Cloud Service Orchestration Using Constraint Programming

ANESTOS NIKOLAOS-EKTORAS
Cloud Service Orchestration Using Constraint Programming

Anestos Nikolaos-Ektoras

Master of Science Thesis
Software Engineering of Distributed Systems
School of Information and Communication Technology
KTH Royal Institute of Technology

September 2016, Stockholm, Sweden

Examiner: Prof. Christian Schulte, KTH
Academic Supervisor: Dr. Mats Carlsson, SICS
Supervisor: Dr. Calin Curescu, Ericsson Research

TRITA-ICT-EX-2016:106
Abstract

Cloud applications and services are frequently built using multiple tiers and current trends such as micro-services further increase componentization, allowing us to place each component in a different physical machine in a distributed cloud. Ericsson owns and manages very large networks, which offer diverse infrastructure in terms of computational power, storage but most importantly position in the network. Typically, a machine which is closer to the edge of the network (closer to the end user) will have limited resources but it will offer less latency, for a higher price.

At the same time, several enterprise/industrial areas expect to benefit from the cloud business model in a large-scale distributed environment. These types of applications have very diverse end-2-end Service-Level Agreements (SLA) to be fulfilled, while at the same time the cloud environment needs to optimize processing, storage, and networking costs. Moreover, customers might want to change and adjust SLAs/requirements themselves using self-management portals.

The objective of this project is to model the network and services offered by Ericsson. Then, given the SLA, finding a valid solution of the problem, using a constraint solver. A solution is a set of physical machines that host the components the required service is composed from. This approach has many challenges since the same service can be composed from different sets of components. The connected components form a connectivity graph, where nodes in the graph are connected by physical links. But, since the connection is described by higher level components (composed by simpler components), this graph can also be expressed as a tree. Leaves in the tree are the nodes that compose the higher-level services and the ones that must be hosted in the infrastructure. The characteristics of each leaf-node depend on its parent and/or siblings in the component tree. Finally, since the components are normally connected, the physical connection between nodes in the network must be taken into consideration.

The proposed model is evaluated in several cases, in order to identify how the number of the software components and the infrastructure topology affect the solution finding. The results are promising, showing fast resolution of the problem instances, varying for each test case, from a few seconds to a couple of minutes.

Keywords
Distributed Systems, Constraint Programming, Cloud Services, Configuration
**Abstrakt**

Molnapplikationer och tjänster är ofta byggda med flera nivåer och nuvarande trender såsom mikro-tjänster ökar ytterligare komponentiseringen, vilket tillåter oss att placera varje komponent i en annan fysisk maskin på ett distribuerat moln. Ericsson äger och förvaltar väldigt stora nätverk som erbjuder varierande infrastruktur när det gäller beräkningskraft, lagring och framför allt position i nätverket. Typiskt kommer en maskin som är närmare kanten av nätet (närmare slutanvändaren) att ha begränsade resurser, men det kommer att erbjuda mindre latens till ett högre pris.

Samtidigt räknar flera företag / industriområden med att dra nytta av moln affärsmodelltjänster i en storskalig och distribuerad miljö. Den här typen av applikationer har väldigt olika end-to-end varierande servicenivåavtal (SLA) som skall uppfyllas, medan moln miljön behöver optimera bearbetnings, lagrings och nätverks kostnader. Dessutom, kan kunden komma att vilja ändra SLA / krav själva med hjälp av självhantering portaler.


Den föreslagna modellen utvärderas i flera fall, för att identifiera hur antalet programvarukomponenter och infrastrukturens topologi påverkar resultatet av lösningen. Resultaten är lovande och visar snabb lösning av problemets instanser, varierande för varje testfall, från några sekunder till ett par minuter.

**Nyckelord**

Distribuerade system, Constraint Programming, molntjänster, konfiguration
Acknowledgements

I would like to thank my supervisor at Ericsson, Calin Curescu, for regarding me as a candidate that could live up to the demands of this project, but also for his guidance and support throughout the course of the thesis. Likewise, I would like to thank my academic examiner at KTH Christian Schulte and my academic supervisor Mats Carlsson for their priceless inspiration and feedback in the writing of this report. Finally, I want to thank my father, mother and brother for their continuous support during my studies.
Contents

1 Introduction ........................................................................................................................................ 19
  1.1 Overview ................................................................. 19
  1.2 Problem ......................................................................... 19
  1.3 Purpose ......................................................................... 20
  1.4 Goal ............................................................................ 20
  1.5 Methodology .................................................................. 20
  1.6 Benefits, Ethics and Sustainability ................................. 21
    1.6.1 Benefits ................................................................. 21
    1.6.2 Ethics ...................................................................... 21
    1.6.3 Sustainability .......................................................... 21
  1.7 Delimitations .................................................................. 21
  1.8 Contributions ............................................................... 21
  1.9 Outline ......................................................................... 21

2 Background ................................................................................................................................. 24
  2.1 Constraint Programming .................................................. 24
    2.1.1 Constraint Satisfaction Problem .................................. 24
    2.1.2 Constraint Optimization Problem ................................ 24
    2.1.3 Variables .................................................................... 24
    2.1.4 Constraints .................................................................. 25
    2.1.5 Constraint Programming System .................................. 26
    2.1.6 Constraint Propagation & Search .................................. 26
  2.2 Example: Sudoku ............................................................ 27
    2.2.1 Modeling a Sudoku puzzle as a CSP ............................ 28
  2.3 Configuration .................................................................... 29

3 Problem Analysis ....................................................................................................................... 31
  3.1 Composing the end-2-end service ...................................... 31
  3.2 Deploying over the Distributed Cloud ................................. 32

4 Model ................................................................................................................................. 34
  4.1 Modeling the Catalog ....................................................... 34
6.2 Future work .....................................................................................................................58

References ................................................................................................................................61

Appendix A ..................................................................................................................................64

Appendix B ....................................................................................................................................70
List of Figures

Figure 2.1 A solved Sudoku puzzle ................................................................. 28
Figure 3.1 Two implementation of a service A ................................................... 32
Figure 3.2 Distributed Cloud Topology ............................................................. 32
Figure 3.1 Example of a Configuration Tree with $c = 2$ and $l = 2$ ................. 40
Figure 5.1 Model behavior when catalog increases in size ............................. 51
Figure 5.2 Model behavior when infrastructure increases in size ................... 52
Figure 5.3a Test case 3 - first scenario (sla1) ..................................................... 53
Figure 5.3b Test case 3 - second scenario (sla2) ............................................. 54
Figure 5.3c Test case 3 - third scenario (sla3) .................................................. 54
Figure 5.4a Test case 8 - first scenario (sla1) ................................................... 56
Figure 5.4a Test case 8 - second scenario (sla1) ............................................. 56
Figure 5.4a Test case 8 - third scenario (sla3) .................................................. 56
List of Tables

Table 5.1 Parameters for each test case .............................................................. 50
Table 5.2 Results for all test cases - scenarios .................................................. 53
Table 5.3 Results for all test cases – scenarios with BAB ................................. 55
1 Introduction

This chapter provides the reader with an introduction to the problem area and also significant data about this dissertation. Section 1.1 is an overview, introducing the reader to the area. Section 1.2 introduces the problem. Section 1.3 and 1.4 state the purpose and the goal of this project respectively. Section 1.5 briefly presents the methodology used during this project. Section 1.6 discusses the benefits, ethics and sustainability related to this project. Section 1.7 presents project's delimitations. Section 1.8 states this thesis contributions. Finally, Section 1.9 contains the outline for this dissertation.

1.1 Overview

5G (5th generation mobile networks) is the next generation of mobile telecommunication standards. Up until this point, all previous generations improved, or added some functionality to the existing standards. The 1st generation (1G) focused on voice, the 2nd generation (2G) added the functionality for text messaging, the 3rd generation (3G) allowed users to use mobile networks for data and the 4th generation (4G) improved those functionalities. 5G networks, while significantly increasing speeds, have many other requirements, such as high capacity, low latency and the support for any-to-any communication [1].

These types of networks will have a great impact on industry and society, allowing the development of applications that are not possible with current technologies, such as Internet of Things (IoT). New types of business opportunities will become possible for Telecommunication companies which will operate those networks. Telecommunication companies will be able to offer services, hosted in any part in their Distributed Cloud, to other companies or users.

This project requires a categorization of the services that can be offered and a model – mapping of the Distributed Cloud that can host any of them. More specifically, a list of software components (a catalog) that offer a service and a cloud network which consists of infrastructure nodes, connected with physical links are required. The nodes location varies, a node might be a server in a data center, located on a smaller, regional or metropolitan level data center, or even in a mobile base station. The model should be able to identify which software components (services) are needed to be deployed and where (which node in the network), based on a Service Level Agreement (SLA).

This thesis project was performed at the department of Cloud Technologies of Ericsson Research in Stockholm, Sweden.

1.2 Problem

Manual service creation and deployment works for a small set of well-known use cases, however the solution space of component and deployment choice on a distributed cloud un-
nder a varying set of SLAs grows exponentially. Thus this thesis aims to a combined orchestration, blending automated end-2-end service creation with an automated deployment. How does such a system behave when the size of catalog and network grow?

1.3 Purpose

The aim of this thesis is to come forward with an algorithm that is optimized to perform automatic decomposition and deployment of an end-to-end Cloud Service which is composed by several atomic functions.

More specifically, Constraint Programming methods are used to model the aforementioned problem and a solver framework is utilized, to optimize the solution finding.

1.4 Goal

The main goal of the degree project is to formulate a representation of Cloud Service Orchestration problems, in a common data structure. Transform this representation to a Constraint Programming model, allowing us to solve the problem, given an end-to-end Cloud Service. Furthermore, this project identifies how different search strategies affect the search space, and evaluates how the size of the search space (catalog) affects the solving process, for the selected solver.

1.5 Methodology

Initially, an in-depth literature study is carried out by the author, to identify the previous work in the area. Then, after decomposing the problem, it is mapped to a Constraint Satisfaction Problem (CSP). This model is implemented afterwards, in the Constraint Programming (CP) language that fulfills most of problems needs.

The experimental method is used in order to investigate how to model the problem and transform it to a Constraint Satisfaction Problem (CSP). This method is characterized by the process of seeking an optimal solution to the problem, by changing one variable at a time [2].

The evaluation of the implementation is performed using a Quantitative Research, measuring the performance of the proposed model using diverse experiments in respect to their parameters, in order to falsify or validate the hypothesis of the thesis. The deductive approach is normally coupled with Quantitative methods, since the theories can be confirmed by the data analysis.
1.6 Benefits, Ethics and Sustainability

1.6.1 Benefits

This thesis project aims to help automate the process of service deployment in a Distributed Cloud. This will help both customers and companies who offer these services. The procedure can be faster, more effective and not affected by human errors.

1.6.2 Ethics

This project conforms with the IEEE Code of Ethics [3]. Notably, the author is honest and realistic in making claims or estimates, always considering the available data. He does not accept any form of bribery and is fair with his colleagues and co-workers. Moreover, the author gives credit to the contribution of others whilst never claiming the work of others as his own. Finally, this project is trying to identify a solution for a technological problem, by examining a fitting usage of existing technologies, without ignoring potential consequences.

1.6.3 Sustainability

Modeling the Cloud Service Orchestration problem in Constraint Programming can be very important to companies that offer these types of services. From a sustainability point of view, it can accelerate or fully automate the process of selecting which services will compose the end-to-end service and where to host them in the distributed cloud. Also, the solution will not be prone to human errors. This can help businesses save both time and money but most importantly limit the spent resources which, even marginally, can improve in energy consumption.

1.7 Delimitations

The main complication in respect to this problem is that it has not been clearly formulated yet. The infrastructure for hosting cloud services already exists, but the specifications of the problem come from a theory of what can be achieved by using new technologies.

1.8 Contributions

The main contributions of this thesis, are the modeling of the Distributed Cloud infrastructure, the modeling of the service composition and deployment problem in Constraint Programming and the evaluation of the model considering various parameters.

1.9 Outline

Chapter 2 presents the fundamentals of Constraint Programming. Chapter 3 discusses the problem in more detail. Chapter 4 presents the model and design choices made during the course of this project as well as information about the implementation of the model in a CP
solver. Chapter 5 presents the experimental results and evaluation of the implementation. This dissertation concludes in Chapter 6, where the results of the project are discussed in addition to the recommended future work.
2 Background

Section 2.1 introduces the reader to Constraint Programming in order to be familiar with the concepts used in the rest of the thesis. Section 2.2 presents a common problem solved with Constraint Programming.

2.1 Constraint Programming

Constraint Programming (CP) is a programming paradigm which is used to solve combinatorial problems. It is a declarative paradigm, which means that it does not require the specification of how to solve a problem, but rather determine the properties of a solution. A real-world problem is translated to a Constraint Satisfaction Problem (CSP) during modeling. The CSP defines a set of variables, a set of possible values each variable might be assigned to (called domain) and a set of constraints on these variables. A constraint is a relation between a variable and the set of values for this variable. A solution to the CSP is one when all variables are assigned to a value and no constraints are violated. Constraint Programming has been used in a broad number of fields, solving problems like scheduling, vehicle routing, networks and more. This section is based on [4].

2.1.1 Constraint Satisfaction Problem

A Constraint Satisfaction Problem is defined as a triple \( P = (X, D, C) \) where \( X \) is a \( n \)-tuple of variables \( X = (x_1, x_2, ..., x_n) \), \( D \) is a \( n \)-tuple of finite domains \( D = (D_1, D_2, ..., D_n) \in \mathbb{Z} \) such that \( x_i \in D_i \) and \( C \) is a \( t \)-tuple of constraints \( C = (C_1, C_2, ..., C_t) \). A constraint \( c \) is defined by its variables \( \text{var}(c) = (x_1, x_2, ..., x_n) \in X^n \) and its solutions \( \text{sol}(c) \subseteq D^n \). An assignment \( a \in V \rightarrow X \), appoints a value to each variable from its corresponding domain. Assignment \( a \) is solution of a constraint \( c \) when for all the \( m \) variables in \( \text{var}(c), a(x_m) \in \text{sol}(c) \).

2.1.2 Constraint Optimization Problem

A Constrained Optimization Problem (COP) is an extended CSP where the solutions are evaluated over an objective function. Solving a COP has the extra purpose of minimizing or maximizing the objective function. A very common algorithm used for Constraint Optimization is Branch and Bound (BAB). BAB tries to eliminate branches of the search tree by proving that no better solutions can be found in those branches than the one already found.

2.1.3 Variables

This section covers the types of CP variables used in this thesis, namely Integer and Boolean variables. More types of variables exist in CP but are not relevant in the proposed model.
**Integer Variable**

An Integer variable is the most frequently used variable in CP. This type of variable has a finite domain that consists of integers.

**Boolean Variable**

A Boolean variable is a variable whose domain only contains two Boolean values: true and false. This can be considered a special case of Integer values where the variable can only have zero or one as its domain. But normally, Constraint Programming systems provide different implementations of Boolean variables, optimized accordingly.

### 2.1.4 Constraints

The implementation of Constraints differs from system to system, but there is a big number of constraints that are essential to any system. The research community has done a great job identifying those constraints and their variants (also called Global Constraints). The constraints used in this thesis are briefly presented below as described in [5].

**Equality:** $eq(var1, var2)$

The equality constraint enforces that var1 and var2 have the same value.

**Greater or Equal:** $geq(var1, var2)$

The greater or equal constraint enforces that the value of var1 is greater or equal than the value of var2.

**All Different:** $alldifferent(variables)$

The alldifferent constraint enforces that a set of variables will have pair-wise distinct values. Synonyms of this constraint are distinct, alldiff, alldistinct and more.

**Domain:** $in(var, values)$

The domain constraint enforces the variable var to take values in the values collection. Alternative names for this constraint are member, in_set, dom.

Multiple variations of the above constraints are implemented in most of the Constraint Programming Systems available as well as many, more complex constraints.

**Reified Constraints**

Reified constraint (or meta-constraints) are variations of constraints which are extended with a Boolean control variable $b$ [6]. If the constraint holds, then $b = 1$ (true) is propagated, if the constraint is violated the $b = 0$ (false) is propagated. Also when $b$ is true, the constraint is
propagated and when \( b \) is false, the negation of the constraint is propagated. Propagation is explained in section 2.1.6.

2.1.5 Constraint Programming System

Constraint Programming Systems (alternative called solvers) are either standalone programs, or libraries, created in popular languages like C++ or Java, which provide the implementation of several global constraints, a constraint propagation mechanism and a search mechanism usually with a variation of built-in search strategies.

2.1.6 Constraint Propagation & Search

Constraint Programming Systems use a combination of two functions in order to solve a CSP. Those functions, called Constraint Propagation and Search run iteratively. At first, Constraint Propagation is executed, this way the solver removes illegal values from variable domains. As soon as propagation has completed its job – in the sense that applying the propagators again doesn’t reduce the domain of any variable – and no domain has failed – there are still possible values for every variable – search is performed. The solver will take a choice, following a strategy defined by the model and select a variable to search on, splitting the search space in two sub-problems. One where the selected variable has a specific value and one where that variable is not equal to that value (the value is selected again using a strategy defined in the model i.e. smallest value). In each node in the search tree, Constraint Propagation is done first and then, if there are still possible options for some variables, the search continues. As soon as all variables are assigned to specific values, and satisfy all the constraints posed to them, a solution is found. Each iteration is implemented as a backtracking search procedure, exploring the choices of the search tree as sub-problems of the problem.

**Constraint Propagation**

Constraint Propagation is the procedure that prunes illegal values from a variable domain according to the posted constraints. Each constraint posted during modeling is translated into a set of propagators. The propagators prune values from a variable domain that are not part of a legal solution according to its constraints. When a Propagator is unable to remove values from a variables domain, it stops being relevant for the current iteration, we say the propagator is at fixpoint. Propagators are implemented in Constraint Programming Systems for each of the constraints offered by them.

**Search**

During the solve process, the system always applies the propagators first, in order to minimize the search space of the problem. Then, if applying the propagators doesn’t reduce the domain of any variable (all the propagators are at a fixpoint), the problem is split into sub-problems by making a choice on a variable. There are several strategies on which variable to select and what kind of choice to make in order to try to find a solution. To name a few, the variable selection can be on the variable that has the fewest values, the one that has the smallest minimal value, a random variable and more. As soon as a variable is selected, the
solver will select a value for that variable (for example smallest value available, maximum value available, center value and more) and will split the search tree into two sub-problems. On the one side of the search tree, the selected variable will be assigned the selected value and on the other side the selected value will be removed from the domain of the selected variable. Propagation will run again and this process will continue recursively until the search tree is exhausted. This process is called Branching.

**Branch and Bound**

The Branch and Bound algorithm searches for solutions that minimize or maximize the value of an objective function, instead of every legal solution. This is accomplished by bounding whole branches of the search tree, and eliminating (failing) branches for which there is proof that not a better solution will be found. The usage of such an algorithm transforms the problem from a Constraint Satisfaction Problem (CSP) to a Constraint Optimization Problem (COP).

### 2.2 Example: Sudoku

One of the most widely used examples to introduce Constraint Programming is the Sudoku puzzle game. Sudoku was invented by Howard Garns in 1979 [7] and has been popular ever since, printed in newspapers, magazines, websites and puzzle booklets. A Sudoku puzzle is played by a single player. In its typical form, it consists of a 9x9 grid which is divided in 9 boxes of 3x3 cells. Each cell is either blank or contains a digit between 1 and 9. The cells which are filled with a value initially, are clues (hints) to help the player solve the puzzle. The objective is to fill all the cells with digits between 1 and 9 while conforming to the following rules:

- Each row must contain each digit exactly once
- Each column must contain each digit exactly once
- Each 3x3 grid must contain each digit exactly once

A legal Sudoku puzzle instance must have exactly one solution. A Sudoku puzzle instance with the hints in black color and the solutions in red color is shown in Figure 2.1. The model discussed below is a slight variation of the one proposed in [8].
2.2.1 Modeling a Sudoku puzzle as a CSP

In order to model the Sudoku puzzle as a CSP, we define the following variables and constraints.

**Variables**

The puzzle board consists of 81 cells, thus it is convenient to define 81 variables, with initial domain the integers 1 to 9. Variable $x_1$ represents the first cell on the top left of the grid, $x_2$ the one on the right of $x_1$ etc. When a cell doesn’t have any neighbor cell on its right, the row changes and we continue to count as expected intuitively.

$$x_1, x_2, x_3 \ldots x_{80}, x_{81} \in \{0..9\}$$

**Constraints**

The hints of the puzzle can be direct assignments on the corresponding variables, for example, for the hints in Figure 2.1 we can state the following:

$$x_1 = 5, x_2 = 3, x_3 = 7, x_{10} = 6, x_{13} = 1, x_{14} = 9, x_{15} = 5, x_{20} = 9 \text{ e.t.c.}$$

Then, according to the rules of the puzzle we have the following constraints:

a. Each row must contain each digit exactly once:
   
   for $i = 0$ to $i = 8$
   
   $$\text{alldifferent}(x_{i+9+1}, x_{i+9+2}, x_{i+9+3}, \ldots x_{i+9+9})$$

b. Each column must contain each digit exactly once
   
   for $i = 1$ to $i = 9$
   
   $$\text{alldifferent}(x_{i+1}, x_{i+9}, x_{i+18}, \ldots x_{i+72})$$

c. Each 3x3 grid must contain each digit exactly once
   
   for $i = 1$ to $i = 9$
   
   $$\text{alldifferent}(x_{i+3-2}, x_{i+3-1}, x_{i+3}, x_{i+3+7}, x_{i+3+8}, x_{i+3+9}, x_{i+3+16}, x_{i+3+17}, x_{i+3+18})$$

Figure 2.1 A solved Sudoku puzzle
**Solution**

Normally, Sudoku puzzles are designed in such a way that they have only one valid solution and can be used by only using propagation, requiring no search during the solving process. This is not true for constraint problems in general. In fact, the hard difficulty Sudoku puzzles require search in order to be solved.

**2.3 Configuration**

These days, an increasing amount of services or products require customization. This trend is pushing to the creation of component based systems, allowing users to satisfy their specific requirements. Some common examples are travel packages or car manufacturing. These are configuration problems which are usually solved by CP methods.

Configuration is the task of composing a customized system out of generic components. This kind of problem is modeled by two elements. A catalog, which contains the description of components with their technical and functional characteristics, and the requirements or preferences of the configuration concerning those characteristics. A solution is a configuration that satisfies all the requirements and when needed optimizes according to the preferences [9]. These types of problems are closely related to the problem this project addresses.
3 Problem Analysis

This thesis focuses on the problem of orchestrating an end-2-end service creation and deployment. To achieve this, we need to compose an end-to-end cloud service in a hierarchical manner and deploy it over an infrastructure topology.

3.1 Composing the end-2-end service

The composition process starts from a Service Level Agreement (SLA) with specific requirements and is resolved to a set (actually a graph w.r.t to their connectivity) of software components, that are mapped over the infrastructure topology as represented by the infrastructure model.

The SLA bounds the possible solutions in respect to price, latency and some special characteristics, like for example the geographical location of the infrastructure nodes that will host the services. It also represents knowledge about the inbound traffic this end-to-end service will have, a property that is required in order to calculate several properties of the components in a solution (such as component price, infrastructure node price per hour, infrastructure link throughput and more).

The software components offer either an Atomic or an Abstract Service. Note that usually several components can implement the same function, and those chosen during resolution need to fulfill the requirements described in the SLA. Moreover, an Abstract Service can depend on a set of components that also contains one or more Abstract Services, which in their turn have to be resolved. Figure 3.1 displays two different implementations of an end-2-end Service, the first one is composed by three software components, where one of them is an Abstract Service composed itself by two other Atomic Services. The second implementation of the same Abstract Service is composed by just two Atomic Services.

The Abstract Services represent a functionality achieved by combining several other Services which hereafter we call dependencies. They also hold information about the connectivity between the dependencies, which needs to be identified during the solving process.

No matter how many layers of Abstract Services are required in order to resolve the end-2-end service, the leaves of this graph will be Atomic Services, which have to be mapped over the infrastructure topology.
3.2 Deploying over the Distributed Cloud

The topology of the network in question is also called Distributed Cloud, which consists of diverse nodes, scattered in different physical locations. A node might be placed in a large Data Center, typically far away from the end user, in smaller Data Centers, in a national, regional or metropolitan level, up to nodes located in the base stations of mobile networks. Normally the closer a node is to the end user, will offer less latency but also less computational power, for a higher price. Figure 3.2 portrays the typical layout of a Distributed Cloud network.

As explained in Section 3.1, after decomposing the end-2-end Service into software components (Atomic Services), the selected components need to be mapped over the distributed cloud. The mapping has to respect the connectivity of the components.
4 Model

In the following section, the model proposed in this thesis is explained in detail. The model is decomposed into several sub problems, lowering the complexity without ignoring any of the requirements. Section 4.1 describes the proposed model of the catalog and section 4.2 describes the proposed model of the infrastructure topology.

4.1 Modeling the Catalog

The component catalog contains two different types of components, which are used in an end-to-end cloud service composition. Each entry in the catalog is an instance of a component, distinguishable by a unique name or id. Moreover, each entry offers a functionality (i.e. Transcoder, Cache, Media Transport), but there can exist multiple instances offering the same functionality with some variations in their parameters, like different licensing price, required memory, CPU, storage and more.

4.1.1 Atomic Service Component

The first type of component is the Atomic Service, which has a clear set of characteristics, based on the input traffic of this component and some fixed characteristics like licensing cost. When such a component is required in a solution, it should be mapped in a single infrastructure node in the Distributed Cloud.

The JSON model of an Atomic Service:

```
{
    "name": "Media Server Amazon",
    "provides": "Media Server",
    "input_traffic_port0": 100,
    "output_traffic_port0": 150,
    "input_traffic_port1": 200,
    "output_traffic_port1": 250,
    "input_traffic_port2": 100,
    "output_traffic_port2": 0,
    "execution_latency": 100,
    "required_processing": 4,
    "required_memory": 200,
    "required_storage": 100,
    "licensing_price": 100,
    "special_capabilities": [ "Stockholm" ],
    "anti_affinity_set": [],
    "affinity_set": []
}
```
• **name**: A unique name or ID that distinguishes this component over all other component instances.

• **provides**: A string that represents the functionality of the component. Many component instances can have the same value.

• **input_traffic_port#**: The maximum input traffic in KB this component can have on each port.

• **output_traffic_port#**: The output traffic in KB of each port.

• **execution_latency**: The time in ms which is required to run the function provided by this component.

• **required_processing**: The processing power in number of cores this component will consume over the infrastructure node.

• **required_memory**: The memory in MB this component will consume over the infrastructure node.

• **required_storage**: The storage in MB this component will consume over the infrastructure node.

• **licensing_price**: The licensing price of the component.

• **special_capabilities**: A set of strings that represent some characteristics the infrastructure node has to match. This can be either location, bounding the service to be hosted in a specific city/region/country or other characteristics of the infrastructure node like the presence of specific hardware (i.e. GPU, mpeg decoder etc.).

• **anti_affinity_set**: A list of offers, with whom this service must not be hosted in the same infrastructure node.

• **affinity_set**: A list of offers, with whom this service should be hosted in the same infrastructure node.

### 4.1.2 Abstract Service Component

The second type of component is the Abstract Service. This component, is composed by two or more other Abstract or Atomic Service components as dependencies (up to a maximum of c dependencies). It also describes the connections between its dependencies in detail, stating which dependencies are connected and specifying the port used for that connection. This type of components, when required in a solution, should not be mapped in an infrastructure node, as they are abstract collections of other components.

The JSON model of an Abstract Service:

```json
{
    "name": "Media Service1",
    "provides": "Media Service",
```
"dependencies" : [
    "Access Point", "Media Transport", "Media Server"
],
"dependency_connections" : [
    [0, 1, 2],
    [0, 2],
    []
],
"connection_ports" : [
    [0, 2, 1],
    [1, 2],
    []
],
"special_capabilities" : [ "Stockholm", "Kista" ]
}

- **name**: A unique name or ID that distinguishes this component over all other component instances.

- **provides**: A string that represents the functionality of the component. Many component instances can have the same value.

- **dependencies**: A set of strings that contains the dependency requirements (in the form of strings of offers) of this Abstract Service. This list should be of maximum size c.

- **dependency_connection**: A array of arrays of integers representing how the dependencies of this component are connected. In the example above, for the first path Access Point connects to Media Transport and Media Transport connects to Media Server, for the second path Access Point connects to Media Server.

- **connection_ports**: A array of arrays of integers representing which ports are used from each dependency in order to have the connectivity described in dependency_connections. In the example above, for the first path, Access Point connects to Media Transport using port 0 and port 2 respectively and Media Transport connects to Media Server using port 2 and port 1 respectively.

- **special_capabilities**: A set of strings that represent some characteristics that all the infrastructure nodes used to create this Abstract Service have to match.

The Atomic Services that compose an Abstract Service, can have more than one connection, for example dependency #1 connects to dependency #2 and dependency #3, dependency #2 connects to dependency #3 and so on. Those connections are described in the Abstract Service’s properties. In order to limit the possible combinations, in the proposed model, we limit the number of possible paths to three. The reason behind this choice is that we identified three possible ports for each Atomic Service and the mapping between ports is
one to one. However, this property can be easily extended to add more ports and connections between Atomic Services.

4.2 Modeling the Network

As explained in section 3.2, a Distributed Cloud consists of diverse nodes, scattered in different physical locations. The nodes are connected with physical links with characteristics such as latency and throughput which we need to consider in our model. The model is encapsulated in a JSON representation which is easy to translate into common standard languages used for representing cloud topologies, like TOSCA [10].

4.2.1 Infrastructure nodes

The infrastructure nodes, as described in the JSON model below, contain all the information required by the solver in order to identify its core properties, like processing power, memory and storage and determine whether it can be used in a solution.

The JSON model of an infrastructure node:

```json
{
    "node_name": "Stockholm Kista 13",
    "nodeID": 13,
    "availability": 999,
    "processing_price": 2,
    "max_processing": 8,
    "memory_price": 2,
    "max_memory": 2048,
    "storage_price": 1,
    "max_storage": 1000,
    "special_capabilities": ["Stockholm", "Kista", "mpeg" ]
}
```

- **name**: The name of the node. This property is not used in the resolution, but can be helpful when displaying the solution.
- **nodeID**: A unique ID for this infrastructure node as an integer. This is used to the model of the physical links as explained later.
- **availability**: An integer ranging from 0 to 1000 which represents the percentage of time the node is expected to be alive. Here 999 is equivalent to 99.9%.
- **processing_price**: A multiplier used for calculating the processing cost, depending on the usage of the processing power of this infrastructure node.
- **max_processing**: The maximum processing power in number of cores this infrastructure node can offer.
• **memory_price**: A multiplier used for calculating the memory cost, depending on the usage of memory of this infrastructure node.

• **max_memory**: The maximum memory in MB this infrastructure node can offer.

• **storage_price**: A multiplier used for calculating the storage cost, depending on the usage of storage of this infrastructure node.

• **max_storage**: The maximum storage in MB this infrastructure node can offer.

• **special_capabilities**: A set of strings with characteristics this infrastructure node has, as explained in the description of the Atomic Services.

### 4.2.2 Infrastructure Links

A physical link between two infrastructure nodes is described in this model. It provides information about the ID of the two nodes it connects, plus some extra characteristics required by the solver in order to determine if the link can be used. Note that the links are not directed. The same link is used when sending data from either of the two nodes listed in the JSON model.

The JSON model of an infrastructure link:

```json
{
    "node1_id": 2,
    "node2_id": 4,
    "latency": 200,
    "traffic_price": 200,
    "max_throughput": 10000
}
```

- **node1_id**: The ID of the first infrastructure node is attached to.

- **node2_id**: The ID of the second infrastructure node this links is attached to.

- **latency**: The latency in ms this link adds to a resolution.

- **traffic_price**: A multiplier used for calculating the traffic cost, depending on the usage of the link

- **max_throughput**: The maximum throughput of this link in KB.

### 4.2.3 Requirements - Service Level Agreement (SLA)

The SLA describes the end-to-end service the solver has to decompose and map over the infrastructure topology.
The JSON model of a Service Level Agreement:

{
    "SERVICE_TO_PROVIDE" : "Media Service",
    "MAX_PRICE" : 500,
    "MAX_LATENCY_1" : 400,
    "MAX_LATENCY_2" : 300,
    "MAX_LATENCY_3" : 250,
    "AVAILABILITY" : 999,
    "TRAFFIC" : 50,
    "GLOBAL_SPECIAL_CAPABILITIES" : [ "Stockholm" ]
}

- **SERVICE_TO_PROVIDE**: A string representing the end-to-end service the model solves for.
- **MAX_PRICE**: The maximum total price per hour of the solution.
- **MAX_LATENCY_#**: The maximum latency in ms the solution can have for each path.
- **AVAILABILITY**: An integer ranging from 0 to 1000 which represents the percentage of time an infrastructure node is expected to be alive. For example, a node that is expected to be alive 99.5% of the time is represented in the JSON model as 995. All of the infrastructure nodes assigned to a solution must have availability greater than or equal to this value.
- **TRAFFIC**: This is the input traffic in KB of the end-to-end service. The Atomic Service which will be the entry point in the solution is required to have traffic greater than or equal to this value.
- **GLOBAL_SPECIAL_CAPABILITIES**: A set of strings with characteristics the end-to-end service requires. All of the characteristics in this set have to be satisfied by all the infrastructure nodes assigned to a solution.

### 4.3 Variables

A solution of an instance of the problem is a tree where the root node is the end-to-end service the system solves for (described in the SLA). The children of each node are its dependencies. The total nodes that have no children (leaves of the tree graph) are the Atomic Services that compose the end-to-end service which will be mapped to infrastructure nodes.

In order to bound the maximum size of this graph produced by decomposing the end-to-end service, we introduce two constants $c$ and $l$ that represent the children each node can have and the tree's maximum levels respectively. Thus the maximum total number of nodes in the tree is: $m = \sum_{i=0}^{l-1} c^i$. 
Throughout the rest of the thesis, this tree is called Configuration Tree. Using this method, we bound the maximum Atomic Services used in a solution to \(c^{l-1}\). The main reason for bounding the possible combinations is that by using dynamic binding we would have to be sure that the catalog is correct. We avoid cases where if there exists an Abstract Service A that depends on another Abstract Service B and so on, and one of the children some levels down depends again on A, leading to an infinite loop. In a different scenario, where the correctness of the catalog can be guaranteed, we could use a method that does not need any information of the solution size as proposed in [11].

![Configuration Tree](image)

**Figure 3.1 Example of a Configuration Tree with \(c = 2\) and \(l = 2\)**

The catalog consists of \(s\) components (services, either Abstract or Atomic), the network consists of \(n\) infrastructure nodes and the Configuration Tree has \(m\) nodes as explained above.

We identify the following sets of Variables:

- \(component[m] \in \{0..s\}\)
  
  We create one variable for each possible position in the Configuration Tree. The domain of each variable is 0 to \(s\), representing which component from the catalog is assigned to each position.

- \(infrastructureNode[m] \in \{0..n\}\)
  
  We create one variable for each possible position in the Configuration Tree. The domain of each variable is 0 to \(n\), representing which infrastructure node will host the component assigned in the corresponding position. The \(infrastructureNode[i]\) assigned value, will host the component assigned in the \(component[i]\) variable.

Those two arrays contain the search variables of the problem. The constraints presented in the following section guarantee that the solutions provided by the solver are valid. In order to be able to post some of the constraints, some extra variables will be introduced in the next section, which are not searched by the solver. These variables are also matched to each node in the Configuration Tree but their value depends on the parent of each node.
As in the model proposed in [12], the first entry in the catalog is a dummy component, which offers no service and is used to identify when a node in our Configuration Tree does not require to be mapped to a service.

Similarly, the first entry in the list of infrastructure nodes, is a dummy entry with no links to other nodes and characteristics which are equal to zero.

4.4 Constraints

The constraints that formulate this problem are presented in this section. The constraints are posted before the search process. Each of the constraints below, are implemented in all major Constraint solvers, which makes the realization of this model to a program trivial.

4.4.1 Functionality

Since we know what functionality the of end-to-end service we want exposed at the root node must have, we constraint the component variable of the first node of the tree to be one of the positions of the catalog that offer this service. For example, if the end-to-end service is provided by components in the catalog in position 1,2:

\[
in(\text{component}[0],\{1,2\}) \quad (1)\]

For each of the nodes in the Configuration Tree, starting from the root node and iterating using pre-order traversal, when the node is not a leaf and the selected component (mapped to that node) is an Abstract service component, constraint the child nodes to what each dependency requires as an offer.

\[
\text{node}[i].\text{hasChildren \ is \ true \ and \ component[i] \ is \ Abstract} \Rightarrow
\in(\text{component}[j],\text{dependency}[i][\text{position}(i)]) \text{for all children of node } i \quad (2)\]

Each component in the catalog provides a service. This is modeled as a list of strings where each component \(i\) in the catalog provides the service \(\text{service}[i]\). Thus we can create lists of which components offer each service e.g. "Cache" is provided by entries in the catalog \([1,2,5,7]\). We use the \(\in\) constraint in order to eliminate illegal values from the \(\text{component}[i]\) variables when we decompose the Abstract Services.

Abstract Services have a list of dependencies. An Abstract Service can have maximum \(c\) dependencies (the maximum number of children a node in the Configuration Tree can have). The list of dependencies in the model, states what the dependencies provide, so an Abstract Service’s \(\text{dependency}[i]\) is a list of lists of components from the catalog that offer that service.

4.4.2 Mapping over infrastructure nodes

Only the Atomic Services need to be mapped in an infrastructure node, thus the variables in the \(\text{infrastructureNode}[m]\) array are constraint to the dummy node (value equal
to zero) when the service mapped in the corresponding position of the Configuration Tree is an Abstract Service, greater than zero otherwise.

\[\text{infrastructureNode}[i] \in \begin{cases} 1..n, & \text{if component}[i] \text{ is an Atomic Service} \\ 0, & \text{otherwise} \end{cases} \quad (3)\]

Similarly, to the catalog, each node in the network is uniquely identifiable by an ID. Hence we construct an array of strings of length n, containing the infrastructure nodes names, plus a dummy entry in the first position.

Each infrastructure node has some concrete characteristics, like available processing power, available RAM and available storage. All of these values are stored to arrays, matching the position of the node in the aforementioned node list to the property the array represents. Furthermore, each node has a lease price per hour, depending on the usage of its properties. The constraints posted for these parameters follow in the next sections.

4.4.3 Traffic – Paths

Path discovery

In order to discover which nodes in the Configuration Tree have to be connected, we iteratively explore the usage of Abstract Services and consider only the entry and exit nodes for each path. By using reified constraints on the component variable, and by knowing which values correspond to Abstract Services, we identify which child nodes have to be connected when a node is assigned to an Abstract Service. We use two arrays of variables for each path, \(\text{path#}[m] \in \{0..\text{linkSize}\}\) and \(\text{path#Bool}[m] \in \{0,1\}\). \(\text{path#}[i]\) is an indication of which link is used for the connection between the node selected in \(\text{infrastructureNode}[i]\) and its successor. The dummy link, with value equal to zero is used when the node in the Configuration Tree is not used in that path, or the two connected nodes are hosted in the same infrastructure node (there is no need for physical connection using an infrastructure link). \(\text{path#Bool}\) is one for a node in the Configuration Tree when that node is used in a path and zero otherwise. A similar technique, with variables which point to successor of nodes is used in the circuit constraint [13].

Constraining the connectivity

During the discovery of the nodes of the Configuration Tree that we find nodes that must be connected, we eliminate illegal pairs of infrastructure nodes in order to allow only the ones which have a link connecting them.

When two nodes \(i\) and \(j\) in the Configuration Tree have to be connected, we remove all possible combinations of nodes that are not physically connected by posting the following constraint:

\[\text{infrastructureNode}[i] == n1 \Rightarrow \text{in}(\text{infrastructureNode}[j], \text{connections}[n1]) \] (4)
Traffic

The input traffic of the end-2-end Service is defined in the SLA. Here we make an assumption that the entry point of the end-2-end Service will be the port paired to the first path, for the first Atomic Service in a solution. We constraint the input of that component to be greater or equal than the TRAFFIC property defined in the SLA. As expected, we compare the output of the selected Atomic Service for the first node in the path, with the second one and require it the latter to be greater or equal than the former. This constraint also applies for all three ports of an Abstract Service, if they are required on any path.

\[
\text{path1Bool}[i] = 0 \text{ and path1Bool}[j] = 0, \quad j < i \\
\Rightarrow \text{component}[i].\text{input\_traffic\_1} \geq \text{sla\_traffic}\]

(5)

\[
\text{path\#Bool}[i] = 1 \&\& \text{path\#Bool}[j] = 1 \&\& \text{path\#Bool}[k] = 0, i < k < j \\
\Rightarrow \text{component}[i].\text{output\_traffic\_#} \leq \text{component}[j].\text{input\_traffic\_#}
\]

(6)

4.4.4 Latency

The maximum allowed latency of a solution is described in the SLA. Since we have distinguished the three possible paths, it is convenient to have the possibility to ask for different values of latency in each path. The latency of a path is the combination of the summary of the execution latencies of the Atomic Services used in that path and the summary of the latencies of the infrastructure links used in that path. The total number cannot exceed the MAX\_LATENCY\_# specified in the requirements.

\[
\text{leq}(\sum\text{path\_#}, \text{MAX\_LATENCY\_#}) \\
\sum\text{path\_#} = \sum_{i=0}^{m} \text{component}[i].\text{execution\_latency} + \text{path\#}[i].\text{latency}
\]

when \text{path\#Bool}[i] = 1

(7)

4.4.5 Link Throughput

Infrastructure Links have a property of maximum throughput. It is obvious that we don't want the traffic flowing through any link to be more than the link's throughput, thus we constraint the summary of the traffic to be less than that property's value for each link used.

\[
\text{leq}(\text{throughput\_sum}, \text{infrastructureLink}[i].\text{max\_throughput}) \\
\text{throughput\_sum} = \sum_{i=0}^{m} \text{component}[i].\text{output\_traffic}
\]

when \text{path\#Bool}[i] = 1, for all paths

(8)

4.4.6 Infrastructure Node

As presented in 4.2, the infrastructure nodes have three properties, processing power, ram and storage. Obviously, we cannot assign services to a node that require more than this node has available. Thus the sum of the services properties that are assigned to an infrastructure node, cannot be higher than the nodes availability for this property.
Node Availability (uptime). The node availability is usually expressed as a percentage of the time the node is expected to be functional (i.e. 99.9%). This percentage can be mapped over integers from 0 to 1000 (or more if we want to increase accuracy), so 99.9% is actually 999 in our model. This way, constraining the infrastructure nodes selected to have availability higher or equal to the one stated in the requirement SLA becomes trivial.

The infrastructure nodes that will host a service, selected in a solution, must fulfil the SLA requirement for availability. Since all the $\text{infrastructureNode}[i]$ variables that host a service have value greater than zero, we remove from their domain the infrastructure nodes that don’t have high enough availability.

$$\text{in}(\text{infrastructureNode}[i], \text{infrastructure_node_subset})$$

Where $\text{infrastructure_node_subset}$ is an array of infrastructure nodes with availability greater or equal than the required value.

4.4.7 Price

The total price of the end-2-end service orchestration and deployment is the summary of the following:

- Licensing cost: The summary of the license cost for all the Atomic services used to construct the required top level service.

  $$\text{licensingPrice} = \sum_{i=0}^{m} \text{component}[i].\text{licence_price}$$

- Infrastructure Node cost: The summary of costs for the required usage of the infrastructure nodes. This depends on the consumption of the node's capabilities (processing power, memory, storage).

  $$\text{infrastructureNodePrice} = \sum_{i=0}^{m} \text{component}[i].\text{required_memory} \times \text{infrastructureNode}[i].\text{memory_price} + \text{component}[i].\text{required_storage} \times \text{infrastructureNode}[i].\text{storage_price} + \text{component}[i].\text{required_processing} \times \text{infrastructureNode}[i].\text{processing_price}$$

- Infrastructure Link cost: Each link has a property of $\text{traffic_price}$ which multiplied by the total traffic coming through a link give us the cost for using that link in a solution.

  $$\text{infrastructureLinkPrice} = \sum_{i=0}^{m} \text{path#}[i].\text{traffic_price} \times \text{component}[i].\text{output_traffic_port#}$$
Intuitively, the total price of the orchestration and deployment of a solution, the summary of the costs above, should not exceed the MAX_PRICE provided in the requirements SLA.

\[
\text{cost} = \text{sum}(\text{licencingPrice}, \text{infrastructureNodePrice}, \text{infrastructureLinkPrice}) \\
\leq (\text{cost}, \text{sla.MAX\_PRICE})
\]

(14)

4.4.8 Special Capabilities

The \textit{special\_capabilities} set property is present in infrastructure nodes, listing characteristics of the node, like location, hardware and more. The Atomic Services used to construct a solution have a set of characteristics they need in order to function properly. The nodes that host those Atomic Services must always conform to that set, in other words, an infrastructure node can host an Atomic Service only if the \textit{special\_capabilities} set of the service is a subset of the node's \textit{special\_capabilities} set. Thus we restrict the options of where a service can be hosted by removing the values which correspond to infrastructure nodes that don’t have all the required \textit{special\_capabilities}.

\[
\text{eq}(\text{component}[i], j) \Rightarrow \text{in}(\text{infrastructureNode}[i], \text{inf\_node\_subset})
\]

where \textit{inf\_node\_subset} is an array of infrastructure nodes that have all the \textit{special\_capabilities} that the service in position \( j \) in the catalog requires.

The same logic applies to the similar property of the Abstract Services, though here, since the Abstract Service is constructed by several components, all those components that are Atomic Services must be hosted in nodes that have a \textit{special\_capabilities} set which is a superset of the Abstract Service's one.

4.4.10 Anti-Affinity

In order to avoid hosting services that are not possible to work together in the same node, as stated by the services Anti-Affinity set, we use the \textit{all\_different} constraint. If a node \( n1 \) in the Configuration Tree is assigned to component \( i \) and this component has component \( j \) in its Anti-Affinity set, every node \( n2 \) that has assigned to it the component \( j \) must be different than \( n1 \).

\[
\text{eq}(\text{component}[n1], i) \& \& \text{in}(\text{component}[n2], \text{AntiAffinitySet}[i]) \Rightarrow \text{neq(\text{infrastructureNode}[n1], \text{infrastructureNode}[n2])}
\]

(16)

4.4.11 Affinity

The reverse logic applies to the Atomic Services \textit{affinity} set. When an Atomic Service is used in a solution, and this component's affinity set contains other Atomic services used in the solution, they must be hosted in the same infrastructure node.

\[
\text{eq}(\text{component}[n1], i) \& \& \text{in}(\text{component}[n2], \text{AffinitySet}[i]) \Rightarrow \text{post eq(\text{infrastructureNode}[n1], \text{infrastructureNode}[n2])}
\]

(17)
4.5 Symmetries

This model has several symmetries which we can eliminate by adding some extra constraints to the problem. One of them is the usage of identical infrastructure nodes, similar to what is suggested in [12] for a Rack Configuration Problem. Two nodes are identical if they are located in the same area (for example the same data center) and offer the same capabilities in terms of physical memory, storage and processing power.

Also, another symmetry that could be present in a model like the one proposed in this thesis, is the positioning of the dependencies of an Abstract Service in the children nodes in the Configuration Tree. For example, in a Configuration Tree with maximum children $c = 3$, and for an Abstract Service with 2 dependencies, there are three possible sets of positions. We eliminate this symmetry by assigning the dependencies in the position suggested in the dependencies array of its JSON model. Namely, the first dependency will be on the first child of the node in the Configuration Tree, the second dependency on the second child and finally the last child will host the dummy service.

4.6 Search Strategy

In the following sub-sections, we discuss the Variable and Value selection strategies respectively. The decision variables as presented in the beginning of this chapter are $component[]$ and $infrastructureNode[]$.

4.6.1 Variable Selection

For Variable selection, the author chose the $CHOOSE_FIRST_UNBOUND$ strategy, which chooses the first variable from the array $component[]$ that does not have a singleton domain. This strategy indeed, makes sense with the iterative process used to decompose the Abstract Services, since the Configuration Tree is explored in pre-order traversal.

4.6.2 Value Selection

For Value selection, in this thesis, the $ASSIGN_MIN_VALUE$ strategy is used, which selects the smallest value in the domain of the selected variable. Other possible strategies are to select the maximum value available in the domain, split the domain in half and select values from the lower or upper half and more. Although, the Value selection has almost no impact in the search process, due to the randomness introduced in the evaluation, as explained in the next chapter, there is no specific order in the list of software components or infrastructure nodes.
4.6.3 Branch and bound

The proposed model will find all legal solutions of a medium sized problem (in respect to Configuration Tree, catalog and infrastructure size) in a small amount of time as it is presented in the next section. Although, listing all legal solutions is not always ideal. When the problem has hundreds or thousands of solutions, selecting the best one is not trivial. With the usage of Branch and Bound, we can assure that the last presented solution by the solver, is the best one, in respect of an objective function of our choice. In our case we explore the usage of Branch and Bound when trying to minimize the cost (total price) of the solution.
5 Evaluation

In this section we describe the test-bed created in order to evaluate the model over diverse experiments in terms of catalog size, size of the infrastructure and Configuration Tree.

5.1 Implementation

Despite the existence of object-oriented languages designed for configuration [14] and the existence of a generic modeling language for Constraint Programming, widely accepted by the community [15], the implementation of the model is done in OR-Tools [16]. OR-Tools is an open source software suite for combinatorial optimization, available for free use under Apache License 2.0. The suite is maintained by Google and is originally written in C++. The library is available through SWIG [17] for Python, Java and .NET for all major operating systems. It contains a constraint programming solver, interface to several linear programming and mixed integer programming solvers, knapsack algorithms and graph algorithms (shortest paths, min cost flow, max cost flow, linear sum assignment).

The main decision points that lead the author to choose OR-Tools for the implementation was that the suit is open source, well documented and efficient, but most importantly that it contains the min cost flow graph algorithm, which is an inevitable addition to this project in the future. As discussed already we only consider nodes in the network that are directly connected. This is a delimitation which limits the possible solutions of a configuration. Future work could focus on solving the routing between nodes that are not directly connected and the graph algorithms that the suite has implemented will help adding this feature to the model.

For this thesis project, the author chose the Java interface of the library, primarily because of the wide adoption of the language which supply it with many other useful libraries, such as the FasterXML library [18] used for reading the JSON files containing the information about software components, infrastructure nodes and links.

The implementation follows the framework proposed in [19], where the model and instance specific data are separated in the interest of re-usability and model maintenance.

5.2 Test-bed

The experiments run on an Intel Core i7-2620M 2.7GHz 4 core computer with 8GB RAM, running 64bit Ubuntu 14.04. The JAVA interface of OR-Tools version 3574 is used. The information for each experiment are gathered from the statistics of the library. The statistics relevant to our results are:

- **Wall Time**: The time it took the solver to finish the search. This time does not include the initialization of the catalog. The time it takes for the system to read the JSON files is constant in each test case and not relevant to the model behavior.

- **Solutions**: The number of feasible solutions the solver found for the problem case.
- **Failures**: The number of failures in the search tree. Note that in OR-Tools, failure occurs whenever the solver has to backtrack, whether it is because of a real failure or a solution.

- **Branches**: The number of branches in the search tree.

### 5.3 Experiment Results

In order to evaluate the behavior of the model, we run eight different test cases, with variation on the parameters, catalog size and infrastructure size, as portrayed in Table 5.1. The first column in the table (# of Nodes in the Configuration Tree) corresponds to the total number of nodes \( m \) as explained in section 4.3. The second column, infrastructure size, is the total number of infrastructure nodes plus the total number of links between them. The third column, Catalog size, corresponds to the sum of Abstract and Atomic Services in the catalog. Solution depth is the maximum depth in the Configuration Tree the solution will be found; one (1) means that the search is done for an Atomic service as the end-2-end service required by the SLA, two (2) means that the end-2-end service required will be composed by up to maximum dependencies (column 4) Atomic Services, and finally three (3) means that the end-2-end service required will be composed by at least one Abstract Service in the first level. The maximum dependencies column states what is the maximum number of dependencies an Abstract Service can have in our test. Finally, the last column, Solutions, shows if the search will stop at one solution (one), find the best solution (BAB) or find all legal solutions (all) before it ends. All the test cases presented in this chapter are done over a randomly created set of software components, infrastructure nodes and links.

Section 5.2.1 discusses the solution of Test 1, section 5.2.2 the ones of Test 2. Test 4 to 8 have similar behavior and are presented in section 5.2.3. Finally, section 5.2.4 shows the impact of Branch and Bound search in the execution time. Each test was executed ten times, the Wall Time in the results below is the average of those executions.

<table>
<thead>
<tr>
<th>Test 1</th>
<th>Test 2</th>
<th>Test 3</th>
<th>Test 4</th>
<th>Test 5</th>
<th>Test 6</th>
<th>Test 7</th>
<th>Test 8</th>
</tr>
</thead>
<tbody>
<tr>
<td># of Nodes in Configuration Tree</td>
<td>4</td>
<td>4 to 400</td>
<td>1-21</td>
<td>1-21</td>
<td>1-21</td>
<td>1-21</td>
<td>1-21</td>
</tr>
<tr>
<td>Infrastructure Size</td>
<td>40</td>
<td>30</td>
<td>20</td>
<td>60</td>
<td>20</td>
<td>60</td>
<td>60</td>
</tr>
<tr>
<td>Catalog Size</td>
<td>25-500</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>Solution depth</td>
<td>1</td>
<td>1</td>
<td>1, 2, 3</td>
<td>1, 2, 3</td>
<td>1, 2, 3</td>
<td>1, 2, 3</td>
<td>1, 2, 3</td>
</tr>
<tr>
<td>Maximum dependencies</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Solutions</td>
<td>one</td>
<td>one</td>
<td>BAB, all</td>
<td>one, BAB, all</td>
<td>one, BAB, all</td>
<td>one, BAB, all</td>
<td>one, BAB, all</td>
</tr>
</tbody>
</table>

**Table 5.1 Parameters for each test case**
5.3.1 Model behavior depending on catalog size

In order to evaluate the behavior of the model when the catalog size increases, for the first test case, we setup twenty scenarios with the same parameters in the number of nodes in the Configuration Tree, maximum dependencies equal to three $c = 3$ and depth equal to one $l = 1$. As cloud infrastructure, in this test, we have 20 infrastructure nodes with 20 connections (links). In each scenario, we increase the catalog size, starting from 20 components up to 500. We show the results of those test regarding the search time (Wall time) in Figure 5.1. The results show linear increase of the time required to find a solution in respect to the increase of the catalog size.

![Figure 5.1 Model behavior when catalog increases in size](image)

5.3.2 Model behavior depending on infrastructure size

The second test case evaluates the behavior of the model when the infrastructure nodes and links increase. Again, we setup twenty scenarios with the same parameters in the number of nodes in the Configuration Tree, maximum dependencies equal to three $c = 3$ and depth equal to one $l = 2$. As software component catalog we use 30 services (10 Abstract and 20 Atomic Services). The cloud infrastructure increases for each scenario, starting from 10 infrastructure nodes and 10 infrastructure links, up to 200 and 200. The results of those tests regarding the search time (Wall time) are shown in Figure 5.2. The results show linear increase of the time required to find a solution in respect to the increase of the number of infrastructure nodes and links.
5.3.3 Model behavior depending on Configuration Tree size

The test cases 3 to 8 are evaluating the effect of the increase of number of nodes in the Configuration Tree. Table 5.2 display the results of running those test ten (10) times each. The Maximum Wall Time (ms) is the maximum average time it took the solver to find all solutions of the test instance. Each test case is divided into three scenarios where the SLA requires an Atomic Service (sla1), an Abstract Service with solution involving only the first level in the Configuration Tree (sla2) and an Abstract Service with solution involving also the second level. The maximum time is required when the maximum number of children is four \(c = 4\) and depth is two \(l = 2\), in the Configuration Tree.

Each scenario is evaluated in different environments. For test cases test 3 and 4, where the number of maximum dependencies is two, we test using two, three and four children in the Configuration Tree and the all the combinations of depth zero, one and two. Test cases 5 and 6, where the maximum dependencies are 3, we use three and four children. Finally, in test cases 7 and 8 we use only four children. The results are as expected, for the first scenario the solver finds solutions in all those settings, for the second scenario the solver cannot find a solution for depth equal to zero and finally for the third scenario the solver can only find a solution for depth equal to two. In Figure 5.3, we display the results of all the scenarios of Test 3 for a complete search in comparison with stopping after the first solution is found. The results also show that in a small search tree, there is no real value in stopping after the first solution. Although, as the Configuration Tree and the catalog size increases, the time required to finish the search grows much faster than the time required to find one (see Appendix A – Figure B15). We emit the results for the rest of the test cases, since they behave equivalently, but they are available in Appendix A.
<table>
<thead>
<tr>
<th>Scenario</th>
<th>Solutions</th>
<th>Failures</th>
<th>Branches</th>
<th>Wall Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Test 3</strong></td>
<td>sla1</td>
<td>3</td>
<td>4</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>sla2</td>
<td>9</td>
<td>0</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>sla3</td>
<td>6</td>
<td>22</td>
<td>54</td>
</tr>
<tr>
<td><strong>Test 4</strong></td>
<td>sla1</td>
<td>5</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>sla2</td>
<td>7</td>
<td>13</td>
<td>38</td>
</tr>
<tr>
<td></td>
<td>sla3</td>
<td>372</td>
<td>42</td>
<td>826</td>
</tr>
<tr>
<td><strong>Test 5</strong></td>
<td>sla1</td>
<td>7</td>
<td>0</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>sla2</td>
<td>2</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>sla3</td>
<td>44</td>
<td>5</td>
<td>96</td>
</tr>
<tr>
<td><strong>Test 6</strong></td>
<td>sla1</td>
<td>10</td>
<td>0</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>sla2</td>
<td>8</td>
<td>8</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>sla3</td>
<td>1612</td>
<td>35</td>
<td>3292</td>
</tr>
<tr>
<td><strong>Test 7</strong></td>
<td>sla1</td>
<td>20</td>
<td>0</td>
<td>38</td>
</tr>
<tr>
<td></td>
<td>sla2</td>
<td>16</td>
<td>5</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td>sla3</td>
<td>1144</td>
<td>5</td>
<td>2296</td>
</tr>
<tr>
<td><strong>Test 8</strong></td>
<td>sla1</td>
<td>37</td>
<td>1</td>
<td>74</td>
</tr>
<tr>
<td></td>
<td>sla2</td>
<td>63</td>
<td>9</td>
<td>142</td>
</tr>
<tr>
<td></td>
<td>sla3</td>
<td>294</td>
<td>62</td>
<td>710</td>
</tr>
</tbody>
</table>

Table 5.2 Results for all test cases - scenarios

![Figure 5.3a Test case 3 - first scenario (sla1)](image-url)
5.3.4 Branch and Bound

In a real world situation, we would not like to have a list of all the possible solutions, ideally we would want the solver to come up with the best possible assignment of software components to infrastructure nodes. In order to achieve this, we use the Branch and Bound algorithm and consider the last solution the solver outputs. Branch and bound tries to find only better solutions, in respect of an Objective function, instead of every legal solution.

The test cases here are the same as in the previous section, but the search tree is bounded by the total price of the solution. Here we experience a decrease of the solutions found, which is especially high in scenarios where the solver found a big number in the previous tests. For example, in test case 6, scenario three (sla3) the solver found 1612 solutions in the previous settings but using Branch and Bound algorithm, this number is greatly reduced to 8. Also for this scenario, the number of Branches and Failures are greatly reduced. We
observe this decrease in Failures and Branches in almost every case, and most importantly a trend for less time needed by the solver to complete the search. The effects of using Branch and Bound in Test 8 are shown in Figure 5.4. We focus on this test, which is the more complicated in respect to nodes in the Configuration Tree and dependencies of the Abstract Services because it experiences the most dramatic changes than all the other test cases. Surprisingly, the third scenario of this test case finds the optimal solution in similar times as when we stopped the search after finding one. The rest of the test cases are available to the reader for reference in Appendix B.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Solutions</th>
<th>Failures</th>
<th>Branches</th>
<th>Wall Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Test 3</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sla1</td>
<td>1</td>
<td>5</td>
<td>9</td>
<td>6006</td>
</tr>
<tr>
<td>sla2</td>
<td>3</td>
<td>1</td>
<td>6</td>
<td>5353</td>
</tr>
<tr>
<td>sla3</td>
<td>2</td>
<td>14</td>
<td>30</td>
<td>5388</td>
</tr>
<tr>
<td><strong>Test 4</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sla1</td>
<td>3</td>
<td>0</td>
<td>4</td>
<td>34606</td>
</tr>
<tr>
<td>sla2</td>
<td>2</td>
<td>12</td>
<td>26</td>
<td>36700</td>
</tr>
<tr>
<td>sla3</td>
<td>2</td>
<td>46</td>
<td>91</td>
<td>34850</td>
</tr>
<tr>
<td><strong>Test 5</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sla1</td>
<td>3</td>
<td>1</td>
<td>6</td>
<td>4925</td>
</tr>
<tr>
<td>sla2</td>
<td>1</td>
<td>5</td>
<td>10</td>
<td>4867</td>
</tr>
<tr>
<td>sla3</td>
<td>1</td>
<td>7</td>
<td>13</td>
<td>4603</td>
</tr>
<tr>
<td><strong>Test 6</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sla1</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>39670</td>
</tr>
<tr>
<td>sla2</td>
<td>3</td>
<td>7</td>
<td>18</td>
<td>37204</td>
</tr>
<tr>
<td>sla3</td>
<td>8</td>
<td>41</td>
<td>88</td>
<td>40042</td>
</tr>
<tr>
<td><strong>Test 7</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sla1</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>5313</td>
</tr>
<tr>
<td>sla2</td>
<td>3</td>
<td>7</td>
<td>18</td>
<td>4843</td>
</tr>
<tr>
<td>sla3</td>
<td>5</td>
<td>15</td>
<td>31</td>
<td>4753</td>
</tr>
<tr>
<td><strong>Test 8</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sla1</td>
<td>7</td>
<td>3</td>
<td>17</td>
<td>45462</td>
</tr>
<tr>
<td>sla2</td>
<td>3</td>
<td>17</td>
<td>38</td>
<td>39018</td>
</tr>
<tr>
<td>sla3</td>
<td>10</td>
<td>73</td>
<td>161</td>
<td>44467</td>
</tr>
</tbody>
</table>

Table 5.3 Results for all test cases – scenarios with BAB
Figure 5.4a Test case 8 - first scenario (sla1)

Figure 5.4a Test case 8 - second scenario (sla1)

Figure 5.4a Test case 8 - third scenario (sla3)
6 Conclusion and future work

6.1 Conclusion

In this thesis, we introduce the problem of Cloud Service Orchestration and deployment over a Distributed Cloud Network. We propose a model which maps all the relevant properties of nodes and links in the network into JSON files, in order to be easily extended or used in conjunction with existing representations. We also propose a model for the Software components in the same format, capturing their essential characteristics into two categories, the Abstract and Atomic Services. Using this distinction, we can use combinations of them in order to construct higher level end-to-end services.

The proposed model is firstly implemented using the Google OR-Tools library and then evaluated over a variation of parameters in respect to the number of infrastructure nodes, links, software components and maximum nodes in the Configuration Tree. The results are encouraging, showing efficient solution finding and overall acceptable behavior when the search space grows. The model is also evaluated with the usage of Branch and Bound algorithm, showing promising results.

6.2 Future work

The results obtained during the evaluation part of the thesis, inspire the exploration of more complicated cases, derived from the expectations of features such a system should have to be applicable in real world scenarios.

**Considering paths with more than one hop**

In the proposed model, we only consider directly connected nodes. A system that can be used in real world scenarios, would require the solver to select infrastructure nodes that are not directly connected. In fact, two nodes can be indirectly connected through several paths and finding the path that adds the less latency or price to the solution is an instance of Min Cost Flow problem [20].

**Ordering of software components and infrastructure nodes**

One other area that might worth exploring is the ordering of software components and infrastructure nodes depending on the price or latency they add in a solution. Then by using a minimum value selection in the branching strategy we might be able to find faster the optimal solution in the search tree.

**Dynamic Value Selection**

Finally, another potential extension could be the dynamic value selection. To illustrate this better with an example, the solver knows the maximum price and latency it wants to find in a solution. Whenever it has to make a choice on which value to select for a variable, it can
consider any previous choice already made and by using a cost function, in order to identify if it needs to select software components that add the less cost or latency to the solution.
References


Appendix A

Figure A1 Test case 4 - first scenario (sla1)

Figure A2 Test case 4 - second scenario (sla2)

Figure A3 Test case 4 - third scenario (sla3)
Figure A4 Test case 5 – first scenario (sla1)

Figure A5 Test case 5 - second scenario (sla2)

Figure A6 Test case 5 - third scenario (sla3)
Figure A7 Test case 6 – first scenario (sla1)

Figure A8 Test case 6 – second scenario (sla2)

Figure A9 Test case 6 – third scenario (sla3)
Figure A10 Test case 7 – first scenario (sla1)

Figure A11 Test case 7 – second scenario (sla2)

Figure A12 Test case 7 – third scenario (sla3)
Figure A13 Test case 8 – first scenario (sla1)

Figure A14 Test case 8 – second scenario (sla2)

Figure A15 Test case 8 – third scenario (sla3)
Appendix B

Figure B1 Test case 3 - first scenario (sla1)

Figure B2 Test case 3 - second scenario (sla2)

Figure B3 Test case 3 - third scenario (sla3)
Figure B4 Test case 4 - first scenario (sla1)

Figure B5 Test case 4 - second scenario (sla2)

Figure B6 Test case 4 - third scenario (sla3)
Test case 5 - first scenario (sla1)

Test case 5 - second scenario (sla2)

Test case 5 - third scenario (sla3)
Figure B10 Test case 6 - first scenario (sla1)

Figure B11 Test case 6 - second scenario (sla2)

Figure B12 Test case 6 - third scenario (sla3)
Figure B13 Test case 7 - first scenario (sla1)

Figure B14 Test case 7 - second scenario (sla2)

Figure B15 Test case 7 - third scenario (sla3)