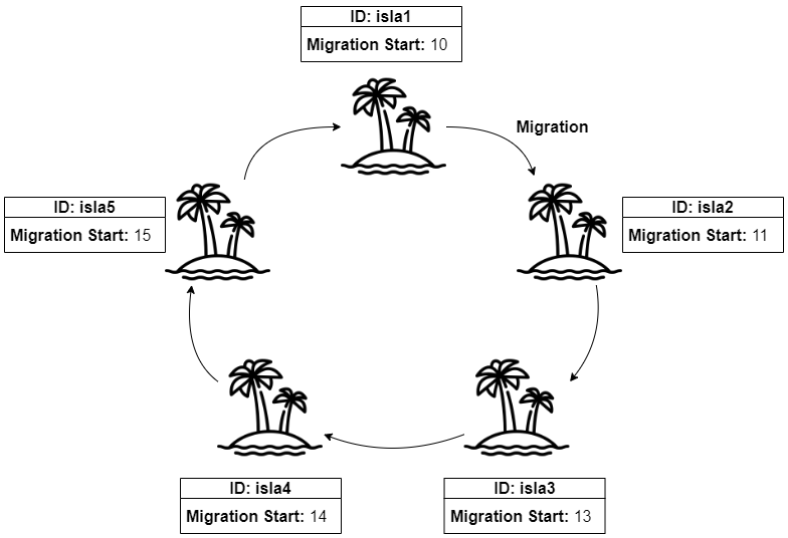


Figure B.1: Islands Topology

5.1 Create a new B11Multiplexer problem with Island Distribution according to the specifications. Pay special attention in the Islands configuration. You should follow the given topology B.1.

5.2 Save the problem. When saving the problem make sure to include your team's number so you can identify it later.

5.3 Run the problem created on the previous steps. **Note: if the problem doesn't show in the list, click on the refresh button.**

5.4 Explore the results.

5.5 Please answer Section 4 of the survey.

Thank you for you participation.

C User Experience Survey

User Profile

1. What is your course of study?

2. In which semester do you study?

3. How could you rate your skills in Evolutionary Algorithms and Distributed Computation? ['None', 'Poor', 'Average', 'Good', 'Excellent']

Task 1 - Local Problem Execution

4. How could you rate your overall experience with DECS so far? [0-10]

5. How could you rate the following aspects regarding this task? [1-5]

6. Did you managed to complete this task?

7. Please enter the Wall-Clock time of the job executed in this task.

8. Please enter the concrete solution of the job executed in this task. (Set of parameters and respective values)

Task 2 - Distributed Evaluation

9. How could you rate your overall experience with DECS so far? [1-10]

10. How could you rate the following aspects regarding this task? [1-5]

11. Did you managed to complete this task?

12. Please enter the Wall-Clock time of the job executed in this task.

13. Please enter the concrete solution of the job executed in this task. (Set of parameters and respective values)

Task 3 - Result Analysis

14. How could you rate your overall experience with DECS so far? [1-10]

15. How could you rate the following aspects regarding this task? [1-5]

16. Did you managed to complete this task?

17. Which was the most efficient execution?

Task 4 - Know more about your cluster

18. How could you rate your overall experience with DECS so far? [1-10]

19. How could you rate the following aspects regarding this task? [1-5]

20. Did you managed to complete this task?

21. Please enter the number of cores of all slave machines in your cluster.

Task 5 - Island Distribution

22. How could you rate your overall experience with DECS so far? [1-10]

23. How could you rate the following aspects regarding this task? [1-5]

24. Did you managed to complete this task?

25. Please enter the Wall-Clock time of the job executed in this task.

26. Please enter the concrete solution of the job executed in this task. (Multiplexer Map)

Final Questions

27. How could you rate DECS in the following aspects? ['Bad', 'Poor', 'Fair', 'Good', 'Excellent']

28. How can you rate your overall experience with DECS? [1-10]

29. Would you use DECS in the future? [1-10]

30. In which task did you had more difficulties?

31. Which area of DECS should be improved? ['User Interface', 'Internal Operation', 'Availability', 'Usability', 'None']

32. Based on your experience, did you felt any improvements when using distribution methods for Evolutionary Algorithms? ['Yes', 'No']

33. How can you rate what you've learnt in this activity? [1-10]

D User Experience Data

Results Spreadsheet

Description:

This supplementary Excel spreadsheet contains the collected and processed data from the User Experience Study survey. Each sheet corresponds to a specific section of the survey. The final sheet, titled "Overall_Task_Analysis", provides a continuous analysis of all tasks.

Filename:

`DECS_User_Experience_Results.xlsx`

File Path:

`\BA_Digital_Appendix_Bruno_Guiomar\DECS_Evaluation`
`\User_Experience_Study\`

E Source Code

DECS
Distributed Evolutionary Computing System

E.1 DECS

Description:

This supplementary folder contains the coordinator component source code of DECS, the proposed system in this thesis. Additionally, the JavaDocs documentation source code, respective license and introductory instructions are provided.

Folder Path:

`\BA_Digital_Appendix_Bruno_Guiomar\DECS_Source_Code\DECS`

Repository:

`https://github.com/BrunoPss/DECS`

JavaDocs:

`https://brunopss.github.io/DECS/`

E.2 DECS-Slave

Description:

This supplementary folder contains the Slave component source code of DECS. Additionally, the JavaDocs documentation source code, respective license and introductory instructions are provided.

Folder Path:

`\BA_Digital_Appendix_Bruno_Guiomar\DECS_Source_Code\DECS_Slave`

Repository:

`https://github.com/BrunoPss/DECS-Slave`

JavaDocs:

`https://brunopss.github.io/DECS-Slave/`

Erklärung zur selbständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original