NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

AFONSO VICENTE MARTINS BOUCHO

BSc in Computer Science

CONSTRAINT REASONING FOR RESOURCE OPTIMIZATION IN HETEROGENEOUS CLOUD/EDGE INFRASTRUCTURES

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
April, 2024

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

CONSTRAINT REASONING FOR RESOURCE OPTIMIZATION IN HETEROGENEOUS CLOUD/EDGE INFRASTRUCTURES

AFONSO VICENTE MARTINS BOUCHO

BSc in Computer Science

Adviser: Jorge Carlos Ferreira Rodrigues da Cruz
Assistant Professor, NOVA University Lisbon

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
April, 2024

Constraint reasoning for resource optimization in heterogeneous Cloud/Edge infrastructures

Copyright © Afonso Vicente Martins Boucho, NOVA School of Science and Technology, NOVA University Lisbon.
The NOVA School of Science and Technology and the NOVA University Lisbon have the right, perpetual and without geographical boundaries, to file and publish this dissertation through printed copies reproduced on paper or on digital form, or by any other means known or that may be invented, and to disseminate through scientific repositories and admit its copying and distribution for non-commercial, educational or research purposes, as long as credit is given to the author and editor.

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v6.10.17) [26].

Acknowledgements

I would like to thank my advisor, Professor Jorge Cruz, for all the help and availability during the development of this dissertation. I also want to thank NOVA School of Science and Technology for providing the resources and knowledge throughout the course. Finally, a special thanks to my family and friends for all the support because without them I would not be able to be in this position.

Abstract

The increase of connected devices with different hardware constraints and the growth of different network access systems contribute to a highly heterogeneous environment inside our cloud/edge infrastructures. With all this diversity, it is important to have a cloud/edge environment that can efficiently use resources and optimize the flow of requests to extract the best performance out of each operation or request made by users.

Using constraint reasoning, we can mitigate the problem of heterogeneity in cloud/edge infrastructures by optimizing the resource usage inside the system. Furthermore, it involves using algorithms and tools to analyze the constraints and limitations of the resources available in the system and find the best ways to allocate and use them to achieve specific goals.

The constraint reasoning solution can only be implemented in this problem by understanding key components such as cloud/edge infrastructures, queuing theory and microservices. Moreover, this solution is expected to bring one or more ways to optimize the resources in each computational node in the system.

Keywords: Constraint reasoning, Cloud/Edge computing, Queuing Theory, Microservices

Resumo

O aumento de dispositivos conectados com diferentes restrições de hardware e o crescimento de diferentes sistemas de acesso à rede contribuem para um ambiente heterogêneo dentro das infraestruturas de *cloud/edge*. Com toda esta diversidade é importante ter um ambiente *cloud/edge* que consiga otimizar o fluxo de pedidos e utilizar de forma eficiente os recursos para extrair o melhor desempenho de cada operação ou pedido feito pelos utilizadores.

A utilização de programação com restrições permite mitigar o problema da heterogeneidade nas infraestruturas de *cloud/edge*. O uso de programação com restrições neste problema tem como finalidade encontrar as melhores formas de alocar esses recursos através da análise e aplicação de algoritmos para formular restrições e limitações dos recursos disponíveis.

A partir do entendimento de componentes-chave, como infraestruturas de *cloud/edge*, teoria de filas de espera e microserviços é possivel implementar uma solução que incorpore programação com restrições. Além disso, a solução proposta visa trazer uma ou mais formas de otimizar os recursos em cada nó computacional do sistema.

Palavras-chave: Programação com restrições, Computação na Cloud/Edge, Filas de Espera, Microserviços

Contents

List of Figures viii

List of Listings ix

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Problem . . . 1
- 1.3 Approach and Contributions . . . 2
- 1.4 Document Structure . . . 3

2 Related Work 4
- 2.1 Cloud/Edge Computing . . . 4
  - 2.1.1 Cloudlets . . . 5
  - 2.1.2 Fog Computing . . . 5
  - 2.1.3 Importance . . . 7
  - 2.1.4 Computational Nodes . . . 7
  - 2.1.5 Limitations . . . 8
- 2.2 Microservices . . . 8
  - 2.2.1 Importance . . . 9
  - 2.2.2 Features . . . 10
  - 2.2.3 Limitations . . . 10
- 2.3 Constraint Programming . . . 11
  - 2.3.1 Constraint Propagation . . . 13
  - 2.3.2 Libraries . . . 14
  - 2.3.3 Limitations . . . 14
- 2.4 Queuing Theory . . . 15
  - 2.4.1 M/M/s Model . . . 17

2.4.2 Jackson Networks 19

3 Model Design and Implementation **21**
3.1 Problem 21
3.1.1 Simple Example 22
3.2 Constraint Model 24
3.2.1 Constants 25
3.2.2 Variables 26
3.2.3 Constraints 28
3.2.4 Constraint Reasoning 32
3.3 Implementation 34
3.3.1 Constants and Variables 34
3.3.2 Constraints 34
3.3.3 Constraint Solving 37
3.3.4 Solutions 39

4 Evaluation and Testing **42**
4.1 First Stage 43
4.1.1 Results 45
4.2 Second Stage 46
4.2.1 Results 47
4.3 Real-Life Scenario 52

5 Conclusions **56**
5.1 Future Work and System Limitations 57

Bibliography **58**

Appendices

A Code **62**

List of Figures

2.1 Edge computing deployment scenarios [14] 6
2.2 Microservices Architecture [40] 9
2.3 Basics of a queue ([23] ch.17, figure 17.1) 15
2.4 Rate Diagram of Birth and Death Processes ([23] ch.17, figure 17.4) . 16
2.5 Rate Diagram of M/M/s Model ([23] ch.17, figure 17.5) 17
2.6 Jackson Network Diagram [1] 20

3.1 Network graph 24
3.2 Minimal Cost Solution Diagram 41

4.1 "Tree"Diagram used in the first stage of tests 44
4.2 First Stage Results 45
4.3 Second Stage Control Case Results with different Optimizations . . . 48
4.4 Execution times of 50 different feasible configurations with different number of nodes, clients and microservices for Cost Minimization . . 49
4.5 Execution times of 50 different feasible configurations with different number of nodes, clients and microservices for Total Service Time Minimization 50
4.6 Execution times of 50 different feasible configurations with different number of nodes, clients and microservices for Optimal Pareto 51
4.7 Solution graph of the final scenario with Cost Minimization 53
4.8 Solution graph of the final scenario with Total Service Time Minimization 53
4.9 Solution graph of the final scenario with Optimal Pareto (solution 2) 54
4.10 Solution graph of the final scenario with Cost Minimization (4 nodes activated) 55

List of Listings

3.1 Allocation Constraint (3.4) 35
3.2 Cost Constraint (3.5) 35
3.3 Time Constraint (3.20) 36
A.1 Variables and Constants 62
A.2 Auxiliary Variables 63
A.3 Allocation Constraints 65
A.4 Cost Constraints 65
A.5 Flow Constraints 65
A.6 Queue Constraints 66
A.7 Time Constraints 69

1 Introduction

1.1 Motivation

Upon the arrival of numerous types of devices, architectures, infrastructures, and communication technologies, cloud/edge environments adapted themselves and made improvements to support all of the heterogeneity. The increased support of the upcoming technologies provided users with a better experience overall while using this kind of infrastructure.

The study and the embracement of new technologies to keep up with the rise of heterogeneity is still an undergoing theme. However, there are several reasons why this will not stop; security, scalability, fault tolerance and resource management are only a few of the reasons that are always trying to be optimized to keep up with the latest technology features.

The use of microservices plays a huge role in helping some of the heterogeneity in cloud/edge infrastructures. Since every microservice is independent of one another, this facilitates the microservices nodes to only work with specific frameworks or architectures, avoiding the problem of deploying resources that are not needed in each node.

Since cloud providers are the ones who offer the resources and take into account each aspect mentioned before, they need to make the right choices and adapt to every upcoming technology in a sustainable way.

1.2 Problem

One key objective of studying heterogeneous cloud/edge infrastructures is to improve resource utilization by optimizing the allocation of computing resources such as CPU, memory, and storage. This may involve improving the algorithms used for resource scheduling and identifying new techniques for optimizing

resource allocation. In addition, reducing latency, or the amount of time it takes for data to be transmitted from one location to another, is another important objective.

Although these highly distributed systems composed of various nodes and servers provide a vast amount of resources that users can use on demand, managing such a complex system is not easy. The allocation of resources needs to be optimized to ensure that the users receive the best possible service.

One approach to addressing heterogeneous/cloud infrastructures' challenges is using microservices. Microservices are a set of small, independent, and loosely coupled services that work together to form a more extensive application. In heterogeneous cloud/edge infrastructures, microservices can be used to break down a large application into smaller, more manageable components. This makes it easier to optimize the allocation of resources and improve the system's overall performance.

Another challenge that needs to be addressed is optimizing the process of handling requests in a heterogeneous cloud/edge computing environment. This involves finding the most efficient and effective ways to manage the resources and services in the network so that requests can be handled promptly and reasonably. In order to achieve this, it is necessary to consider the various factors that impact the performance of the network, such as the different types of nodes and the microservices deployed on them. Moreover, it is crucial to develop a robust and flexible framework that can accommodate the changing demands and constraints of the environment and provide a consistent and reliable solution for request handling.

1.3 Approach and Contributions

Constraint reasoning is a computational technique that can be applied to solve complex problems, such as resource allocation and managing and handling requests in a diverse cloud and edge computing environment. In this context, constraint reasoning involves mathematical models representing the relationships between the different system elements inside a cloud/edge scenario.

The constraint reasoning approach in this dissertation focuses on resource allocation and request handling in cloud and edge computing environments and consists of two main steps. First, it builds a model with all the relationships between components inside the cloud/edge environment using constraints, which describe the limitations and relationships between each system element. Secondly, it uses the model's information to search for a consistent configuration that meets the

constraints. This process involves a combination of constraint propagation and search, where the model explores different system configurations and evaluates their feasibility against the constraints. The final configuration produced by the constraint reasoning model represents a set of consistent and optimized resource allocations and settings for managing and handling requests in the cloud/edge environment.

The approach developed during this dissertation has two main contributions:

- **Applying constraint reasoning to cloud/edge environments:** The main objective of this project was to solve the problem described above using constraint reasoning and apply its capabilities to solve major problems inside this environment, in particular, optimizing resource allocation and the flow of requests. Consequently, improving decision-making capabilities, reducing waste, and minimizing the cost of processing requests in cloud/edge environments.

- **Building an agile solution:** The constraint model that propels the whole constraint reasoning needs to provide a flexible solution to keep up with the heterogeneity of the diverse and rapidly evolving computing landscape present in cloud/edge scenarios, enabling further constraints and optimizations to be introduced, taking into account the characteristics of all components present in the system.

1.4 Document Structure

The document is organized into four more chapters. Chapter 2 covers the relevant related work by explaining cloud/edge environments, microservices, constraint programming and queuing theory. Then, Chapter 3 details the constraint model created and its implementation with constraint programming, followed by Chapter 4, which provides all the testing and evaluation to which the model created in Chapter 3 was submitted. Finally, Chapter 5 outlines the conclusions of this work and lists areas where the model could improve and the possibility for future work.

2

Related Work

2.1 Cloud/Edge Computing

Cloud computing can be explained as a method to provide computer system resources on-demand to a user. The user does not have a direct active management role, meaning that the cloud computing provider is the one that takes care of that. This computational paradigm is becoming increasingly popular among users because it follows a pay-as-you-go model; this means that you are only getting and paying for the computing power, data storage, and other resources you really need[30].

With the introduction of cloud computing and the management of the servers that are required to be able to achieve what users demand, cloud providers, in order to maintain consistency, low latency, and performance, continued to explore this whole idea further until reaching what we called nowadays as edge computing. Edge computing could be an extension of cloud computing and can be viewed as a service provided by the cloud provider that holds system resources. This idea has in mind the user; instead of demanding system resources directly from the cloud server, the server creates an edge node near the client with the needed system resources. The client-side (user requesting system resources) will be able to have whatever kind of resource in a faster manner because its request is being sent to the edge node near the client and not directly to the cloud server. Moreover, this approach allows for faster data processing and better response times, as the edge node can store data locally and process it quickly, reducing the need to send data back and forth between the client and the cloud server. Additionally, edge computing also enables the deployment of local services and applications, which can help reduce network congestion and improve overall network performance[36, 35].

However, one of the potential downsides of edge computing is that it may result

in a fragmented network, with different edge nodes having different configurations and resources. This can lead to compatibility and integration issues, as well as increased complexity in terms of management and maintenance. In addition, the cost of deploying and maintaining edge nodes can also be higher compared to traditional cloud computing, especially for small and medium-sized businesses that do not have the resources to support the additional infrastructure.

Despite these challenges, the benefits of edge computing are significant, and its use is likely to grow as more and more organizations embrace the IoT and big data and seek to find ways to process and analyse data closer to the source[36]. Overall, edge computing is a promising solution for organizations that need to balance the benefits of cloud computing with the need for low latency, fast processing, and improved data privacy and security.

After the birth of edge computing, several other architectures were born, namely cloudlets and fog computing.

2.1.1 Cloudlets

Cloudlets are an innovative concept in cloud computing that brings the power of cloud computing to the edge of the network, close to mobile devices. They are located only a single hop away from mobile devices and are designed to provide high-performance computing capabilities. A cloudlet can be considered a computer or a cluster of computers, where all are virtualised to offer maximum flexibility and versatility [16].

Operating on high-bandwidth networks and disconnected from the cloud servers, cloudlets offer a unique combination of low latency and high-performance computing capabilities. This makes them ideal for applications that require real-time data processing and fast response times. The virtualisation of cloudlets also allows for scalability and versatility, making it possible to quickly and easily scale up or down depending on the application's needs.

By providing a bridge between the cloud and the edge, cloudlets are changing the way we think about cloud computing. They offer a more flexible, scalable, and versatile solution that is designed to meet the needs of modern applications and devices.

2.1.2 Fog Computing

Fog computing is another architecture that deserves recognition in the world of cloud computing and edge computing. The idea behind fog computing emerged as

a response to the increasing complexity and heterogeneity of the edge computing environment. With the rise of the Internet of Things (IoT) and Big Data, the need for a more efficient and scalable solution to handle the massive amount of data generated by these devices has become increasingly important. Aiming to address these challenges by creating an abstract layer where a large number of diverse devices can communicate and cooperate with each other, fog computing brings computing power closer to the edge, where it is needed most, rather than relying solely on the cloud. This helps to reduce latency, improve responsiveness, and increase overall system efficiency[29, 10].

The design complements cloud computing by serving as an intermediary between edge devices and the cloud. It acts as a bridge, facilitating communication and data exchange between the edge devices and the cloud. This helps to balance the load on the cloud, reduce network congestion, and ensure the efficient use of resources.

In summary, fog computing is a crucial architecture that provides a valuable solution to the challenges faced by the edge computing environment. It is an innovative concept that brings computing power closer to the edge and provides a more efficient and scalable solution to handle the growing demands of IoT and Big Data.

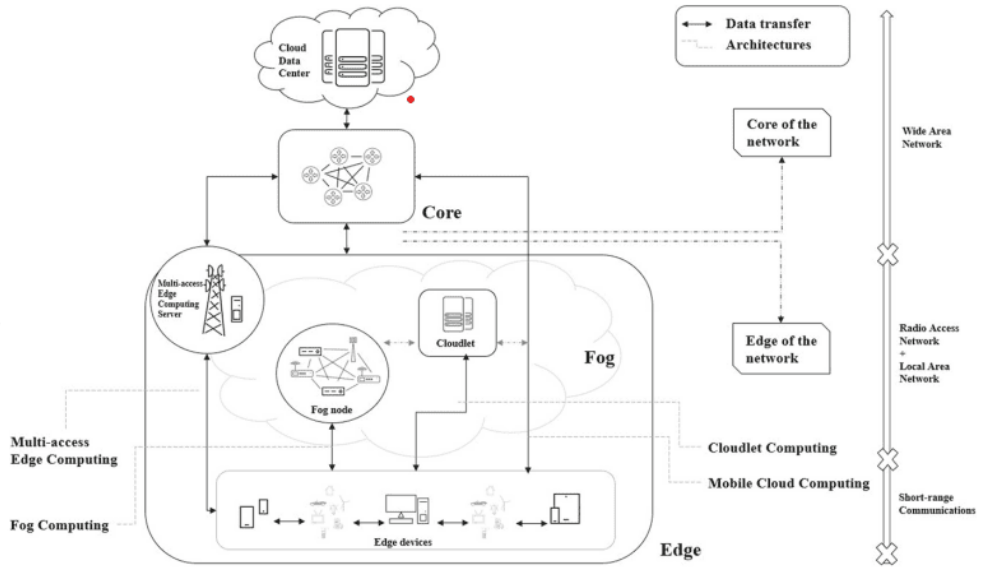

Figure 2.1: Edge computing deployment scenarios [14]

2.1.3 Importance

In general, cloud computing is a service that openly offers resources that an average user and client sometimes cannot afford to allow organizations to easily access and scale computing resources, such as storage and processing power, on-demand over the internet, by reducing costs, improving agility, and enabling organizations to focus on their core competencies.

Edge computing extends the cloud computing paradigm and has the expertise to allow data to be processed closer to the source of data rather than sending all data to centralised data centers for processing. This can reduce latency, improve response time, and increase the efficiency of data transfer. As a result, edge computing is particularly useful for IoT devices and other connected devices that generate large amounts of data and for applications that require real-time processing [36].

Together, cloud and edge computing can provide a powerful and flexible solution. Cloud computing can provide centralised resources and scalability, while edge computing can provide the low latency and real-time processing capabilities needed for specific use cases.

2.1.4 Computational Nodes

There are a lot of relevant features in both computing paradigms infrastructures, but the ones that we will be focusing on in this work will be the computational nodes and the interaction between them.

In cloud computing, computational nodes are represented by servers or virtual machines that are used to perform a wide range of computing tasks, such as running applications, storing and processing data, and handling network traffic. They are typically located in data centers and are connected to the internet, allowing users to access the resources and services they provide from anywhere in the world. In addition, these computational nodes are often virtualised, which means they are not bound to any physical machine, offering greater flexibility and the ability to move computational tasks around between other nodes [5].

The design around these nodes makes them easy to scale up or down to meet the user's needs; everything around the management side of things the cloud provider does it. In case of any type of failure, these nodes can quickly adapt and are configured to be fault-tolerant by continuing to operate even if one node fails.

In the edge paradigm, the computational nodes are specialised devices that are designed to perform specific tasks, such as data processing, storage, and networking. They are located at the edge of the network, near users and devices,

and are used to perform tasks that require low latency and high bandwidth. These nodes are typically connected to the cloud through a high-bandwidth, low-latency network connection and play a crucial role in supporting the processing and analysis of data at the edge of the network, helping to improve the performance and responsiveness of applications and services [4].

Interaction between nodes is usually made through the internet. The internet acts as the communication channel through which the nodes can exchange data and resources, and it allows for the nodes to be accessed and controlled remotely. The use o the internet is beneficial to allow wider geographic flexibility in order to reduce costs and improve efficiency. Furthermore, a load balancer is often used to route requests to the appropriate node to ensure that the workload is distributed evenly among the nodes.

2.1.5 Limitations

Although the number of similarities between cloud and edge computing is significant, they are quite different in some aspects, and they have their own limitations.

Cloud computing servers are located within the internet, so the availability of these servers relies on the distance between the users and them. The larger this distance is, the greater the latency will be. In addition, its centralised infrastructure has issues in terms of location awareness and becomes more vulnerable to integrity/security attacks. At last, the high jitter is also a problem because there is a great variation in delay between each packet sent over the network [3].

On the other hand, edge computing solves some of the limitations but adds other problems by having a limited amount of resources compared to cloud computing, a higher cost of maintenance and management, compatibility and integration issues, and a lower geographic aperture.

2.2 Microservices

Microservices is a design pattern for building service-oriented software systems that utilise small, independent services to perform specific tasks. This architectural style enables a software system to be broken down into smaller, more manageable components, with each component communicating with other components over a network. This enables the creation of complex software systems that are still easy to maintain and update.

The adoption of microservices in cloud-hosted environments has been driven by the benefits that it provides. The deployment and updates of microservices

are much easier, as each service is small and independent. Also, providing better fault isolation, as failures in one service will not necessarily bring down the entire system. The loose coupling between services also allows for greater flexibility and scalability, enabling the dynamic detection and binding of devices and services on a network[6, 15].

By breaking down monolithic software systems into smaller, independent components, the microservices architecture enables developers to develop, deploy, and update individual services faster and more efficiently. As a result, microservices have become an increasingly popular design pattern for building software systems in cloud-hosted environments[39].

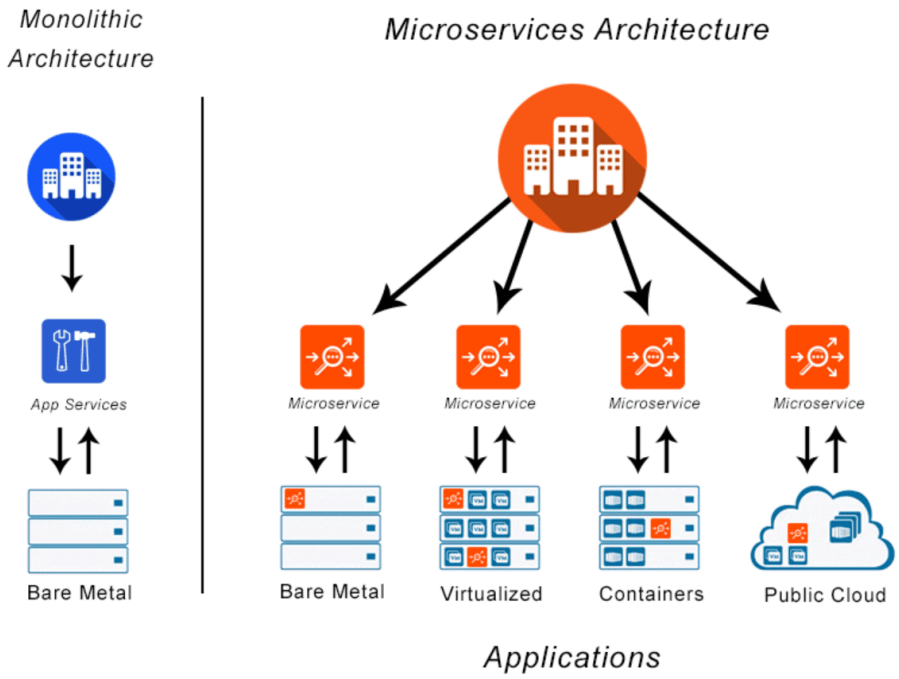

Figure 2.2: Microservices Architecture [40]

2.2.1 Importance

As mentioned before, the massive role of breaking down monolithic architectures became a necessity for cloud providers. Moreover, the whole structure for the software deployment as a set of microservices allows the could providers to offer higher scalability and a more optimised usage of cloud resources; this enables a more dynamic way of rearranging the software in order to meet the user's needs [33].

In contrast to monolithic architectures, the microservices partitioning model grants the segmentation of application data and avoids having a single monolithic

data storage. The use of this partitioning model becomes more relevant when we start performing operations like storing and accessing large amounts of data [39].

Lastly, the relevance of using microservices in edge nodes is quite important in this work and can be viewed as the embodiment of each microservices in the edge nodes working independently from each other in order to fulfil a service communicating through a network with each other.

2.2.2 Features

We can see why users found this so appealing by diving a bit further into this architectural model [32]. The design follows a loosely coupled style, allowing the services to be developed, deployed, and scaled independently, grating each service with minimal dependencies on other services. Due to its capability of supporting different languages, frameworks, and databases, microservices can use the best piece of technology for each service, and by having each of them responsible for a specific task opens the door to a more modular and viable system.

Additionally, the availability makes this architecture function during failure events or other disruptions achieved through the use of redundancy and fail-over mechanisms. Moreover, the stateless feature does not save any information on previous requests making them more scalable and fault-tolerant because each request is handled independently.

2.2.3 Limitations

Like every single architecture, there are drawbacks, and microservices are no exception to it. Starting off with the complexity of the system requires managing multiple services and interactions between them, and the deployment is often more difficult than monolithic architectures. Network latency, data consistency, and service discovery can be quite challenging when we talk about distributed systems. Additionally, since you have a sharding style architecture, this will require the testing of all individual services making it more time-consuming and complex when compared to monolithic architectures. In case there is an increase in dependencies between services, it is harder to upgrade or replace them with other individual services[15].

Another key aspect of microservices is the huge pursuit of equilibrium between security and performance; this can be viewed as a limitation because in order to get the performance, we need to sacrifice security and vice-versa. Security is a vital point in microservices, and if someone gets control of one component in this

infrastructure, it can ruin the whole thing [37, 24]. The risk of having a threat in the system would be the propagation of attacks through microservices dependencies; most of these attacks have in mind the origin of the data and how trustworthy it is [12].

2.3 Constraint Programming

Constraint programming is a type of programming that aims to solve problems by describing a set of constraints and then using these constraints to search for a solution.

Providing a rich declarative language that is ideal for describing combinatorial problems, constraint programming offers a variety of algorithms and techniques for solving these problems automatically. Making it an effective tool for solving complex optimization problems, scheduling problems, and many other types of issues that are difficult to solve using traditional programming techniques. Furthermore, constraint programming is highly adaptable and can be used in a variety of different domains, from scheduling and resource allocation to decision-making and planning problems [28, 2, 17, 38].

The use of constraint programming can only take action when we have a clear model of our system, so it is important first to define our Constraint Satisfaction Problem.

A Constraint Satisfaction Problem (CSP) is revolved around four main notions constraints, variables, domains and solutions. The first three notions are often identified as a triplet <X, D, C>, where X is the set of variables present in our problem, D is the domains that hold the values which variables can be instantiated with, and C holds the set of constraints that the problem must satisfy. The domains of the variables may vary; they can either be discrete or continuous, depending on the variable modelled. At last, solutions are values assigned to all variables satisfying all constraints in C.

Constraint solvers employed in this type of programming are used to automatically search for solutions to problems that can be modelled as CSPs. In addition to finding solutions, these solvers can also be used to optimise solutions based on one or more criteria, find all possible solutions, or even prioritise constraints using preferences. The solvers can operate using systematic methods, such as backtracking or branch and prune algorithms, or through local search methods that may not be comprehensive.

For finite domains [22], branching creates new subproblems from an existing

problem by fixing the values of the variables in the subdomains and then recursively applying the same branching process to the remaining variables. Interleaved with pruning eliminates subproblems that are no longer viable by removing values that are inconsistent with the constraints in the problem. Backtracking is often used in finite domains. After checking each assignment's consistency with the constraints as it goes along, if an assignment leads to a conflict with a constraint, the solver backtracks to the previous assignment and tries a different value. This process is repeated until either a solution is found or it is determined that no solution exists. The search progresses by branching at each node to represent the different possible assignments to the next variable [20, 25].

In continuous domains [13], since the variable's domain is a Cartesian product of intervals $I_1 \times ... \times I_n$ (a box), branching splits the box into sub-boxes (usually two). After every branch, a pruning step is applied to eliminate boxes or reduce them without removing any solutions.

Pruning is achieved through algorithms that apply constraint propagation and consistency techniques. In the case of continuous domains, consistency techniques are based on interval analysis [31], like hull-consistency and box-consistency [8]. On the side of finite domains, arc consistency or node consistency techniques are the most common [27].

Efficiency is always a crucial aspect of the search process methods, so heuristics are regarded as one of the focal points to be aware of before applying any kind of search. Heuristics are techniques or methods that help guide the search process by prioritising certain branches of the solution space. Various types of heuristics can be used in constraint solvers, such as value ordering heuristics, variable ordering heuristics, and constraint ordering heuristics [9, 19]. Value ordering heuristics help guide the search process by prioritising certain values for a given variable. Finally, constraint ordering heuristics help guide the search process by prioritising certain constraints over others. A common approach in selecting the order of variables in constraint solvers is prioritising the variable with the smallest number of potential values in its domain. On the other hand, most value ordering heuristics are designed based on selecting the value with the highest probability of success or forming a solution.

Besides heuristics, randomization and restart strategies [21] are other ways of improving efficiency. Randomization and restart strategies are alternative approaches to heuristics in search methods for constraint solvers. In randomization, the algorithm makes random choices in selecting the next variable or value to assign, making it less likely to get stuck. The idea behind restart strategies is that starting over from a different initial state can help avoid getting stuck in the same

place again if a search becomes stuck. In both cases, these strategies aim to escape from being trapped in suboptimal solutions by providing a way to randomly explore different parts of the solution space. These methods can be combined with heuristics to further improve the performance of the search process in constraint solvers.

Considering the role solvers play in the whole constraint programming structure, the need for even better performance and extended robustness, constraint propagation came as a complement to enhance solvers' search methods.

2.3.1 Constraint Propagation

Constraint propagation reduces the search space of a constraint satisfaction problem (CSP) from the existing constraints. In constraint propagation, information from one constraint is used to reduce the possible solutions for another constraint. The process of reducing the search space continues until a solution is found or it is determined that no solution exists. The goal of constraint propagation is to simplify the problem, reduce the number of possible solutions, and increase the efficiency of the search for a solution [18, 27]. Local consistency and global constraints help constraint propagation to be more efficient and consume less processing time.

Local consistency in constraint programming refers to the process of ensuring that a given set of variables, as well as their respective values, satisfy the constraints that have been modelled in a problem. This involves checking each variable's value with its associated constraint and determining if the other variables in the constraint support it. In other words, local consistency aims to enforce a relationship between the variables and their values based on each problem's constraints [27]. The local consistency process involves pruning to remove and update variable values that are found to be inconsistent with the constraints. By limiting the possible values of each variable, the search space can be reduced, making it easier to find a solution.

Global constraints do not have a clear description because they can vary depending on the problem's variables. Generally, these types of constraints are defined on a set of variables and are efficient and specialised for the pruning process. Instead of expressing each relation separately in the system, some relations can be grouped into a single global constraint [7]. A typical application of a global constraint is the *all-different* constraint [34]. This constraint forces the values from every pair of variables to be different, granting a tighter and more efficient pruning process because it is done on a set of variables rather than individually.

2.3.2 Libraries

Constraint libraries provide a set of tools, APIs, and modelling languages that abstract the underlying complexities of solvers, enabling users to focus on problem modelling and solution strategies. There are a lot of constraint programming libraries, but we will only delve into the two most commonly used.

2.3.2.1 MiniZinc

MiniZinc is a sophisticated and high-level constraint modelling language designed to address the challenges posed by Constraint Satisfaction Problems (CSPs) and Constraint Optimization Problems (COPs). Unlike many programming languages, MiniZinc is tailored explicitly for expressing and solving constraint-based problems, offering higher abstraction and ease of use.

2.3.2.2 CHOCO Solver

CHOCO Solver is a robust and versatile Java library specifically crafted to address Constraint Satisfaction Problems (CSPs) and Constraint Optimization Problems (COPs). As an open-source tool, it has gained prominence for effectively tackling combinatorial problems by providing a robust set of features for constraint-based modelling and solution strategies. The CHOCO's library allows integration with another constraint library called IBEX, which specializes in continuous variables and mathematical constraints, enhancing the capabilities for solving a wider range of constraint satisfaction and optimization problems and making the ideal library to use during the implementation.

2.3.3 Limitations

Despite having a vast amount of advantages, constraint programming also has its downsides. There are certain types of problems where constraint programming does not stand out and can even be slower than traditional programming. To top all that, problems that do not possess easily stated constraints or a declarative way of describing them are difficult to model in this type of programming.

In summary, constraint programming is a powerful tool for solving certain types of problems, but its use may also have limitations and drawbacks that should be carefully considered in the model formulation.

2.4 Queuing Theory

The analysis of requests and how they are handled are factors in modelling a constraint reasoning solution to this problem, and this is when queuing theory comes in.

Queuing theory is a mathematical study of waiting for lines or queues [23]. It is used to model and analyse the behaviour of systems where customers or requests are queued for service. Furthermore, it provides a framework for understanding the interactions between customers and servers and the impact of various system parameters on performance metrics, such as response time, wait time, and utilisation.

In queuing theory, a queue is represented by a mathematical model that includes the arrival rate of customers, the service rate of servers, and the queue discipline, which determines the order in which customers are served. The mathematical models can be used to calculate key performance metrics such as the expected number of customers in the queue, the expected wait time, and the expected response time.

The structure of a queuing system is basic; there is an input source, a queue, a service mechanism and the served customers. The input source is the number of people arriving in the queue, and the queue is where customers wait patiently for their turn. After waiting, customers enter the service mechanism, where they are served by one or more servers, before leaving the system (figure 2.3).

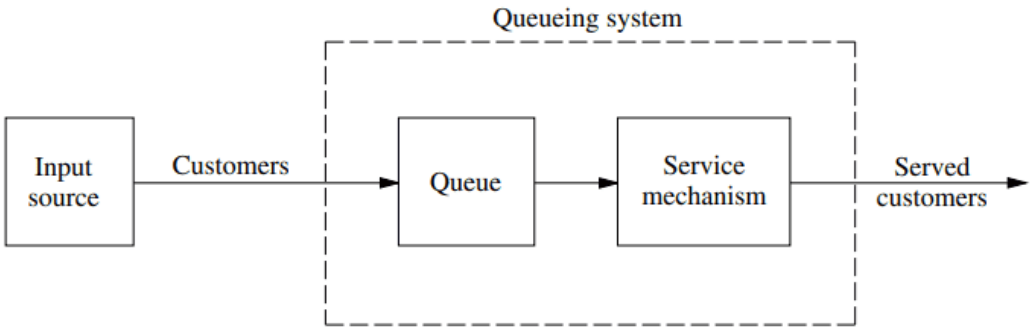

Figure 2.3: Basics of a queue ([23] ch.17, figure 17.1)

Overall, the most rudimentary queuing models follow a birth-and-death process, meaning that the arrival of customers and their departure from the queue is represented by "births"and "deaths,"respectively (figure 2.4).

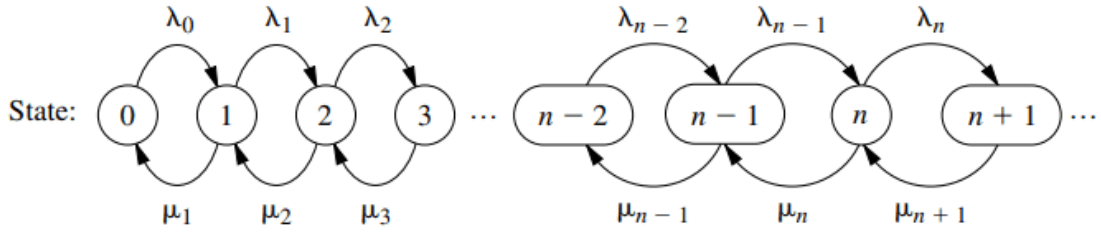

Figure 2.4: Rate Diagram of Birth and Death Processes ([23] ch.17, figure 17.4)

This study allows us to analyse the behaviour of requests in our main problem as they wait in a queue for processing and to determine the optimal utilisation of resources by each node. With this, organizations can make informed decisions about system design, capacity planning, and service level objectives and optimise their systems for improved performance and customer satisfaction.

Here, we can look at the terminology used in Queuing theory [23] (ch.17.2):

- State of the system - number of customers in the queuing system
- Queue Length - number of customers waiting for service to begin, state of the system minus the number of customers being served.
- $N(t)$ - number of customers in the queuing system at time ($t >= 0$)
- $P_n(t)$ - the probability of exactly n customers in the queueing system at time t
- λ - mean arrival rate
- μ - mean service rate
- s - number of servers given by the model
- $\rho = \frac{\lambda}{s\mu}$ - utilization ratio
- P_n - probability of exactly n customers in queueing system
- L - expected number of customers in the queueing system
- L_q - expected queue length, excludes customers being served
- W - waiting time in the system for each individual customer
- W_q - waiting time in queue for each individual customer, excluding service time
- C_n - factors for the birth-and-death process

2.4.1 M/M/s Model

The M/M/s model is a special case of the birth-and-death model [23] (ch.17.6) used in queueing theory to describe the behaviour of a queueing system with multiple servers (figure 2.5).

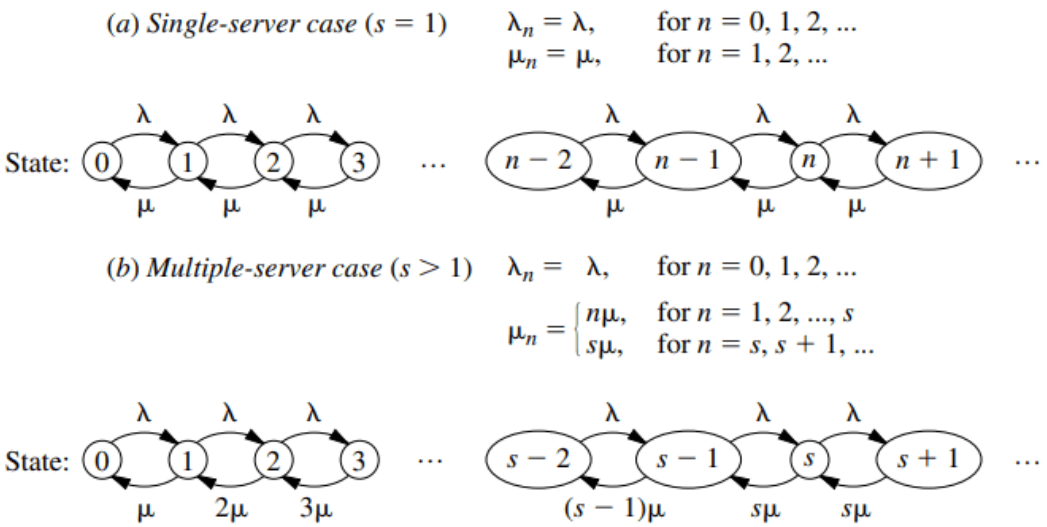

Figure 2.5: Rate Diagram of M/M/s Model ([23] ch.17, figure 17.5)

In this model, it is assumed that the inter-arrival times between customers are exponentially distributed, which means that the time between arrivals of customers is random and follows an exponential distribution. Furthermore, the service times of customers are also assumed to be exponential, meaning that the time it takes for each customer to be served is random and follows an exponential distribution.

The number of customers in the system is defined as the number of customers waiting in line and the number of customers served. The system's state is characterised by the number of customers in the system, and the system's behaviour can be described by the probabilities of being in each possible state.

The M/M/s model can give us information about the expected waiting times used in the modelling of the constraints through these formulas:

- $$C_n = \begin{cases} \frac{(\lambda/\mu)^n}{n!} & n = 1, 2, ..., s \\ \\ \frac{(\lambda/\mu)^s}{s! s^{n-s}} & n = s, s+1, ... \end{cases}$$
- $$P_0 = 1 / \sum_{n=0}^{s-1} \frac{(\lambda/\mu)^n}{n!} + \frac{(\lambda/\mu)^s}{s!} \frac{1}{1 - \lambda/(s\mu)}$$

- $$P_n = \begin{cases} \frac{(\lambda/\mu)^n}{n!} P_0 & 0 \leq n \leq s \\ \\ \frac{(\lambda/\mu)^s}{s! s^{n-s}} P_0 & n \geq s \end{cases}$$
- $L_q = \frac{P_0 (\lambda/\mu)^s}{s!(1-\rho)^2} \rho$
- $W_q = \frac{L_q}{\lambda}$
- $W = W_q + \frac{1}{\mu}$
- $L = \lambda(W_q + \frac{1}{\mu}) = L_q + \frac{\lambda}{\mu}$

2.4.2 Jackson Networks

Jackson networks are an adequate representation of our problem because each facility and its services can be translated as the nodes and microservices operating inside them.

In order to understand the whole concept of Jackson Networks, we first need to look at the principle of infinite queues in series.

Infinite queues in series are present when customers in a system want to perform a fixed sequence of services. In this model, the services performed are represented as service facilities with an infinite queue capacity. Moreover, with a fixed sequence and a group of services, the system becomes a network where each customer follows a series of services.

Although Jackson Networks inherit most of the properties of the infinite queues in series, the main dissimilarities between these two networks are evident. Starting with the non-fixed entry point to the system means that every facility can have customers arriving both from the outside of the system and from other facilities. The other significant difference is the non-fixed sequence that needs to be followed by each individual on the system; this opens the door for customers to jump from facility to facility without a restricted order. Besides these differences, one particular asset stands out from the inheritance from infinite queues in series, the analysis of each service facility as M/M/s model. Measuring performance in each service facility, rather than needing to analyse the whole system to do it, makes room for simplification [11].

In a Jackson network, the servers are connected in a network topology, where customers can enter the network at any of several entrance nodes and leave the network at any of several exit nodes. The customers move through the network and wait in queues at each server.

The Jackson network model is widely used in many applications, such as designing and analysing computer networks, telecommunication networks, transportation systems, and production systems. It provides a powerful tool for understanding and optimising complex systems' performance and helps make informed decisions about system design and resource allocation.

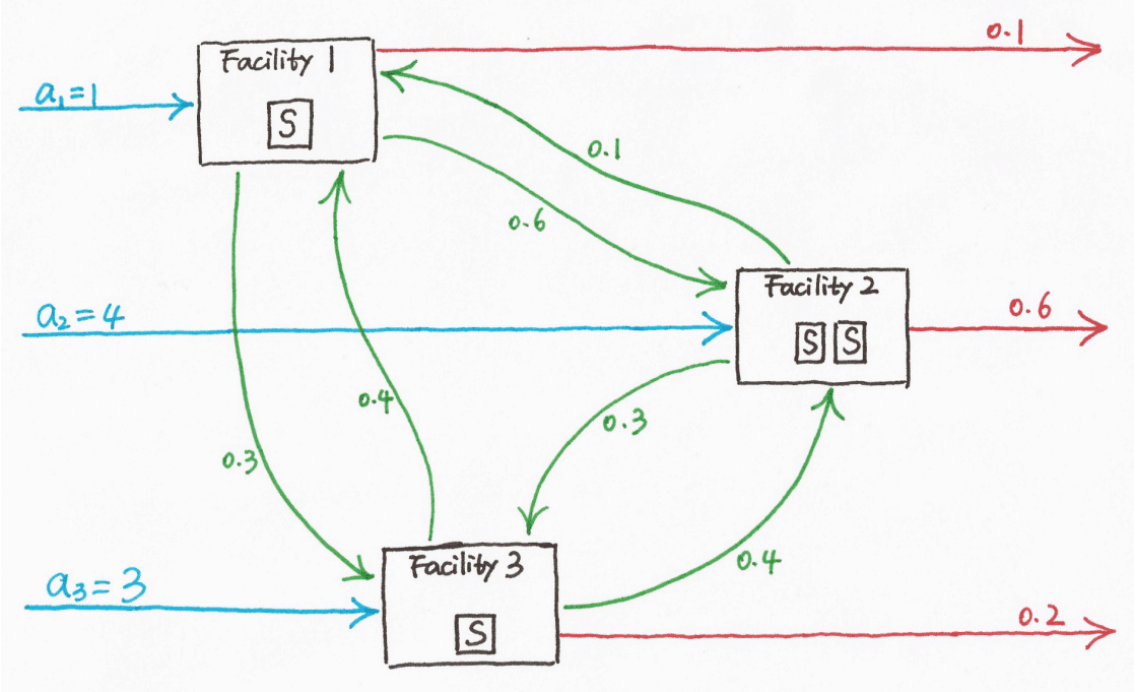

Figure 2.6: Jackson Network Diagram [1]

The image above (figure 2.6) shows a Jackson network with three facilities, each with one or more available services. Each blue arrow represents the mean arrival rate to each facility. Green arrows represent the probability of a customer switching from one facility to another, and the red arrow symbolizes the probability of exiting the system after the service is done on the facility. Furthermore, this image demonstrates how the solution to our problem will work. The facilities will be the edge nodes, the services microservices and the arrows for the requests done in the system.

3

Model Design and Implementation

This chapter describes an overview of the problem that our model is trying to solve and the design and implementation routes taken when developing the solution to meet the objectives mentioned in 1.

3.1 Problem

The cloud/edge environment is modelled as a group of nodes and clients where the clients request the nodes to run an application represented as a sequence of microservices.

Every node in the system has three known specifications: their storage space, the number of cores and an index of their velocity compared to a standard CPU. Additionally, the nodes can communicate with both clients and other nodes; each interaction has a time associated. In the case of node-to-node interactions, this represents the travel time between nodes, and in the case of a client-to-node interaction, it is the latency between client and node.

Clients are responsible for making the requests to run the application; each client has a mean request rate attached to it, which is an expected number of requests per unit of time.

The application the clients request is divided into a sequence of small partitions called microservices that must be handled sequentially by the nodes: the next microservice is only processed after the previous is finished. Individually, the microservices are deployed inside the nodes, that will process the incoming requests. Depending on the demand of requests, these microservices may have one or multiple replicas inside each node. Furthermore, each microservice represents

a part of the application, so they have a memory space needed to be stored inside a node limiting the number of replicas that can be stored in each node.

Every node activated and microservice deployed has a cost, which dictates the resources spent inside the cloud/edge environment. This work aims to efficiently minimize and manage the resources spent when the clients want to run the application. This will be modeled as a constraint optimization problem to search configurations that satisfy the constraints from the available resources and optimize costs and service time.

3.1.1 Simple Example

To illustrate the kind of problems we are trying to model, a very small example is presented with 3 processing nodes and 2 clients that request to run an application partitioned into 2 sequential microservices.

The information about the available nodes is shown in table 3.1. In this example, node #1 has 10 cores, 10 units of storage space and a processing velocity similar to a standard CPU. Both nodes #2 and #3 are twice faster but with less cores (7 and 5 respectively). Node #3 has a smaller storage memory capacity (8 units).

	Node #1	Node #2	Node #3
Cores	10	7	5
Storage	10	10	8
Velocity	1	2	2

Table 3.1: Node resources

Table 3.2 shows the information with respect to the client requests. Here, clients #1 and #2 are considered, which in fact represent 2 groups of clients that make requests from identical geographical regions. The number of requests a group demands to the system is the sum of requests made from the set of clients in that same group. Whereas client #1 has an expected number of 3 requests per unit of time, client #2 mean request rate is smaller (2 per unit of time).

	Client #1	Client #2
Requests	3	2

Table 3.2: Client requests

In the example, the application requested by the clients is partitioned into 2 sequential microservices, #1 and #2. Both, memory space and service time, must be specified for each microservice as illustrated in table 3.3. Microservice #1 is smaller, occupying a single unit of storage space, and faster to process (it takes 2

units of time if processed by a standard CPU). Microservice #2 is larger (2 storage units) and slower to process (3 units of time in a standard CPU).

	Microservice #1	Microservice #2
Memory	1	2
Time	2	3

Table 3.3: Microservice attributes

The costs associated with the usage of the available resources and deployment of the microservice replicas must be specified as illustrated in table 3.4. Each node has an activation cost triggered when one or more microservice replicas are deployed. In the example, the usage of node #2 is more costly (2 cost units) than the usage of nodes #1 or #3 (1 cost unit). Additionally, each node has a cost per deployed replica of each microservice. In our example, for every node, each deployed replica of each microservice has 1 unit cost, except in case of deployment of replicas of microservice #1 in node #3 (which is 2 cost units per replica).

	Node #1	Node #2	Node #3
Activation cost	1	2	1
Microservice #1	1	1	2
Microservice #2	1	1	1

Table 3.4: Resource costs

To complete the specification of the problem is necessary to provide information about the network travel times between nodes and clients as illustrated in figure 3.1. In the graph the vertices are nodes (represented as rounded squares) and clients (represented as circles). Edges are labelled with the travel times between the connected vertices.

All such information can be summarised in two tables. The travel times table 3.5 specify the distance, in units of time, between any pair of nodes. The latency table 3.6 specify the distance, in units of time, between each client and each node.

	Node #1	Node #2	Node #3
Node #1	0	2.24	2.24
Node #2	2.24	0	1.41
Node #3	2.24	1.41	0

Table 3.5: Travel times

	Node #1	Node #2	Node #3
Client #1	0.71	1.58	1.58
Client #2	1.8	0.5	1.12

Table 3.6: Latency

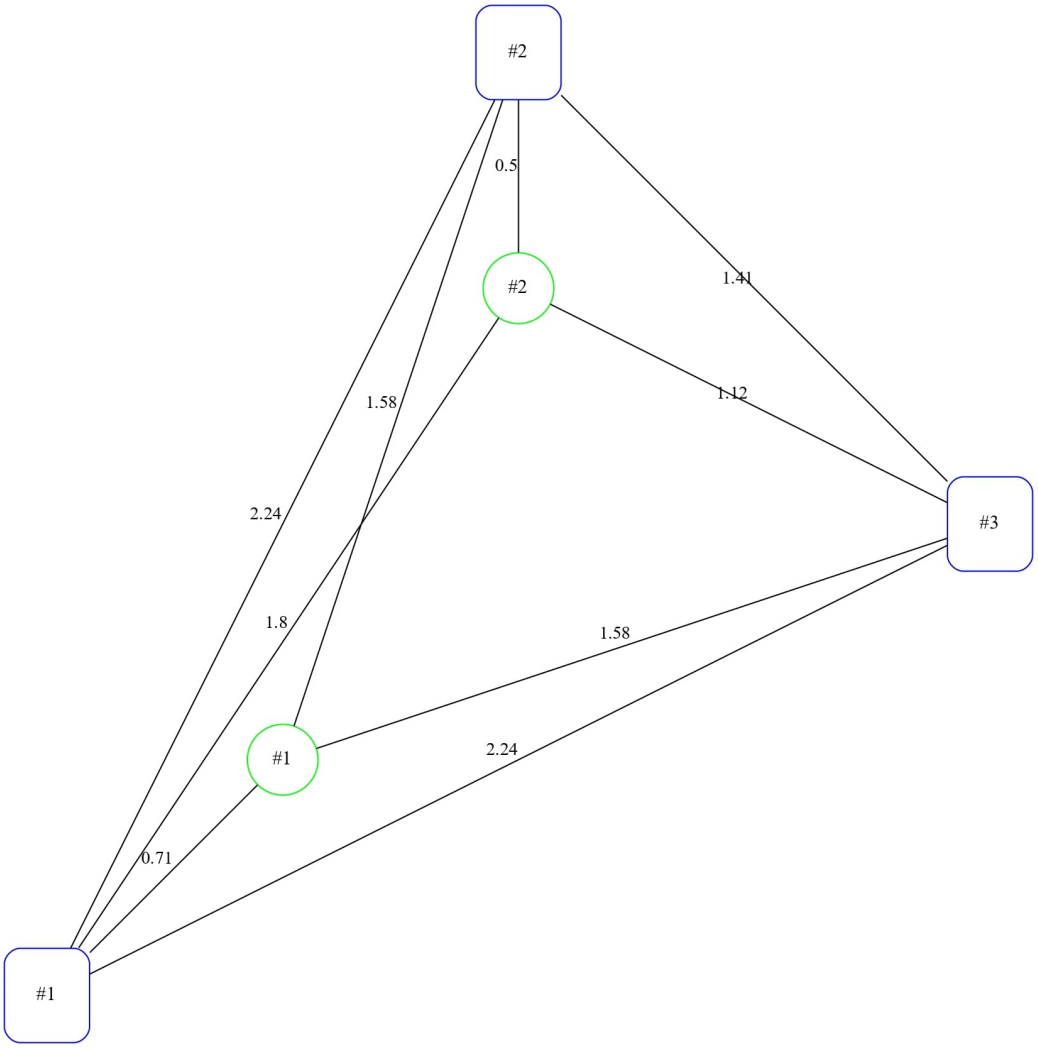

Figure 3.1: Network graph

3.2 Constraint Model

The fundamental concept of constraint programming is that the constraints are specified by the user, and a general purpose constraint solver is used to resolve them. Constraints are relationships, and a constraint model specifies which relationships should exist between the problem variables and constants.

To represent our cloud/edge problem we must identify its constants and

variables and establish a set of constraints to model their relationships.

3.2.1 Constants

The constants of the constraint model capture the input specifications of the cloud/edge environment. These include information about nodes, clients, microservices, costs and times.

Nodes

$\#i$ is the number of available processing nodes

For each node i: $(1 \leq i \leq \#i)$

- n_i is the number of cores
- s_i is the storage space
- v_i is the processing velocity compared with a standard CPU

Clients

$\#j$ is the number of clients

For each client j: $(1 \leq j \leq \#j)$

- λ_j is the expected number of requests per unit time

Microservices

$\#k$ is the number of microservices of the application

For each microservice k: $(1 \leq k \leq \#k)$

- m_k is the memory space needed to store one replica
- μ_i is the expected mean service rate on a standard CPU

Costs

c is a cost matrix with $\#k + 1$ rows and $\#i$ columns where:

- $c_{0,i}$ is the activation cost of node i $(1 \leq i \leq \#i)$
- $c_{k,i}$ is the cost per replica of microservice k in node i $(1 \leq k \leq \#k)$

Times

t is a time matrix with $\#i$ rows and $\#i$ columns where:

- t_{i_1,i_2} is the travel time between nodes i_1 and i_2 $(1 \leq i_1, i_2 \leq \#i)$

l is a latency matrix with $\#j$ rows and $\#i$ columns where:

- $l_{j,i}$ is the latency time between client j and node i $(1 \leq j \leq \#j, 1 \leq i \leq \#i)$

In the case of the example presented in section 3.1.1, the following constants

would be defined:

$$\#i = 3, \quad n = \begin{bmatrix} 10 & 7 & 5 \end{bmatrix}, \quad s = \begin{bmatrix} 10 & 10 & 8 \end{bmatrix}, \quad v = \begin{bmatrix} 1 & 2 & 2 \end{bmatrix}$$
$$\#j = 2, \quad \lambda = \begin{bmatrix} 3 & 2 \end{bmatrix}$$
$$\#k = 2, \quad m = \begin{bmatrix} 1 & 2 \end{bmatrix}, \quad \mu = \begin{bmatrix} 2 & 3 \end{bmatrix}$$
$$c = \begin{bmatrix} 1 & 2 & 1 \\ 1 & 1 & 2 \\ 1 & 1 & 1 \end{bmatrix}, \quad t = \begin{bmatrix} 0 & 2.24 & 2.24 \\ 2.24 & 0 & 1.41 \\ 2.24 & 1.41 & 0 \end{bmatrix}, \quad l = \begin{bmatrix} 0.71 & 1.58 & 1.58 \\ 1.8 & 0.5 & 1.12 \end{bmatrix}$$

3.2.2 Variables

Variables are *unknowns* that can assume values from a specific domain. Some quantities, such as the number of replicas deployed in the processing nodes, are naturally represented by integer valued variables (it makes no sense to deploy half of a replica). Other variables, such as those to represent time, are typically real valued variables. Additionally, depending on the constraint solver ability to deal with finite or continuous domains, quantities such as the expected number of requests, may either be directly represented as real valued variables or discretized into integer valued variables.

In the constraint model, we can split the variables into two groups, one being the decision variables and the other the dependent variables. The decision variables (or solution variables) characterize our solution. Once assigned values to the solution variables, that satisfy all the model constraints, a candidate solution is obtained. All the values for the dependent variables are determined as a function of such assignment and can be obtained from the constraints of the model.

Our goal is to determine the deployment of microservices replicas into the processing nodes and to define the flow of requests for processing the microservices on the available nodes. The solution variables capture both, the deployment and the flow information.

Solution Variables

The matrix of replicas x with $\#k$ rows and $\#i$ columns where:

$x_{k,i}$ is the number of replicas of microservice k in node i $\quad (1 \leq k \leq \#k)$ $\quad (1 \leq i \leq \#i)$

The matrix of requests y with $\#j$ rows and $\#i$ columns where:

$y_{j,i}$ is the number of requests from client j to node i $\quad (1 \leq j \leq \#j, 1 \leq i \leq \#i)$

A vector of $\#k-1$ matrices $z^{(1)}, z^{(2)}, \ldots, z^{(\#k-1)}$ with $\#i$ rows and columns where:
$z^{(k)}_{i_1,i_2}$ is the number of requests of microservice $k+1$ to be processed at node i_2 after finishing microservice k at node i_1 $(1 \leq i_1, i_2 \leq \#i, 1 \leq k < \#k)$

Note that the number of matrices created on the last decision variable will vary based on the number of microservices of the application ($\#k$). Let's say the number of microservices is 2; there will be only one matrix to the forward variable, and it would be the request passed from the node where the microservice 1 was processed to the node where the microservice 2 will be processed.

Both deployments and flows cannot be negative, so all cells of the solution matrices have non negative domains. Moreover the cells of the deployments matrix x are integer valued and the cells of the flows matrices y and $z^{(k)}$ can either be real or integer valued.

The dependent variables are used to access the quality of the solutions both in terms of cost and expected execution time. For a particular solution, its total cost depends on the nodes that are used and the number of replicas deployed in each node.

Cost Variables

Integer vectors A and C with $\#i$ elements where:
A_i is the cost associated with the activation of node i $(1 \leq i \leq \#i)$
C_i is the cost associated with the number of replicas in node i $(1 \leq i \leq \#i)$
Integer variable $Cost$ representing the total cost of the solution

The computation of service times are based on concepts of queuing theory (see section 2.4), such as, mean request rates, mean service rates, utilization ratio and expected waiting times for each processing node.

Queue Variables

Real matrices μ, λ, ρ and W with $\#k$ rows and $\#i$ columns where:
$\mu_{k,i}$ is the mean service rate for microservice k at node i $(1 \leq k \leq \#k)$
$\lambda_{k,i}$ is the mean request rate for microservice k at node i $(1 \leq i \leq \#i)$
$\rho_{k,i}$ is the utilization ratio of microservice k at node i
$W_{k,i}$ is the expected waiting time in system for microservice k at node i

The time variables represent the expected service times predicted from the queue variables.

Time Variables

Real matrix S with $\#k$ rows and $\#i$ columns where:

$S_{k,i}$ is the expected service time to terminate all remaining microservices after the request of microservice k at node i $(1 \leq k \leq \#k, 1 \leq i \leq \#i)$

Real vector T with $\#j$ elements where:

T_j is the expected total service time for client j $\quad (1 \leq j \leq \#j)$

Real variable $Time$ representing the expected total service time of the solution

Note that, even for small problems, its constraint model has a large set of variables and a huge solution space.

The number of solution variables x, y and z is, respectively, $\#k \times \#i$, $\#j \times \#i$ and $(\#k - 1) \times \#i^2$. The cost variables are $2\#i + 1$, the queue variable are $4(\#k \times \#i)$, and the time variables are $\#k \times \#i + \#j + 1$. The total number of variables of a problem with $\#i$ processing nodes, $\#j$ clients and $\#k$ processes is:

$$(\#k - 1) \times \#i^2 + (6\#k + \#j + 2) \times \#i + \#j + 2 \tag{3.1}$$

In our small example with $\#i = 3$, $\#j = 2$ and $\#k = 2$, there are: $2 \times 3 + 2 \times 3 + (2 - 1) \times 3^2 = 21$ solution variables; $2 \times 3 + 1 = 7$ cost variables; $4(2 \times 3) = 24$ queue variables; and $2 \times 3 + 2 + 1 = 9$ time variables. This gives a total of 61 variables.

The size of the solution space is determined by the Cartesian product of the domains of the solution variables. The domain of each $x_{k,i}$ is limited by the number of cores of the processing node and so its value is an integer between 0 and n_i ($n_i + 1$ possibilities). The domain of each $y_{j,i}$ is limited by the number of requests of the client and so its value is an integer between 0 and λ_j ($\lambda_j + 1$ possibilities). The domain of each z_{i_1,i_2} is limited by the total number of requests from all clients and so its value is an integer between 0 and $\sum_{j=1}^{\#j} \lambda_j$ ($1 + \sum_{j=1}^{\#j} \lambda_j$ possibilities).

For a problem with $\#i$ processing nodes (where node i has n_i cores), $\#j$ clients (where client j makes λ_j requests per time unit) and $\#k$ processes, the size of the Cartesian product of the domains of the solution variables is:

$$\prod_{i=1}^{\#i} (n_i + 1)^{\#k} \times \prod_{j=1}^{\#j} (\lambda_j + 1)^{\#i} \times \left(1 + \sum_{j=1}^{\#j} \lambda_j\right)^{(\#k-1)\times\#i^2} \tag{3.2}$$

In our small example with $\#i = 3$ ($n = [10, 7, 5]$), $\#j = 2$ ($\lambda = [3, 2]$) and $\#k = 2$, the size of the solution space is:

$$= (10 + 1)^2 \times (7 + 1)^2 \times (5 + 1)^2 \times (3 + 1)^3 \times (2 + 1)^3 \times (1 + 3 + 2)^{(2-1)\times 3^2}$$
$$= 11^2 \times 8^2 \times 6^2 \times 4^3 \times 3^3 \times 6^9 = 121 \times 64 \times 36 \times 64 \times 9 \times 10077696$$
$$= 1618272231358464 \approx 1.6 \times 10^{15}$$

3.2.3 Constraints

The constraints define the relationships between the variables that are relevant for modeling our problem. Constraint based reasoning discard solutions that do not

satisfy the relationships specified in the constraints.

For modeling the cloud/edge environment we need constraints to: restrict the allocation of microservice replicas in the processing nodes; compute the associated costs; represent the flow of requests between clients and nodes; and use queuing theory to predict the expected service times.

The allocation constraints restrict the allocation of microservice replicas into the processing nodes according to their number of cores and storage capacity.

Allocation Constraints

The number of microservice replicas deployed in a node cannot exceed its number of cores:

$$1 \leq i \leq \#i : \quad \sum_{k=1}^{\#k} x_{k,i} \leq n_i \tag{3.3}$$

The total memory space occupied by the replicas at a node cannot exceed its available storage space:

$$1 \leq i \leq \#i : \quad \sum_{k=1}^{\#k} m_k x_{k,i} \leq s_i \tag{3.4}$$

The cost constraints establish the relation between the cost variables and the allocation variables.

Cost Constraints

The activation cost is associated with a node if there is at least one allocated replica:

$$1 \leq i \leq \#i : \quad A_i = c_{0,i} \left(\sum_{k=1}^{\#k} x_{k,i} > 0 \right) \tag{3.5}$$

The cost associated with the number of replicas allocated at a node is proportional to each cost per replica:

$$1 \leq i \leq \#i : \quad C_i = \sum_{k=1}^{\#k} c_{k,i} x_{k,i} \tag{3.6}$$

The total cost is the sum of the costs associated with each node:

$$Cost = \sum_{i=1}^{\#i} (A_i + C_i) \tag{3.7}$$

The flow constraints establish restrictions on the flow of requests between clients and nodes.

Flow Constraints

The client requests are distributed among all nodes:

$$1 \leq j \leq \#j : \quad \lambda_j = \sum_{i=1}^{\#i} y_{j,i} \tag{3.8}$$

After finishing the first microservice at a node, the requests of the next microservice are distributed among all nodes: $(\#k \geq 2)$

$$1 \leq i_1 \leq \#i : \quad \sum_{j=1}^{\#j} y_{j,i_1} = \sum_{i_2=1}^{\#i} z^{(1)}_{i_1,i_2} \tag{3.9}$$

After finishing the k^{th} microservice at a node, the requests of the next microservice are distributed among all nodes: $(1 \leq k < \#k - 1)$

$$1 \leq i_1 \leq \#i : \quad \sum_{i=1}^{\#i} z^{(k)}_{i,i_1} = \sum_{i_2=1}^{\#i} z^{(k+1)}_{i_1,i_2} \tag{3.10}$$

The queue constraints relate the replica allocations and request flow to the concepts of queuing theory such as mean request rate, mean service rates and expected waiting times at the processing nodes.

Queue Constraints

The mean service rate of a microservice at a node is proportional to the speed of the node CPU compared to the speed of the standard CPU:

$$1 \leq k \leq \#k, 1 \leq i \leq \#i : \quad \mu_{k,i} = v_i \mu_k \tag{3.11}$$

The mean request rate of the first microservice at a node is the sum of requests from all clients:

$$1 \leq i \leq \#i : \quad \lambda_{1,i} = \sum_{j=1}^{\#j} y_{j,i} \tag{3.12}$$

The mean request rate of k^{th} microservice at a node is the sum of requests from all nodes after finishing the previous microService:

$$2 \leq k \leq \#k, 1 \leq i \leq \#i : \quad \lambda_{k,i} = \sum_{i_1=1}^{\#i} z^{(k-1)}_{i_1,i} \tag{3.13}$$

The utilization ratio of a microservice at a node is its mean request rate divided by the service rate:

$$1 \leq k \leq \#k, 1 \leq i \leq \#i : \quad \rho_{k,i} = \frac{\lambda_{k,i}}{x_{k,i}\mu_{k,i}} \tag{3.14}$$

The utilization ratio must be strictly smaller than 1 to achieve a steady state:

$$1 \leq k \leq \#k, 1 \leq i \leq \#i : \quad \rho_{k,i} < 1 \tag{3.15}$$

The expected waiting time for a microservice at a node is given by the M/M/S queue model rules:

$$1 \leq k \leq \#k, 1 \leq i \leq \#i :$$

$$\begin{aligned} P_{k,i} &= \frac{1}{\sum_{n=0}^{x_{k,i}-1} \frac{(\lambda_{k,i}/\mu_{k,i})^n}{n!} + \frac{(\lambda_{k,i}/\mu_{k,i})^{x_{k,i}}}{x_{k,i}!}(\frac{1}{1-\lambda_{k,i}/x_{k,i}\mu_{k,i}})} \\ W_{k,i} &= \frac{P_{k,i}\lambda_{k,i}^{x_{k,i}}}{(x_{k,i}-1)!\mu_{k,i}^{x_{k,i}-1}(x_{k,i}\mu_{k,i}-\lambda_{k,i})^2} + \frac{1}{\mu_{k,i}} \end{aligned} \tag{3.16}$$

The time constraints establish restrictions to compute the expected service times based on the expected waiting times at the processing nodes and their geographical locations.

Time Constraints

The expected service time of the last microservice is the expected waiting time of its processing node:

$$1 \leq i \leq \#i : \quad S_{\#k,i} = W_{\#k,i} \tag{3.17}$$

The expected service time of a sequence of microservices is the expected waiting time for processing the first plus the sum of the travel time to the next processing node with the expected service time of the remaining sequence (averaged by the number of requests):

$$1 \leq k < \#k, 1 \leq i \leq \#i : \quad S_{k,i} = W_{k,i} + \sum_{i_2=1}^{\#i} \frac{z_{i,i_2}^{(k)}}{\sum_{i_2^1=1}^{\#i} z_{i,i_2^1}^{(k)}} (t_{i,i_2} + S_{k+1,i_2}) \tag{3.18}$$

The expected service time of a client is the sum of the latency to the first processing node with the expected service time of the all microservices sequence (averaged by the number of requests):

$$1 \leq i \leq \#j : \quad T_j = \sum_{i=1}^{\#i} \frac{y_{j,i}}{\sum_{i_1=1}^{\#i} y_{j,i_1}} \left(l_{j,i} + S_{1,i}\right) \tag{3.19}$$

The expected total service time of a solution is the sum of the expected service time of each client averaged by the number of requests:

$$Time = \sum_{j=1}^{\#j} \frac{\lambda_j}{\sum_{j_1=1}^{\#j} \lambda_{j_1}} T_j \tag{3.20}$$

3.2.4 Constraint Reasoning

In the development of the model, there are two ways to integrate constraint reasoning: either by adding constraints to existing variables or by optimizing them.

The addition of constraints in existing variables can be pretty vast, but a few examples can show this type of reasoning in the model.

Solution Variables

A node is activated when its activation cost is higher than zero. By counting the number of nodes with activation costs higher than zero, we can arrive at the number of activated nodes. Through constraint reasoning, we can introduce a constraint that forces a specific number of nodes n to be activated, directly impacting how the solution variables are arranged.

The number of nodes activated is at least n when there are at least n nodes with an activation cost higher than zero:

$$\sum_{i=1}^{\#i} (A_i > 0) \geq n \tag{3.21}$$

Cost Variables

The cost of the solution can also be constrained to meet certain criteria. In this example, if we wanted to impose a ceiling on the cost limitCost, a constraint could be added by making the total cost lower or equal to that.

$$Cost \leq limitCost \tag{3.22}$$

Queue Variables

Guarantee a given utilization ratio ρ_{min} for all microservices inside the nodes to ensure that the microservices are not underutilized.

$$1 \leq k < \#k, 1 \leq i \leq \#i : \quad \forall_{k,i} \quad \rho_{k,i} \geq \rho_{min} \tag{3.23}$$

Time Variables

Total service time is one of the time variables, where constraint reasoning can also be applied by introducing a constraint that does not allow solutions above a certain time, limitTime.

$$Time \leq limitTime \tag{3.24}$$

The following optimizations were the ones that were found relevant to the main scope of the problem:

Cost Optimizations

- Minimization of the Total Cost: The cost quantifies how expensive the solution is, and translating this to our problem, we want to save the most resources. Subsequently, the cost is directly related to the number of microservice replicas deployed, making the minimization of this property one of the key objectives when developing the algorithm.

Time Optimizations

- Minimization of the Expected Total Service Time: The expected total service time symbolizes the amount of time wasted to complete all client requests in normal conditions, meaning without any failure, which is why it is an expectancy and not an absolute value. This time is affected by how long the chain of microservices is, the times of communication between nodes and clients and the times of communication between nodes. Although time is not a resource, minimizing it when the system is performance-focused is essential.

- Minimization of the Worst Expected Total Service Time for a Client: On the same note as the previous optimization, the worst expected total service time for a client is also an expectancy. Still, it symbolizes the maximum time wasted for a client to complete all of his requests. This minimization tries to minimize the time wasted for the client when the service is taking longer. As previously said, time may not be a resource, but minimizing this value is pivotal when creating an environment where clients have a lower discrepancy in service times.

Cost and Time Optimizations

- Optimal Pareto: The optimal Pareto, in our problem, involves finding solutions that reflect the best possible compromise between total cost and the expected total service time. A solution is considered Pareto optimal if no solution could improve one objective without degrading the other. This optimization can help analyze the best trade-offs and balance between both objectives.

3.3 Implementation

This section will focus on the implementation of the Constraint Model with CHOCO solver and standard Java programming. It will also discuss the search method used and how solutions are displayed after the model's execution.

3.3.1 Constants and Variables

Constants from each problem are stored in a file that serves as the input for our problem. Internally, during each execution, each constant takes the values written on that file. This allows the constraint model to be created more dynamically and makes it easier to change between values without altering the constraint model.

In CHOCO's solver, every variable implemented must have a lower and an upper bound; from the choices made during the implementation, every variable declared with the CHOCO's library has Integer domains. Variables with real domains are only dealt with after all the discrete values are found through CHOCO's solver. Since the discrete values are all positive, every variable has a lower bound of zero and an upper bound that goes all the way to its possible maximum. The upper bound value is significant because it directly impacts the execution of the solver and the number of solutions given; if this value is too low, there is a possibility that some variables are being deprived of taking some values, which contributes to a lower amount of solutions. On the other hand, a larger upper bound can make the solver try values that are out of the scope of the problem.

3.3.2 Constraints

When implementing the constraint model, there was the need to split the constraints into two groups. The first group comprised only constraints that worked with integer domains, and the second group had constraints that worked with real domains. This separation was made to facilitate the integration of the constraints inside the CHOCO's solver due to its nature of only being able to define constraints with integer domain variables.

Besides the time constraints, every type of constraint belongs to the first group of constraints, and all of them are handled through CHOCO's solver constraint model.

The time constraints are only applied after a solution is found on the CHOCO's constraint model; the values present on these constraints are calculated through standard Java programming outside the CHOCO's constraint model. Although

these values are not constrained directly, they result from a function on the solution variables constrained directly through CHOCO's constraint model.

Inside the CHOCO's constraint model, constraints served a dual purpose. Aside from representing every kind of integer domain constraint, they were also instrumental in forming auxiliary variables. These variables are crucial in maintaining values from more complex calculations or states within the algorithm. Specifically, they capture and store the aggregate values of sums across various rows and columns in the matrices involved; these constraints can be consulted in the appendix A. This approach is essential as it simplifies and streamlines the process of applying subsequent constraints. By having these sums readily available in the auxiliary variables, the algorithm can more efficiently apply further constraints, thus enhancing its overall performance and accuracy.

Below, we can see the implementation of three different constraints: the first one is an allocation constraint, the second is a cost constraint, and the third is a function to recreate a time constraint; the remaining constraints and functions used to recreate the time constraints can be seen in the appendix section A.

```
1    for (int i = 0; i < nodes; i++) {
2      md.scalar(x[i], microservicesSpecs[0], "=", m[i]).post();
3      md.arithm(m[i], "<=", nodeSpecs[i][0]).post();
4    }
```

Listing 3.1: Allocation Constraint (3.4)

In this piece of code, we can see two distinct constraints from CHOCO's solver, the "arithm"and the "scalar". The for loop goes through all the nodes in the problem, and the "scalar"constraint is a weighted sum constraint that ensures that the sum of the products of variables and their corresponding coefficients equals a given value. In this case, it ensures that the sum of x[i] multiplied by microservicesSpecs[0] is kept in m[i]. The x[i] is an array that keeps all of the replicas of each microservice inside node i; if there are three microservices, the vector would have a length of three, one for each microservice and their respective replicas inside that node. The microservicesSpecs[0] is an array that keeps each microservice's memory when deployed into a node. The m[i] keeps the total memory occupied by all of the microservices replicas inside the node, and its value is assigned through the "scalar"constraint after the calculations. The "arithm"constraint ensures that the memory occupied in each node (m[i]), previously calculated, does not exceed the available storage inside the node, which is stored in the matrix nodeSpecs.

```
1    for (int i = 0; i < nodes; i++) {
```

```
2       md.ifThenElse(md.arithm(x_row[i], ">", 0), md.arithm(a[i], "=",
    costs[0][i]), md.arithm(a[i], "=", 0));
3     }
```

Listing 3.2: Cost Constraint (3.5)

This second piece of code has another type of CHOCO's solver constraint: a conditional constraint ifThenElse. Like standard Java programming, a constraint is applied if the condition is met; if not, another constraint is used. In this specific case, if the sum of all of the microservices' replicas inside a node is greater than zero, meaning if there are any microservices' replicas inside a node, then the activation cost for that node is the activation cost in the cost table. Otherwise, the activation cost is zero because there are no replicas, so the node does not need to be activated.

```
1
2 private static double calculateTotalServiceTime(double[] timePerClient
    , int[] clientRequests,int clients) {
3     double totalServiceTime = 0;
4     for(int i = 0; i< clients; i++) {
5     double part1 = clientRequests[i];
6     double part2 = 0;
7
8     for(int k = 0; k < clients; k++) {
9         part2 += clientRequests[k];
10    }
11
12    double part3 = timePerClient[i];
13    double part4 = (double) part1 / part2;
14    totalServiceTime += part4 * part3;
15
16    }
17    return totalServiceTime;
18 }
```

Listing 3.3: Time Constraint (3.20)

The last example shows a time constraint that was calculated through a function in standard Java programming. This function calculates the total service time by applying the formula described on 3.20. The client requests are acquired through the file containing the constants of the problem, and the times per client are calculated through the time constraint 3.19, which is the result of a function on the solution variables.

3.3.3 Constraint Solving

When solving a constraint model, a solver must be created and tuned in to adjust the solver's search method to be more effective and objective-driven in our problem.

The search method is crucial to our problem because it indicates the order by which variables will be instantiated. CHOCO's default search system is a **domOverWDegSearch**. This means prioritizing variables with smaller domains and higher weighted degrees during the instantiation. The weighted degree of a variable is the sum of the weights of constraints involving that variable.

In this model, only discrete values are manipulated directly due to the reasons discussed in section 3.3.2; this makes the decision variables and the total cost the focal points when searching for optimal solutions. Since half of the optimizations are cost-related, and the cost is the only discrete value out of the three, the search method must revolve around minimizing that value. To achieve that, the search system chosen was the **inputLBOrderSeach**; this search strategy needs the order of the variables being instantiated and forces them to start always from the lower bound of their domain.

The order chosen for our problem is the total cost first, followed by the other decision variables, guaranteeing that every single solution found is ordered from the minimum to the maximum cost.

Although CHOCO's library has a built-in way of minimizing variables with integer domains, which are the ones being handled directly through CHOCO's constraints, variables being restrained through time constraints such as the total service time and the maximum total service time spent for a client, needed a way of being minimized since they are not directly constrained through CHOCO. The search strategy implemented is a custom minimization of the cost and ensures that the cost of the solution or solutions arriving are on a cost-ascendant order, with the use of an integer number (delta), representing the offset values that the cost will take further away from the global minimum and restraining the solver's search space this opens up a way for a local minimization of the variables within the time constraints. The delta value does not apply to the cost minimization because it implies achieving the minimal cost first.

For the optimizations proposed in 3.2.4, besides the cost optimization, all utilize the delta value and start by finding the minimal cost solution and then going through every single solution from the minimal cost to (minimal cost + delta). Suppose the minimal cost is x, and the delta equals 2. In that case, the solver runs until the last solution with the cost (x+2) and keeps updating the solution or solutions according to the optimization chosen. The minimization of the expected

total service and the minimization of the worst expected total service time for a client keep updating the minimal solution until the search space is finished and return the solution that best fits their objective, in this case, the solution with the least total service time and the solution with the least maximum expected total service time for a client. The Optimal Pareto works the same way as the previous two, but instead of keeping one solution, it keeps a set of solutions. Since cost minimization is used in every optimization, its implementation has no extra steps as the others because the first solution returned is guaranteed to be cost-minimal.

Similarly to the constants, the constraint model leading optimization is stored in a file with an integer number (delta) and a boolean that lets the algorithm accept solutions with the exact cost and expect total service time.

Apart from the leading optimization present in the file during execution, there is the option of having an additional file containing one or more goals that filter out even more solutions by specifying a maximum total cost, a maximum expected total service time, a maximum expected total service time for a particular client, a maximum for the maximum expected total service time, minimum utilization ratio for a specific microservice inside a particular node and a minimum utilization ratio for all microservices.

3.3.4 Solutions

The solutions are the results the constraint model gives after its execution, and solutions vary depending on the type of problem and objective chosen. However, their structure remains the same for every single optimization; the variables that form a solution are always the solution variables, the total cost, the maximum expected total service time for a client and the expected total service time.

Looking back to our simple example 3.1.1 and both the minimization of the cost and total service time minimization as our optimizations, the solution would look like this:

Cost Minimization

**** solution 1 ****

x:

$$\begin{bmatrix} 0 & 0 \\ 2 & 1 \\ 0 & 0 \end{bmatrix}$$

y:

$$\begin{bmatrix} 0 & 3 & 0 \\ 0 & 2 & 0 \end{bmatrix}$$

z0:

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Cost = 5
Max Time for a Client: 2.94
Total Service Time: 2.51

Total Service Time Minimization
delta = 2

**** solution 1 ****

x:

$$\begin{bmatrix} 0 & 0 \\ 3 & 2 \\ 0 & 0 \end{bmatrix}$$

y:

$$\begin{bmatrix} 0 & 3 & 0 \\ 0 & 2 & 0 \end{bmatrix}$$

z0:

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Cost = 7
Max Time for a Client: 2.04
Total Service Time: 1.61

Looking at our solution through the decision variable x, we can see the number of replicas of each microservice in each node; through y, we can see where the requests from the clients initially went; and through z, since there are only two microservices in the initial problem there is only one matrix, we can trace where the requests, processed inside a node where the first microservice was deployed, went to process the second microservice.

Unlike the cost minimization solution, the total service time minimization has an extra parameter, which is the maximum offset from the minimal cost where the solver searches for better solutions, delta. In this case, the best solution was in the last cost value allowed by the offset because the minimal cost is 5, so the delta goes through all the solutions less or equal to cost 7.

The diagram 3.2 shows how the solutions can be translated into our cloud/edge environment; the numbers in each arrow represent the number of requests sent to that node, and the number in parenthesis is the time it takes for the requests to arrive at that node. The set of numbers inside each microservice is their queuing model M/M/s, which inside node 2 the microservice 1 five requests are arriving. The value 2 is the number of servers our queue line has, in this case, the number of replicas available to process the requests, which is 1 replicas. Each replica can process 4 requests due to the velocity index inside node 2, which is twice as fast as a standard CPU; the standard CPU velocity would take only 2 requests at the time for the microservice 1. For the total service time minimization the diagram generated would be same, only with one more replica in both of the microservices.

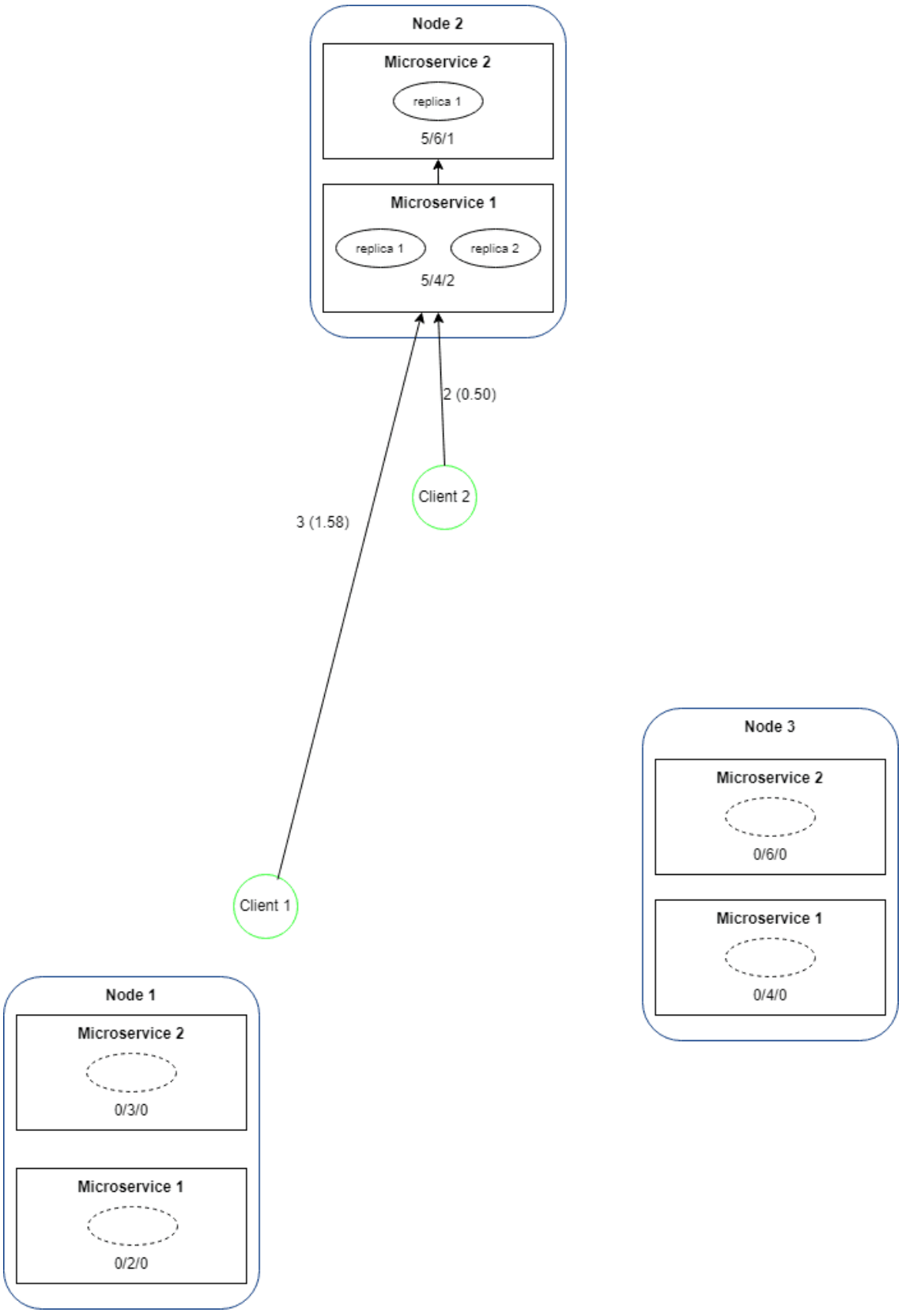

Figure 3.2: Minimal Cost Solution Diagram

4

Evaluation and Testing

This section delves into the model's behaviour in different problems and objectives; every problem involved was created to test different cloud/edge scenarios.

When testing the model, there was the need to create not only a single problem but a set of them to test the limits of our solver and emulate different types of cloud/edge environments. Two-stage tests were made to reproduce the vast number of scenarios and a final example that validates the model's use for a potential real-life scenario. The first stage of tests aims to find the impact of each component present in the cloud/edge environment on the model and the boundaries where the model can operate. The second stage tests the impact of the cost on the execution times and the impact of varying each element in a cloud/edge scenario separately. The final scenario is a possible real-life problem the model tries to solve to show its applicability to those scenarios.

During the testing phase, not all the optimizations were tested; the cost minimization was the only optimization tested in the first stage, and in the second stage, every optimization besides the maximum time per client minimization was tested since it was similarly implemented and functioning the same way as the total service time minimization. The choice of only testing the cost minimization in the first wave of tests is because this optimization is present in every other optimization implemented, so the impact of each component would not be any different in the other optimizations.

In the second stage of tests, both the objective and deltas values chosen are pivotal when executing the model. Minimizing the total cost is the only objective, where solutions are not kept and immediately returned the moment they are found. On the other hand, every other optimization that involves a continuous value keeps updating the minimal solution until the search space is over. The delta's influence comes in due to the lack of control on continuous variables inside the solver, forcing every solution to be analyzed; this offset keeps the algorithm

from running infinitely by limiting the search space. The higher the delta value, the further we get from the minimum, exponentially increasing the number of solutions to be analyzed.

4.1 First Stage

The first stage of testing the model is designed to measure the complexity of the cloud/edge scenarios where the model can perform. The idea was to start from a simple and elemental structure of two nodes, two clients and two microservices and then make it more complex step by step.

Two phases can occur when testing a structure: a generation phase and an evaluation phase. The generation phase consists of creating sixty different configurations of the structure being analyzed, and an evaluation phase dictates whether the structure can be feasible through the model based on the model's performance in the configurations generated in the generation phase. A structure is feasible if fifty of the sixty configurations are solvable within the solver's time limit; however, a structure is unfeasible if more than ten configurations can not be solved within the solver's time limit. Although the generation phase always occurs, the evaluation may not happen if one of the two cases is presented: the model previously has been able to solve more complex structures than the one being tested, making it automatically feasible, or the model previously could not solve simpler structures than the one being tested, making it automatically unfeasible. Two structures, A and B, A is more complex than B if at least one of the components is strictly higher than B's components and the other ones are equal or higher ((nodesA > nodesB and clientsA >= clientsB and microservicesA >= microservices B) or (nodesA >= nodesB and clientsA > clientsB and microservicesA >= microservices B) or (nodesA >= nodesB and clientsA >= clientsB and microservicesA > microservices B)); A is simpler than B if B is more complex than A.

After running the first tests and concluding that the most simple example is feasible, the following structures were created incrementally by adding nodes, microservices and clients. This incremental method works like a "tree", being x the number of nodes, y the number of clients and z the number of microservices, the "tree"starts with the structure xyz. Every time a structure is feasible, the "tree"grows three new branches: a branch containing a structure with one more node, another with one more client and another with one more microservice than its predecessor. If the xyz structure is feasible, an (x+1)yz, an x(y+1)z, and an xy(z+1) will be created, as the figure shows.Furthermore, when an expansion

of the tree occurs, the fifty configurations solved in the parent structure,xyz, are passed down to the descendant structures, (x+1)yz,x(y+1)z and xy(z+1), to be modified and generate the subsequent configurations according to next structure's characteristics.

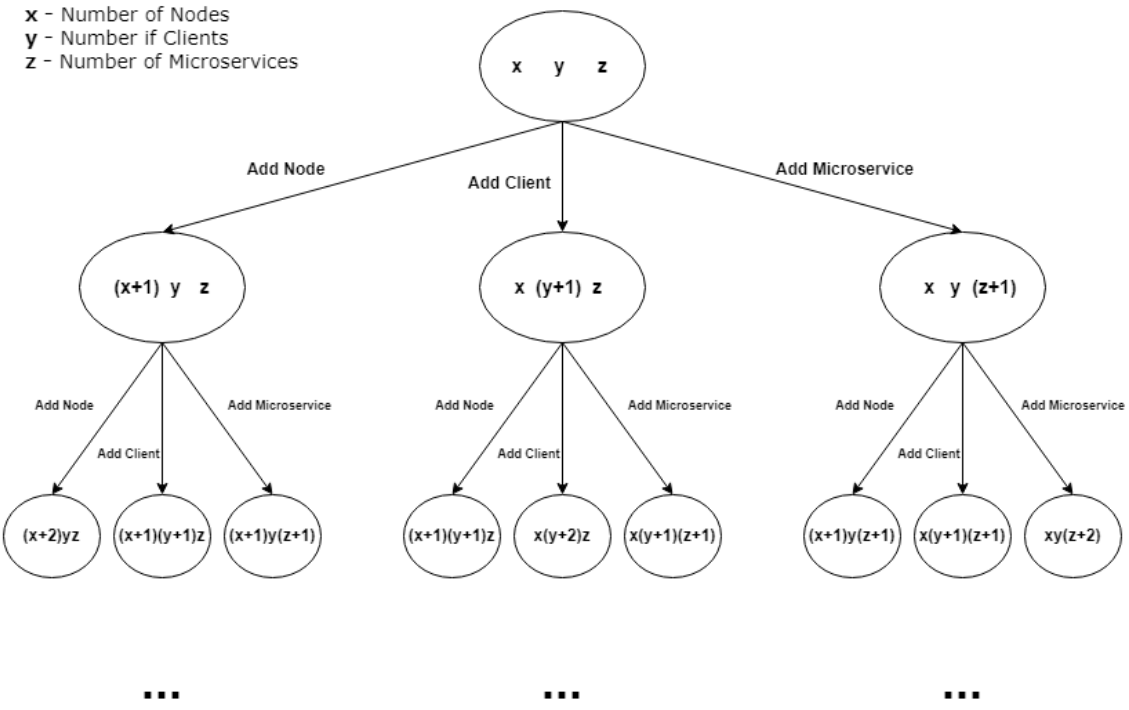

Figure 4.1: "Tree"Diagram used in the first stage of tests

Only the configurations from the primal structure are entirely randomly generated; every structure that follows builds on those existing configurations and adds one node, one client or one microservice at a time, with their constants also randomized. Despite passing down the fifty solvable configurations, not all are used to generate the next sixty to be evaluated by the upcoming structure. Each time a configuration is solved, the problem and the resolution time it took are kept until we have the fifty solved configurations needed to move on to the next structure. With the data gathered, the population of problems and their respective resolution times go through a sampling process.

The sampling process starts by calculating the mean and standard deviation of the population, a new population L of fifty numbers drawn from a Gaussian distribution with these statistics is made. From this new population, a recalculation of the mean and standard deviation are made, and then a Gaussian probability density function (PDF) is applied; this describes the probability distribution of each continuous variable inside L. After applying the PDF, a normalization of

these probabilities is needed to ensure they sum up to 1, and a fifty-sample selection starts by randomly picking elements from L based on their normalized probabilities, forming a new population L1. The L1 population has a sample of both the problems and their resolution times picked from the original population; the sampled population can have the same problem more than once because it is a probability-based process. The whole purpose of the sampling process is to minimize and restrain the number of outliers present in the configurations solved with a particular structure, increasing the likelihood of problems that have resolution times nearer the mean of the population to be selected.

The whole "tree"process used to explore all the possible combinations of structures ends when the "tree"can grow no longer, and all of the branches created were evaluated through the model. A stoppage criterion was applied to both mitigate the time spent and to get results within a computable time frame; a limited number of nodes, clients and microservices of 10 and a solver time limit of 1 minute were the conditions that every structure evaluated was submitted to during this first stage of tests.

4.1.1 Results

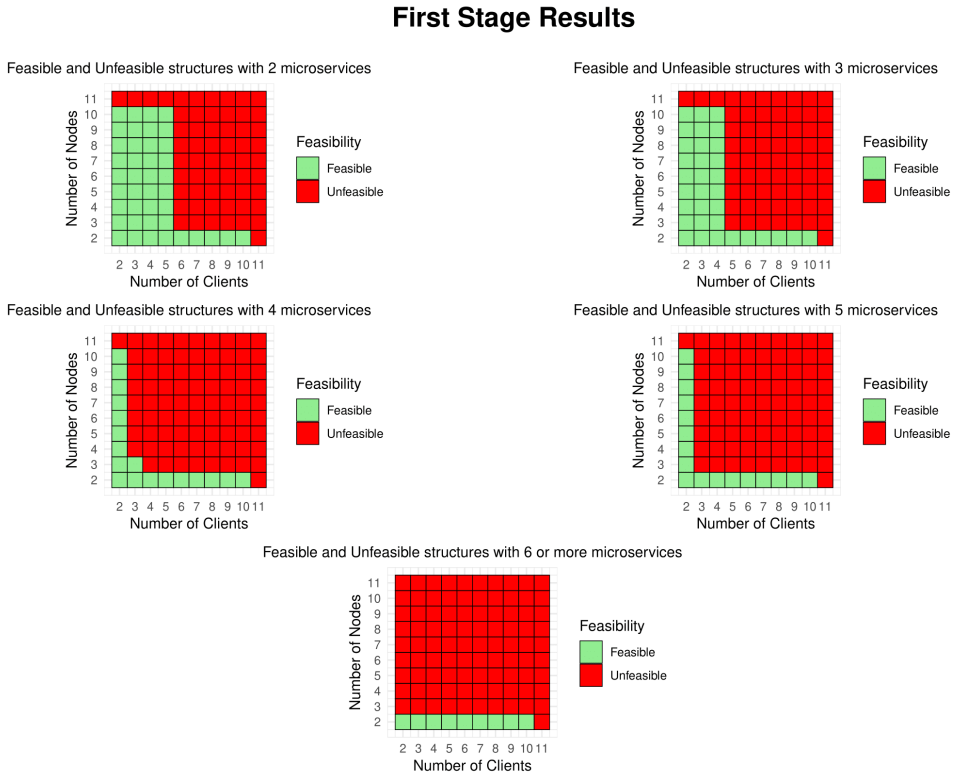

Figure 4.2: First Stage Results

After the first stage of tests, the results can be expressed through a set of two-dimensional tile plots, describing the feasible and unfeasible structures through the model. Despite being three different components in a structure, the microservices were the components staying out of the set of plots to better visualize the boundaries where the model performs 4.2.

Through the analysis of the plots above, the "boundary"structures where the model can perform are 10/5/2, 10/4/3, 10/2/5,3/3/4,2/10/10 where the numbers represent the number of nodes, number of clients and number of microservices, respectively; every structure simpler than any of these structures is also feasible. From the set of plots, there is also an apparent decay of feasibility when the number of microservices increases. Structures with four microservices make the model struggle to solve them in under one minute, and only the structures with two clients are feasible independently of the number of nodes.

When the number of microservices surpasses five, the tile plot remains the same until the ten-microservice mark. The only structures feasible are the ones that do not go over two nodes. This abrupt collapse of the number of nodes happens due to the lack of solutions in this type of structures, meaning that the model to solve the 2-node structures with more than five microservices, most of the configurations it encounters do not have any solutions, and it wastes most of its time proving that by going through all of the possible options; when the number of nodes increases the model could have solutions but has a hard time to find them in the time limit imposed. With this, we can see that the impact of the number of microservices is instrumental in increasing the complexity of the structure, making it harder for the model to solve.

4.2 Second Stage

The second stage of tests is highly dependent on the results from the first stage. The number of nodes, microservices, and clients is initially pinned into default values using the data gathered in the previous stage. The default values used are our control case. An initial study is conducted to measure the impact of the cost of each solution on the resolution times taken from fifty different feasible configurations for this default structure. A following study is made which compares the distribution of fifty different feasible configurations when there is an increase of nodes, clients, or microservices separately on the control case.

The model generates the configurations being solved in each structure, following the same pattern as the ones created in the first stage. This time, the only

difference is the time limit imposed by the solver, which is ten minutes. Furthermore, the second stage of tests also focuses on the total service time minimization and Optimal Pareto, aside from cost minimization.

During the second study in this stage, a set of plots was created, one for each variation in the optimization. Resolution times above ten minutes were not counted in for the creation of the plots, but their number was kept as a factor to consider when measuring the impact of each tested variable.

With the results of the first part of testing, the control case chosen had the following structure and delta value:

- Number of Nodes = 8
- Number of Microservices = 3
- Number of Clients = 3
- Delta = 2

These values were chosen because they are a good initial benchmark for the model to start. Both the number of microservices and clients can be explained as the middle ground between the minimum and the boundaries found in feasible structures during the first stage of tests. The number of nodes is higher due to its flexibility in reaching higher values with the number of clients and microservices chosen for our control case, as seen in the first stage of tests. The value of delta is also a factor to consider in this second part of testing because, aside from the cost minimization, the search space needs to be limited to get results within the solver's time limit. Since the variation is done separately, the components that are not varied always take their default values from the control case.

4.2.1 Results

The second stage of test results is divided into four segments: cost versus resolution time, cost minimization, total service time minimization, and optimal Pareto. Each segment offers unique insights into the model's performance with different structures and optimization objectives.

4.2.1.1 Cost vs Resolution Time

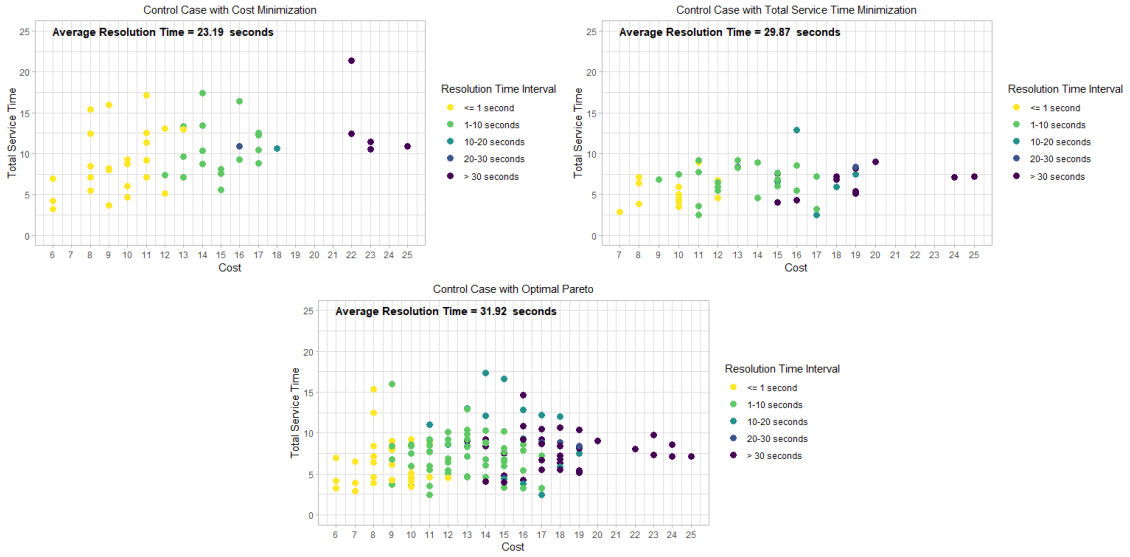

Figure 4.3: Second Stage Control Case Results with different Optimizations

After running each optimization on the control case, a set of plots was made with the resolution time of each configuration and their respective costs and service times.

As we can see from the plots, in any of the optimizations, we can dissect three different areas: the first area, defined by the yellow points, at the lowest cost and represents the quickest configurations solved; the second area, defined by green and blue points, is located in the middle portion of the plot and represents the configurations that are near the average resolution time, the third and last area defined by the violet points represents the configurations that took longer to be solved and are usually situated on the highest costs.

Through the analysis, the cost of the solution is directly related to the resolution time, making the model slower when the cost of the solution is higher. This is a consequence of the search method used to find the solutions, which explicitly makes the solver assign values to the cost starting from its lower bound. Although this type of search method guarantees that the first solution has a minimal cost, if the cost is too far away from the lower bound, the algorithm needs to go through a higher set of values until it finds the solution, making the resolution times longer.

Despite the fifty random feasible configurations for our control case being the same ones being solved in every optimization, points in each optimization are scattered differently. When comparing the cost optimization to the total service time minimization plot, we can see many points dropping to lower service times as expected since it is the optimization's role. On the Optimal Pareto, there are

more points in the plot because, in every configuration, there is a set of solutions instead of only one like the other two. Moreover, the average resolution time increases from the cost minimization to the others since they need to go through every single solution found within the delta's threshold.

4.2.1.2 Cost Minimization

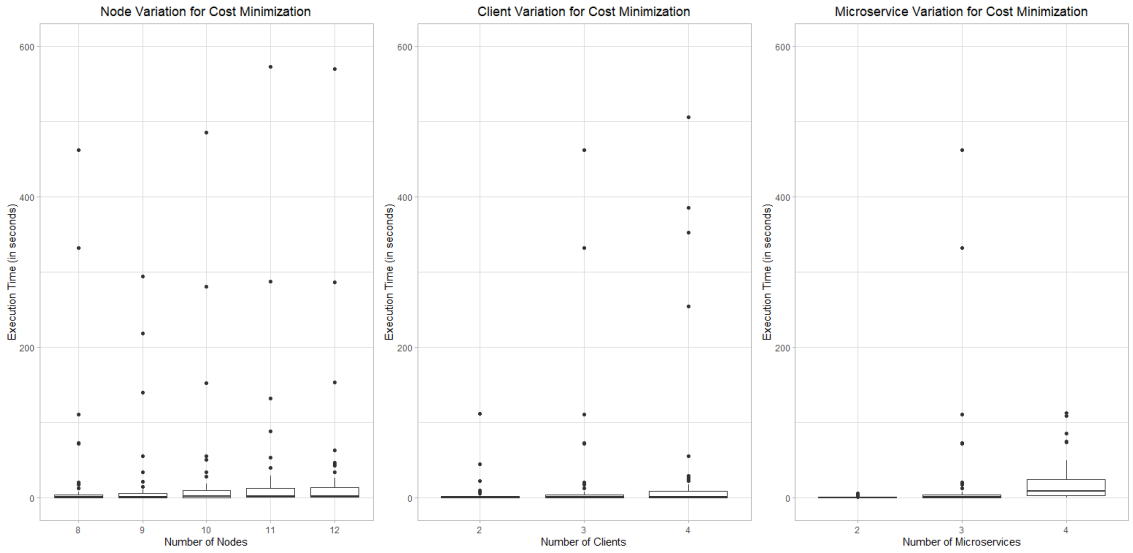

Figure 4.4: Execution times of 50 different feasible configurations with different number of nodes, clients and microservices for Cost Minimization

Through the set of box plots above, we can see how the execution times throughout the fifty different configurations with different numbers of nodes, clients, and microservices spread themselves in this optimization.

Looking into the node variation, we can see from the first boxplot that a higher number of nodes contributes to increased resolution times. The box lengths also increase, meaning a higher dispersion of execution times when using a structure with more nodes. Additionally, the outliers on structures with more nodes display longer execution times. Another key aspect considered was the number of configurations in each node structure over the 10-minute mark. From the 8-node structure to the 12-node structure, the configurations over the 10-minute mark were 2,4,4,6,7 respectively. Through the number of fails, we can see a gradual and consistent increase that the nodes can even go a bit further since they still have a gap until reaching ten or more beyond the time limit.

Similarly to the node variation, both microservices and client variations show an increase in the execution times the higher their values are. Furthermore, the width of the box plots in both client and microservice variations increases rampantly from

3 to 4; the resolution times' dispersion, especially on the microservice variation, surpasses the 12-node structures. Conversely, regarding the node variation, the number of configurations above the 10-minute mark is still low and consistent in both client and microservices' variations, non-surpassing the two configurations above 10 minutes.

4.2.1.3 Total Service Time Minimization

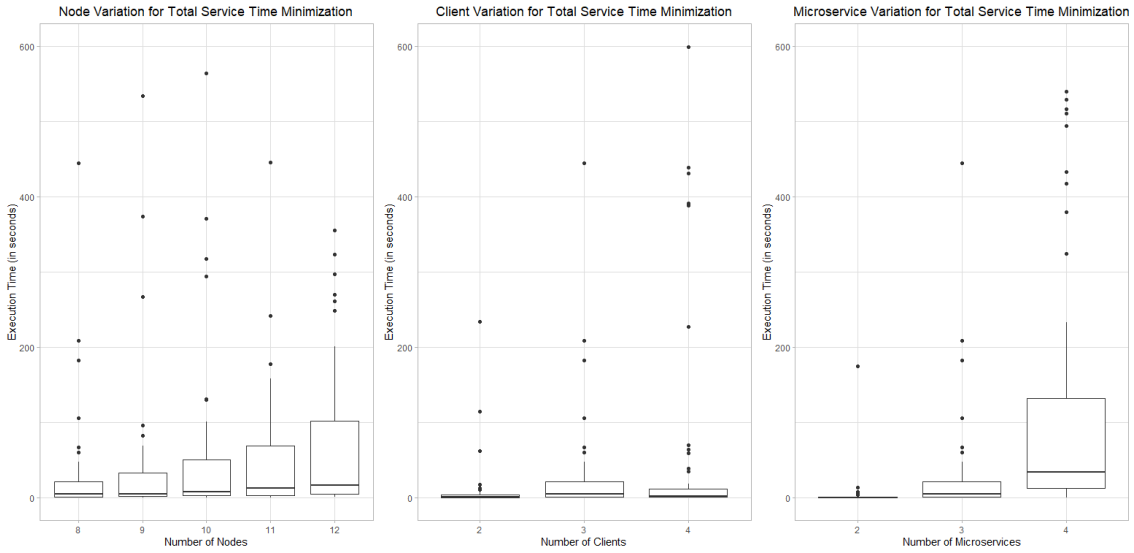

Figure 4.5: Execution times of 50 different feasible configurations with different number of nodes, clients and microservices for Total Service Time Minimization

After the second stage of tests, compared to the cost minimization, the plots show a notable increase in execution times and a more dispersed box plot distribution throughout the different node structures. In contrast to the cost optimization, the number of configurations during the node variation tests over 10 minutes rose in all structures nearing and, in some cases, matching ten configurations beyond the time limit. So, in this optimization in every node structure, the model took more time to solve the configurations while having a sizeable number of configurations above the 10-minute mark, making this optimization less prone to run more complex node structures with this time limit.

The number of clients and microservices present two different scenarios; the impact of the number of clients was almost nonexistent when there was a variation; in fact, the box plot from three clients to four clients reduces its mean and disparity. On the other hand, the number of microservices makes the resolution times peak when increasing from three to four microservices, making it the structure with the highest mean of all the structures tested in this optimization. These two types

of scenarios can be explained through the impact of microservices and nodes in each solution; contrary to the cost minimization, the difference this time is that the model goes through many solutions and keeps updating them until finding the minimal solution that fulfils the lowest total service time, during this process of finding and updating solutions, structures with a higher number of nodes and microservices take longer to meet the constraints in each solution rather than structures with more clients. This becomes visible in this optimization because there are cases where the model goes through thousands or more solutions to find the minimal solution.

4.2.1.4 Optimal Pareto

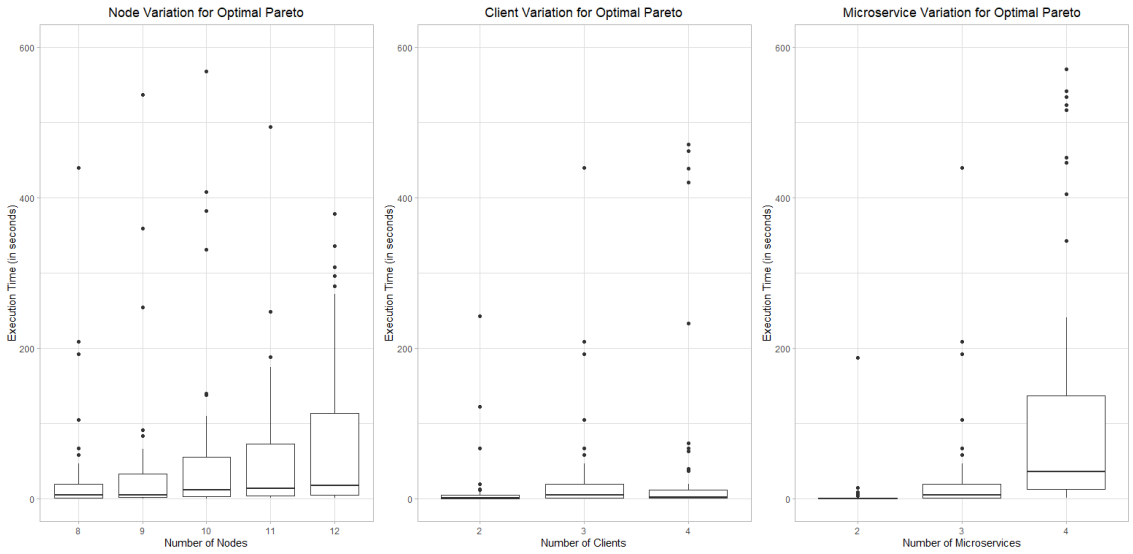

Figure 4.6: Execution times of 50 different feasible configurations with different number of nodes, clients and microservices for Optimal Pareto

As expected, many results from this optimization's second stage of tests are similar to those from the total service time minimization due to the same configurations being solved and their implementations being alike. Different node structures in the optimal Pareto have similar distributions and almost the same number of configurations above the 10-minute mark; from the 6-node structure to the 12-node structure, these values vary from six to ten configurations.

The same two scenarios occurred on microservice and client variations: the microservices had a considerable leap from three to four, and the clients had a slight decrease in disparity but a higher number of outliers from three to four. Also, identical to the previous optimization, while having two different outcomes when increasing clients and microservices from the control case, the number of

configurations above the 10-minute mark is the same on both and is equal to ten, making the model less liable to run structures with higher number of clients and microservices within the solve's time limit.

4.3 Real-Life Scenario

After testing the model in different scenarios and searching for the limits of the model when certain time limits were imposed, there was the need to validate some of the model's capabilities with a structure that could emulate a possible real-life cloud/edge environment. A structure of twenty nodes, five clients, and three microservices was tested while considering some of the model's limitations from the tests made in the previous stages. The number of nodes was one of the most flexible components to increase, and the clients are really impactful but way less impactful than the number of microservices. Also, there was the need to keep the number of microservices low so that the model could solve the problem and not get overwhelmed by the number of constraints and matrices created.

Below, we can find the graphs created with the solutions given through the model in the different optimizations tested in this final scenario.

The graphs show, on a 10x10 coordinated axis, the client's and nodes' geographical location, with blue points representing the clients and orange ones the computational nodes. The black arrows symbolize the initial requests made by the clients to the nodes to start processing the first microservice. After the first microservice is finished, the requests forward to the second microservice are represented by the green arrows, and the red arrows represent the requests forward from the second to the third microservice. Moreover, each arrow has a number that defines the number of requests to be processed.

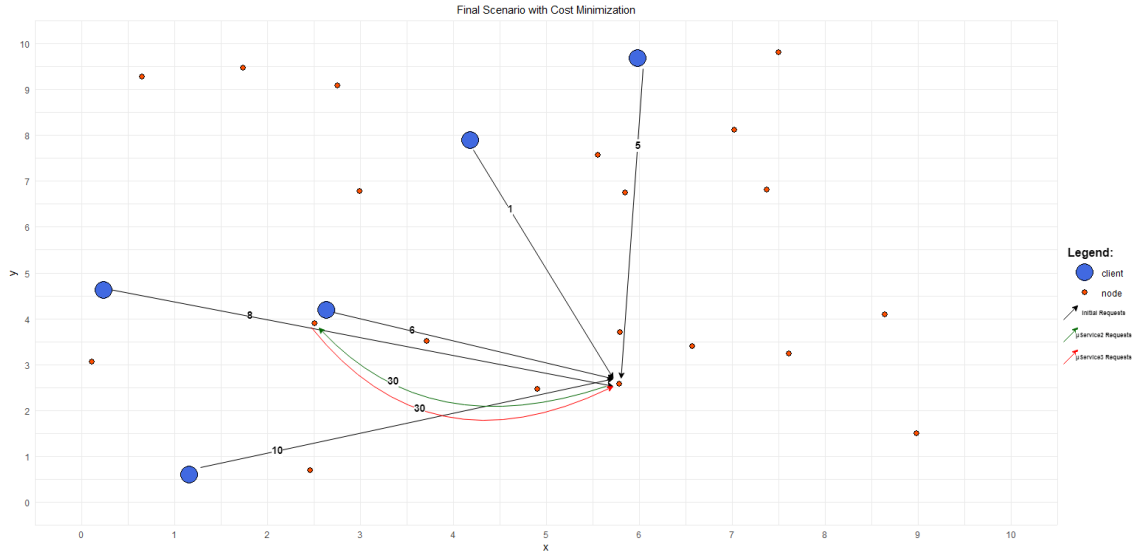

Figure 4.7: Solution graph of the final scenario with Cost Minimization

Figure 4.7 shows the flow of requests in the cost minimization using only two nodes, one node processing the first and third microservices and a second node processing the second microservice, given the requests from five different clients, keeping the cost minimal at 13 with a total service time of 12.97 units of time.

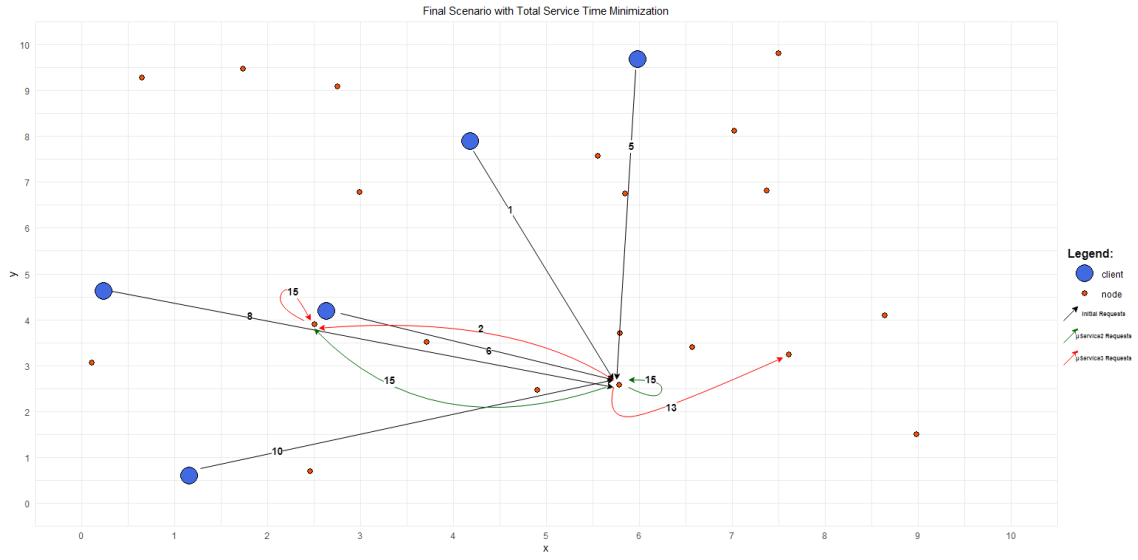

Figure 4.8: Solution graph of the final scenario with Total Service Time Minimization

The same scenario running with the total service minimization with a delta of 2, in figure 4.8, shows a different flow of requests in order to minimize the total service time, and the routes taken by some requests are different, in this case, the requests split themselves to process the second microservice into two

different nodes, and the same goes for the third microservice, making the cost of the solution higher at 15 with a lower total service time of 9.42 units of time.

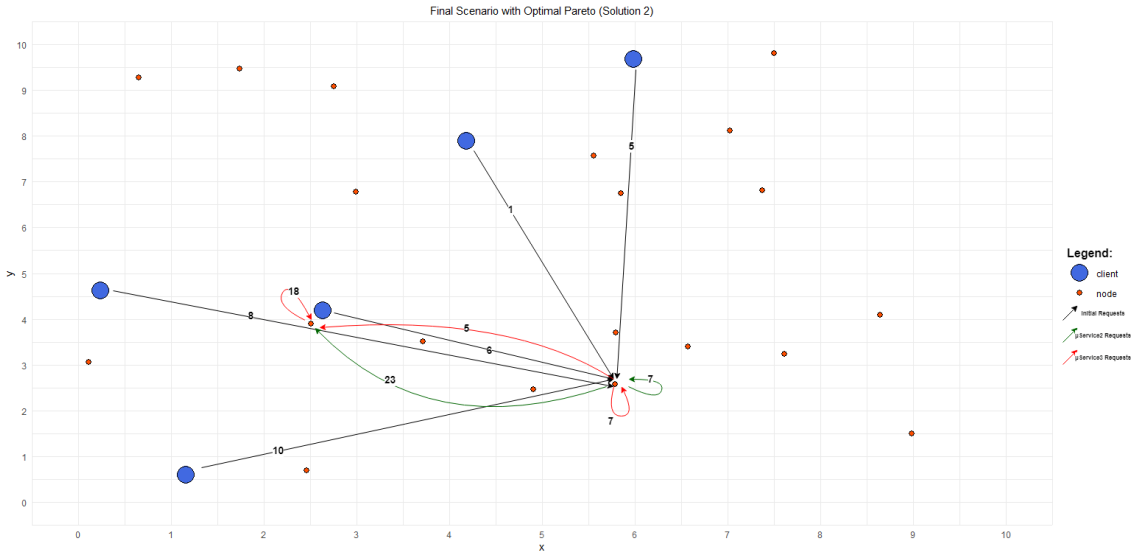

Figure 4.9: Solution graph of the final scenario with Optimal Pareto (solution 2)

The Optimal Pareto in this scenario, also executed with a delta of 2, presents three different solutions: one with the minimal cost at 13 with a total service time of 12.97 units of time 4.7, which is the same as the Cost minimization; another with the (minimal cost + 1) at 14 with a total service time of 9.80 units of time; and a last one, which is the same as the Total service minimization with (minimal cost + 2) at 15 with a lower total service time of 9.42 units of time 4.8. Throughout the figure 4.9, we can see the compromise between the total service time and cost, with a first graph showing the flow of requests entirely focused on cost minimization, a second one starting to process the third microservice inside the nodes where the second microservice was processed instead of passing around all of the requests to the node which processed the first microservice and a last graph showing a more focused total service time minimization with the third microservice in one of the nodes being processed on a third node to reach the lowest possible total service.

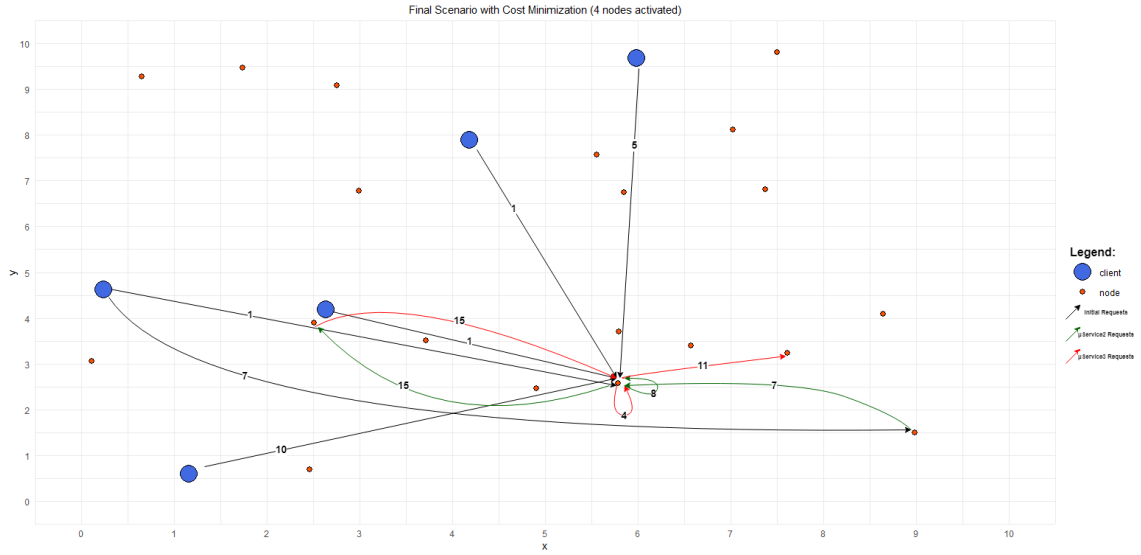

Figure 4.10: Solution graph of the final scenario with Cost Minimization (4 nodes activated)

A final execution was made to minimize the cost, and an extra constraint was added to the model. This additional constraint demanded the model to activate four nodes by applying one of the constraint reasoning approaches detailed in 3.21; the purpose was to show the flow of requests and the cost of the solution while forcing four different nodes to be used. As shown in figure 4.10, the solution uses four different nodes, increasing the cost of its solution from the initial 13 to 16 with a total service time of 13.05 units of time. The flow of requests seems more diverse due to the solver's priority of reaching the lowest cost possible configuration.

Lastly, the resolution time of all of these optimizations was also kept; the cost minimization was the quickest, with 4 minutes, due to its minimal cost; the total service time minimization and the optimal Pareto took 44 minutes and 48 minutes, respectively, as a consequence for evaluating every single solution from 13 to 15. After introducing a new constraint into the model, the cost minimization took a significant leap in the resolution time from 4 to 56 minutes because the cost increased from 13 to 16.

5

Conclusions

In this dissertation, a constraint model was developed to improve resource allocation and the flow of incoming client requests to run an application split into microservices inside a heterogeneous cloud/edge environment.

Therefore, with constraint programming, the model aimed to automatically find a solution, using available information passed down through a file about nodes, clients, and microservices to better optimize the overall resources represented in the model as the cost of the solution. Aside from the cost optimization, during the implementation, time variables, such as the total service time and the maximum total service time per client, were seen as necessary to optimize as well. Although they can be optimized through the model, their implementations depend highly on minimal-cost solutions.

The evaluation of the first stage results in 4.2 showed the boundaries where the model could give a solution within one minute while having a ceiling imposed on all the components tested. The second stage of results highlights the cost's impact on the model's resolution time shown in 4.3, making it clear that different configurations within the same structure with higher activation cost and replica deployment costs are prone to take longer to solve. Furthermore, the separate variation of components in the control case chosen in the second stage helps us understand the influence of each component on the model's solving time. However, after the first two stages of tests, we concluded that the structures run by the model were only a fraction of how a cloud/edge scenario would look like, so a final example was tested to validate this model's implementation to a more approximate real-life cloud/edge environment.

The last example tested with the model wanted to emphasize one of the core themes of this dissertation, which is the heterogeneity inside the cloud/edge environments. As such, the structure used had a huge discrepancy between the number of nodes and the other two components to promote various types of

computational nodes existing in a cloud/edge environment. In the scope of the problem, both the microservices and clients can be grouped up; this means that the number of microservices that split the application can be reduced by grouping them. A lower number of microservices boosts the chance of making the model solve the structure by not overwhelming it with all of the extra constraints and variables that a higher number of microservices demands; the same can be applied to the number of clients, with each one representing a group of clients in the same geographical region, if we broaden that geographical region we can group up other groups of clients reducing the general number of clients passed to the model. On the other hand, for the study of our model's applicability to heterogeneous environments, the computational nodes do not make sense to be grouped up since we want to replicate a diversified cloud/edge environment. Additionally, introducing a constraint on this final example and its solution shows the model's adaptability to upcoming constraints, making it a flexible blueprint for future implementations.

This implemented model has shown that through a constraint reasoning approach, there is a way to optimize resources in heterogeneous cloud/edge infrastructures. However, the model's performance is highly dependent on the number of microservices, the number of clients and the cost of the solution.

5.1 Future Work and System Limitations

Regarding the constraint model, there is always room for improvement and innovation. The model has some limitations, such as running the microservices sequentially, consequently reducing the number of scenarios in which the microservices could be processed inside the cloud/edge scenarios. The search strategy can be inefficient when the cost passed on to the model is really high. Lastly, optimizing time variables would be more efficient if the model could directly manipulate those values, saving time for those optimizations instead of sometimes evaluating hundreds or thousands of solutions.

Future work would be extracting the full potential of CHOCO's solver library, which supports the IBEX constraint library, specializing in real domains, and producing a constraint model where real domain variables and integer domain variables work together. Additionally, with this implementation, a different search strategy could be implemented to efficiently solve the model with other optimizations like the total service time minimization, optimal Pareto or other future optimizations.

Bibliography

[1] URL: https://courses.washington.edu/inde411/QueueingTheoryPart5.pdf (cit. on p. 20).

[2] K. Apt. *Principles of constraint programming*. Cambridge university press, 2003 (cit. on p. 11).

[3] M. Armbrust et al. "A View of Cloud Computing". In: *Commun. ACM* 53 (2010-04), pp. 50–58. DOI: 10.1145/1721654.1721672 (cit. on p. 8).

[4] M. Asim et al. "A Review on Computational Intelligence Techniques in Cloud and Edge Computing". In: *IEEE Transactions on Emerging Topics in Computational Intelligence* 4.6 (2020), pp. 742–763. DOI: 10.1109/TETCI.2020.3007905 (cit. on p. 8).

[5] F. M. Aymerich, G. Fenu, and S. Surcis. "An approach to a Cloud Computing network". In: *2008 First International Conference on the Applications of Digital Information and Web Technologies (ICADIWT)*. 2008, pp. 113–118. DOI: 10.1109/ICADIWT.2008.4664329 (cit. on p. 7).

[6] A. Balalaie, A. Heydarnoori, and P. Jamshidi. "Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture". In: *IEEE Software* 33.3 (2016), pp. 42–52. DOI: 10.1109/MS.2016.64 (cit. on p. 9).

[7] N. Beldiceanu. "Global Constraints as Graph Properties on a Structured Network of Elementary Constraints of the Same Type". In: *Principles and Practice of Constraint Programming – CP 2000*. Ed. by R. Dechter. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 52–66. ISBN: 978-3-540-45349-9 (cit. on p. 13).

[8] F. Benhamou et al. "Revising Hull and Box Consistency". In: *International Conference on Logic Programming*. 1999 (cit. on p. 12).

[9] C. Bessiere and J.-C. Régin. "MAC and Combined Heuristics: Two Reasons to Forsake FC (and CBJ?) on Hard Problems". In: *International Conference on Principles and Practice of Constraint Programming*. 1996 (cit. on p. 12).

[10] F. Bonomi et al. "Fog Computing: A Platform for Internet of Things and Analytics". In: *Big Data and Internet of Things*. 2014 (cit. on p. 6).

[11] P. J. Burke. "The Output of a Queuing System". In: *Operations Research* 4.6 (1956), pp. 699–704. DOI: 10.1287/opre.4.6.699. eprint: https://doi.org/10.1287/opre.4.6.699. URL: https://doi.org/10.1287/opre.4.6.699 (cit. on p. 19).

[12] F. Callegati et al. "Data Security Issues in MaaS-enabling Platforms". In: *International Forum on Research and Technologies for Society and Industry*. Bologna, Italy, 2016-09. URL: https://hal.inria.fr/hal-01336700 (cit. on p. 11).

[13] E. Carvalho, J. Cruz, and P. Barahona. "Reasoning with Uncertainty in Continuous Domains". In: *Integrated Uncertainty Management and Applications*. Ed. by V.-N. Huynh et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 357–369. ISBN: 978-3-642-11960-6. DOI: 10.1007/978-3-642-11960-6_34. URL: https://doi.org/10.1007/978-3-642-11960-6_34 (cit. on p. 12).

[14] G. Carvalho et al. *Edge computing: Current trends, research challenges and future directions - computing*. 2021-01. URL: https://link.springer.com/article/10.1007/s00607-020-00896-5 (cit. on p. 6).

[15] C. H. Costa et al. "Sharding by Hash Partitioning - A Database Scalability Pattern to Achieve Evenly Sharded Database Clusters". In: *International Conference on Enterprise Information Systems*. 2015 (cit. on pp. 9, 10).

[16] A. V. Dastjerdi et al. "Fog Computing: Principles, Architectures, and Applications". In: *ArXiv* abs/1601.02752 (2016) (cit. on p. 5).

[17] R. Dechter, D. Cohen, et al. *Constraint processing*. Morgan Kaufmann, 2003 (cit. on p. 11).

[18] E. C. Freuder. "Synthesizing Constraint Expressions". In: *Commun. ACM* 21.11 (1978-11), pp. 958–966. ISSN: 0001-0782. DOI: 10.1145/359642.359654. URL: https://doi.org/10.1145/359642.359654 (cit. on p. 13).

[19] I. P. Gent et al. "An empirical study of dynamic variable ordering heuristics for the constraint satisfaction problem". In: *Principles and Practice of Constraint Programming — CP96*. Ed. by E. C. Freuder. Berlin, Heidelberg:

Springer Berlin Heidelberg, 1996, pp. 179–193. ISBN: 978-3-540-70620-5 (cit. on p. 12).

[20] S. W. Golomb and L. D. Baumert. "Backtrack Programming". In: *J. ACM* 12.4 (1965-10), pp. 516–524. ISSN: 0004-5411. DOI: 10.1145/321296.321300. URL: https://doi.org/10.1145/321296.321300 (cit. on p. 12).

[21] W. D. Harvey. *Nonsystematic backtracking search*. stanford university, 1995 (cit. on p. 12).

[22] S. Heipcke. "Comparing constraint programming and mathematical programming approaches to discrete optimisation—the change problem". In: *Journal of the Operational Research Society* 50.6 (1999), pp. 581–595. DOI: 10.1057/palgrave.jors.2600730. eprint: https://doi.org/10.1057/palgrave.jors.2600730. URL: https://doi.org/10.1057/palgrave.jors.2600730 (cit. on p. 11).

[23] F. Hillier and G. Lieberman. *Introduction to operations research*. 2001 (cit. on pp. 15–17).

[24] S. Kim et al. "High-assurance synthesis of security services from basic microservices". In: *14th International Symposium on Software Reliability Engineering, 2003. ISSRE 2003.* (2003), pp. 154–165 (cit. on p. 11).

[25] G. Kondrak and P. van Beek. "A theoretical evaluation of selected backtracking algorithms". In: *Artificial Intelligence* 89.1 (1997), pp. 365–387. ISSN: 0004-3702. DOI: https://doi.org/10.1016/S0004-3702(96)00027-6. URL: https://www.sciencedirect.com/science/article/pii/S0004370296000276 (cit. on p. 12).

[26] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/master/template.pdf (cit. on p. ii).

[27] A. K. Mackworth. "Consistency in networks of relations". In: *Artificial intelligence* 8.1 (1977), pp. 99–118 (cit. on pp. 12, 13).

[28] K. Marriott and P. Stuckey. *Programming with Constraints - An Introduction*. English. United States of America: The MIT Press, 1998. ISBN: 0 262 13341 5 (cit. on p. 11).

[29] B. A. Martin et al. "OpenFog security requirements and approaches". In: *2017 IEEE Fog World Congress (FWC)*. IEEE. 2017, pp. 1–6 (cit. on p. 6).

[30] P. Mell and T. Grance. *The NIST definition of cloud computing*. 2011-09. URL: https://csrc.nist.gov/publications/detail/sp/800-145/final (cit. on p. 4).

[31] R. E. Moore. *Interval analysis*. Vol. 4. Prentice-Hall Englewood Cliffs, 1966 (cit. on p. 12).

[32] S. Newman. *Building microservices*. "O'Reilly Media, Inc.", 2021 (cit. on p. 10).

[33] F. Oliveira et al. "Delivering software with agility and quality in a cloud environment". In: *IBM Journal of Research and Development* 60 (2016-03), 10:1–10:11. DOI: 10.1147/JRD.2016.2517498 (cit. on p. 9).

[34] C.-G. Quimper and T. Walsh. "Beyond Finite Domains: The All Different and Global Cardinality Constraints". In: *Principles and Practice of Constraint Programming - CP 2005*. Ed. by P. van Beek. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 812–816. ISBN: 978-3-540-32050-0 (cit. on p. 13).

[35] H. El-Sayed et al. "Edge of Things: The Big Picture on the Integration of Edge, IoT and the Cloud in a Distributed Computing Environment". In: *IEEE Access* 6 (2018), pp. 1706–1717 (cit. on p. 4).

[36] W. Shi et al. "Edge Computing: Vision and Challenges". In: *IEEE Internet of Things Journal* 3.5 (2016), pp. 637–646. DOI: 10.1109/JIOT.2016.2579198 (cit. on pp. 4, 5, 7).

[37] Y. Sun, S. Nanda, and T. Jaeger. "Security-as-a-Service for Microservices-Based Cloud Applications". In: *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)* (2015), pp. 50–57 (cit. on p. 11).

[38] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. Cambridge, MA, USA: MIT Press, 1989. ISBN: 0262081814 (cit. on p. 11).

[39] M. Villamizar et al. "Cost comparison of running web applications in the cloud using monolithic, microservice, and AWS Lambda architectures". In: *Service Oriented Computing and Applications* 11 (2017), pp. 233–247 (cit. on pp. 9, 10).

[40] *What is microservices? microservices definition and related faqs*. 2022-05. URL: https://avinetworks.com/glossary/microservice/ (cit. on p. 9).

A

Code

```
1
2       //Constants
3       int nodes = data.getNodes();
4    int clients = data.getClients();
5    int microservices = data.getMicroservices();
6    int[] clientRequests = data.getClientRequests();
7    int[][] nodeSpecs= data.getNodeSpecs();
8    int[][] microservicesSpecs = data.getMicroservicesSpecs();
9    int[][] costs= data.getCosts();
10      double[][] nodeTimes =  data.getNodeTimes() ;
11   double[][] latency = data.getLatency();
12
13   // Model and Solver
14   Model md = new Model("Thesis");
15   Solver sv = md.getSolver();
16
17   // Decision Variables
18      IntVar[][] x = md.intVarMatrix("X", nodes, microservices, 0,
   data.maxNodeCores());
19    IntVar[][] y = md.intVarMatrix("Y", clients, nodes, 0, data.
   totalRequests());
20    List<IntVar[][]> zS = new ArrayList<IntVar[][]>();
21
22    for(int i = 0; i < microservices -1 ;i ++) {
23      IntVar[][] z = md.intVarMatrix("Z", nodes, nodes, 0, data.
   totalRequests());
24      zS.add(z);
25    }
26
27    //Dependent Variables
28    IntVar[] m = m = md.intVarArray("m", nodes, 0, data.maxNodeCores()
   *data.maxNodeMemory());
```

```
29      IntVar[] a = a = md.intVarArray("a", nodes, 0, data.
    maxMicroserviceActivationCost());
30      IntVar[] c = md.intVarArray("c", nodes, 0, data.
    maxMicroserviceCost());
31      IntVar[] s = md.intVarArray("s", nodes, 0, data.
    maxMicroserviceActivationCost() + data.maxMicroserviceCost());
32      IntVar[][] l = md.intVarMatrix("l", nodes, microservices, 0,  data
    .totalRequests());
33      IntVar C = md.intVar("C", 0, (data.maxMicroserviceActivationCost()
     + data.maxMicroserviceCost())*microservices);
34      double[][] probabilityMatrix = new double[nodes][microservices];
35      double[][] waitingTime = new double[nodes][microservices];
36      double[][] serviceTime = new double[nodes][microservices];
37          double[][] utilizationRatio = new double[nodes][microservices
    ];
38      double[] timePerClient = new double[clients];
39      double totalServiceTime = 0.0;
40
41
42
43
```

Listing A.1: Variables and Constants

```
1  // Auxilary Variables
2       x_row = md.intVarArray("x_row", nodes, 0, microservices*data.
    maxNodeCores());
3      for (int i = 0; i < nodes; i++) {
4        md.sum(x[i], "=", x_row[i]).post();
5      }
6
7      x_col = md.intVarArray("x_col", microservices, 0, nodes*data.
    maxNodeCores());
8      for (int i = 0; i < microservices; i++) {
9        md.sum(getColumn(x, i), "=", x_col[i]).post();
10     }
11
12     y_row = md.intVarArray("y_row", clients, 0, nodes*data.
    totalRequests());
13     for (int i = 0; i < clients; i++) {
14       md.sum(y[i], "=", y_row[i]).post();
15     }
16     y_col = md.intVarArray("y_col", nodes, 0,clients*data.
    totalRequests());
17     for (int i = 0; i < nodes; i++) {
18       md.sum(getColumn(y, i), "=", y_col[i]).post();
```

```
19      }
20
21      zSRow = new ArrayList<IntVar[]>();
22      for(int k =0;k < microservices -1; k++) {
23
24        IntVar[][] currentZ = zS.get(k);
25        IntVar[] z_row = md.intVarArray("z_row", nodes, 0, data.
    totalRequests());
26
27        for (int i = 0; i < nodes; i++) {
28          md.sum(currentZ[i], "=", z_row[i]).post();
29        }
30        zSRow.add(z_row);
31      }
32
33      zSCol = new ArrayList<IntVar[]>();
34      for(int k =0;k < microservices -1; k++) {
35
36        IntVar[][] currentZ = zS.get(k);
37        IntVar[] z_col = md.intVarArray("z_col", nodes, 0, data.
    totalRequests());
38
39        for (int i = 0; i < nodes; i++) {
40          md.sum(getColumn(currentZ, i), "=", z_col[i]).post();
41        }
42        zSCol.add(z_col);
43      }
44
45      l_row = md.intVarArray("l_row", nodes, 0, microservices*data.
    totalRequests());
46      for (int i = 0; i < nodes; i++) {
47        md.sum(l[i], "=", l_row[i]).post();
48      }
49      l_col = md.intVarArray("l_col", microservices, 0, nodes*data.
    totalRequests());
50      for (int i = 0; i < microservices; i++) {
51        md.sum(getColumn(l, i), "=", l_col[i]).post();
52      }
53
```

Listing A.2: Auxiliary Variables

```
1    // -----------------------(3.3)-----------------------
2
3    for (int i = 0; i < nodes; i++) {
4      md.arithm(x_row[i], "<=", nodeSpecs[i][2]).post();
5    }
6
7    // -----------------------(3.4)-----------------------
8
9    for (int i = 0; i < nodes; i++) {
10     md.scalar(x[i], microservicesSpecs[0], "=", m[i]).post();
11     md.arithm(m[i], "<=", nodeSpecs[i][0]).post();
12   }
```

Listing A.3: Allocation Constraints

```
1    // -----------------------(3.5)-----------------------
2        for (int i = 0; i < nodes; i++) {
3      md.ifThenElse(md.arithm(y_col[i], ">", 0), md.arithm(x[i][0], ">
     ", 0), md.arithm(x[i][0], "=", 0));
4    }
5
6    for (int i = 0; i < nodes; i++) {
7      md.ifThenElse(md.arithm(x_row[i], ">", 0), md.arithm(a[i], "=",
     costs[0][i]), md.arithm(a[i], "=", 0));
8    }
9
10   // -----------------------(3.6) and (3.7)-----------------------
11
12   for (int i = 0; i < nodes; i++) {
13     md.scalar(x[i], getCosts(costs, i), "=", c[i]).post();
14     md.arithm(a[i], "+", c[i], "=", s[i]).post();
15   }
16
17   md.sum(s, "=", C).post();
```

Listing A.4: Cost Constraints

```
1
2    // -----------------------(3.8)-----------------------
3
4    for (int i = 0; i < clients; i++) {
5      md.arithm(y_row[i], "=", clientRequests[i]).post();
6    }
7
8        // -----------------------(3.9) and  (3.10)
     -----------------------
9    for(int k = 0; k < zSRow.size();k++) {
```

```
      IntVar[] currentZRow = zSRow.get(k);
      for (int i = 0; i < nodes; i++) {
        if(k==0) {
          md.arithm(currentZRow[i], "=", y_col[i]).post();
        }
        else {
          IntVar[] lastZCol = zSCol.get(k-1);
          md.arithm(currentZRow[i], "=", lastZCol[i]).post();
        }
      }

     }

    for(int k = 0; k < zSCol.size();k++) {
      IntVar[] currentZCol = zSCol.get(k);
      for (int i = 0; i < nodes; i++) {
        md.ifThenElse(md.arithm(currentZCol[i], ">", 0), md.arithm(x[i
    ][k+1], ">", 0), md.arithm(x[i][k+1], "=", 0));
      }
     }



```

Listing A.5: Flow Constraints

```
// ----------------------(3.11)-----------------------
    u = new int[nodes][microservices];
    for (int i = 0; i < nodes; i++)
      for (int j = 0; j < microservices; j++)
        u[i][j] = nodeSpecs[i][1] * microservicesSpecs[1][j];


// ----------------------(3.12)-----------------------

    for(int i = 0; i < nodes; i++) {
      IntVar[] firstRowZ = zSRow.get(0);
      md.arithm(l[i][0], "=", firstRowZ[i]).post();
    }

// ----------------------(3.13)-----------------------

        for(int k = 0; k < zSCol.size();k++) {
      IntVar[] currentZCol = zSCol.get(k);
      for(int i = 0; i < nodes; i++) {
        md.arithm(l[i][k+1], "=", currentZCol[i]).post();
```

```
21        }
22      }
23
24 // ----------------------(3.14)-----------------------
25
26  private double[][] fillingUtilizationRatio(IntVar[][] x,IntVar[][] l,
    int[][] u,int nodes,int microservices) {
27
28      double[][] utilizationRatio = new double[nodes][microservices];
29      for(int i = 0; i< nodes ;i++) {
30        for(int j = 0;j < microservices; j++) {
31          if(x[i][j].getValue() > 0) {
32            utilizationRatio[i][j] =  (double) l[i][j].getValue() / (u[i
    ][j] * x[i][j].getValue());
33          }
34          else {
35            utilizationRatio[i][j] = 0.0;
36          }
37      }
38      }
39
40      return utilizationRatio;
41 }
42
43 // ----------------------(3.15)-----------------------
44
45      for (int i = 0; i < nodes; i++) {
46        md.ifThen(md.arithm(x[i][0], ">", 0), md.arithm(md.intScaleView(
    x[i][0], u[i][0]), ">", y_col[i]));
47      }
48
49          for(int k = 0; k < zSCol.size();k++) {
50        IntVar[] currentZCol = zSCol.get(k);
51        for (int i = 0; i < nodes; i++) {
52          md.ifThen(md.arithm(x[i][k+1], ">", 0), md.arithm(md.
    intScaleView(x[i][k+1], u[i][k+1]), ">", currentZCol[i]));
53        }
54       }
55
56 // ----------------------(3.16)-----------------------
57
58 private double[][] fillingWaitingTimeMatrix(double[][]
    probabilityMatrix, IntVar[][] x, IntVar[][] l, int[][] u,int nodes,
    int microservices) {
59
```

```
60      double[][] waitingTime = new double[nodes][microservices];
61      for(int i = 0; i< nodes ;i++) {
62      for(int j = 0;j < microservices; j++) {
63
64          if(x[i][j].getValue() > 0) {
65            double part1 = probabilityMatrix[i][j] * Math.pow(l[i][j].
    getValue(),x[i][j].getValue());
66            double part2 = (double) factorial(x[i][j].getValue()-1);
67            double part3 = Math.pow(u[i][j],x[i][j].getValue()-1);
68            double part4 = Math.pow(((x[i][j].getValue()*u[i][j]) - l[i
    ][j].getValue()),2);
69            double part5 = (double) 1 / u[i][j];
70
71            waitingTime[i][j] = ( (double) part1 / (part2*part3*part4))
    + part5;
72          }
73          else {
74            waitingTime[i][j] = 0.0;
75          }
76          }
77      }
78
79      return waitingTime;
80  }
81
82  private double[][] fillingProbMatrix(IntVar[][] x, IntVar[][] l, int
    [][] u,int nodes,int microservices) {
83
84      double[][] probabilityMatrix = new double[nodes][microservices];
85
86      for(int i = 0; i< nodes;i++) {
87      for(int j = 0;j < microservices; j++) {
88        double auxSum = 0;
89        if(x[i][j].getValue() > 0) {
90
91        for(int n = 0; n < x[i][j].getValue() ; n++) {
92
93            double div = (double) l[i][j].getValue() / u[i][j];
94
95            double part1 = (double) Math.pow(div,n) / factorial(n);
96
97            double part2 =  (double) Math.pow(div, x[i][j].getValue()) /
     factorial(x[i][j].getValue());
98
```

```
99            double div2 = (double) l[i][j].getValue() / (x[i][j].
    getValue() * u[i][j]);
100
101           double part3 =  1 / (1 - div2);
102
103           auxSum += part1 + (part2 * part3);
104         }
105         double prob = (double)  1 / auxSum;
106
107         if(prob < 0)
108             probabilityMatrix[i][j] = 0;
109         else
110             probabilityMatrix[i][j] = prob;
111       }
112       else {
113         probabilityMatrix[i][j] = 0;
114       }
115
116     }
117     }
118 }
```

Listing A.6: Queue Constraints

```
1
2 // ----------------------(3.18)-----------------------
3
4 private double[][] fillingServiceTimeMatrix(double[][] waitingTime,
    IntVar[][] z, double[][] nodeTimes,int nodes) {
5
6     double[][] serviceTime = new double[nodes][nodes];
7
8   for(int k = 0; k < nodes;k++) {
9       serviceTime[k][1] = waitingTime[k][1];
10  }
11
12  for(int i = 0; i< nodes;i++) {
13      double bigSum = 0.0;
14      for(int m = 0; m< nodes; m++) {
15        if(z[i][m].getValue() > 0) {
16          double part1 = z[i][m].getValue();
17          double part2 = 0.0;
18
19          for(int k = 0; k < nodes; k++) {
20              part2 += z[i][k].getValue();
21          }
```

```
22
23            double part3 = nodeTimes[i][m] + serviceTime[m][1];
24
25            double part4 = (double) part1 / part2;
26
27            bigSum += part4 * part3;
28          }
29
30        }
31      serviceTime[i][0] = waitingTime[i][0] + bigSum;
32    }
33  }
34
35  // ----------------------(3.19)-----------------------
36
37  private double[] fillingTimePerClient(IntVar[][] y, double[][] latency
        , double[][] serviceTime, int clients, int nodes) {
38
39      double[] timePerClient = new double[clients];
40
41    for(int i = 0; i< clients;i++) {
42      double bigSum = 0.0;
43      for(int m = 0; m < nodes; m++) {
44
45        double part1 = y[i][m].getValue();
46        double part2 = 0.0;
47
48        for(int k = 0; k < nodes; k++) {
49          part2 += y[i][k].getValue();
50        }
51
52        double part3 = latency[i][m] + serviceTime[m][0];
53        double part4 = (double) part1 / part2;
54
55        bigSum += part4 * part3;
56
57      }
58
59      timePerClient[i] = bigSum;
60
61    }
62
63      return timePerClient;
64  }
65
```

```
66 // ----------------------(3.20)-----------------------
67
68 private double calculateTotalServiceTime(double[] timePerClient, int[]
       clientRequests,int clients) {
69
70     double totalServiceTime = 0
71     for(int i = 0; i< clients; i++) {
72
73     double part1 = clientRequests[i];
74     double part2 = 0;
75
76     for(int k = 0; k < clients; k++) {
77         part2 += clientRequests[k];
78     }
79
80     double part3 = timePerClient[i];
81     double part4 = (double) part1 / part2;
82
83     totalServiceTime += part4 * part3;
84
85     }
86
87     return totalServiceTime;
88
89 }
```

Listing A.7: Time Constraints