Mixed-Tenancy Platform provisioning SaaS Applications:
System Alteration Over Life Cycle

Approved: _____ Date: 5/23/2013
Committee Chair

Approved: _____ Date: 20. May 2013
Committee Member

Approved: _____ Date: 23-May 2013
Committee Member

h_da

HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

UNIVERSITY OF WISCONSIN
PLATTEVILLE

Mixed-Tenancy Platform provisioning SaaS Applications: System Alteration Over Life Cycle

A Thesis presented to the Graduate Faculties
University of Wisconsin-Platteville (UWP) and
University of Applied Sciences Darmstadt (h_da)

In Partial Fulfillment of the Requirement for the Degree

Master of Science

By
S.N. Candide Orou-Yorouba

Committee Chair:	Yan Shi, Ph.D. (UWP)
Committee Member:	Prof. Dr. Urs Andelfinger (h_da)
Committee Member:	Prof. Dr. Christoph Wentzel (h_da)

Start Date:	June 1st, 2013
End Date:	December 1st, 2013

Statement of Authorship

I hereby declare that this thesis is written on my own solely by using the sources quoted in the thesis body. All contents (including text, figures, and tables) which are taken from sources, both verbally and in terms of meanings are clearly marked and referenced.

The thesis has not been previously submitted to any other examination board, neither in this nor in a similar form.

Frankfurt am Main, November 29th, 2013

S.N. Candide
Orou-Yorouba

Acknowledgments

On my journey accomplishing my thesis, many different individuals played a significant role. There were highlights and pitfalls on this journey. However, despite the difficulties, I was able to achieve the goals that I had set for this thesis thanks to the following key individuals.

Foremost, I would like to express my sincere gratitude to my committee members, Dr. Yan Shi, Dr. Urs Andelfinger and Dr. Christoph Wentzel for their guidance, continuous support, and motivation. Their insightful comments, and challenging questions helped me during the time of my research and writing this thesis.

My sincere appreciation also goes to Dr. Stephan Verclas from *T-Systems International GmbH* for giving me the opportunity to work on this research. Moreover, a special thank you goes to my enthusiastic company adviser, Stefan T. Ruehl, who was a great mentor throughout the entire process. Thank you for the encouragement you gave me and the trust you had in me. Another thank you goes to my fellow colleagues who always had an open ear and time for brain storming.

Last but not the least, I would like to thank my parents for their unyielding support. Finally, I would also like to thank my friends in the U.K. and in the U.S. for proof-reading this document and giving me sharp-witted feedback.

You all made it happen! Thank you!

Abstract

The delivery of software to the customers over the internet - Software as a Service (SaaS) - is becoming a dominant mechanism for the consumption of software. This generally follows a combination of two different architecture models, differentiated by the level of isolation: single or multi-tenancy. A research project at the Clausthal University of Technology introduced a new architecture model, namely the mixed-tenancy architecture, which combines both multi and single-tenancy on the component of a composite application. This enables customers to choose which resource (e.g. server, cluster, application instance, etc.) of which application component (e.g. finance component, logistic component, etc.) they are (or are not) willing to share with other specific tenants. Another research at the Darmstadt University of Applied Science proposed a prototype to prove the feasibility of this approach by realizing a mixed-tenancy platform. This research demonstrated the real-world applicability of the Mixed-Tenancy approach.

Though, the variability over time of the customers' sharing requirements has yet to be addressed. In fact, new customers may need to be granted access to the application, others may need to be removed or moved. Also, due to different reasons, customers may decide to alter their initial sharing requirements, which then leads to a new deployment configuration.

To tackle this challenge, this thesis introduces an approach based on the abstraction of the graph theory concepts, and which enables service providers to transform the old deployment configuration into the new one with the minimum cost possible. The idea is to build a weighted multigraph representing the possible transformations and eventually apply a weighted shortest path algorithm, since the transformations are connected by a cost function. This approach led to exponentially growing multigraphs, whose processing is unfeasible. That is, only a portion of the graph can be considered, and the optimal route can only be approximated. The thesis looks at different algorithms available in the literature, then implements and evaluates a solution for finding an optimal cost effective path in the partially known graph.

Abstract (German Version)

Heutzutage ist die Bereitstellung von Software an den Kunden über das Internet - Software as a Service (SaaS) - weit verbreitet. Dies erfolgt in der Regel durch zwei verschiedene Architekturmodelle, die sich durch den Grad der Isolation unterscheiden: Einzel- und Mehrmandantenfähigkeit, bzw. *Single-Tenancy* und *Multi-Tenancy.* Ein Forschungsprojekt an der Technische Universität Clausthal führte ein neues Architekturmodell ein, das *Mixed-Tenancy* Modell. Dieses Modell steht für die verteilte Bereitstellung von mehrfach instanziierten Komponenten einer Applikation. Dies ermöglicht den Kunden die Auswahl der Ressourcen (z.B. Server, Cluster, Applikationsinstanz, usw.), die sie bereit sind zu teilen. Diese Kundenwünsche werden in einer sogenannten *Deployment Konfigurationsdatei* zusammengefasst. Im Rahmen einer weiteren Forschung an der Hochschule Darmstadt wurde eine prototypische *Mixed-Tenancy* Plattform entwickelt. Ziel war es, die Realisierbarkeit dieses Ansatzes zu beweisen. Diese Plattform dient dazu, Multi-Tenancy-Applikationen anhand der Kundenvorgaben automatisch bereitzustellen.

Auf der Plattform lässt sich nur die initiale Deployment Konfiguration bereitstellen. Jedoch könnten Kunden aus verschiedenen Gründen ihre ursprünglichen Anforderungen ändern. Dies führt zu einer neuen Deployment Konfiguration. Das Ziel dieser Arbeit ist es, die alte Deployment Konfiguration in die Neue mit minimalen Kosten umzuwandeln ohne die ganze Applikation neu bereitstellen zu müssen. Der vorgeschlagene Lösungsansatz basiert auf dem Konzept der Graphentheorie. Die Grundidee besteht darin einen gewichteten Multigraph aufzubauen. Dieser Graph enthält alle möglichen Deployment Konfigurationen, die aus der initialen Deployment Konfiguration erreichbar sind. Die Knoten des Graphen sind durch Kanten verbunden, welche die Kosten der Transformation enthalten. Schließlich soll mit einem kürzesten Pfad Suchalgorithmus die kosteneffizienteste Route von der initialen Deployment Konfiguration zu der Ziel Deployment Konfiguration gesucht werden. Es wird gezeigt, dass dieser Ansatz zu exponentiell wachsenden Graphen führt und dass der Graph nur teilweise verarbeitet werden kann. Daraufhin untersucht diese Arbeit verschiedene Algorithmen aus der Fachliteratur. Anschließend implementiert und bewertet sie eine Lösung, die es ermöglicht den kürzesten Pfad in dem teilweise bekannten Graph zu finden.

Contents

List of Figures

List of Tables

List of Algorithms

List of Listings

> *The mere formulation of a problem is far more often essential than its solution, which may be merely a matter of mathematical or experimental skill. To raise new questions, new possibilities, to regard old problems from a new angle requires creative imagination and marks real advances in science.*
>
> – *Albert Einstein*

1 Motivation & Objectives

1.1. Motivation

Software as a Service (SaaS) is becoming a dominant mechanism for the consumption of software over the internet [Sengupta and Roychoudhury, 2011]. The delivery of the software to the customers (also known as tenants) in the cloud generally follows a combination of two different architecture models [Chong and Carraro, 2006]: *single* and *multi-tenancy*, which are differentiated by the level of isolation. In a single tenancy architecture model, a single instance of an application serves only one tenant, whereas on the other hand, in a multi-tenancy architecture model, a single instance of an application serves multiple tenants. Tenants unarguably prefer the former model because of the strict isolation of the application instances [Chong et al., 2006]. However, because of the waste of resources that this model generates, the service providers prefer the latter approach, in which the resources are more efficiently used [Bezemer and Zaidman, 2010a].

To overcome this challenge for the satisfaction of both tenants and providers, Ruehl et al. [Ruehl et al., 2012] introduced the concept of Mixed-Tenancy Software Model, in which the tenants state which components of the application[1] or even which level of infrastructure (e.g. server, farm/cluster) shall be multi-tenant and which ones rather be single-tenant. Moreover, the tenants can decide with whom this resource (application component or infrastructure level) shall rather be/not be shared with. Using an optimization algorithm based on the constraints defined by the tenants, Ruehl et al. generate

[1] This applies to service-oriented applications (SOA).

an optimal deployment configuration, which describes the best way to deploy the system by stating, for example, how many servers, clusters and instances of an application components are needed to deploy the system at a minimum cost. Based on this deployment configuration, Reinhardt [Reinhardt, 2013] developed a prototype of a Mixed-Tenancy Platform, which addressed and solved two challenges:

- Deployment of an application according to the deployment configuration
- Establishment of the communication among components

The result of his research is a platform that provides the initial infrastructure computed by Ruehl et al. and which enables the communication among the application's components in a very specific way. The initial constraints expressed by the tenants are taken into account to set up and deploy the Mixed Tenancy Application. However, for different reasons, such as financial reasons or to gain a competitive advantage, customers may change their deployment constraints; new customers may also need to be granted access to the application; at the same time, other customers' access to the application may have to be revoked. The interesting question arising from these few facts, and which is to be investigated in this thesis, is:

> How is it possible to transform any given initial deployment configuration to any given target deployment configuration at a minimum cost?

1.2. Delimitation of the study & Objective target

The study conducted in this thesis is not about software migration strategies as some work in this area has already been done and solutions for this problem already exist in the industry [Piazza, 2009, Transvive, 2011]. Indeed, the term *software migration* used in this context does not refer to techniques of moving a software from an old system architecture to another (e.g. migration to object-oriented software development [Auer and Dobler, 2000]) as the software and the architecture remain the same. Furthermore, it does not refer to techniques of moving an existing application to cloud environments. Indeed, Fehling et al. already investigated this topic and presented in [Fehling et al., 2013] diverse best practices as well as an implementation of how to migrate of a service-based application to the cloud. In this thesis, however, the migrations that are mentioned, refer to a change of (deployment) settings within the same application.

The goal of the study conducted in this thesis is to determine a set of sequential operations that are to be realized in order to transform any given initial deployment configuration into any given new or target configuration. Though, the realization of the deployment is

part of another research, whose results will be presented at the *Darmstadt University of Applied Science* by the end of January 2014.

1.3. Significance of the study

A research by Gartner showed a worldwide growth of the use of cloud services with SaaS being the second largest fast-growing area [Pettey and van der Meulen, 2012a] [Pettey and van der Meulen, 2012b]. Mostly driven by cost-saving expectations, tenants increasingly choose to use SaaS instead of local applications [Blokdijk, 2008] [Bayrak et al., 2011]. Moreover, tenants are reluctant to share their resources (e.g. database server and/or instance) with one another because they fear data breaches. Indeed, a recent research by the Cloud Security Alliance identified data breaches in the cloud, as one of the top problems in Cloud Computing [Alliance, 2013]. As a matter of fact, in November 2012, researchers from the University of North Carolina, the University of Wisconsin-Madison and the RSA Corporation released a paper [Zhang et al., 2012] describing how a virtual machine could use a side channel timing information to extract private cryptographic keys being used in other virtual machines on the same physical server.

Consequently, a SaaS provider would, in general, try to drive commonality amongst the requirements of different tenants, and at best, offer a fixed set of customization options. However, many tenants would also come with custom requirements (e.g. Tenant#1 would share an instance with Tenant#2 but not with Tenant#3), which may be a pre-requisite for them to adopt the SaaS system [Jiang et al., 2012]. Furthermore, these requirements are very likely to change over time.

So far, the Mixed-Tenancy platform [Reinhardt, 2013] [Ruehl et al., 2012] on which this study is built upon, does not address this important aspect. Due to financial reasons or to gain a competitive advantage or other reasons, a tenant may request to change the sharing requirements initially stated. The "easiest" way for the service provider would be to delete the old configuration, and redeploy the system with the new configurations. This approach could lead to the tenants loosing data, hence all the work done on the application up to that point. Obviously, this approach is not an option for the tenants, as this would lead to a huge financial lost, several millions of US Dollars [Institute, 2013]. In fact, the deployment configuration modifications should be done without hindering the existing tenants. This means that implementing the migration plan should not interfere with the services provided to the other existing tenants. This induces the strong requirement of zero-downtime, as downtime per hour can go up to $4,500,000 depending on the type of business [Bezemer and Zaidman, 2010b]. To avoid this, the approach proposed in this thesis gives the service provider more flexibility to transform the initial deployment

configuration in the newer one, without having to redeploy the whole application.

1.4. Methodology & Thesis Overview

In order to compute a migration plan to transform an initial deployment configuration to a target deployment configuration, this thesis makes use of the concept of the graph theory. The basic idea is to build a weighted multigraph containing different deployment configurations which represent the nodes of the graph. The initial and the target configurations should also be part of the graph. The migration plan would then consist of the cost-efficient route between the initial and the target configuration. The solution that is proposed is purely technical and does not take into account any underlying process. Furthermore, performance as well as security are two other topics that will not be discussed in this study.

Figure 1.1 depicts the overall architecture.

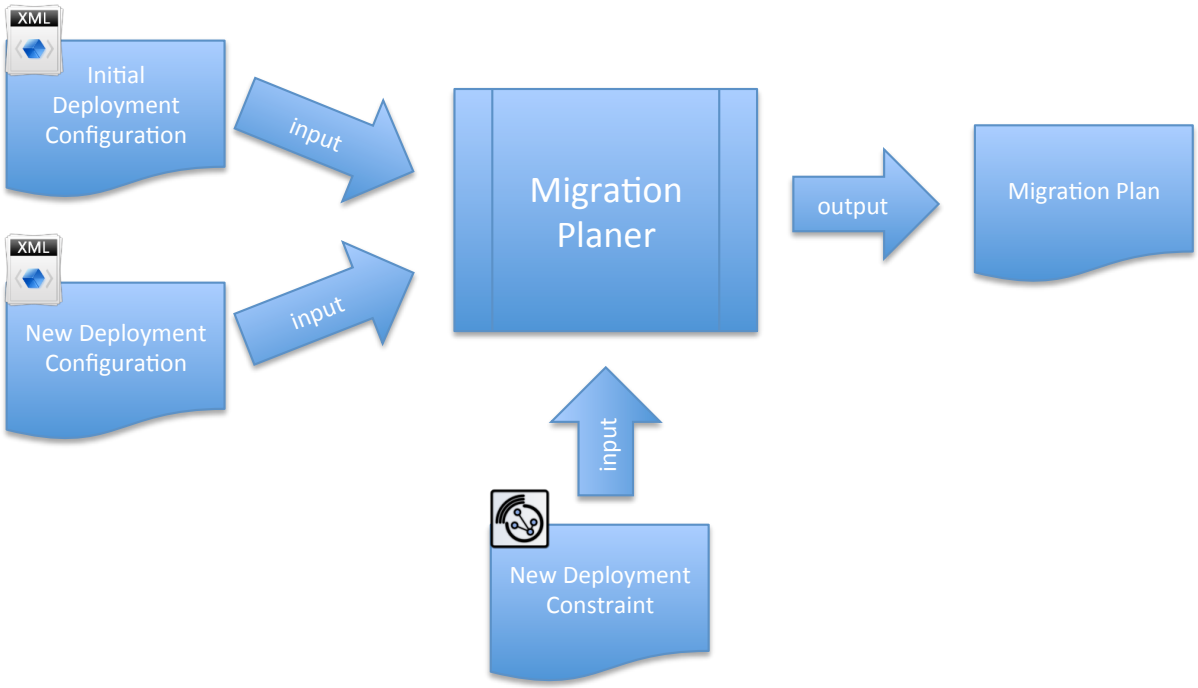


Figure 1.1.: Overall Architecture

The *Migration Planner*, which represents the main component of this architecture, takes as input both the initial and the target deployment configurations. The *Migration Planner* also takes the new deployment constraints as input. These contain the customers' sharing requirements. The output is a *Migration Plan* containing the successive operations, which are to be implemented in order to transform the initial configuration to the target configuration.

This thesis is composed of eight chapters. This first chapter stated the question that this thesis deals with. This chapter also defined the limitations of this study as well as its significance.

Chapter 2 gives more details on the subject matter. This chapter introduces various paradigms and concepts that are essential for understanding this thesis. In fact, it defines different terminologies in regards of the sense in which they are used in this context. In the literature, these terminologies often have various meanings. The aim of this chapter is to specify which of these meanings will be used throughout this thesis. Following, Chapter 3 makes a quick review of the work done so far in the relevant literature. The purpose is to prove that the problem this thesis addresses, was not addressed so far.

Chapter 4 explains the problem in question in this research in different words using a real-world scenario. Here, it traces the problem back to a specific field of Computer Science. This research is continued with different algorithms that are stemming from this field and which are analyzed in Chapter 5 as an attempt of a solution. This chapter ends by identifying the algorithm, which suits best in this context. Chapter 6 outlines the important steps of its implementation by introducing different technologies. The algorithm is eventually evaluated in Chapter 7.

The thesis ends with Chapter 8 which summarizes the results of this thesis. Finally, this chapter also briefly addresses other topics that could possibly derive from this research, and which could be worked on in the future.

> *The world has a soul and whoever understands that soul can also understand the language of many things*
>
> – *Paulo Coelho*

2

Fundamental Terms

Before going deeper into the research topic in further chapters, this chapter lays the foundations for a better understanding of the research area. Some important terms are defined in the context of this thesis. This chapter starts by introducing both the concepts of *Cloud Computing* and *Software as a Service*. Furthermore, the single and multi-tenancy architectures are described before defining the concept of Mixed-Tenancy architecture, which this research is built on. The chapter ends by giving a general overview about graphs and online algorithms, as further chapters show that they are the basis for the solution proposed in this thesis.

2.1. Cloud Computing

Cloud Computing is a new term for a long-held dream of computing as a utility [Parkhill, 1966], which rose up as a commercial reality in the Information Technology (IT). Cloud Computing is about using computer systems over the internet. Nevertheless, there is no existing agreement on a unique definition of the term.

Armbrust et al. state that,

> *Cloud Computing refers to both the applications delivered as services over the Internet and the hardware and systems software in the datacenters that provide those services. The services themselves have long been referred to as Software as a Service (SaaS). The datacenter hardware and software is what*

we will call a Cloud [Fox et al., 2009].

For the German Federal Office for Security in Information Technology (*Bundesamt für Sicherheit in der Informationstechnologie* - BSI),

> *Cloud Computing designates the providing, the use and the billing of IT services, adapted dynamically to the customers' needs, over a network. The providing and the use of these services are thereby exclusively made available through defined technical interfaces and protocols. The range of services offered as part of Cloud Computing includes the entire information technology spectrum, and also includes Infrastructure (such as processing power, memory capacity), Platforms and Software* [BSI, 2011].

Another German institution, namely the Federal Association for Information Technology, Telecommunications and New Media (*Bundesverband Informationswirtschaft, Telekommunikation und neue Medien* - BITKOM), which represents over 1,700 companies, defines Cloud Computing as

> *A form of need-based and flexible use of IT systems. These are provided in real time as a service over the Internet and billed on usage. Thus, cloud computing allows a redistribution of the capital according to the operating expenses* [BITKOM, 2009].

The Research Institute Gartner sees Cloud Computing as

> *A style of computing where scalable and elastic IT capabilities are provided as a service to multiple customers using Internet technologies* [Plummer, 2009].

The American National Institute of Standards and Technology's (NIST) definition of Cloud Computing is, however, the most widely accepted and is viewed as a valuable contribution toward providing a clear understanding of Cloud Computing Technologies and Cloud Services [Liu et al., 2011].

> *Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction* [Mell and Grance, 2011].

This is the definition adopted in this thesis. Figure 2.1 illustrates the main parts of this definition.

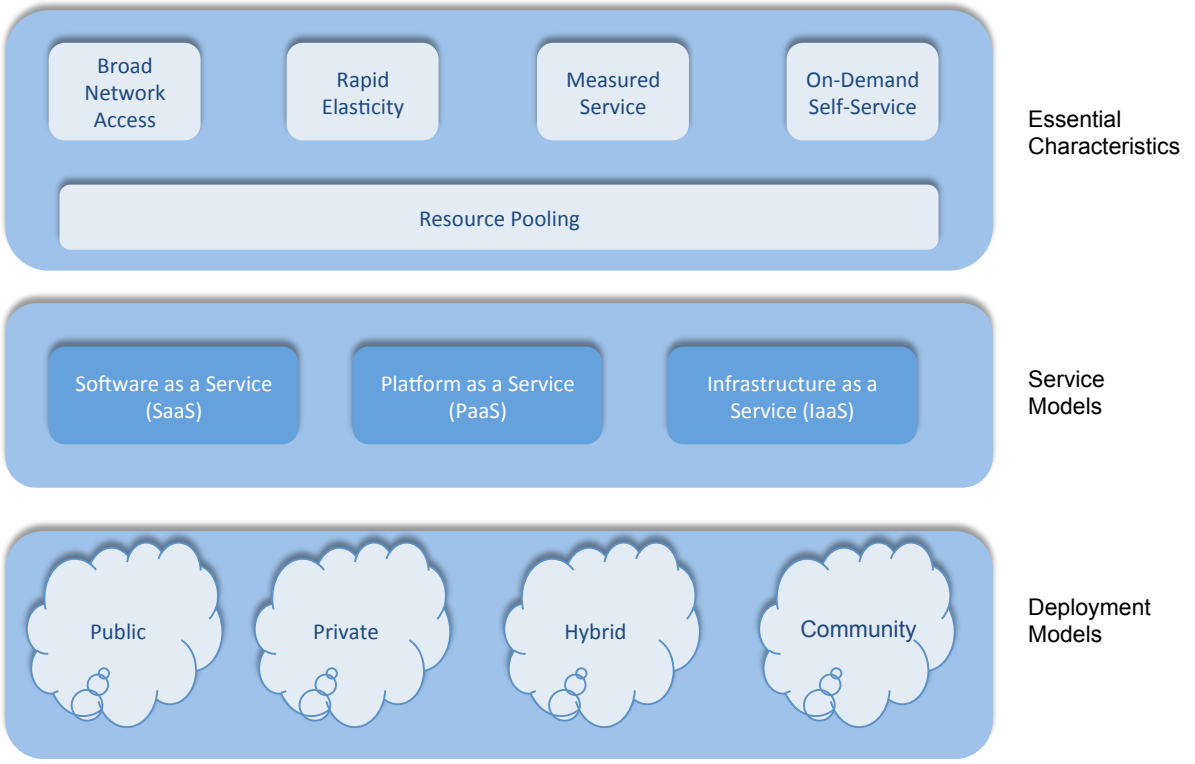


Figure 2.1.: NIST Visual Model of Cloud Computing Definition [Brunnette and Mogull, 2009]

As shown in Figure 2.1, the NIST definition provides a unifying view of five essential characteristics that all cloud services exhibit:

On-demand self-service : A consumer can unilaterally provision computing capabilities such as server time and network storage as needed automatically, without requiring human interaction with a service provider [Mell and Grance, 2011].

Broad network access : Capabilities are available over the network and accessed through standard mechanisms that promote use by heterogeneous thin or thick client platforms (e.g., mobile phones, laptops, and PDAs) as well as other traditional or cloud-based software services [Mell and Grance, 2011].

Resource pooling : The provider's computing resources are pooled to serve multiple consumers using a multi-tenant model (see Section 2.3), with different physical and virtual resources dynamically assigned and reassigned according to consumer demand. There is a degree of location independence in that the customer generally has no control over or knowledge of the exact location of the provided resources [Mell and Grance, 2011].

Rapid elasticity : Capabilities can be rapidly and elastically provisioned - in some cases automatically - to quickly scale out; and rapidly released to quickly scale in. To the consumer, the capabilities available for provisioning often appear to be unlimited

and can be purchased in any quantity at any time [Mell and Grance, 2011].

Measured service : Cloud systems automatically control and optimize resource usage by leveraging a metering capability at some level of abstraction appropriate to the type of service (e.g. storage, processing, bandwidth, or active user accounts). Resource usage can be monitored, controlled, and reported - providing transparency for both the provider and consumer of the service [Mell and Grance, 2011].

Furthermore, NIST acknowledges three service models, which are to be made available to cloud users, namely *a*) cloud software as a service (SaaS), *b*) cloud platform as a service (PaaS), *c*) cloud infrastructure as a service (IaaS). The model followed in this thesis is the first one, **SaaS**, which Section 2.2 discusses intensively. **PaaS** provides the consumer with an infrastructure, where consumer-created or acquired applications created using programming languages and tools supported by the provider can be deployed. **IaaS** provides the users processing, storage, network and other fundamental computing resources enabling the deployment and hosting of arbitrary software, including operating systems.

Finally, NIST also identifies four deployment models about how the infrastructure that provides these services can be shared regardless of the service model utilized:

Public Cloud : The infrastructure is made available to the general public or a large industry group.

Private Cloud : The infrastructure is made available to a single person or organization.

Community Cloud : The infrastructure is shared by a number of organizations and supports a group or community that has the same concerns.

Hybrid Cloud : The infrastructure is a composition of two or more clouds (private, public, community).

2.2. Software as a Service (SaaS)

An application is referred to as Software as a service, or just SaaS, when it is being provided as a service according to the definitions of Cloud Computing given in Section 2.1. Hence, SaaS can be defined as

> *A software deployed as hosted service and provided by the customer over the internet* [Mell and Grance, 2011].

The consumer is provided with the capability to use the provider's applications running on a cloud infrastructure. These applications can be accessed from various client de-

vices such as mobile phones, tablets or computers using a web browser [Liu et al., 2011]. The provider is responsible for the management of the entire underlying infrastructure, including network, servers, operating systems, storage, or even individual application capabilities, with the possible exception of limited user specific application configuration settings [Liu et al., 2011]. SaaS bears various advantages for both service consumers and service providers, specifically the fact that it is faster to implement as on-premises Software; in addition, it can be less expensive for the consumer [Gartner, 2009]. Furthermore, Gartner reports a rapid worldwide growth of SaaS in the industry and expects more in the next few years [Pettey and van der Meulen, 2012a] [McCall and van der Meulen, 2011] [Pettey and van der Meulen, 2012b] [Desisto et al., 2012].

Service consumers do not have any software maintenance task anymore; instead, they are offered a multitude of services to support their business. SaaS enables these services to be purchased at lower cost as the consumers pay only for those services that will be used. Moreover, there is mostly no extra hidden costs. The services are offered as subscriptions, which include all costs, as opposed to on-premises software where the consumers may have to pay extra for maintenance, training, or troubleshooting for example. In addition, as the services are provided over the internet (within an internet browser), the consumers do neither need to maintain an IT-Infrastructure, nor need to concern themselves with upgrading the software version . The consumer is given the choice to subscribe only for the services required by his business, which also come with a Service Level Agreement (SLA). The SLA defines all the commitments made by the provider to the customer related with maintenance, reliability, availability, security, performance, etc. [Verma, 2011].

The advantages for service providers are also fairly robust. The SaaS provider can now easily install, centrally manage, maintain and control the software from their own workstations. Furthermore, the providers would benefit from a higher demand (as SaaS can be less expensive for organizations [Gartner, 2009] or attract organizations having less IT infrastructure or capital to purchase licensed software). This would also enable more accurate predicted cash-inflows and good Customer Relationship Management (CRM). SaaS is less expensive for the consumers as their customers may share the same kind of software, services and resources, thus resulting in using less manpower [Verma, 2011].

2.3. Single & Multi Tenancy Applications

According to F. Chong et al., researchers at *The Microsoft Corporation*, trust, or the lack thereof, is the number one factor blocking the adoption of software as a service [Chong et al., 2006]. There is no doubt that data (e.g. data about product, customers, employees, etc.) are the most important asset for any business, but in order to take advantage of

SaaS, the customer (e.g. an organization) should trust the SaaS provider with its data to be controlled centrally. Different data architecture models have been developed to ensure the customer of the safety of their data. These models are referred to as *single-tenancy* and *multi-tenancy* data architecture. In the context of this thesis, the terms "customer", "consumer", and "tenant" will be used interchangeably to designate the SaaS consumer. In case the tenant is an organization, the employees will be referred to as "users".

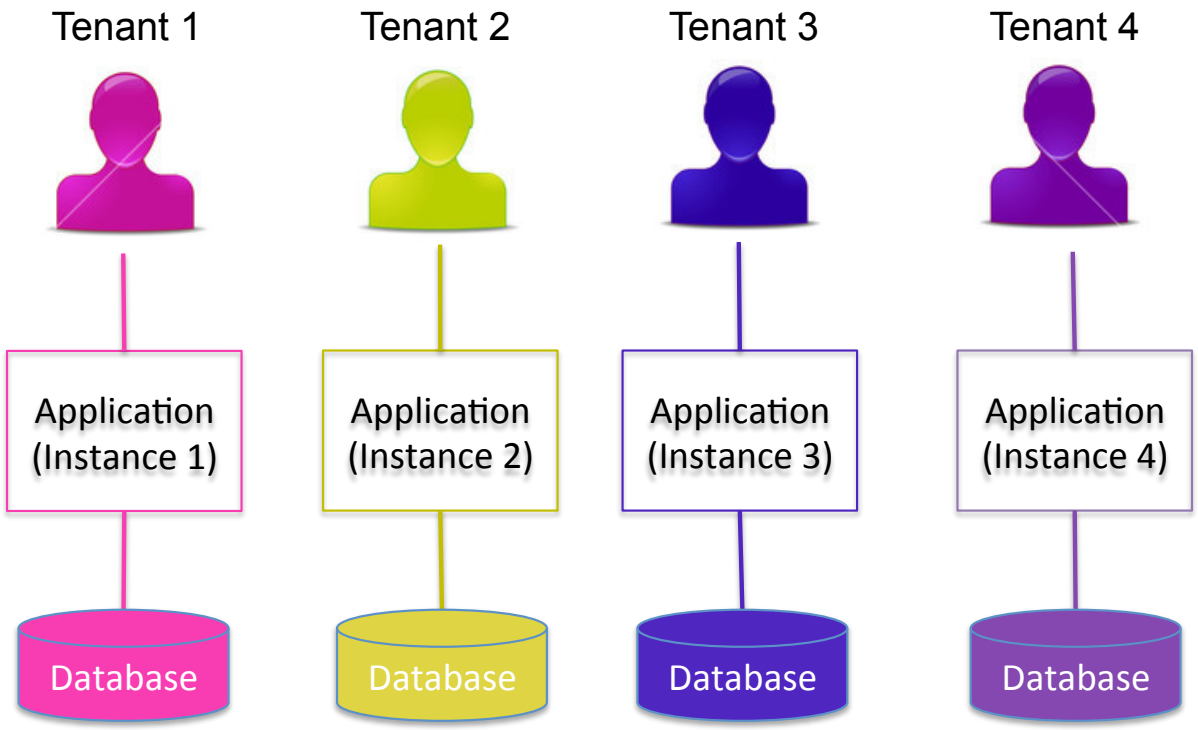


Figure 2.2.: The Single-Tenancy Data Model

In a single tenancy SaaS architecture, as illustrated in Figure 2.2, each tenant has its own copy of the software or a separate instance thereof. In addition, the underlying supporting infrastructure is set up solely for the one particular tenant, meaning the tenant has a distinct database, which is either running on an individual server or sometimes segregated using extensive security controls to create an individual virtual server network. The tenants, in such an architecture model, can have the software and its functionality fully configured to their own "taste". Furthermore, the level of isolation is very high, which makes the system very reliable and almost non-vulnerable to external security threats. For example, the risk of data stealing is relatively small [Bezemer and Zaidman, 2010b].

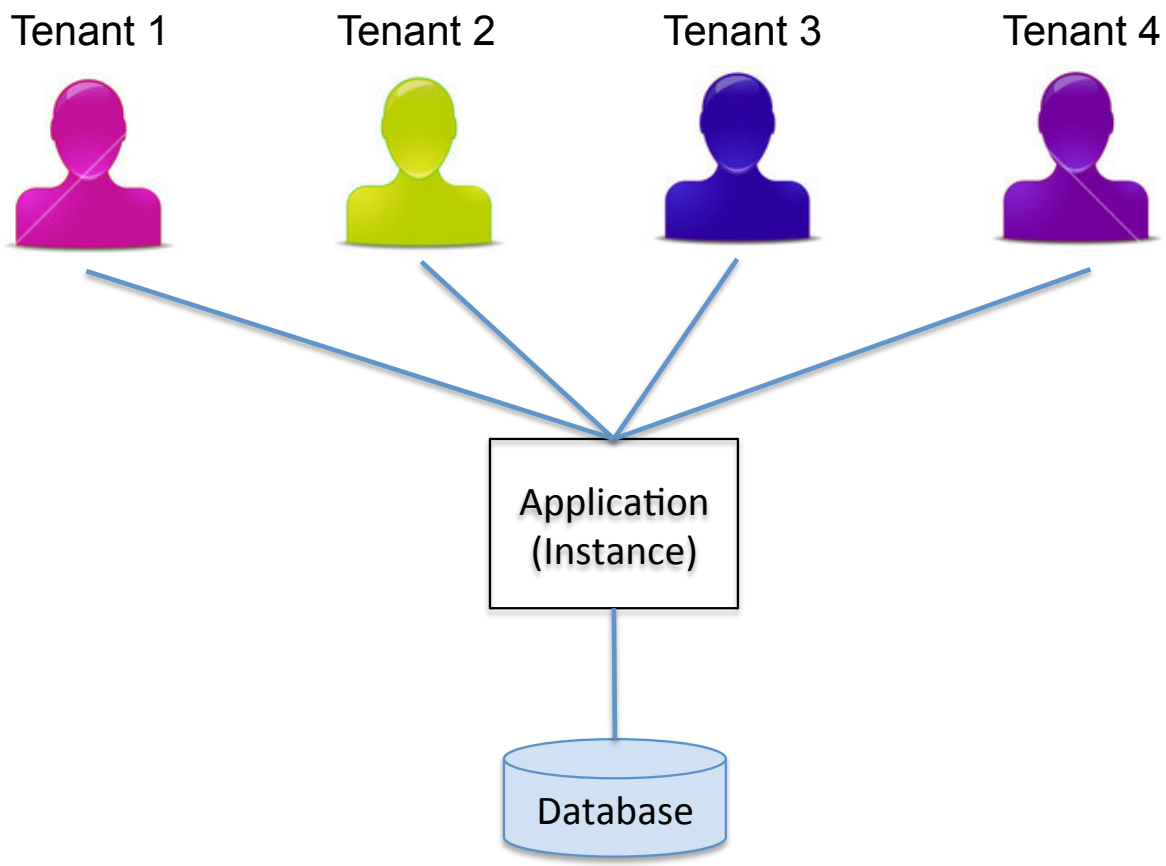


Figure 2.3.: A Multi-Tenancy Data Model

On the other hand, a Multi-Tenancy Application as shown in Figure 2.3 and which should not be confused with Multi-User Application, enables the tenants to share the same hardware resources, by offering them one shared application and a database instance, while allowing them to configure the application to fit their needs as if it runs on a dedicated environment [Bezemer and Zaidman, 2010b]. Thus, the tenants all have their data on the same server. However, there exist different approaches to separate the data, while being on the same server [Chong et al., 2006]:

Separate Database: The tenants have their data stored in different databases, but on the same server.

Shared Database, Separate Schema: House multiple tenants in the same database, with each tenant having its own set of tables that are grouped into a schema created specifically for the tenant.

Shared Database, Shared Schema: Use the same database and the same set of tables to host multiple tenants' data. A given table can include records from multiple tenants stored in any order; a Tenant ID column associates every record with the appropriate tenant.

From the SaaS provider point of view, a multi-tenancy data model allows a higher utilization of hardware resources as well as easier and cheaper application maintenance [Bezemer and Zaidman, 2010b]. This results in lower overall costs, as well as higher market competitiveness as services can be offered at a lower price than competitors. For the SaaS consumer, the cost factor plays an important role in the process while deciding to adopt such a model. Nevertheless, many consumers do not feel comfortable taking the risk of sharing important and confidential information, particularly if the application stores the user's data [Verma, 2011] as they fear any security breach [Everett, 2009].

2.4. Mixed Tenancy Applications

As can be deduced from previous Section 2.3, a Single-Tenancy Application ensures the consumer with a higher level of security for more cost, whereas in the less expensive multi-tenant environment, a security breach can result in the exposure of data to other, possibly competitive, tenants. This makes security issues such as data protection [Guo et al., 2007] very important. Many cases of data theft have occurred in the past [Alliance, 2013]. The customers would unarguably prefer a single-tenancy approach at lower cost [Chong et al., 2006]. The service providers, on the other hand, prefer a multi-tenancy approach where they have less maintenance work to do and can have the resources better utilized [Bezemer and Zaidman, 2010a]. In order to find a compromise between these two points of view, the concept of Mixed-Tenancy (MT) Data Model has been introduced [Ruehl et al., 2012].

MT can only be applied to component-based applications, such as Enterprise Resource Planning (ERP) systems (e.g. OpenERP). It gives the tenants the possibility to decide, whether they would share an instance of a given component, and if so, which other tenants they would share it with. While, for privacy reasons, the tenants cannot choose which other specific tenant they would share the component with, they can however indicate which other group of tenants they are willing to share/not share an instance with. For example, consider a market leading Investment Bank in the United States of America, and another big bank in Germany, as tenants. Due to privacy reasons, they don't know that they're both customers from the same MT-Provider. As competitors, they would obviously not agree on sharing an instance of the "Finance and Accounting" component for example; one of them could request not to share the specific instance (or any other instance) of this specific component (or any other component) with any other competitor. The requirements statement could go further, as tenants could even specify which other regions they are willing to share an instance with or not. For example, a big company in the United States of America could state that it would not share the component "Logistic" with any other European Company, but rather with an Asian company. The reader is

asked to refer to Appendix A for details about other deployment models. It is important to remark that the Deployment Constraints are mutually exclusive: two tenants may share an instance only if they both agree; if one does not, no sharing occurs [Ruehl et al., 2013].

Application Component Instances (ACI) are just an example of the resources that a tenant may decide to share. Those resources are decomposed into levels and are referred to as *Deployment levels* (DL). A DL, as illustrated in Figure 2.4 represents a level of infrastructure stack (such as a Virtual Machine - VM - or an ACI), and should be able to be instantiated more than once. Moreover, each lower DL should be able to run multiple instances of its higher corresponding DL simultaneously. For example, a VM and an Application Server (AS) can be defined as DLs, as multiple instances of an AS can run on a VM. However, VM and Operating Systems (OS) are not suitable as DLs as a only one Operating System can run on a dedicated VM.

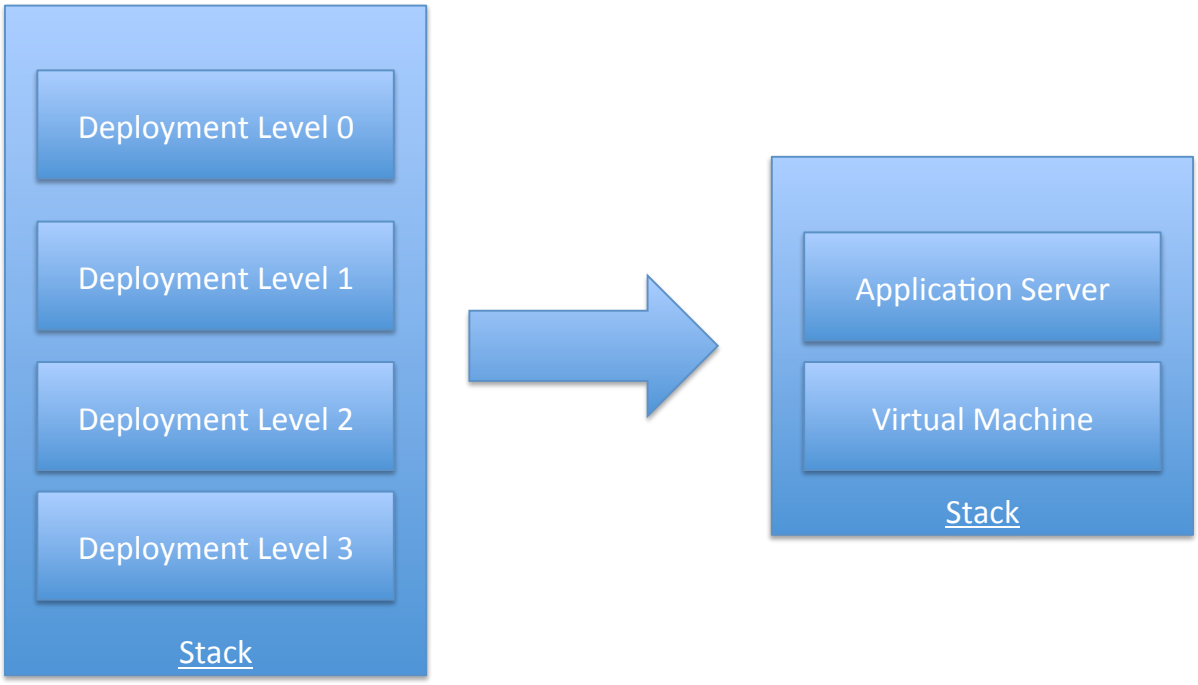


Figure 2.4.: Deployment Levels

In the context of this thesis, only two levels will be considered, namely "servers" (or VM) and "instances". MT allows the tenants to not only decide which Application Component they are willing to share, but also which Deployment Level. Thus, the tenants are given the ability to share (or not) an instance of component A with another tenant, or also a server. Though, sharing a higher DL automatically leads to sharing its corresponding lower DL. For example, sharing an instance of an Application Server leads to sharing the underlying Virtual Machine.

The result of processing the customers' requirements is a Deployment Configuration,

which describes how the different resources are to be deployed. This states the relationship between the different resources, as shown by the Entity Relationship Model in Figure 2.5.

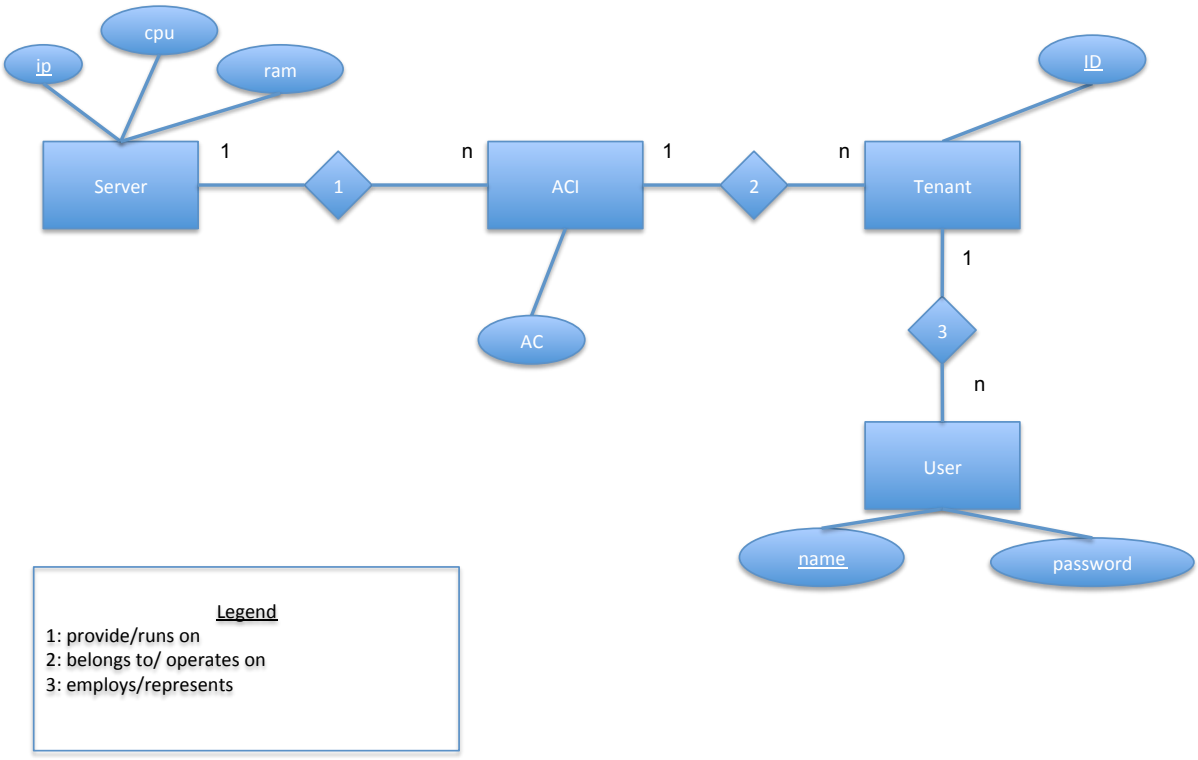


Figure 2.5.: Deployment Configuration: Entity Relationship Model

The Deployment Configuration states which Application Component Instance runs on which server, and who are the tenants that are allowed to access it. The technical implementation of the Deployment Configuration could be an XML file [Reinhardt, 2013].

Mixed-Tenancy related definitions

The following terms are frequently used in this thesis. This section aims to centrally define them in the context of their use. The given definitions are the ones adopted throughout this research.

***Definition 1**: Customer/Tenant*
The entity called *Customer* or *Tenant* is a company that consumes an application provided by the Service Provider. The term *Customer* is used to refer to this entity from a business perspective. The technical term for the same entity is *Tenant* [Ruehl, 2014]. Both terms will be used interchangeably throughout this thesis.

***Definition 2**: User*
A *User* is a person or employee that has access to an application and interacts with it.

Each *User* belongs to exactly one *Customer/Tenant* [Ruehl, 2014].

***Definition 3**: Application Component*
An *Application Component* (AC) is defined as a building block of an application that encapsulates atomic functionality (e.g. Component *Idea* of OpenERP) [Ruehl, 2014].

***Definition 4**: Deployment Constraint*
A *Deployment Constraint* is a customer's description of if or with which other tenants resources may be shared [Ruehl, 2014]. It is necessary that each tenant gives a number of Deployment Constraints for the deployment of an entire application.

***Definition 5**: Deployment Configuration*
The Deployment Configuration is a transfer document, which describes a deployment. For example, it could state which Application Component's Instance (ACI) runs on which server, and which tenants are granted permission to access it [Ruehl, 2014].

***Definition 6**: Deployment Level*
A Deployment Level (DL) is a slice of the infrastructure stack for which the Service Provider may offer the possibility to realize Mixed-Tenancy. A slice encapsulates one to many layers of the infrastructure stack [Ruehl, 2014]. However, in the context of this research, only two DLs will be considered, namely Server and ACI.

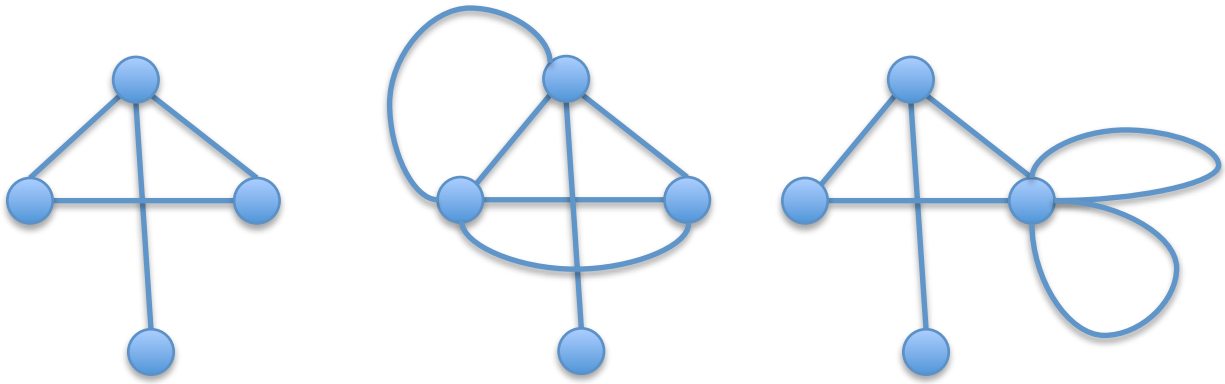
***Definition 7**: Deployment Unit*
Once a deployment has been created there will be one or many instances of every Deployment Level. These instances are referred to as *Deployment Units* (DU) [Ruehl, 2014]. An ACI thus represents a DU.

2.5. Graphs

The concept of "Graphs" in Mathematics as well as in Computer Science has been introduced several centuries ago, in year the year 1736 more precisely[2] [Biggs et al., 1976] [Bondy and Murty, 1976] [Bollobás, 1982]. This is being introduced here in this thesis since the solution resulting from the research is based on the concept of graphs, as the main chapters will show.

[2]It took, however, 200 years before the first book on graph theory was written. This was done by König in 1936. ("Theorie der endlichen und unendlichen Graphen", Teubner, Leipzig, 1936. Translation in English, 1990.)

A graph, formally denoted as G(V,E), is a structure consisting of a set of vertices V = v1,v2, . . . and a set of edges E = e1,e2, . . .; each edge e has two endpoints, which are vertices, and are not necessarily distinct [Even, 2011, Christofides, 1975]. In other words, a graph is a set of nodes or points connected to each other by a line. The points or nodes are the vertices, whereas the lines represent the edges. Figure 2.6 shows the different kind of graphs existing.

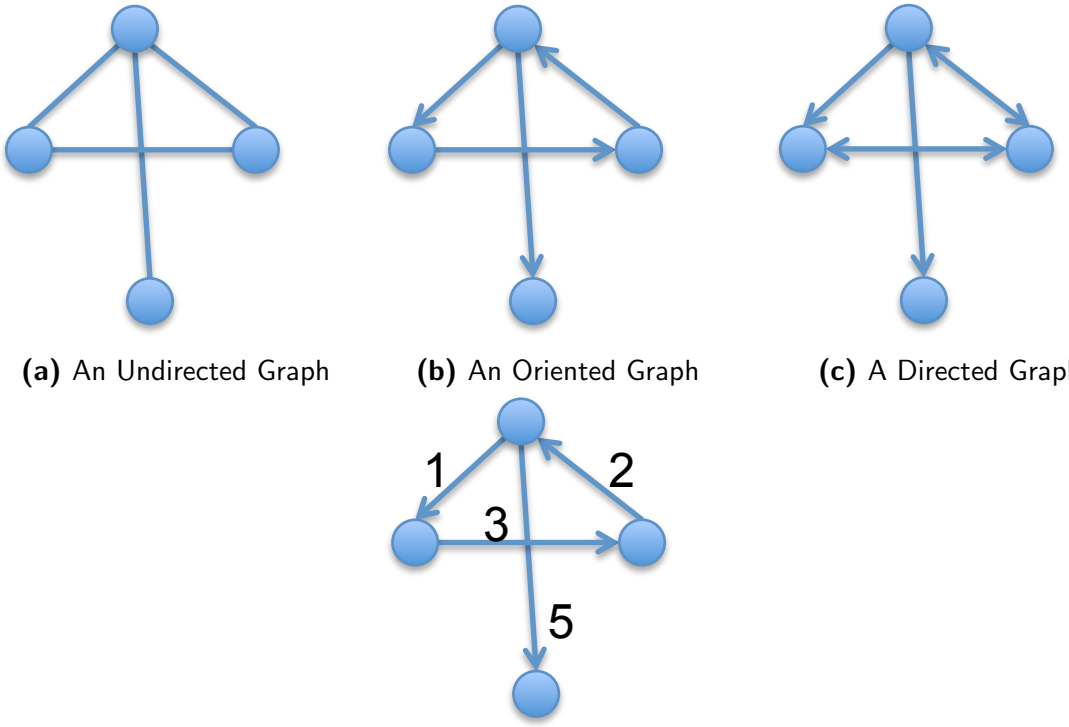


(a) A Simple Graph **(b)** A Multigraph **(c)** A Pseudograph

Figure 2.6.: Different sorts of Graphs (1)

Graphs in which at most one edge (i.e., either one edge or no edges) may connect any two vertices are called *simple graphs*. If multiple edges are allowed between vertices, the graph is known as a *multigraph*. Vertices are usually not allowed to be self-connected, but this restriction is sometimes relaxed to allow such "graph loops". A graph that may contain multiple edges and graph loops is called a *pseudograph* [Weisstein, 2013a].

The edges of graphs may also be complemented with directedness. Figure 2.7 represents another classification of graphs based on the direction of the edges.



(a) An Undirected Graph **(b)** An Oriented Graph **(c)** A Directed Graph

(d) A Network or Weighted Graph

Figure 2.7.: Different sorts of Graphs (2)

A normal graph in which edges are undirected is said to be *undirected*. Otherwise, if arrows may be placed on one or both endpoints of the edges of a graph to indicate directedness, the graph is said to be *directed*. A directed graph in which each edge is given a unique direction (i.e., edges may not be bidirected and point in both directions at once) is called an *oriented graph*. A graph or directed graph together with a function which assigns a positive real number to each edge (i.e., an oriented edge-labeled graph) is known as a *network* or *weighted graph* [Weisstein, 2013a].

2.6. Introduction to Online Graph Search Algorithms

Most Graph related problems in Computer Science consists of finding an optimal path between a given initial node and a given target node [Russell and Norvig, 2010]. Further chapters of this thesis will show that the challenge faced in this research is also about finding an optimal path in a graph. For a better understanding of the algorithms that will be discussed (in Chapter 5, this section will introduce and explain a special sort of Graph Search Algorithm.

Offline Search Algorithms compute a complete solution with full knowledge of the landscape. For example, *Selection Sort* can be only performed on a list which is fully known

at beginning. In terms of graphs, this would mean that the graph is, a priori, fully known. In contrast, *Online Search Algorithms* do not have any information about the environment in advance. *Insertion Sort*, for example, doesn't require the entire list to be given before the sorting starts. In terms of graphs, this means that the full graph is not required at the beginning. The following scenario by Russel and Norvig [Russell and Norvig, 2010] summarizes the idea which motivates Online Search Algorithms:

> *Consider a newborn baby: it has many possible actions but knows the outcomes of none of them, and it has experienced only a few of the possible states that it can reach. The baby's gradual discovery of how the world works is, in part, an online search process.* [Russell and Norvig, 2010]

It can be deduced from the given scenario that online search algorithms process the data piece-by-piece and increasingly gain more information about the environment. In a non-deterministic environment, online search algorithms would focus on the contingencies that actually arise rather than those that might happen but probably won't; they are also useful in unknown environments, where it is not known in advance what states exist and which actions shall be taken [Russell and Norvig, 2010]. Like in most graph search algorithms, the aim of online search algorithms is to reach a target state while minimizing the cost. The goal is to have a total path cost as close as possible to the case where the search environment is fully known in advance. This is known as the competitive ratio:

$$\text{Competitive Ratio} = \frac{\text{Cost of the online algorithm}}{\text{Cost of an optimal offline algorithm}}$$

Another important aspect of Online search algorithms is that they usually make use of a heuristic function, an *admissible heuristic function* more precisely. Heuristic functions are the most common form in which additional knowledge of the problem is imparted to the search algorithm [Russell and Norvig, 2010]. The heuristic function *h(n)* of a node n represents the estimated cost of the cheapest path from that state at node n to the goal state. h(n) is admissible when it does no overestimation.

> *You do not really understand something unless you can explain it to your grandmother.*
>
> – *Albert Einstein*

3

Related work

Cloud Computing in general, and SaaS in particular have been widely analyzed from both a technical and a economical perspective [Bayrak et al., 2011] [Carraro and Chong, 2006] [Dillon et al., 2010] [Luna Garcia et al., 2012] and its challenges have also been addressed. For example, in [Sengupta and Roychoudhury, 2011], Sengupta and Roychoudhury outline a set of research issues in the design, testing and maintenance of multi-tenant SaaS systems. They highlight some of the interesting optimization questions arising from the process of unifying the different requirements of the SaaS consumers. Sun et al. explore in [Sun et al., 2008] the configuration and customization issues as well as the challenges faced by SaaS vendors while drawing a straight line between customization and configuration. They recognize that, although SaaS applications are usually developed with highly standardized software functionalities to serve as many clients as possible. Still a lot of clients ask for functions variants according to their unique business needs, which should happen through easy configuration and customization. They proposed a competency model as well as a methodological framework to help SaaS providers plan and evaluate their capabilities and strategies for service configuration and customization. This research is more about the variability of the software functionalities.

In [Jiang et al., 2012], Jiang et al. take a closer look at the basic data models to implement a cost-efficient multi-tenancy model. The basic data models are differentiated by the database isolation: (*1*) Independent Database data Model (IDM), (*2*) Shared Database & Individual Schema data Model (SDISM) and (*3*) Shared Schema data Model (SSM). According to Jiang et al., these three basic models do not offer the tenants sufficient

options for both cost efficiency and the security insurance of their sensible data at the same time. To solve this dilemma, [Jiang et al., 2012] propose a new mixed data model, named Mixed Multi-tenancy Data Model (MMDM), which strives to achieve a certain degree of balance by introducing two sub-models: Mixed Database data Model (MDM) and Mixed Schema data Model (MSM) with a relatively low-cost and high isolation. Basically, MMDM combines the advantages of the original basic models: some of the tenants' data stored on IDM, and other data is stored on SSM, while meeting the different requirements of tenants' data safety and pricing from a comprehensive point of view. In addition, the research provides a Configurable Multi-tenancy data Model Group (CMMG), which can cater to the tenant requirements of the customized data model. Also, the research proposes an approach to migrate a single-tenancy database to CMMG. In their work, Jiang et al. address the topic of customization of SaaS applications on rather a technical perspective: from the database point of view. In the model suggested, the tenants still use the same instance of the SaaS application, but are given a new way of ensuring data safety while keeping the costs low.

In [Mietzner et al., 2009], utilizing techniques from the field of Software Product Lines (SPL), Mietzner et al. introduce the notion of variability into SaaS applications. The variability of both the SaaS application and its underlying requirements can be triggered by either the customer (*Customer-driven Variability*) or by technical reasons (*Realization-driven Variability*). Using the Orthogonal Variability Model (OVM) described in [Pohl et al., 2005], Mietzner et al. applied the SPL concepts of external and internal variability to the problems of customization of SaaS applications. The results have been tested on a real-world application. The variability introduced in [Mietzner et al., 2009] is rather based on the functionalities that the SaaS application offers. The research suggests a new way of enabling SaaS providers to offer cost-effective multi-tenancy applications customized to the unique requirements of their customers.

Another research by Ruehl and Andelfinger [Ruehl and Andelfinger, 2011] also examines the use of SPL in multi-tenancy SaaS applications. However, this work focuses on flexibility instead of variability. The authors address the two important types of flexibility that are important for Information Systems (IS) according to Gebauer and Schober [Gebauer and Schober, 2006], namely *Flexibility-to-Use* and *Flexibility-to-Change*. The former type refers to a manufacturing machine's ability to perform different operations while the latter describes the potential adaptability to implement a major change, which is an adjustment or a modification that requires a fresh system setup, re-compilation, re-installation or re-testing [Gebauer and Schober, 2006]. In their research, Ruehl and Andelfinger proposed a model that allows the creation of highly flexible SaaS applications. The basic idea is to provide customers with an application that suits their unique requirements. A so-called *Execution Engine*, which acts as a router, has been introduced with the aim to seamlessly

re-direct the logged-in tenant to the corresponding unique configuration, by binding the available application components at run-time. In summary, this work proposes another technique for SaaS providers to attract more customers by easily developing multi-tenancy applications customized for each consumer.

In a further research, Ruehl et al. [Ruehl et al., 2012] look at SaaS applications from a different view angle and introduce a new Data Model, the Mixed-Tenancy Data Model. Mixed-Tenancy has already been detailed in Section 2.4 of this thesis. The model is motivated by the same reasons as [Jiang et al., 2012], namely the fact that the tenants would like to save cost by adopting multi-tenancy applications, but are still reluctant because of data security. Mixed-Tenancy gives the tenants more flexibility. This allows them to choose which component of the SaaS application shall be multi-tenant (shared with another tenant) and which ones rather be single-tenant (used only by them). Capturing and unifying the customers' requirements are necessary steps in such an approach and represent a fundamental challenge. Using Semantic Web Technologies, Ruehl et al. also address this challenge in a further research in [Ruehl et al., 2013]. The outcome of this latter research is a Deployment Configuration file, which summarizes the different customers' sharing requirements, known as deployment constraints.

In [Reinhardt, 2013], Reinhardt analyzes the results of Ruehl et al. in [Ruehl et al., 2012] and in [Ruehl et al., 2013]. Furthermore, he investigates the technical constraints to implement a Mixed-Tenancy Data Model. Indeed, Reinhardt examines in which ways a Mixed-Tenancy Platform can be implemented and proposes a prototypical system tested with a dummy application. The testing consists of realizing an initial deployment configuration on the platform. The proposed platform bears its own logic which should be kept separated from the SaaS application's logic. Moreover, the communication among the components of the SaaS application represents a core part of the research. Reinhardt suggests a connector-based model in which the components communicate with one another over the *Execution Engine* (introduced in [Ruehl and Andelfinger, 2011]) through so-called connectors using the Representational State Transfer - REST - described in [Richardson and Ruby, 2007]. A proof of concept using a real-world complex component-based application, OpenERP, is realized by Rupprecht in [Rupprecht, 2013]. Rupprecht elaborates some guidelines to successfully deploy OpenERP on the Mixed-Tenancy Platform.

The different researches summarized above addressed the question of variability in SaaS applications. However, this was more about the software functionalities variation rather than the variation of the customers' sharing requirement. The focus of this thesis is of two levels. On the one hand, the thesis considers only Mixed-Tenancy SaaS Applications. On the other hand, it addresses the variability over time of the customers' deployment constraints.

> *Rarely do we find men who willingly engage in hard, solid thinking. There is an almost universal quest for easy answers and half-baked solutions. Nothing pains some people more than having to think.*
>
> – *Martin Luther King, Jr.*

4

Towards a Migration Plan

Conveying the initial deployment configuration into a newer one, based on the continuous changing users' requirements, has a set of rules to respect. After describing these rules, this chapter discusses the general field of computer science to which the question addressed in this thesis belongs to.

4.1. Requirements

A migration plan, i.e. a set of different operations leading from an initial deployment configuration to a target configuration, shall be computed. Various criteria are to be applied to this migration plan. This thesis will concentrate on two of them.

4.1.1. The Migration Cost

The cost of the migration represents the first criterion. Indeed, migrating a tenant to another ACI, or an ACI to another server, etc. all bear a cost, which is to be held as low as possible. The cost of a migration may depend on many factors such as the

- human capital needed to perform the task,
- complexity of the operation,
- scalability,
- financial value,
- number of distinct operations needed

to be completed,

- application being deployed,
- etc.

Even the infrastructure environment in which the application is being deployed can influence the cost. For example, deploying an application in the Company XYZ's Computer Centers may have a different cost as an application deployed in the Computer Centers of another Company ZYX. The unit of the cost may also be different depending on the tenants' wishes. However, the fact that the tenant wants to keep the costs as low as possible would hardly change.

4.1.2. The Deployment Constraints

The second criterion is based on the deployment constraints initially specified by the tenants and which have been collected and processed by Ruehl et al. to compute the optimal deployment configurations [Ruehl et al., 2012]. The deployment constraints could be viewed as a sort of agreement between tenant and service vendor in which the tenants specify the other types of tenants they may share a resource with [3]. The resources to be shared are grouped into so-called *Deployment Levels*[4](DL) and are hierarchically related. The constraints can be set for each DL. However, when tenants share a lower level DL, it means that the upper level DL is automatically shared. In the context of this thesis, only the levels "server" and "application component instance" (ACI) will be considered. Figure 4.1 shows their hierarchical relationship, which is very simple in this case.

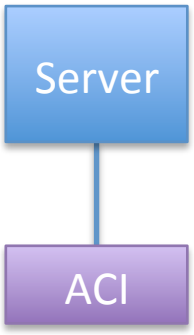


Figure 4.1.: Deployment Levels hierarchy

As a consequence, two tenants which agree to share an instance automatically share the server on which this instance is running. Though, if they share a server, they don't have to share an instance. Ruehl et al. [Ruehl et al., 2013] identified five sharing models, namely private, public, white hybrid, black hybrid and gray hybrid (see Appendix A for

[3]See Definition 4 on page 16.
[4]See Definition 6 on page 16.

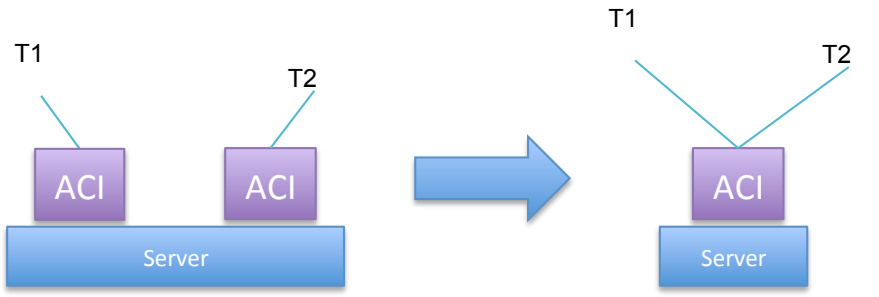
further details). During the computation of the migration plan, these constraints shall be respected at any time, even if the system isn't running at that time.

In summary, the requirements to the migration plan are the following:

- The plan shall bear the lowest cost possible to compute.
- The (new) deployment constraints shall be respected at any time during the process.

4.2. Scenario based problem description

Based on the criteria previously stated in Section 4.1, an algorithm is to be elaborated, which shall depict the best suitable set of operations to get from a given point A, which shall be the initial deployment configuration, to a given point B, which shall represent the target deployment configuration. Let us consider the following trivial examples from Figure 4.2:



(a) Example 1

(b) Example 2

Figure 4.2.: Trivial examples

ACI or "Application Component Instance" represents an instance from any given Application Component (AC) such as the component *Idea* from OpenERP[5] (See Definitions 3 and 7 on page 16) for example. In both examples on Figure 4.2, the ACIs are on the same server.

[5]OpenERP is an open-source enterprise resource planning (ERP) software, actively developed, written, supported, and organized by OpenERP s.a. `https://www.openerp.com/`

The first example on Figure 4.2a illustrates two tenants T1 and T2. In the initial deployment, they are each deployed alone on a distinct instance of the the given component. However, according to the target configuration, they shall both share the same instance of that component. There may be three approaches to realize this: (1) migrate T1 to the instance of T2, (2) migrate T2 to the instance of T1, or (3) migrate both T1 and T2 to a new instance of the component. Each of these possibilities should be analyzed as they may have different cost. The possibility with the lowest cost is the preferred option. In practice, case (3) can be omitted as it is less likely to be least expensive to realize than either case (1) or case (2). No deployment constraints are defined for this example.

The second example on Figure 4.2b shows an initial state, in which 4 tenants, T1 and T2, and then T3 and T4 share an instance. The target state introduces a new tenant T5. In this state, T1, T3 and T5 share and instance; T2 and T4 also share another instance. Furthermore, according to the deployment constraints, T2 and T5 may not share an instance; neither may T1 and T4 as well as T2 and T3. For a better understanding of this example, let us consider the following scenario:

> In a company, Albert and Henrietta (they represent the tenants) share an office (this is the ACI); so do Karl and Ada (tenants). All four of them work in the same building (this is the server). The company expanded and hired a new employee, Grace (new tenant), who will also work in the same building. In addition, due to internal changes, the offices have to be re-distributed. Albert, Karl and Grace shall now share an office; so do Henrietta and Ada. For unofficial reason, however, Grace and Henrietta can never share an office; neither can Albert and Ada. This also applies to Karl and Henrietta. Furthermore, each employee has to always be in an office; this means that no employee can be waiting in the hallway during the re-allocation (or moving) process. To ease this, there are a lot of empty offices that may be made available to use (for an employee to be waiting in, for example). The Facility Management Business Unit (Service Provider) has to implement those changes; their goal is to find the best and cheapest way to convey the initial offices allocation into the new one while respecting the rules.

Obviously, there are many ways for the Facility Management to solve this problem. They could, for example, move Henrietta to a new office, then move Karl to Albert's office before moving Henrietta to Ada's office. Another way would have been to move Ada to Henrietta's new office. Each of these moves of course bears costs, which are to be taken into account. Possibly, Ada has so many personal belongings in her office that it would be easier and cheaper to move Henrietta to Ada's office (e.g. Facility Management would contract fewer persons for that move).

4.3. Node notation

Following Section 4.4 will introduce the concept of graphs and show how this can be used to solve the problem faced in this thesis. For a better understanding of these graphs, this Section will introduce a new notation, which will be used for the nodes in the graph.

Consider following Figure 4.3.

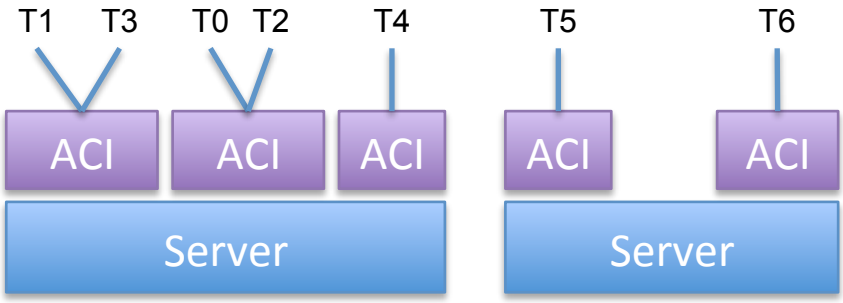


Figure 4.3.: Deployment Configuration to introduce a new notation

Figure 4.3 illustrates a deployment configuration with only one AC. On the first server (from the left to the right), three instances of the given AC are running. The first of these ACI is accessed by Tenants T1 and T3. The second is shared by Tenants T0 and T2. Only Tenant T4 is granted access to the third ACI. The second server supplies two ACIs which each are solely used by Tenants T5 and T6. As following Section 4.4 will show, this configuration in Figure 4.3 represents a node in a graph. This node will be labeled as showed in Figure 4.4:

[[[T1 T3][T0 T2][T4]][[T5][T6]]]

Figure 4.4.: Node Notation

In the notation in Figure 4.4, the first outer bracket represent the state. A state contains servers, which are represented by the first inner brackets. The instances which are running on the server and on which the tenants have an access are represented by the second inner bracket.

4.4. The concept of Graphs

The scenario described above in Section 4.2 maps the challenge this thesis addresses to a real-world problem. From a computer science point of view, each move leads to a new state. The states are thus related to each other by at least one move. For example, as

shown in Figure 4.2a, either moving T1 to T2's instance or moving T2 to T1's instance both lead to the same state in which T1 and T2 share the same instance. Starting at an initial state, there are then a defined set of following states which can be realized by moving exactly one tenant at a time. Therefore, there are a defined number of states that are to be realized to transform the initial state into the target state. This perception of the problem reminds of a known field of computer science: this problem can be translated as a graph, a *weighted directed multigraph* more precisely.

In their book on a modern approach to artificial intelligence, Russel and Norvig [Russell and Norvig, 2010] identified five parts that a problem may consist of:

> *"A problem consists of five parts: the initial state, a set of actions, a transition model describing the results of those actions, a goal test function, and a path cost function. The environment of the problem is represented by a state space. A path through the state space from the initial state to a goal state is a solution"* [Russell and Norvig, 2010].

In the context of this thesis, the different states would represent the vertices or nodes; the nodes are joined by at least one edge. Each edge would bear a cost, which is the cost of the move. The states between the initial state and the target state would represent the actions. Figure 4.5 transforms the first trivial example given in Figure 4.2a to its graph representation.

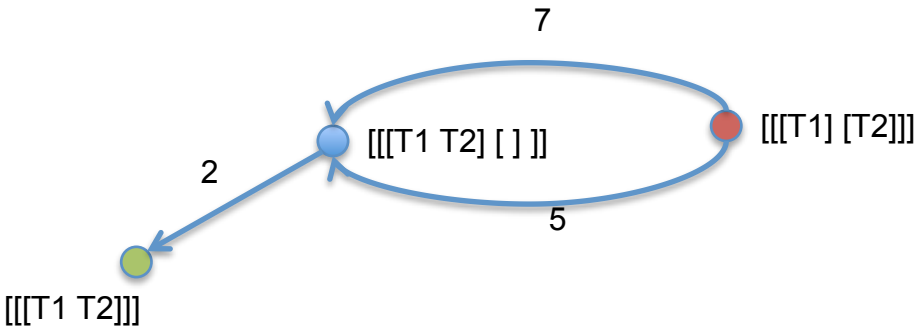


Figure 4.5.: A graph representation of trivial example 1 from Figure 4.2a

The red node is the initial state whereas the green node represents the target state. Please refer to previous Section 4.3 for an elaboration on the node notation.

Representing the problem at hand by a graph breaks down the search for the best suited migration plan to the search of the weighted shortest (respectively cheapest) route in the graph.

4.5. Weighted Graph Search Algorithms

Graph problems pervade computer science, and algorithms for working with them are fundamental to the field [Cormen et al., 2009]. There exists a lot of algorithms which deal with the problem of finding the shortest path in a weighted graph. These algorithms are grouped into informed and uninformed search methods.

Informed search methods have access to more information. Some of them use a heuristic function $h(n)$ which is the estimate or the cost or a path from node n to the target node. The basics algorithms are as follow [Russell and Norvig, 2010]:

The generic best-first search algorithm selects a node for expansion according to an evaluation function.

Greedy best-first search expands nodes n with minimal $h(n)$. It is not optimal but is often efficient.

A* search is a best-first search algorithm in which the cost associated with a node is $f(n) = g(n) + h(n)$, where $g(n)$ is the cost of the path from the initial state to node n. Thus, $f(n)$ estimates the lowest total cost of any solution path going through node n. At each point, a node with lowest f value is chosen for expansion. A* is complete and optimal, provided that $h(n)$ is admissible.

RBFS (recursive best-first search) and SMA* (simplified memory-bounded A*) are robust, optimal search algorithms that use limited amounts of memory; given enough time, they can solve problems that A* cannot solve because it runs out of memory.

The uninformed search methods have access to the problem definition only. The basics algorithms are as follow [Russell and Norvig, 2010]:

Breadth-first search expands the shallowest nodes first; it is complete, optimal for unit step costs, but has exponential space complexity.

Uniform-cost search expands the node with lowest path cost, g(n) , and is optimal for general step costs.

Depth-first search expands the deepest unexpanded node first. It is neither complete nor optimal, but has linear space complexity. Depth-limited search adds a depth bound.

Iterative deepening search calls depth-first search with increasing depth limits until a goal is found. It is complete, optimal for unit step costs, has time complexity

comparable to breadth-first search, and has linear space complexity.

Bidirectional search can enormously reduce time complexity, but it is not always applicable and may require too much space.

Dijkstra's algorithm a special case of A* for which no heuristic function is used.

> *We always hope for the easy fix: the one simple change that will erase a problem in a stroke. But few things in life work this way. Instead, success requires making a hundred small steps go right - one after the other, no slipups, no goofs, everyone pitching in.*
>
> *Atul Gawande*

5 Solution approaches

The search for a solution to the problem described in this thesis led to the exploration of a few algorithms. In a first step, the chapter describes the main idea for the solution proposed. In a second step, the chapter looks into different algorithms which may solve the problem at hand, and also discusses their suitability. Finally Section 5.3 summarizes the different algorithms detailed in Section 5.2.

5.1. Solution Concept: Main Idea

The solution proposed in this thesis is based on multigraphs. The idea is to recursively generate all possible valid deployment configurations which represent the nodes in the multigraph and to eventually find the shortest path between the initial and the target deployment configurations using a weighted shorted path algorithm (see Section 4.5).

A state is a representation of a physical configuration, while a node is a data structure that is part of the graph. However, from now on in this thesis, “node”, “state”, “deployment configuration” and “configuration” will be used interchangeably. A state, in this context, will be defined for only one AC at a time. For example, consider following Deployment Configuration illustrated in Figure 5.1

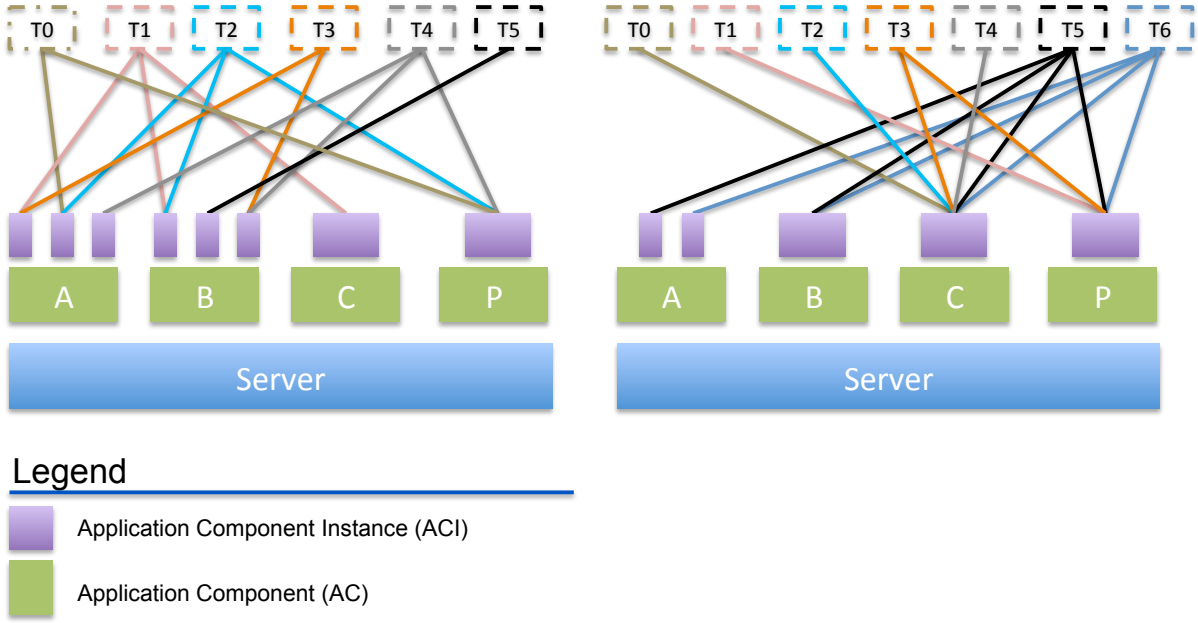


Figure 5.1.: Deployment Configuration

Defining the state from the viewpoint of Application Component A, for example, would result in the partial DC showed in Figure 4.3 and which has been textually represented in Figure 4.4. The idea is to determine a (partial) migration plan for each AC. The general migration plan would then consist of the sequential execution of the partial migration plans. The overall complex problem is then broken into smaller less complex pieces, which are solved successively.

A valid deployment configuration is a configuration which respects the deployment constraints. For example, if Tenant T1 and Tenant T2 are not supposed to share an instance, there shall not exist any node in the multigraph, in which T1 and T2 do share an instance. The valid possible configurations will be referred to as "possible" configurations.

Starting from an initial deployment configuration, a "neighboring" deployment configuration would be a configuration which would result from doing exactly one operation. An operation could be:

- Migrate exactly one tenant, which means moving a tenant from an instance to another instance.
- Delete exactly one empty instance. An empty instance is an instance on which no tenant operates.
- Delete exactly one empty server. An empty server is a server, which does not contain any instance at all.
- Create exactly one empty instance on a server.

- Create exactly one empty server with an empty instance.

A node will be connected to its neighboring node by an edge which, in addition, bears the cost of the operation, which led from the given starting node to the computed target node. More precisely, the multigraph is therefore a weighted multigraph. Furthermore, the edges are directed; hence, the multigraph is also directed. This is needed because the cost of the operation which leads from node n1 to node n2, for example, isn't necessarily the same as the cost of reaching to n1 from n2.

In summary, the idea is to build a weighted, directed multigraph, which consists of all the possible deployment configurations starting from an initial deployment configuration as root of the graph. This multigraph will thus also contain the target configuration. The solution would consist of the shortest and in this case, the cost-efficient path from the initial deployment to the target deployment configuration.

There might exist cases in which the source and target deployment configurations do not contain the same tenants: tenants may have to be removed from or added to the target configuration. This requires a necessary initializing phase before starting to build the graph. In that phase both the start and target configurations may have to be modified. The tenants that are to be removed (i.e. tenants that are present in the source configuration but not in the target configuration) should be directly removed. The cost of this operation can be ignored as it is fixed. At the same time, the tenants that are to be added (i.e. tenants that are present in the target configuration but not in the source configuration) should temporally be removed from the target configuration and added at the end of the graph search process. The costs of these operations can also be neglected as they are constant. The goal is to have a source configuration and target configuration that contain the same tenants.

For simplicity reasons, and also in order to address the largest possible set of states, the algorithms described in the following sections will not consider any deployment constraint at all. This means that the tenants are all allowed to share every unit of a deployment level with each other. Due to the fact that the deployment constraints are variable, not considering them, shall help address a broader and more general problem space.

5.2. Solution approaches: Detailed Description

This Section intends to take a deeper look at few algorithms that have been tried out as an attempt to solve the problem addressed in this thesis. These algorithms have been chosen after a thorough study of the relevant literature. Although a lot more algorithms have also been examined, the Section aims to present only the ones that have also been

partially implemented and tested. It is showed in this section that for different reasons, the algorithm described at last, namely RTA*, is more suitable.

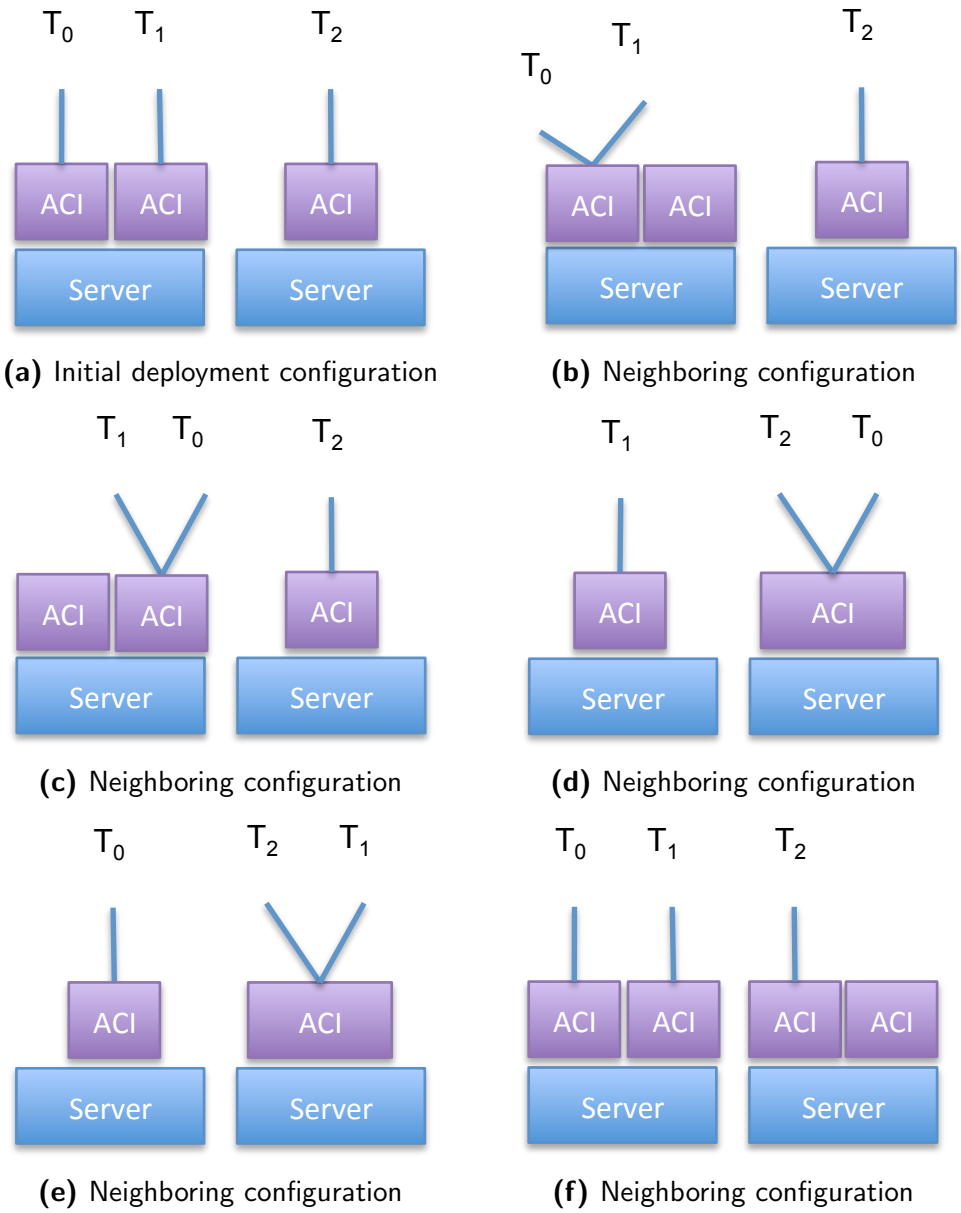
5.2.1. The “Brute-Force” or Exhaustive Approach

The brute-force technique, also known as exhaustive approach, is a very general problem-solving technique that consists of systematically generating and inspecting all data configurations in a large state space that is guaranteed to contain the desired solutions [Nievergelt, 2000].

Applying this technique to the problem in the context of this thesis is a direct translation of the main idea described in Section 5.1. The goal of this solution is to generate all possible (and of course valid) deployment configurations starting from the initial deployment configuration. This will end up in a fully observable, discrete and deterministic environment, in which searching for the “best” migration plan becomes trivial as it is resumed to the use of one of the weighted graph search algorithms mentioned in Section 4.5.

In this approach, the target configuration does not matter at the beginning because it is ensured that it will be generated by the many operations that will follow. All what first matters, is the initial configuration.

Let us consider, for example, the initial configuration in Figure 5.2a which consists of two servers; on the first server, Tenants T0 and T1 both operate on distinct instances, whereas on the second server, Tenant T2 operates alone on an instance.



(a) Initial deployment configuration

(b) Neighboring configuration

(c) Neighboring configuration

(d) Neighboring configuration

(e) Neighboring configuration

(f) Neighboring configuration

Figure 5.2.: Example for neighboring configurations

Figures 5.2b to 5.2f represent some of the possible configurations which can be reached from the initial configuration by, as mentioned earlier in Section 5.1, doing exactly one operation. In Figure 5.2b, Tenant T1 has been moved to the instance of Tenant T2; the opposite happens in Figure 5.2c. In Figure 5.2d, Tenant T0 has been moved to the instance of Tenant T2 on the another server. Tenant T1 is moved to the instance of Tenant T2 in Figure 5.2e. Figure 5.2f shows the operation which consisted of creating a new empty instance on the second server, for either Tenant T0 or Tenant T1 to be moved on, in a next step.

The configurations shown in Figure 5.2 are just a few of the possible configurations realizable starting with the initial configuration. The idea is to generate all neighboring

configurations for each new configuration generated and add them to the graph. The cost of the migration or transformation from one configuration to the derived one shall be computed. The configurations shall be connected by an edge, which contains the cost of the migration as data. In case the generated configuration already existed, there is no need to add it again to the graph; however, an edge shall be drawn from the starting configuration to the generated (in this case, already existent) configuration, and the cost of the migration shall also be computed and stored as data in the edge. Pseudocode 1 summarizes the concept.

Pseudocode 1 Brute-Force Algorithm

```
 1: function computeMigrationPlan(config)
 2:     for all server ∈ config do
 3:         for all aci ∈ server do
 4:             for all tenant ∈ aci do
 5:                 newConfig ← MoveTenantToNextACI
 6:                 cost ← getCost(config, newConfig)
 7:                 if newConfig not exists then
 8:                     addNewConfigToGraph
 9:                     edge ← addDirectedEdge(config, newConfig)
10:                     storeCost(edge)
11:                     return computeMigrationPlan(newConfig)
12:                 else
13:                     edge ← addDirectedEdge(config, newConfig)
14:                     storeCost(edge)
15:                 end if
16:             end for
17:         end for
18:     end for
19: end function
```

For each configuration, the set of neighboring configurations consists of all configurations which result from moving each tenant to each of the other ACIs across the existing servers. This is shown in line 5. A neighboring configuration might also be realized by either adding a new server or a new instance, deleting an empty server or an empty instance. Although these cases have not be considered in the pseudo-code, they should be taken into account while implementing the algorithm. For each of the new configurations thus computed, the cost of the transformation from the original configuration to the new configuration shall be computed (line 6). Next, it shall be checked if the new computed configuration already existed before (line 7). In case it did not, it shall be added to the graph (line 8) and a directed edge from the original configuration to the new configuration shall be added to the graph as well. This edge contains the cost of the configuration (line 10). As these steps shall be repeated for every new configuration, the algorithm can be built recursively, by just calling the function for every new configuration (line 11). In case the new computed configuration was already existent in the graph, only the edge

shall be drawn (line 13) and the cost stored (line 14).

Analysis - A lower-bound

This section will examine the algorithm suggested in Section 5.2.1 and calculate the lower-bound of the number of nodes needed to find the most optimal route from a given initial configuration to a given target configuration. For a better calculation, the sharing constraints will still be ignored.

The number of servers and instances is not important for this calculation as will be shown later in this section. Relevant is only the number of tenants. For any given initial configuration, because the sharing constraints do not exist, both (1) the configuration in which all tenants are together on an instance on a server, and (2) the one in which each tenant is alone on an instance and alone on a server, will always occur throughout the transformation process described in Section 5.2.1. Configurations (1) and (2) will be used as “control configurations” to prove that all configurations have been considered in the calculation. Furthermore, the tenants order is also not important: moving T0 on the instance of T1 or moving T1 on the instance of T2 both result in the same configuration.

The lower-bound computation is based on the notation of the *integer partition*.

> *A partition of a positive integer n is a finite nonincreasing sequence of positive integers* $\lambda_1, \lambda_2, ..., \lambda_r$ *such that* $\sum_{i=1}^{r} \lambda_i = n$. *The* λ_i *are called the parts of the partition* [Andrews, 1998].

More clearly, the integer partition of n, denoted as p(n) gives the number of ways of writing the integer n as a sum of positive integers, where the order of addends is not considered significant [Weisstein, 2013b]. For example, $p(5) = 7$ since there are 7 ways of writing 5 as shown below:

$$\begin{aligned} 5 &= 1+1+1+1+1 \\ &= 2+1+1+1 \\ &= 2+2+1 \\ &= 3+1+1 \\ &= 3+2 \\ &= 4+1 \\ &= 5 \end{aligned}$$

Let us consider, for example, an initial deployment configuration, which contains 3 tenants, namely T0, T1 and T2. The different configurations starting from this initial configuration can be represented as integer partition:

$$\begin{aligned} 3 &= 1 + 1 + 1 && \textbf{(a)} \\ &= 2 + 1 && \textbf{(b)} \\ &= 3 && \textbf{(c)} \end{aligned}$$

(a) The first sum represents the configuration, in which each tenant is alone on a server; this means that each tenant is alone on its instance on its server. As the tenants' order is irrelevant, there is only one possibility for this case:
$a = 1$

(b) The second sum represents the configuration, in which 2 tenants share a server while the other tenant is alone on a server.
The first step in this case consists of determining which two tenants will share a server: this is in fact a combinatorial problem, a combination with no repetition in this context. The combination without repetition of two elements out of three, denoted as ${}^3C_2$ is three. The next step is to determine in which way the two chosen tenants could be assembled with respect to ACIs. In this case, two tenants can either both share an instance or be deployed on an instance each. The number of possibilities can be calculated with integer partitions:

$$\begin{aligned} 2 &= 1 + 1 \\ &= 2 \end{aligned}$$

The first case represents the one in which the two tenants are each on their instance, whereas the second one is about the case in which they both share an instance:
$b_1 =^3 C_2 * p(2) = 3 * 2 = 6$
After the two tenants have been sorted out, there is only one tenant left to sort; the same procedure is to repeat in this case as well: $b_2 =^1 C_1 * p(1) = 1 * 1 = 1$
The number of cases here is thus:
$b = b_1 * b_2 = 6 * 1 = 6$

(c) The third sum represents the configuration, in which all tenants share a server. The tenants can, however, be combined in different ways. These cases can also be represented using the p(3). In fact, they can either all share an instance (case 3 = 3), or be deployed on an instance each (case 1 + 1 + 1), or two tenants can share an instance while the other one is alone on its instance (case 2 + 1). For the first two cases, there is exactly only one configuration possible as the tenants' order isn't relevant. The last case can be calculated exactly like b. Thus:
$c = 1 + 1 + 6 = 8$

A configuration with 3 tenants can therefore lead to 15 ($a+b+c$) different configurations.

Because this calculation does not take into account the configurations which result in creating an empty server or an empty instance as stipulated in Section 5.1, it is considered

to be the minimum number of configurations (or nodes in the graph) generated by an initial configuration with a given number of tenants: this calculation gives lower-bound of the algorithm.

This lower-bound, however, grows really fast with an increasing number of tenants as illustrated in Figure 5.3

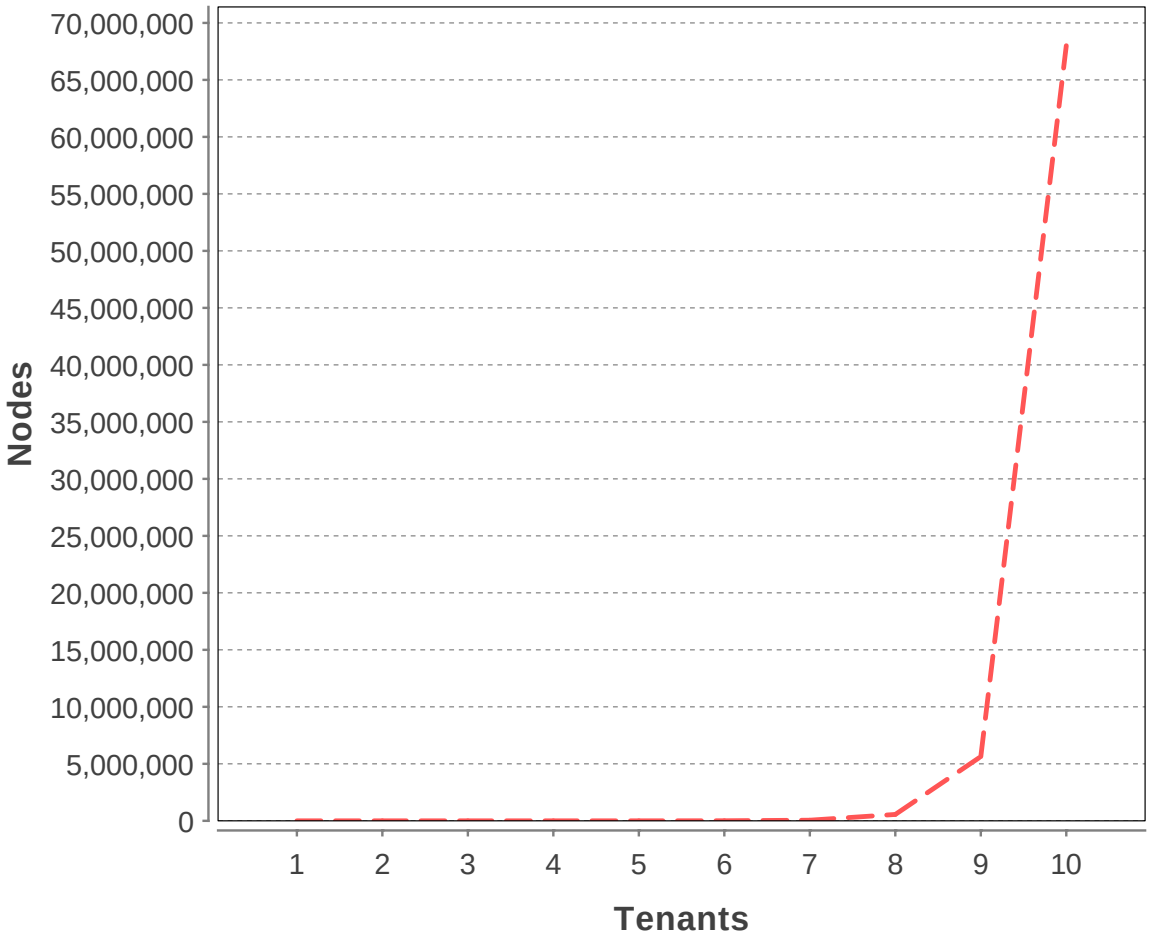


Figure 5.3.: Number of nodes vs. number of tenants

Figure 5.3 shows that the number of nodes or configurations grows exponentially with respect to the number of tenants. A configuration consisting of 10 tenants would generate over 70 Mio. configurations before searching for the set of configurations to sequentially realize in order to transform the initial configuration into the target configuration. Furthermore, for each new node generated, the problem-solving function is being recursively called. That is, the recursive depth is proportional to the number of nodes. In practice, the problem-solving function will call itself recursively an excessive number of times before returning; this could lead to a *StackOverflowError*[6] in Java, for example.

Moreover, the algorithm suggested in Section 5.2.1 produces a lot of overhead or unnecessary nodes. In a productive environment, the number of tenants could increase into the

[6]Thrown when a stack overflow occurs because an application recurses too deeply [Corporation, 2013].

hundreds. The limits of this approach will then be reached quickly in that case. Fixes such as increasing the hardware properties (RAM, CPU cores, etc.), parallel programming or the use of a graph database (such as Neo4j [Neo Technology, Inc., 2013]) aren't real solution as they just postpone the problem.

5.2.2. The Dynamic Approach

The algorithm suggested in previous Section 5.2.1 tries to solve the problem by recursively calling the same problem-solving function with the new computed parameters until a solution is found. This means that the results from a recursion to the other as well as its parameters are of no use. Thus, the problem-solving function could be called several times with the same parameters throughout the search, which could result into memory overflow. *Dynamic Programming* (DP), which is a widely used technique in the field of Artificial Intelligence [Poole and Mackworth, 2010], could be used in such situations. This section will analyze if such an approach is suited to solve the problem at hand.

Dynamic Programming is an algorithm paradigm in which "a problem is solved by identifying a collection of subproblems and tackling them one by one, smallest first, using the answers to small problems to help figure out larger ones, until the whole lot of them is solved" [Dasgupta et al., 2006]. DP thus cuts the main problem into smaller problems and resolves them recursively. Cormen et al. [Cormen et al., 2009] defined two key ingredients that an optimization problem must have in order for dynamic programming to apply, namely *1)* an **Optimal substructure**, and *2)* **Overlapping subproblems**.

A problem exhibits **optimal substructure** if an optimal solution to the problem contains within it optimal solutions to subproblems [Cormen et al., 2009] . In this context, at each node, a set of choices is given (neighboring nodes). The main problem consists of finding one of the shortest paths p between a defined initial state s and a defined target state t. The fact that recursions can be used to solve the problem, as shown in Section 5.2.1, already proves that the main problem can be cut into subproblems. If w[7] is a state on p, the main problem could then be split into subpaths p_1 from s to w and p_2 from w to t (*cut-and-paste argument* [Cormen et al., 2009]). The optimal solution to the main problem is therefore built from optimal solutions to the subproblems.

The second ingredient needed for dynamic programming is **overlapping subproblems.** This means that the space of subproblems must be small. In other words, the recursive problem-solving function shall solve the same subproblems instead of generating new ones. In the quest for an optimal route in this context, each node would generate a set of neighboring nodes, some of which represent the subproblems, as they might be on

[7] w could be the state in which all tenants are alone on their instance, and also alone on a server.

the optimal global path. However, it cannot be ensured that these subproblems are not new. The old subproblems, which have already been solved can certainly have their results stored (in an array or hash table) so that these results can be used in case the subproblem reoccurs rather than solving it again; this process is known as **Memoization** [Cormen et al., 2009]. Furthermore, the subproblems should be smaller than the main problem, and also smaller than each other. For example, Factorial *f(n)* of an Integer n consists of $f(n) * f(n-1)$; the same problem is being resolved repeatedly, and it gets smaller. This is not the case in the context of this thesis. This cannot be ensured that the subproblems are smaller than each other. Also, it cannot not be ensured that a previously computed solution can be used to solve another subproblem.

In summary, DP requires the problem to be cut into overlapping subproblems which exhibit an optimal structure. For the problem at hand, the only node which could be used to cut the main problem into subproblem is the node in which each tenant is alone on its instance, and alone on a server. However, although this node is certainly on a path from the initial state to the target state, it is not necessarily on the shortest path. Moreover, the subproblems do not always overlap. Hence, Dynamic Programming cannot be used to solve the problem.

The solution needed has to make the best choices while building the graph; only the nodes which may lead to the target node should be taken into account and expanded. Following Section 5.2.3 will present a Greedy Approach and discuss its suitability.

5.2.3. The Greedy Approach

The algorithm that is needed shall go through a sequence of steps, with a set of choices at each step and always make the best one. An example for such an algorithm is the *greedy algorithm*. First appearance occurred in the combinatorial optimization literature long ago, namely in year 1971 in an article by Edmonds [Edmonds, 1971]. A greedy algorithm always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution [Cormen et al., 2009].

In the context of the problem faced in Section 5.2.1, applying a greedy algorithm could help reduce the overhead by only analyzing and expanding the best nodes at the moment while building the graph. Indeed, starting at a given initial configuration, the greedy approach would expand only the neighboring configuration with the lowest cost. This will be repeated until the target configuration has been reached. The algorithm can be implemented either recursively or iteratively. The recursive implementation would repeat the same procedure for each best promising configuration at the moment. On the other

hand, the iterative version would need to memorize each of the new configurations computed and choose the best one to continue with. The configurations could, per example, be sorted by their cost and stored in a container (e.g. *ArrayList* or *PriorityQueue* in Java); the algorithm would then always choose the very first element in the container. This reminds of the idea behind the well-known Dijkstra Algorithm, which is also a greedy algorithm [Cormen et al., 2009].

Analysis

Using a greedy algorithm would surely ensure that only the “best” configurations will be taken into account; these would be the configurations which are reached with the lowest cost from the initial configuration. Thus, all other “expensive” configurations are ignored. The total amount of configurations generated by this approach will certainly be less compared to the brute-force approach from Section 5.2.1.

This approach, however, presents some important downfalls. In fact, it is likely that the most optimal short-terms solutions may lead to the worst long-term outcome. Indeed, already at the beginning, the neighboring configuration that the algorithm may choose to expand, and which is the one with the lowest cost, may lead in the wrong direction. It might then never be possible to find the target configuration because a set of wrong decisions will have been continuously taken. However, taking the right decision isn't trivial as the whole environment isn't predetermined.

What is needed is rather an algorithm which may at each node, in addition, recognize the best direction to go and build the graph with the minimum set of configurations as possible. The algorithm shall thus be able to compute the most cost efficient route in an unknown environment. For this purpose, the algorithm shall be able to recognize at any time which of the “cheapest” neighboring configurations would lead to the target configuration. That is, for each new neighboring configuration generated, it shall choose the neighboring configuration which would lead to the target at an overall lower cost. This configuration might not be the cheapest neighboring configuration; in fact, it could even be the most expensive one in case none of the others may lead to the target. These kind of algorithms are known as the *Online Graph Search Algorithms*.

5.2.4. The “Cow-Path” Approach

In [Kao et al., 1996], Kao et al. analyzed generally the problem of searching in an environment, which is basically unknown. More precisely, the research focused on the so-called *w-lane* or *cow-path* problem. The problem has been defined as follows:

> *Consider a cow, Bessie, standing at a crossroads (referred to as the origin) with w paths leading off into unknown territory. On one of the paths there is a grazing field (the goal) at distance n from the intersection, and all of the other paths go on forever; unfortunately, Bessie's eyesight is not very good - she will not know that she has found the field until she is standing in it (i.e. she cannot see down the road). Clearly Bessie must walk at least distance n to get to the field; if she knows which path to take, she will walk exactly distance n. When Bessie has no prior knowledge of which path the field is on, or of the value n, we would like to know how she can find the field while traveling the least distance possible* [Kao et al., 1996].

This scenario clearly reflects the challenge that this thesis is facing. Like Bessie, at the algorithm searched for, starts at a defined initial state, the source state. At each state, there is a defined number of neighboring reachable from that given state; the actual state in which the algorithm is, represents the *intersection*; the neighboring states, more precisely, the paths to the neighboring states, represent the *unknown territory*; their total number is w. In the context of this thesis, n represents the total least expensive cost to the goal state. Obviously, if the algorithm could choose the right path right away at the beginning, it would directly find the shortest path to the goal state and would expand only the nodes on that path. However, like Bessie, the algorithm does not know which paths lead to the goal, and has to find it out, while expanding as least nodes as possible.

In their approach to solve the *Cow-Path* Problem, Kao et al. [Kao et al., 1996] introduced some randomization, and show that this approach is better than the earlier deterministic ones by Baezayates at al. [Baezayates et al., 1993]. In fact, randomized algorithms are known to be easier to analyze and implement, and faster than deterministic algorithms, and some are even able to efficiently solve problems for which no efficient deterministic algorithm exists (e.g. polynomial identity testing) [Motwani and Raghavan, 1995]. A Randomized Algorithm can be seen as an algorithm that performs some of its choices randomly at some states (i.e. “flips a coin”); the same performance as a complicated deterministic algorithm is guaranteed [Gomes and RyanWilliams, 2005]. Furthermore, [Gomes and RyanWilliams, 2005] and [Gomes et al., 1998] found out and proved that the addition of randomness into a computational process can lead to a significant speedup over purely deterministic methods.

The result of Kao et al. [Kao et al., 1996] is a so-called *SmartCow Algorithm*, an online, randomized algorithm for the *cow-path* problem, and which is described in following Pseudocode 2.

Basically, at the beginning, the *SmartCow Algorithm* chooses a random path to investigate (line 4). The algorithm follows this path up until a certain distance d (line 9). In case the

Pseudocode 2 SmartCow Algorithm [Kao et al., 1996]

```
1: function SMARTCOW
2:     r ← A constant geometric ratio > 1
3:     w ← Total number of paths or lanes leaving the current node.
4:     σ ← A random permutation of {0, 1, 2, ... , w − 1};
5:     ϵ ← A random real uniformly chosen from [0,1);
6:     d ← r^ϵ;
7:     p ← 0;
8:     repeat
9:         Explore path σ(p) up to distance d;
10:        if goal not found then
11:            return to origin;
12:        end if
13:        d ← d ∗ r;
14:        p ← (p + 1) mod w;
15:    until goal found;
16: end function
```

goal state has not been found along the chosen path and up until the computed distance d (line 6), the algorithm returns to the source state (line 11), increments the distance to sweep (line 13), and now computes - instead of randomly choosing - the new path to investigate (line 14). To determine the paths to follow, each edge leaving the current (source) node is assigned a number from 0 to $w - 1$, with w being the total number of edges leaving the (current) node. The different permutations of these edges are computed and at the beginning of the algorithm, one is chosen randomly.

The *SmartCow Algorithm* could be applied to the problem faced in this thesis. Starting at the source node, all the neighboring nodes would be computed and numbered from 0 to $w - 1$. This would then be used to build the different permutations. For example, consider 3 edges leaving the source node, namely 0, 1 and 2. The permutations of these numbers are [{0, 1, 2}, {0, 2, 1}, {1, 2, 0}, {1, 0, 2}, {2, 0, 1}, {2, 1, 0}]. At the beginning of the algorithm, one of these paths is chosen randomly and explored up until d. If, for example, the second path has randomly been chosen for exploration, the algorithm would at first visit node 0, and could stop the exploration there in that node in case the distance (or cost, in the context of this thesis) to node 0 already corresponds to d. In a second run, the algorithm would return to the source node, compute a higher d, compute the new path to take, and explore it. According to line 14 in Algorithm 2, the next path would be {1, 2, 0}. This means that edge 1 would be explored at first, and then edge 2 and etc.. Continuing this example would show that in the end, all paths will gradually be explored to incremental distances. This will continue until the goal is found.

Analysis

Applying the *SmartCow* Algorithm to the problem faced in the previous Sections in this chapter could help avoid the overhead mentioned in Section 5.2.1. The use of randomness could possibly help finding the goal with a small number of expansions as possible. Though, in a worse case scenario, all nodes could have to be expanded. The use of randomness can thus not ensure that only the nodes leading to the target goal will be expanded.

Furthermore, *SmartCow* assumes that the number w of outgoing edges at each node is constant. In the context of this thesis, the number w could change from a node to the other, as the nodes do not always have the same number of neighbors. For an example, if a configuration contains an empty ACI, a neighboring configuration could be the one without the empty ACI; a configuration without empty ACI would not have such a neighbor.

Finally, *SmartCow* assumes that there is exactly one path from start state to target state; it then focuses on optimizing the running time. In this thesis, however, the start and target states might be connected by more than one path. The solution then consists of the optimal possible path between start and target nodes.

In summary, although *SmartCow* could solve the problem faced in this thesis, the challenges mentioned in Section 5.2.1 could arise again. What is needed is an algorithm that would soon recognize which nodes are not on the solution path. The algorithm would then ignore those nodes and concentrate only the potential solution nodes (nodes which are the solution path). This means that only a few neighbors will be further expanded at each node.

5.2.5. Minimin Lookahead Search & Alpha Pruning

The solutions discussed in the previous Sections of this chapter computed for each configuration the neighboring configurations and arbitrary choose to move to one of the computed neighbors. There is no of mechanism enabling to detect early enough, if a neighbor is worth expanding or not. The algorithms described in this Section offer such a feature.

The first algorithm is known as *Minimin Lookahead Search* [Korf, 1990]. This consists of exploring the environment (i.e. the graph) to a certain depth starting at the current node (i.e. node where the search is at, at the moment) and eventually evaluating the frontier nodes at the search horizon. Exploring the graph from current node to a depth $d = 2$,

for example, means that, starting at this current node, all the second neighboring nodes (i.e. neighboring nodes reachable by two moves - along the edges) should be computed. A frontier configuration is a configuration found at the defined depth from the actual configuration. The evaluation function used on each frontier node n is the same as the A*-Search (see Section 4.5) cost function, namely $f(n) = g(n) + h(n)$. g represents the known exact cost from the root node to the current node; that is, at the beginning of the algorithm $g(n) = 0$. h represents the heuristic estimated cost from the current node to the target node; hence, at target node, $h(n) = 0$. Additionally, h is required to be admissible: it should not overestimate the exact cost from the current node to the target node but should rather return a lower bound (see Section 2.6). Eventually, the algorithm would choose to move to the direct neighboring node (of the current node), which led to the lowest frontier's *f-value*.

In order to find the frontier node with the minimum cost, all frontier nodes have to be evaluated. This can result in an expensive computing task, given the size of the graphs. The second algorithm, known as *Alpha Pruning* by analogy to *Alpha-Beta Pruning* can be used to ease this task [Korf, 1990]. *Alpha Pruning* requires the evaluation function f introduced earlier in this Section to be monotonic. A function is monotonic if it never decreases (or increases) along a path away from the root node [Thomson et al., 2008]. In fact, if node n_2 is a successor of node n_1, and $k(n_1, n_2)$ the exact cost to get to node n_2 from node n_1, this means:

$$
\begin{aligned}
f(n_2) \geq f(n_1) &\Rightarrow g(n_2) + h(n_2) \geq g(n_1) + h(n_1) && (1)\\
&\Rightarrow g(n_1) + k(n_1, n_2) + h(n_2) \geq g(n_1) + h(n_1) && (2)\\
&\Rightarrow k(n_1, n_2) + h(n_2) \geq h(n_1) && (3)
\end{aligned}
$$

The property in line 3 is referred as the consistency of $h(n)$ [Kumar, 2008]. Thus, monotonicity of $f(n)$ is equivalent to consistency of $h(n)$ [Korf, 1990].

The monotonicity of $f(n)$ is the core of *Alpha Pruning* and it allows to decrease the number of frontier nodes that shall be evaluated before deciding the neighbor node that shall be expanded next. Following Pseudocode 3 outlines the algorithm.

The idea in Pseudocode 3 is to generate all the neighbors of the given starting node up to the given depth. The variable α represents the lowest *f-value* of the frontier nodes and is initialized with ∞ (line 3). This means that the very first better value of α is the *f-value* of the first frontier node encountered. Since the function f is monotonic, the *f-value* of any successor of the current node can only be greater than or equal to the current node's *f-value*, and hence cannot be lower than the value of the frontier node responsible for the current value of α. This allows to prune each generated interior node, which has its *f-value* smaller than α. In addition, α is overwritten anytime the algorithm comes across

Pseudocode 3 Alpha Pruning Algorithm

```
1: function getAlpha(node, depth)
2:     stack ← node;
3:     α ← ∞;
4:     f(node) ← g(node) + h(node);
5:     while stack not empty do
6:         currentNode ← pop(stack);
7:         if α ≠ ∞ & f(currentNode) ≥ α then
8:             for all child of currentNode do
9:                 k ← cost from currentConfig to child;
10:                g(child) ← g(currentNode) + k ;
11:                f(child) ← g(child) + h(child);
12:                if f(child) < α then
13:                    if childDepth = depth or child = target then
14:                        α ← f(child);
15:                    end if
16:                else
17:                    Put child on stack
18:                end if
19:            end for
20:        end if
21:    end while
       return α;
22: end function
```

a lower *f-value* at a frontier.

Analysis

Minimin Search extended by *Alpha Pruning*, applied in the context of this thesis, introduces a new aspect, namely the "reasoning" aspect. The cost function used on the edges should represent the cost of the single operation (e.g. setting up or removing an empty server, creating or removing an ACI, moving a tenant from an ACI to another, etc.), which transformed any given (valid) configuration to a neighboring configuration. The same function can be used as heuristic function to determine the estimated cost of transforming any given configuration into the target configuration. In that case, the heuristic function would add the costs of an arbitrary set of operations which could be applied on the given configuration in order to transform it in the target configuration. However, the deployment constraints should then not be taken into account (by the heuristic function). Using the same approach for both the cost and the heuristic functions, as well as not considering the deployment constraints, in the case of the heuristic function would ensure an admissible heuristic. Furthermore, the consistency of the heuristic function can be assured because of the transformation costs being positive, and given the fact that, if t is the target configuration and s the initial configuration, $h(t) = 0$ and $g(s) = 0$.

The introduction of reasoning at each node of the graph before choosing, which neighboring node will be expanded next would help reduce the number of nodes generated to find the target state. However, this could also lead to Dead-End nodes, or the algorithm could move back to an already expanded node and loop forever. This cannot be avoided by just excluding expanded nodes, as neighbors of the current node may have all been expanded before. Moreover, choosing the depth of the search horizon can be a tricky task: Evaluating many moves ahead leads to better decisions since obtaining better estimates of the possible strategies can help with selecting the appropriate responses in the current situation. However, while trying to mathematically explain this principle, Lustrek et al. proved the opposite: this has been termed as *Pathological Lookahead* and appears when the "quality of a shallower heuristic is higher than the quality of a deeper one" [Luštrek et al., 2006]; this has also been addressed in [Tavcar, 2012]. Finally, as noticed by Korf [Korf, 1990], since the reasoning at each node is based on limited information, an initially promising path can turn out to be less favorable after more information has been acquired down the road, thus requiring the algorithm to backtrack to a previous choice point; this could, again, lead to infinite loops. Hence, a way to prevent infinite loops while allowing backtracking is needed. Following Section 5.2.6 describes such an algorithm.

5.2.6. The Real-Time A* Approach

The main weakness of the algorithm presented above in Section 5.2.5 is its incapability of backtracking to a previous choice node in case the path previously chosen turns out to be less favorable after gathering additional information in the process of exploring it. Backtracking should occur when the estimated cost of continuing the current path exceeds the cost of going back to a previous node, plus the estimated cost of reaching the target from that node [Korf, 1990]. This Section describes the *Real-Time A** (RTA*), an online algorithm, which has been introduced by Korf [Korf, 1990] and which offers such a feature. RTA*is the last solution approach that would be analyzed in this chapter.

As in *Minimin Lookahead Search* and in *A**, RTA*also makes use of a merit function $f(n) = g(n) + h(n)$ to evaluate a node n. $h(n)$ represents the heuristic function in this case as well and, unlike in *Alpha Pruning*, does need to be only admissible; its consistency is not required. Though, $g(n)$ is defined differently. In fact, in RTA*, $g(n)$ represents the actual cost of any given node n from the current node rather than from the root node. Thus, the cost from the root node to the current node is irrelevant, as each node is evaluated relatively to the current node. The activity diagram in Figure 5.4 illustrates the algorithm.

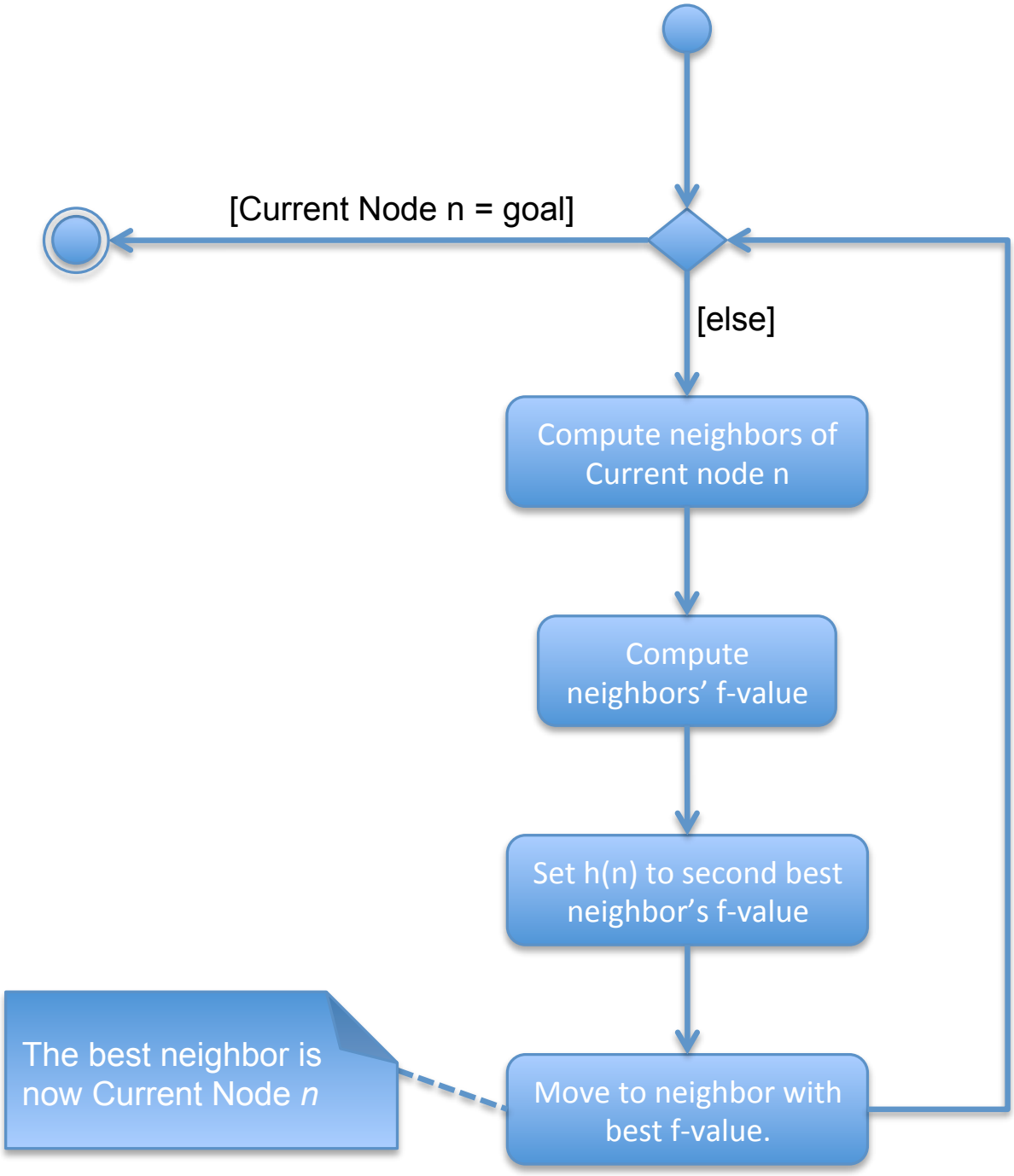


Figure 5.4.: RTA* Algorithm

The basic idea of RTA* as shown in Figure 5.4 is the following: at each cycle, the algorithm expands the current node by generating its neighbors, and evaluates them using the merit function $f(n) = g(n)+h(n)$ introduced earlier in this Section. The algorithm then chooses to move to the best neighbor, that is the one with the minimum *f-value* and stores at the previous node, the second best neighbor's *f-value* as its new $h(n)$. This represents the estimated h cost of solving the problem by returning to this node, from the perspective of the new node [Korf, 1990]. For a previously expanded node, the stored *h(n)-value* is used to evaluate the node whereas for a new node, the heuristic evaluator is called. This

procedure is repeated until the target state is found.

Analysis

RTA*introduces a backtracking mechanism that could help solve the problem described in this chapter in particular, as well as the overall problem addressed by this thesis in general.

The heuristic function needed in this context could be the same as the one proposed earlier in Section 5.2.5: it should be able to compare any given deployment configuration with the target deployment configuration and return a value that would represent a lower bound of the real cost of transforming the given deployment configuration into the target configuration. Here also, the heuristic should not consider the deployment constraints: invalid configurations could be allowed in the process. The cost function, used on the edges of the graph to represent the cost of transforming any given configuration into its neighbor, should follow the same core principle while being applied only on valid configurations.

RTA*would not lead to an infinite loop, given the fact that each time the algorithm makes a move (inclusive backtracking), it increases the *f-value* of the previously expanded node, by the storing the second best neighbor *f-value* as the current node's *h-value*. Hence, even if the algorithm steps into a loop, it would at a certain time breach out and change its search direction to go a step further than the previous time, and gather more information [Korf, 1990]. The heuristic function of any given node could also be augmented by a *Minimin Lookahead Search*. Although this would result in a more expensive heuristic, it would increase the information gathered at the node.

Furthermore, according to Korf [Korf, 1990],

> "In a finite problem space with positive edge costs and finite heuristic values, in which a goal state is reachable from every state, RTA*will find a solution" [Korf, 1990].

In the context of this thesis, these conditions are all satisfied. In fact, the edge costs are expected to be positive, as none of the single operations to transform a configuration into another can cost negatively. The heuristic function described earlier is finite. In addition, the single operation which lead to a deployment configuration can be undone (e.g. adding an empty ACI/remove an empty instance); this ensures that the target node can be reached from any given node.

Finally, since RTA*makes decisions based on limited information (the merit of each node

is computed relatively to the current node) as well, a general statement about its quality cannot be made. It can rather be assured that the decisions taken are optimal relatively to the part of the search space that it has seen so far. Each move made by RTA*is along a path whose estimated cost of reaching the target node is minimum.

5.3. Solution approaches: Summary

Section 5.2.1 describes a brute-force approach, which consists of generating all possible deployment configurations reachable from the given initial deployment configuration as graph nodes. For each new node generated, the algorithm computes all its neighbors and expands them recursively. The target configuration is enclosed in these generated configurations. The (directed) edges in the graph bear the transformation cost. Using well-known weighted graph search algorithms, such as the Dijkstra Algorithm, a cost effective path from the initial to the target configuration can be found. The advantage of such an approach is that the path computed is the best solution, as all possibilities have been taken into account. However, the number of nodes generated by this approach grows exponentially with increasing number of tenants: a configuration with 10 tenants would generate a graph with over 70 Million nodes. Thus, this approach cannot be used in a real-world context with hundreds of tenants.

Section 5.2.2 describes a solution approach based on Dynamic Programming. It consists of cutting the problem into smaller pieces and resolving them recursively. Both of the following criteria are to be met for a problem to be solved by Dynamic Programming:

- The problem exhibits an optimal substructure: an optimal solution to the problem contains within it optimal solutions to subproblems.
- The problem contains overlapping subproblems: the problem-solving function shall solve the same subproblems instead of generating new ones.

Section 5.2.2 shows that the second criterion isn't met by the problem faced in this thesis, as it cannot be guaranteed that the subproblems are smaller than the main problem. Moreover, Dynamic Programming expects full knowledge of the environment (graph), which, as mentioned earlier, is not possible.

In Section 5.2.3, an attempt at a solution, based on a Greedy Approach is analyzed. Instead of generating all the possible configurations as required in the previous approaches, the Greedy Approach chooses to generate and expand only the configuration with the lowest cost: it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution. However the locally best choices could lead to following a

wrong path.

Section 5.2.4 sketches an online algorithm, namely the *SmartCow Algorithm*, which had been introduced to solve the well-known *cow-path*, also known as *w-lane problem*. The algorithm randomly chooses a path and explores it up to a given distance. If the target node hasn't been found, *SmartCow* chooses another path, which will be explored up to a longer distance. This means that the same path can be explored several times with increased distances. This principle is repeated until the target is found. *SmartCow* has been ruled out as a solution for the problem in the context of this thesis for different reasons. For example, it assumes that there is exactly one path connecting the initial and target nodes. Hence, *SmartCow* focuses on optimizing the running time instead of the path costs.

Section 5.2.5 introduces some reasoning. *Minimin Lookahead Search* searches forward from the current node to a fixed depth and applies the A* evaluation function $f(n) = g(n)+h(n)$ on each frontier node n of search horizon before eventually choosing to expand the neighboring node of the current node, which is on the path to the minimum $f(n)$. In case the evaluation function f is monotonic (which is equivalent to the heuristic h being consistent), *Alpha Pruning* can be used to evaluate a fewer number of frontier nodes in the process of determining the one with the minimum *f-value*. This principle is repeated at each new node until the target node is found. However, ignoring the information from previous searches as this algorithm does, will result in infinite loops.

Finally, Section 5.2.6 outlines a more promising algorithm, the *Real-Time A** (RTA*), another online algorithm, which also evaluates each neighboring node n of the current node using a merit function $f(n) = g(n) + h(n)$ choosing to expand the one with the minimum *f(n)-value* next. RTA*, however, defines $g(n)$ as the cost from current node to neighbor node n. The algorithm introduces a backtracking mechanism, which occurs when the estimated cost of continuing the current path exceeds the cost of going back to a previous node, plus the estimated cost of reaching the target node from that node. Furthermore, augmenting the heuristic in RTA*with *Minimin Lookahead Search* (assisted by *Alpha Pruning* would result in a better heuristic, which could speed up the search. Moreover, the algorithm is complete and ensures to find an optimal solution.

A lot more algorithms, such as the *PHA* Algorithm* [Felner et al., 2004], the *Focussed D** [Stentz, 1995] or *D** [Stentz, 1994] have been examined. For example, *D** is capable of planning paths in unknown, partially known, and changing environments in an efficient, optimal and complete manner [Stentz, 1994]; though it has soon been identified as not a contender to solve the problem at hand because of the fact that it expects the cost to change over time.

Among all the algorithms described, the RTA*Algorithm, using a heuristic augmented by *Minimin Lookahead Search* has been identified as the most suitable. Chapter 6 deals with the technical details about its implementation.

> *The computing scientist's main challenge is not to get confused by the complexities of his own making.*
>
> – *E.W. Dijkstra*

6

The Migration Planner

The previous chapter presented different algorithms and thoroughly analyzed them. It ended by choosing the most suited one which this chapter implements. This chapter aims to give a general overview on the specific techniques that the implementation is based. Hence, it describes, for example the external library that has been used for the graph without going too deep in the matter.

This chapter is structured according to Figure 1.1. The implementation is realized in Java as analogy to [Reinhardt, 2013], [Ruehl et al., 2012] and [Ruehl et al., 2013] and also because of the numerous ready-to-use libraries available.

6.1. Loading the Deployment Configurations

In [Reinhardt, 2013], Reinhardt made the optimal Deployment Configuration computed by the *Deployment Configuration Generator* from Ruehl et al. [Ruehl et al., 2012] available as an Extensible Markup Language (XML) file. The use of XML as data format brings advantages such as technology independence, loose coupling, high cohesion of both systems[8].

There exists different standard Application Programming Interfaces (API) to transform an XML file to an object oriented structure readable by a Java application. Examples

[8]Both the Platform developed in [Reinhardt, 2013] and the *Deployment Configuration Generator* from [Ruehl et al., 2012].

include SAX (Simple API for XML), DOM (Document Object Model), JDOM (Java native API) [Harold, 2002]. The Java Architecture for XML Binding (JAXB) also offers the functionality of directly binding an XML file by mapping Java classes to their XML representations, and will be used in this context, in analogy to [Reinhardt, 2013]. JAXB provides two primary features, namely the ability to *unmarshal* XML into Java objects and the inverse, i.e. to *marshal* Java objects into XML. The former feature is the one needed. Though, JAXB requires a *Schema* to be properly defined. A schema is an XML specification that governs the allowable components of an XML document and the relationships between the components [Ort and Mehta, 2003].

The XML Schema has been defined according to the Entity Relationship Diagram in Figure 2.5. This schema presents a slight modification compared to the one proposed by [Reinhardt, 2013]. In fact, [Reinhardt, 2013] modeled the ACs as an entity from which the ACIs derive, thus treating ACs as a Deployment Level. In this thesis, however, ACs have been modeled as attribute of ACIs, as in fact, an ACI (from a specific AC) runs directly on a server. Appendix B presents the code for the XML schema definition while Appendix C shows a concrete example of a deployment configuration.

Now that the schema is ready, it can be bound. Binding a schema means generating a set of Java classes that represents the schema [Ort and Mehta, 2003]. The unmarshalling process can be started as shown in Listing 6.1.

```
1 import javax.xml.bind.JAXBContext;
2 import javax.xml.bind.Unmarshaller;
3 ...
4
5 JAXBContext jaxbContext = JAXBContext.newInstance(
     DeploymentConfiguration.class);
6 Unmarshaller jaxbUnmarshaller = jaxbContext.createUnmarshaller();
7 DeploymentConfiguration config = (DeploymentConfiguration)
     jaxbUnmarshaller.unmarshal(new File(xmlFile));
```

Listing 6.1: Example of a Deployment Configuration

Line 5 creates a *JAXBContext* object by specifying the context path, which is the path to the interface previously generated by the binding compiler. Line 6 create an *Unmarshaller* object, which controls the process of unmarshalling and contains the methods that perform the unmarshalling activity. The *unmarshal* method, called in line 7, is the method which does the actual unmarshalling of the XML document. The *config* object in line 7 then contains the Java object (of type *DeploymentConfiguration*) representation of the given XML file. The *DeploymentConfiguration* class contains a list of servers, of type Server; the Server class contains a list of instances of type ACI, and the ACI class contains a list of tenants of type Tenant. Each of the classes also contains their respective

attributes (i.e. ram, ip, cpucores for Server, ac for ACI and id for Tenant).

The *DeploymentConfiguration* class is created automatically and is overwritten each time the application is compiled. This makes its extension with user-defined functions such as the constraints validation function (see Section 6.3 unfeasible. To overcome this, a new class with the same name has been created in a new package, as Java does not allow two classes of the same name in the same package. The new class is given to the automatically generated *DeploymentConfiguration* class as parameter in its constructor as Listing 6.2 indicates.

```
1  package migrationplan;
2  ...
3  public class DeploymentConfiguration {
4    private List<Server> servers;
5    ...
6    public DeploymentConfiguration() {
7      servers = new ArrayList<>();
8    }
9    public DeploymentConfiguration(generated.DeploymentConfiguration
       generatedConfig) {
10     this();
11     for (generated.DeploymentConfiguration.Server server :
       generatedConfig.getServer()) {
12       servers.add(new Server(server));
13     }
14   }
15   ...
16   public Boolean isValid() {
17     ...
18   }
19 }
```

Listing 6.2: DeploymentConfiguration Java class

The new *DeploymentConfiguration* class is created in a package called *migrationplan* while the automatically generated *DeploymentConfiguration* class is located in the *generated* package as shown in lines 1, 9 and 11. The later is then loaded in the former's overloaded constructor (line 9) and its objects are passed as well (line 12). The same procedure is repeated for the Server, ACI and Tenant classes. Embedding the automatically generated *DeploymentConfiguration* class in the manually created one allows the use of only one *DeploymentConfiguration* class with all the features needed. This then changes the unmarshalling procedure of Listing 6.2 as shown in Listing 6.3.

```
1 ...
2 DeploymentConfiguration config = new DeploymentConfiguration((generated
     .DeploymentConfiguration) jaxbUnmarshaller.unmarshal(new File(
     xmlFile)));
3 ...
```

Listing 6.3: Loading XML file in Java object

The *config* in line 2 is now an instance of the new manually created The *DeploymentConfiguration* class, which can be complemented with user-defined functions.

6.2. Designing the graphs: The GraphStream Library

Chapter 4 based the solution approach adopted in this thesis on the concept of graphs. The *GraphStream Library* provides a lot of built-in features that enable an easy interaction with graphs and is used to model these graphs. GraphStream has been chosen by experience.

GraphStream is a graph handling Java library that focuses on the dynamics aspects of graphs [Balev et al., 2013]. Among many other types of graphs (see Section 2.5), the GraphStream Library also proposes classes that allow the modeling of directed multigraphs. Listing 6.4 presents some of the basic features available for GraphStream. This shows that creating graphs with GraphStream, as well as adding nodes and edges to that graph are done with a minimum effort.

```
1  import org.graphstream.graph.Edge;
2  import org.graphstream.graph.Graph;
3  import org.graphstream.graph.Node;
4  import org.graphstream.graph.implementations.MultiGraph;
5  ...
6  //Create graph
7  Graph graph = new MultiGraph("MigrationPlan");
8  //Adding nodes
9  graph.addNode(initialConfig.toString());
10 graph.addNode(neighboringConfig.toString());
11 //Adding edges
12 graph.addEdge(edgeID_1, initialConfig.toString(), neighboringConfig.
     toString(), true);
13 graph.addEdge(edgeID_2, initialConfig.toString(), neighboringConfig.
     toString(), true);
14 ...
```

Listing 6.4: Example of a Deployment Configuration

Listing 6.4 creates a new multigraph in line 7. Lines 8 and 9 add a few nodes to the created graph. The `addNode` function takes as parameter, the unique identifier of the node to be added. This node can be referenced in the graph using this identifier. Lines 13 and 14 draw two edges, both from the given initial deployment configuration to the same neighboring configuration. The `addEdge` function takes as first parameter, the unique edge identifier, then the outgoing and incoming nodes' identifiers, as well as a Boolean parameter, which when set to `true`, renders a directed edge.

GraphStream also provides some ready-to-use weighted graph search algorithms, such as the A* and the *Dijkstra* Algorithms, which will later be used to find the actual shortest (cheapest) path between two given nodes in the graph.

6.3. Valid Deployment Configuration

As mentioned in Section 5.1, a valid Deployment Configuration is a configuration, which respects the Deployment Constraints stated by the tenants in the requirement analysis phase. Each Deployment Level has its own constraints. [Ruehl et al., 2013] modeled the Deployment Constraints as undirected graphs as Figure 6.1 illustrates.

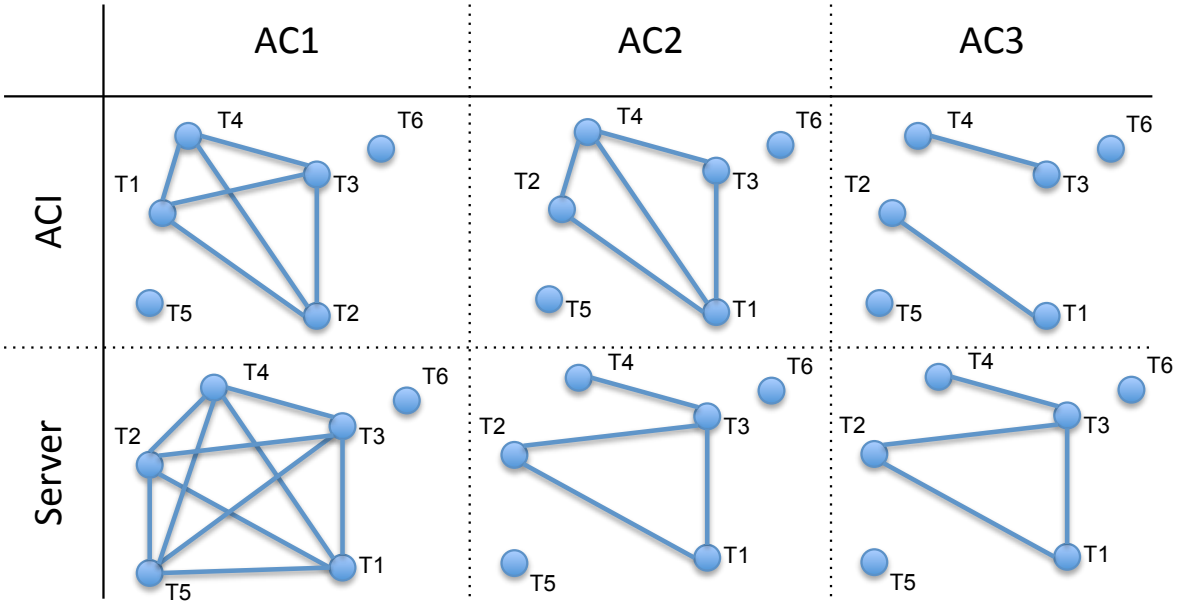


Figure 6.1.: Deployment Constraints [Ruehl et al., 2013]

From Figure 6.1, it can be deduced that two tenants are allowed to share a resource (i.e. ACI or Server) if the nodes representing them are connected by an edge in the graph. For example, Tenants T4 and T3 are allowed to share an instance of AC3; tenants T4 and T1 are not allowed to share a server of AC3. Thus, checking if a deployment configuration is valid results into checking if there is an edge between each of the tenants (more precisely,

between the nodes representing the tenants) on that deployment level.

The manually created *DeploymentConfiguration* class introduced in Section 6.1 has been extended with an *isValid()* function, which checks the constraints as depicted in Listing 6.5.

```
public Boolean isValid() {
  // Iterate through servers
  ...
  // Iterate through tenants on the servers
  ...
  Node nodeOfTenant1 = serverConstraintsGraph.getNode(tenant_1.toString
   ());
  if (!nodeOfTenant1.hasEdgeBetween(tenant_2.toString())) {
    return false;
  }
  ...
  //Iterate through ACIs on each server
  ...
  //Iterate through tenants on each ACI
  ...
  nodeOfTenant1 = aciConstraintsGraph.getNode(tenant_1.toString());
  if (!nodeOfTenant1.hasEdgeBetween(tenant_2.toString())) {
    return false;
  }
  ...
  return true;
}
```

Listing 6.5: Deployment Configuration Validation

For the Deployment Level "Server", the function iterates through all tenants present on each server, and checks if these tenants are all connected by an edge in Deployment Constraint graph for the DL-Server. For the Deployment Level "ACI", the function iterates through all tenants and performs the same check, using the constraint graph for ACIs. The built-in Boolean function `hasEdgeBetween()` of the GraphStream Library, which can be called on a graph node with the identifier of the second node as parameter, allows to verify if two nodes of a graph are connected by an edge.

6.4. The Cost & Heuristic functions: Levenshtein Distance

The cost function, which is assigned to each edge of the Migration Planner's graph, should represent what it costs to transform a given configuration into a neighbor. This function should, in reality, take into account several factors such as time, personnel costs, etc. of

either deploying a new empty server, or a new empty instance, or deleting a server or an instance (and all the other operations needed for the transformation). For this, an empiric study in the corresponding computer centers should be done. However, as this work is a prototype, the *Levenshtein Distance*, also known as *Edit Distance*, has been identified for this purpose.

The Levenshtein Distance is a string metric, which defines the number of edit operations that must be performed to transform one string to the other [Cesare and Xiang, 2012]. An edit operation is applied to only one character at a time and includes character deletion, insertion and substitution.

The Levenshtein Distance will be used on the Deployment Configurations' string representations (see Figure 4.4). Although this does not make any sense physically, it, however, bears the logic of finding the cost to transform one configuration into another one. An implementation of the metric can be found in Appendix D. This function will also be used as heuristic to estimate the cost of transforming any given configuration into the target configuration. Using the same function for computing both the cost between two neighboring configurations and the estimate cost to reach the target configuration will ensure that the heuristic stays admissible as required in Section 5.2.6.

6.5. The Migration Planner

The implementation of the *Migration Planner*, the core process in solving the problem addressed by this thesis, can be done by strictly following the algorithms described in both Sections 5.2.5 and 5.2.6. The *DeploymentConfiguration* class introduced in Section 6.1 is given new attributes, namely f_rta, g_rta, h, f_astar, and g_astar, to hold the f, g, and h values computed according to both RTA*and A*. In both cases, the heuristic h is computed by the same way.

RTA*is implemented by storing in a `java.util.List` all previously expanded *DeploymentConfiguration* objects (representing the graph nodes), including of course their h values. Iteratively, the current *DeploymentConfiguration* object's neighbors are computed and their *f-value* calculated. For determining the *f-value* of an already visited *DeploymentConfiguration*, the *h-value* stored in the list is used. For any new *DeploymentConfiguration*, the *h-value* is calculated as mentioned above in Section 6.4. The GraphStream ready-to-use Dijkstra algorithm is used to compute the *g-values*. In principle, the A* algorithm, which is also provided by the GraphStream Library can also be used. The *h-values* are augmented by the Lookahead Search described earlier in Section 5.2.5. No specification regarding its implementation would be given at this point since Pseudocode

3 has been very detailed. Though, an important aspect, which should be mentioned is the fact that the *DeploymentConfiguration* class is added a new variable, which holds the depth of each *DeploymentConfiguration* object in the graph. The depth for the starting *DeploymentConfiguration* object is 0; each of its neighbors gets its depth incremented by 1, and so on.

Moreover, the *DeploymentConfiguration* class is complemented with a new function for generating the neighboring configurations of the current configuration as shown in Listing 6.6.

```
1  package migrationplan;
2  ...
3  public class DeploymentConfiguration {
4    ...
5    public List<DeploymentConfiguration> getNeighbors(
       DeploymentConfiguration targetConfiguration) {
6      List<DeploymentConfiguration> neighbors = new ArrayList<>();
7      //Compute neighbor
8      ...
9      if(neighbor.isValid()){
10       neighbors.add(neighbor);
11     }
12     ...
13     return neighbors;
14   }
15 }
```

Listing 6.6: Computing neighbors

As mentioned in Section 5.1, the neighbors of a given deployment configuration consist of the configurations resulting in adding an empty server or an empty instance, deleting an empty instance or server. The number of neighbors of the current expanded node can be reduced drastically by only generating the meaningful ones. For example, if the target configuration has more servers than the initial configuration, then a set of empty servers (the number needed) should successively be added before moving any tenant. This mechanism enables the `getNeighbors()` function to compute only the neighbors which could lead to the target configuration. Furthermore, as line 9 shows, the computed neighbor is returned only if it is valid (see Section 6.3).

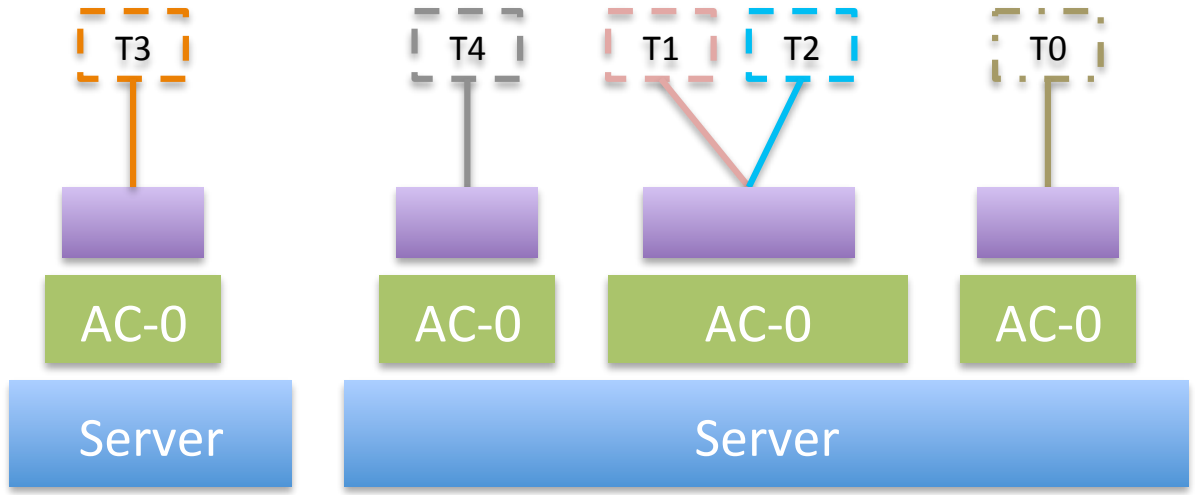
> *An expert is a person who has made all the mistakes that can be made in a very narrow field.*
>
> – *Niels Bohr*

7 Evaluation

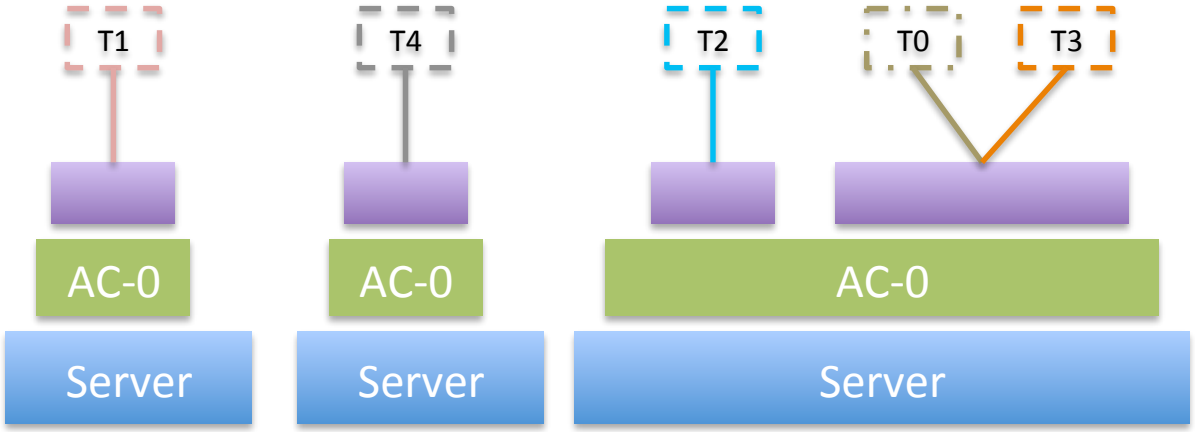
Chapter 5 introduced the *Real-Time A** algorithm, which was implemented in Chapter 6. Chapter 7 now analyzes the quality of this solution. Indeed, the chapter starts by developing different test plans. The results of each of these plans are presented and thereafter analyzed thoroughly.

7.1. Test plan

The testing consists of running the implemented Real-Time A* algorithm with its heuristic augmented by Minimin Search by incrementally increasing the lookahead depth. The depths considered are from 1 to 10. At each run, the cost of the solution as well as the number of nodes expanded, and the time needed to compute the solution will be compared with the cost of the Brute-Force implementation. The Brute-Force approach is supposed to give the optimal solution. The result of such a comparison is known as *Competitive Ratio* (see Section 2.6). The following Figure 7.1a illustrates the initial and target deployment configurations used for the purpose of testing.



(a) Initial Deployment Configuration



(b) Target Deployment Configuration

Figure 7.1.: Deployment Configurations for Testing

Figure 7.1a represents the initial deployment configuration, whereas Figure 7.1b shows the target configuration. Their textual representation is illustrated in Figure 7.2.

Figure 7.2.: Test subjects - Textual representation

The initial deployment configuration consists of two servers: on the first server, tenants T1 and T2 share an ACI whereas tenants T0 and T4 are each deployed alone on their ACIs; on the second server, tenant T3 uses an ACI solely alone. On the other hand, the target deployment configuration consists of three servers: on two of these servers, tenants T1 and T4 each solely use an ACI; on the third server, tenants T0 and T3 share an ACI whereas tenant T4 is deployed alone on its ACI.

In summary, the following will be evaluated:

1. RTA*: Number of nodes expanded vs. lookahead depth.
2. RTA*: Computation time vs. lookahead depth.
3. RTA*: Solution cost vs. lookahead depth.
4. RTA*& Brute-Force: Computation time vs. number of tenants.

The tests are conducted on a 22-Core Intel® Xeon® Processor E5649 (12M Cache, 2.53 GHz, 5.86 GT/s Intel® QPI), 45GB RAM Linux Virtual Machine.

7.2. Results & Analysis

The results of the tests elaborated in previous Section 7.1 are summarized in Table 7.1. These results will also be analyzed in this Section.

Table 7.1.: RTA*vs. Brute-Force Approach

	Lookahead depth	Computation time (min)	Cost	Competitive Ratio	Number of nodes expanded
RTA*	1	0.057	44.00	1.630	54
	2	0.079	31.00	1.148	19
	3	0.405	35.00	1.296	46
	4	0.288	31.00	1.148	20
	5	1.691	33.00	1.222	51
	6	1.417	31.00	1.148	24
	7	4.053	31.00	1.148	20
	8	16.206	31.00	1.148	36
	9	27.613	36.00	1.333	26
	10	41.195	31.00	1.148	21
Brute Force	-	0.332	27.00		16,459.00

Computation time vs. Lookahead depth

The results in Table 7.1 show that the optimal solution, which is returned by the Brute-Force algorithm, is computed in 0.332 minute (≈ 19.92 seconds). Nonetheless, this value is still superior as the time needed by RTA*to compute an approximate solution: 0.057 minute (≈ 3.42 seconds) at depth 1 or a better solution in 0.079 minute (≈ 4.74 seconds) at depth 2. This shows that RTA*is 4 to 6 times faster than the Brute-Force Algorithm.

Furthermore, the chart in Figure 7.3 illustrates the time (in minutes) needed by the implemented *Real Time A* Algorithm* to compute a cost efficient migration plan based

on the scenario introduced earlier in this chapter (see Section 7.1).

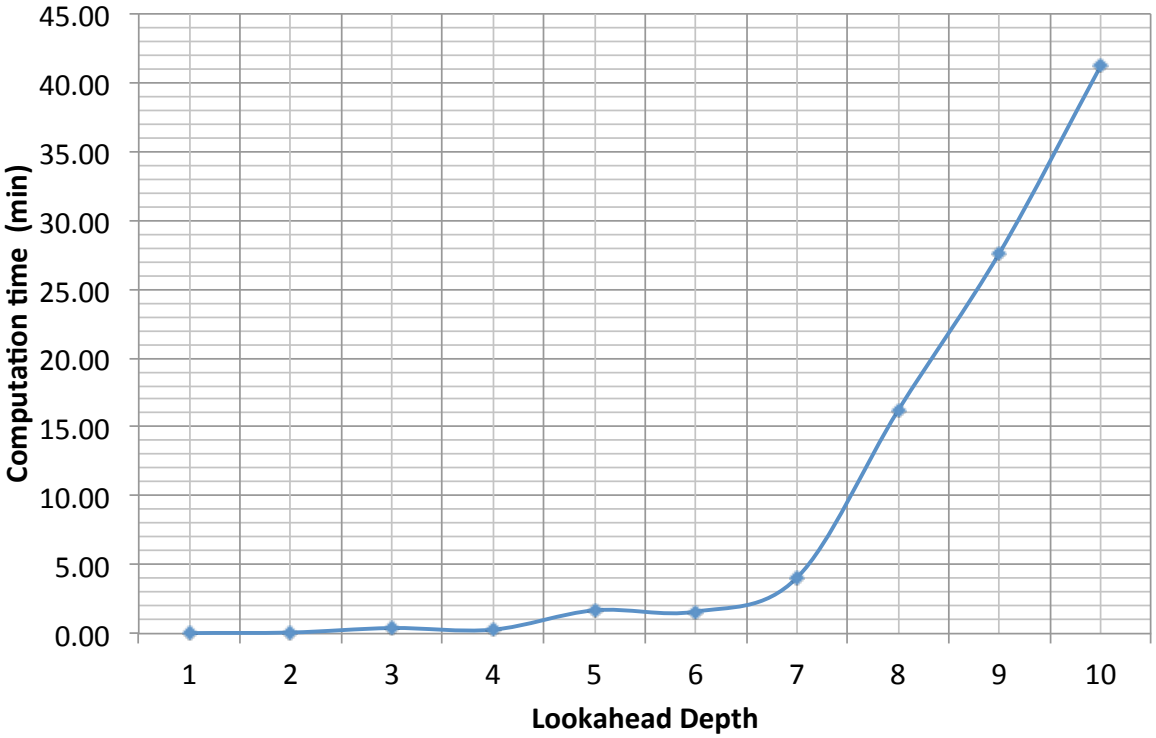


Figure 7.3.: Lookahead depth vs. Duration

The chart reveals an overall exponential growth of the RTA*computation time with increasing lookahead depth. Indeed, increasing the lookahead depth increases the total number of computation needed per move. This means that during the graph built-up, increasing the lookahead depth also increases the time that the algorithm needs to decide which neighboring node should be expanded as next. Theoretically, it is to be expected that the further the algorithm looks ahead, the more precise the solution would be. However, choosing the optimal lookahead depth is not as trivial as it seems to be. In fact, this is problem dependant as it depends on both the relative costs of computation and execution [Korf, 1990]. This case will be analyzed as next.

Cost vs. Lookahead depth

The chart in Figure 7.4 depicts the developing of the cost of the solutions relatively to the lookahead depth.

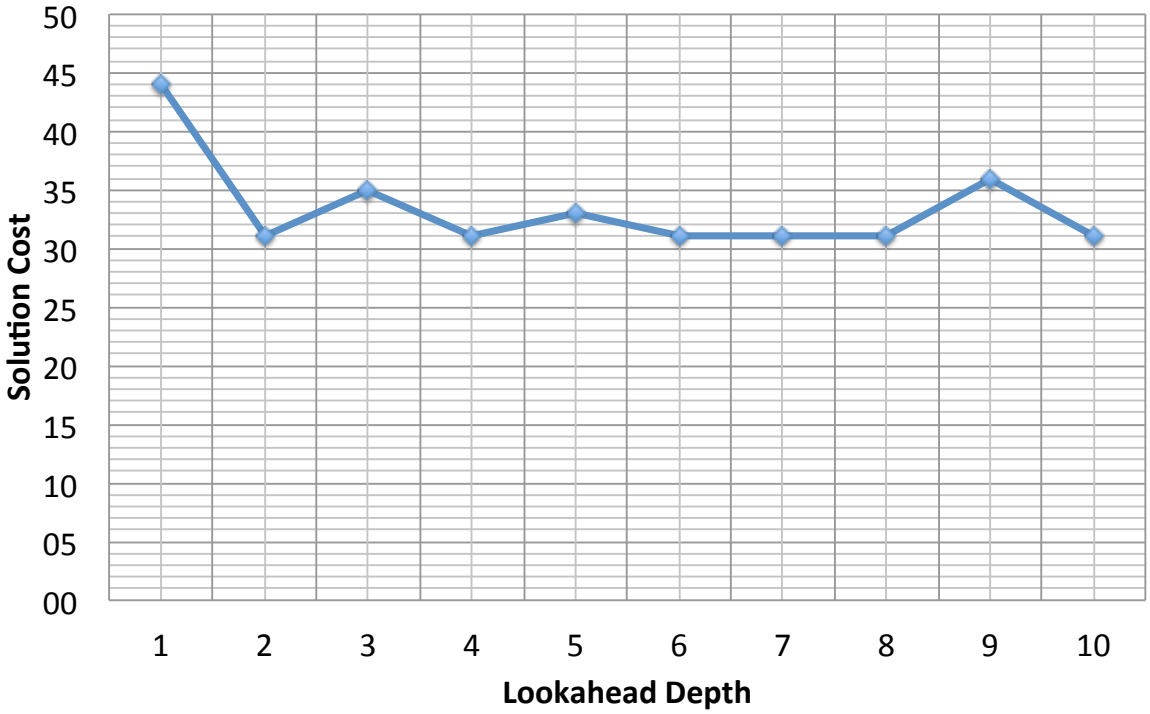


Figure 7.4.: Lookahead depth vs. Cost

The chart shows a quite unexpected course. The costs of the solution was expected to decrease with increasing lookahead depth, hence move closer to the optimal cost computed by the Brute-Force approach. The biggest difference is only between depth 1 and depth 2: the solution computed with a lookahead depth of 1 is 44 while the one computed with a lookahead depth of 2, is 31. However, at depths 3, 5, and 9, increasing the lookahead depth resulted in a slight increase in the average solution cost. Such a behavior is known as *pathology* and was first investigated by Dana Nau [Nau, 1982]. His research was based on the the case of two-player games. He revealed that for certain artificial games, increasing the lookahead depth resulted in poorer play in some cases. Further recent researches in that area have also been done by Lustrek et al. [Luštrek et al., 2006] and Tavcar [Tavcar, 2012].

In his very first work on Real Time Algorithms, Korf [Korf, 1987] analyzed the effects of the pathology phenomenon on RTA*. The work is based on *decision quality* as opposed to *solution quality*: a solution is composed of a large number of individual move decisions; in addition, solution quality is measured by the total length of the solution while decision quality is measured by the percentage of time that an optimal move is chosen [Korf, 1987]. Korf proved that pathology does not show up in terms of decision quality. This means that decision quality improves with increasing lookahead depth while solution quality may decrease. A reason for that might be that *while the probability of mistakes decreases, the cost of any individual mistake can be quite high in terms of overall solution cost* [Korf,

1987].

Number of nodes expanded vs. Lookahead depth

The chart below in Figure 7.5 shows the number of nodes generated by RTA* to find the most cost-efficient possible solution relatively to the lookahead depth.

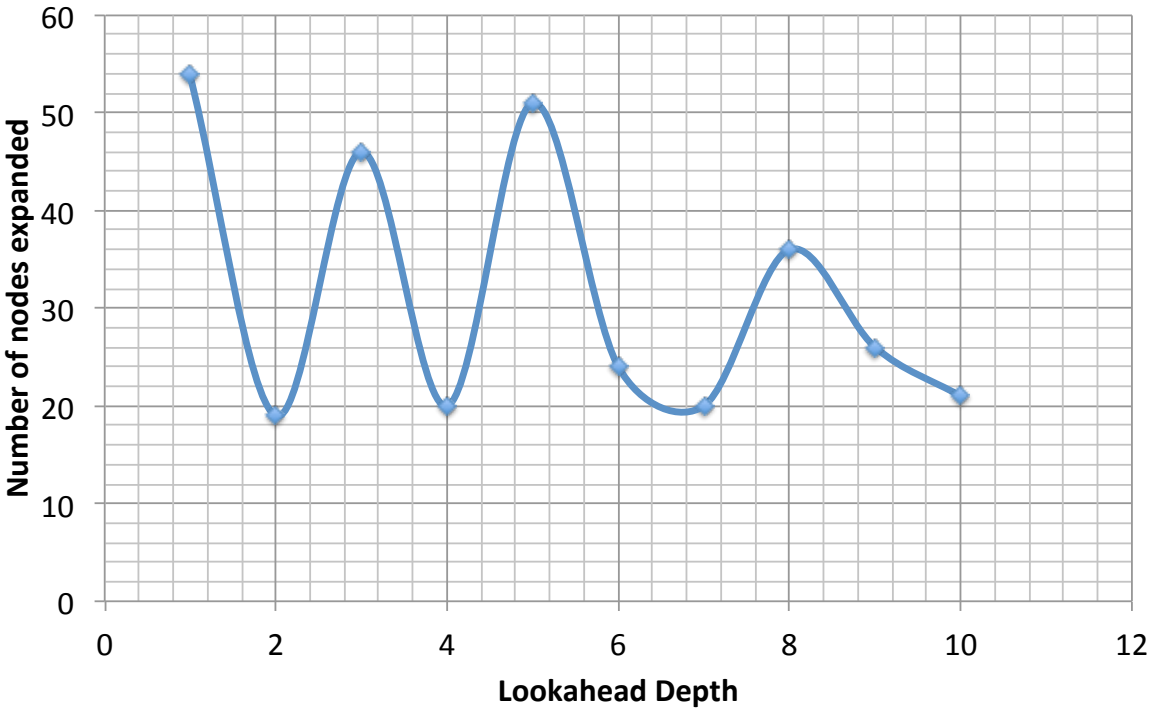


Figure 7.5.: Lookahead depth vs. Nodes expanded

The difference between the average number of nodes generated by RTA* in order to find the solution and the total number of nodes generated by the Brute-Force algorithm is huge. For example, Brute-Force generates over 16,000 nodes. This is more than 300 times the highest number of nodes generated by RTA* (54 nodes at depth 1).

In Section 5.2.1 of this thesis, a lower bound of the number of nodes generated by the Brute-Force Algorithm has been calculated. The same cannot be done for the RTA* Algorithm since the number of nodes generated by this algorithm strongly depends on the specific problem to be solved. Therefore, the number of nodes generated by RTA* could also be reduced using specific conditions applicable to the specific problem.

Computation time: RTA*vs. Brute-Force

Following chart in Figure 7.6 refers to the computation time needed by RTA*on one hand and the Brute-Force algorithm on the other hand, to compute a migration plan for different problems with increasing tenants numbers: from 2 to 10.

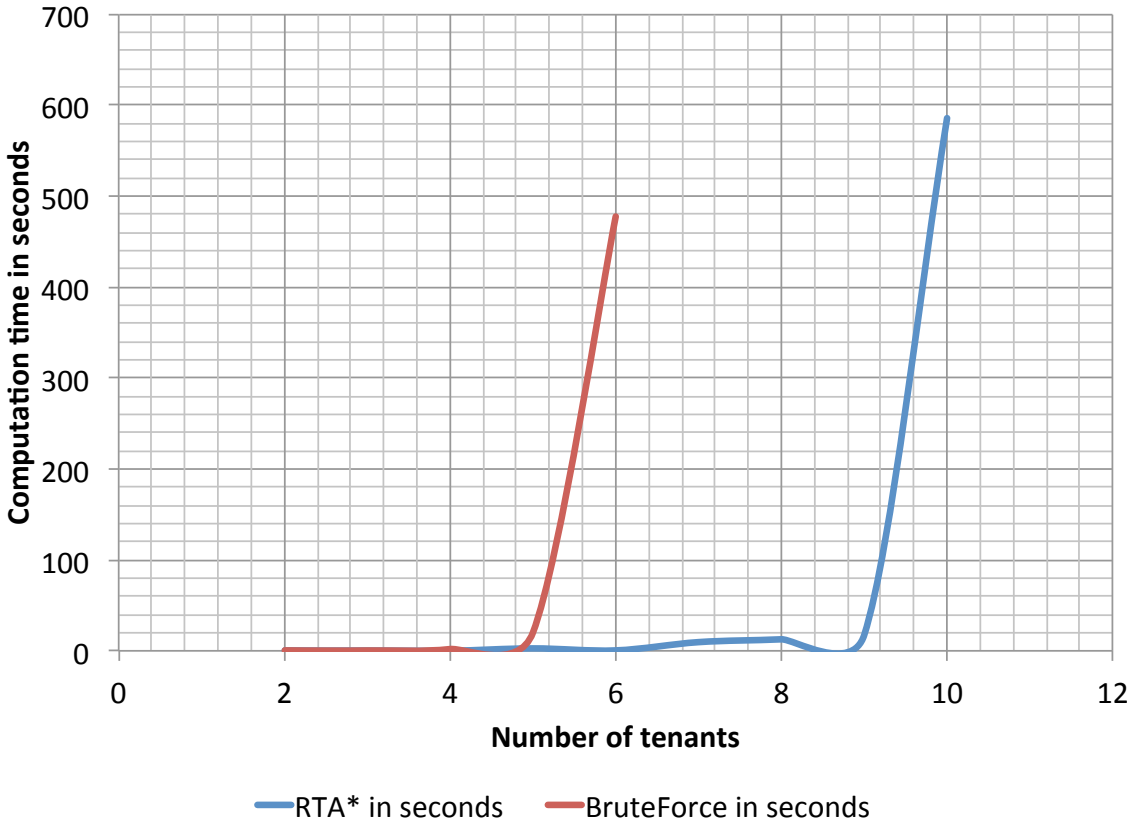


Figure 7.6.: Computation time: RTA*(Lookahead depth = 1) vs. Brute-Force

As expected, the chart shows for both algorithms an exponential growth of the computation time relatively to the number of tenants. At a lower number of tenants (up to 5), the computation time needed for both algorithms is almost the same. The gap between the two algorithms (in regards of computation time) increases drastically already with 6 tenants.

7.3. Summary

In the context of the example, which the evaluation was based upon, the competitive ratio, which has been achieved by RTA*, averaged a value of 1.2. This value is very close to the ideal value of 1. The tests also showed that RTA*strongly depends on the problem to be solved as well as the lookahead depth. However, the optimal lookahead depth cannot be easily identified. One could think of computing different solutions with

different lookahead depths and eventually choose the best one. Furthermore, since in some cases, computing the optimal solution using the Brute-Force approach is not very expensive in terms of computation time, one could think of setting a threshold: the idea is to use Brute-Force for small problems (defined by the threshold) and then RTA*for bigger problem areas.

“There is no real ending. It’s just the place where you stop the story.”

– *Frank Herbert*

“Learn from yesterday, live for today, hope for tomorrow. The important thing is to not stop questioning.”

– *Albert Einstein*

8

Conclusion & Outlook

8.1. Conclusion

In previous researches, Ruehl et al. [Ruehl and Andelfinger, 2011] [Ruehl et al., 2012] [Ruehl et al., 2013] introduced a new Data Model, namely the Mixed-Tenancy Data Model. This model enables tenants to mix the concepts of multi and single tenancy data models on each component of a composite application. This means that they are given the possibility to choose which slice of the infrastructure stack (e.g. Application Component Instance, Server, Cluster, etc.) they are/are not willing to share. A further research by Reinhardt [Reinhardt, 2013] analyzed the feasibility of this model by realizing a Mixed-Tenancy Platform. Reinhardt laid the foundations for deploying a real-world complex composite application on a *Mixed-Tenancy Platform*, by implementing the different customers’ requirements, which are summarized in a so-called *Deployment Configuration*. However, his work did not consider the variability over time of these Deployment Configurations. In fact new customers may need to gain access to the system, others may need to be removed or even moved. The goal of this thesis was to offer SaaS providers, a new way of transforming a previous deployment configuration into the new one without having to delete the actual configuration. Indeed, the research question addressed in this thesis was:

> How is it possible to transform any given initial deployment configuration to any given target deployment configuration at a minimum cost?

The solution that has been proposed in this thesis resolved this question. This solution was based on the concept of the graph theory. Basically, the idea was to build a multigraph with the possible configurations as nodes, and eventually use a weighted graph search algorithm to find the cost-efficient route from the initial configuration to the target configuration. The initial deployment configuration represented the root of the graph. The neighboring configurations consisted of all valid configurations that were reachable by only doing one operation at a time. Such an operation consisted of either moving a tenant to another instance, or creating a new instance, or a new server, or also deleting an empty instance or an empty server. Furthermore, a valid deployment configuration had been defined as a configuration which respected the deployment constraints set by the customer. Using a cost function, the cost of an edge connecting two nodes had been calculated. Since the cost function strongly depends on the real application considered, it has been fully abstracted in this context. Nonetheless, for the purpose of implementation, the Levenshtein Distance has been used. The thesis showed that considering all the valid deployment configurations to build the multigraph (this approach was referred to as the Brute-Force approach) wasn't feasible since the number of nodes in the graph grew exponentially relatively to the number of tenants. In fact, a configuration with 10 tenants could generate a graph with over 70 Mio. nodes. Thereupon, the thesis analyzed different algorithms from the relevant literature and introduced the class of online algorithms. The online algorithms make use of a heuristic function, which estimate the cost of reaching the target node from any node in the graph. The *Real-Time A** (RTA*) Algorithm, which also uses a heuristic function had been identified as most suitable to build the multigraph, among the few algorithms described in this research. Finally, this solution has been evaluated. The evaluation consisted of comparing the results returned by RTA*with the ones returned by the Brute-Force approach, based on an example containing five tenants. This showed that the best competitive ratio achieved by RTA*was 1.1 and the solution was computed in 0.079 minute ($\approx$ 4.74 seconds). This is over 4 times faster than the time needed by the Brute-Force algorithm (0.332 minute, or $\approx$ 19.92 seconds).

8.2. Outlook

There are a few points that this thesis didn't address and which would be essential for this prototype to reach a release level.

First, the solution implemented didn't fully considered the "initialization" and the "finalization" phases introduced in Section 5.1. In fact, it contains only the initialization phase but not the finalization phase. The initialization phase consists of removing from the initial configuration all tenants that are not in the target configuration, directly at

the beginning. In addition, this phase consists of removing and temporally storing all tenants from the target configuration that are not in the initial configuration. The purpose is to have the initial and target configurations with the same tenants to run the RTA*algorithm with. The finalization phase, on the other hand, takes place at the end of the search procedure. This consists of adding back the temporally stored tenants directly where they belong. This could lead to adding a new server, or a new instance, etc.

Another important task would be to define a proper interface for the *Migration Planner's* output. So far, this procedure returns a list of the successive deployment configurations objects that are to realize in order to transform the initial configuration to the target configuration. However, taking into account that another research project[9] is currently investigating how to automatically implement a deployment configuration using *Opscode Chef*[10], a clear interface should be agreed on. In this same context, one could investigate if the *Stateless Component Swapping* migration pattern introduced by Fehling et al. in [Fehling et al., 2013] could be applicable. Also, the different servers' properties haven't been taken into account. For example, questions such as "How many CPUs or RAM should a new virtual machine be created with? " have not been addressed.

Finally, this thesis concentrated only on computing migration plans for each Application Component. This means that the migration plan for an application with n components would consist of n sub-migration plans: RTA*would have to be repeated over and over again. The learning version of RTA*, the *Learning Real-Time A** (LRTA*), which has also been introduced by Korf [Korf, 1990] could help improve the search performance over time. A continuation of this thesis could consist of analyzing if LRTA* is applicable in this context.

[9]The results of this research project by Björn Morr will be presented at the Darmstadt University of Applied Science by the end of January 2014.
[10]http://www.opscode.com/chef/

References

[Alliance, 2013] Alliance, C. S. (2013). Cloud Security Alliance Warns Providers of 'The Notorious Nine' Cloud Computing Top Threats in 2013. In *Top Threats Working Group The Notorious Nine Cloud Computing Top Threats in 2013*, page 8. (Cited on pages 3 and 13.)

[Andrews, 1998] Andrews, G. (1998). *The Theory of Partitions*. Cambridge mathematical library. Cambridge University Press. (Cited on page 37.)

[Auer and Dobler, 2000] Auer, D. and Dobler, H. (2000). A model for the migration to object-oriented software development with special emphasis on improvement of acceptance. In *Technology of Object-Oriented Languages and Systems, 2000. TOOLS-Pacific 2000. Proceedings. 37th International Conference on*, pages 132–143. (Cited on page 2.)

[Baezayates et al., 1993] Baezayates, R. A., Culberson, J. C., and Rawlins, G. J. (1993). Searching in the plane. *Information and Computation*, 106(2):234–252. (Cited on page 43.)

[Balev et al., 2013] Balev, S., Baudry, J., Dutot, A., Pigné, Y., and Savin, G. (2013). Graphstream - a dynamic graph library. `http://graphstream-project.org/`. Project hosted by University of Le Havre. Accessed: 2013-10-21. (Cited on page 57.)

[Bayrak et al., 2011] Bayrak, E., Conley, J. P., and Wilkie, S. (2011). The economics of Cloud Computing. Department Of Economics, Vanderbilt University Nashville, `http://www.accessecon.com/pubs/VUECON/vu11-w18.pdf`. (Cited on pages 3 and 20.)

[Bezemer and Zaidman, 2010a] Bezemer, C.-P. and Zaidman, A. (2010a). Challenges of Reengineering into Multi-Tenant SaaS Applications. Software Engineering Research Group, Delft University of Technology, `http://swerl.tudelft.nl/twiki/pub/Main/TechnicalReports/TUD-SERG-2010-012.pdf`. (Cited on pages 1 and 13.)

[Bezemer and Zaidman, 2010b] Bezemer, C.-P. and Zaidman, A. (2010b). Multi-Tenant SaaS Applications: Maintenance Dream or Nightmare? Technical report, Delft University of Technology. (Cited on pages 3, 11, 12 and 13.)

[Biggs et al., 1976] Biggs, N., Lloyd, E., and Wilson, R. (1976). *Graph Theory 1736-1936*. Clarendon Press. (Cited on page 16.)

[BITKOM, 2009] BITKOM (2009). Cloud Computing - Evolution in der Technik, Revolution im Business. http://www.bitkom.org/files/documents/BITKOM-Leitfaden-CloudComputing_Web.pdf. Accessed: 2013-10-05. (Cited on page 7.)

[Blokdijk, 2008] Blokdijk, G. (2008). *Saas 100 Success Secrets: How Companies Successfully Buy, Manage, Host and Deliver Software As a Service (Saas)*. Lulu.com. (Cited on page 3.)

[Bollobás, 1982] Bollobás, B. (1982). *Graph Theory*. Access Online via Elsevier. (Cited on page 16.)

[Bondy and Murty, 1976] Bondy, J. A. and Murty, U. S. R. (1976). *Graph Theory with Applications*, volume 290. Macmillan London. (Not cited.)

[Brunnette and Mogull, 2009] Brunnette, G. and Mogull, R. (2009). Cloud Security Alliance - Security Guidance for Critical Areas of Focus in Cloud Computing V2.1. http://www.cloudsecurityalliance.org/guidance/csaguide.v2.1.pdf. Accessed: 2013-10-11. (Cited on page 8.)

[BSI, 2011] BSI (2011). Bsi - sicherheitsempfehlungen für cloud computing anbieter. https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Mindestanforderungen/Eckpunktepapier-Sicherheitsempfehlungen-CloudComputing-Anbieter.pdf. Accessed: 2013-10-05. (Cited on page 7.)

[Carraro and Chong, 2006] Carraro, G. and Chong, F. (2006). Software as a Service (SaaS): An Enterprise Perspective. http://msdn.microsoft.com/en-us/library/aa905332.aspx. Accessed: 2013-10-06. (Cited on page 20.)

[Cesare and Xiang, 2012] Cesare, S. and Xiang, Y. (2012). *Software Similarity and Classification*. SpringerBriefs in Computer Science. Springer. (Cited on page 60.)

[Chong and Carraro, 2006] Chong, F. and Carraro, G. (2006). Architecture Strategies for Catching the Long Tail. http://msdn.microsoft.com/en-us/library/aa479069.aspx#docume_topic7. (Cited on page 1.)

[Chong et al., 2006] Chong, F., Carraro, G., and Wolter, R. (2006). Multi-Tenant Data Architecture. http://msdn.microsoft.com/en-us/library/aa479086.aspx. (Cited on pages 1, 10, 12 and 13.)

[Christofides, 1975] Christofides, N. (1975). *Graph theory: An algorithmic approach (Computer science and applied mathematics)*. Academic Press, Inc., Orlando, FL, USA. (Cited on page 17.)

[Cormen et al., 2009] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition. (Cited on pages 29, 40, 41 and 42.)

[Corporation, 2013] Corporation, O. (2013). StackOverflowError (Java Platform SE 7). `http://docs.oracle.com/javase/7/docs/api/java/lang/StackOverflowError.html`. Accessed: 2013-10-11. (Cited on page 39.)

[Dasgupta et al., 2006] Dasgupta, S., Papadimitriou, C. H., and Vazirani, U. V. (2006). *Algorithms*. McGraw-Hill Science/Engineering/Math. (Cited on page 40.)

[Desisto et al., 2012] Desisto, R. P., Natis, Y. V., Greene, J., Tang, T. T., Dharmasthira, Y., Lo, T., and Petri, G. (2012). Gartner 2013 Software as a Service Predictions. `http://blogs.gartner.com/robert-desisto/2012/12/27/gartner-2013-software-as-a-service-predictions/`. Accessed: 2013-10-07. (Cited on page 10.)

[Dillon et al., 2010] Dillon, T., Wu, C., and Chang, E. (2010). Cloud Computing: Issues and Challenges. In *Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on*, pages 27–33. (Cited on page 20.)

[Edmonds, 1971] Edmonds, J. (1971). Matroids and the greedy algorithm. *Mathematical Programming*, 1(1):127–136. (Cited on page 41.)

[Even, 2011] Even, S. (2011). *Graph Algorithms*. Cambridge University Press. (Cited on page 17.)

[Everett, 2009] Everett, C. (2009). Cloud computing: A question of trust. *Computer Fraud & Security*, 2009(6):5–7. (Cited on page 13.)

[Fehling et al., 2013] Fehling, C., Leymann, F., Ruehl, S. T., Rudek, M., and Verclas, S. (2013). Service Migration Patterns - Decision Support and Best Practices for the Migration of Existing Service-based Applications to Cloud Environments. In *Proceedings of the 6th IEEE International Conference on Service Oriented Computing and Applications (SOCA).*, pages 1–8. IEEE. (Cited on pages 2 and 72.)

[Felner et al., 2004] Felner, A., Stern, R., Kraus, S., Ben-Yair, A., and Netanyahu, N. S. (2004). Pha*: finding the shortest path with a* in an unknown physical environment. *J. Artif. Intell. Res. (JAIR)*, 21:631–670. (Cited on page 52.)

[Fox et al., 2009] Fox, A., Griffith, R., Joseph, A., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., and Stoica, I. (2009). Above the Clouds: A Berkeley view of Cloud Computing. In *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS*, volume 28. Electrical Engineering and Computer Sciences University of California at Berkeley. (Cited on page 7.)

[Gartner, 2009] Gartner (2009). Gartner Fact Checks the Five Most-Common SaaS Assumptions. http://www.gartner.com/newsroom/id/889713. Accessed: 2013-10-05. (Cited on page 10.)

[Gebauer and Schober, 2006] Gebauer, J. and Schober, F. (2006). Information system flexibility and the cost efficiency of business processes. *Journal of the Association for Information Systems*, 7(3):122–147. (Cited on page 21.)

[Gomes and RyanWilliams, 2005] Gomes, C. P. and RyanWilliams (2005). Approximation Algorithms. In Burke and Kendall, editors, *Introduction to Optimization, Decision Support and Search Methodologies*. Kluwer. (Cited on page 43.)

[Gomes et al., 1998] Gomes, C. P., Selman, B., Kautz, H., et al. (1998). Boosting combinatorial search through randomization. *AAAI/IAAI*, 98:431–437. (Cited on page 43.)

[Guo et al., 2007] Guo, C. J., Sun, W., Huang, Y., Wang, Z. H., and Gao, B. (2007). A Framework for Native Multi-Tenancy Application Development and Management. In *E-Commerce Technology and the 4th IEEE International Conference on Enterprise Computing, E-Commerce, and E-Services, 2007. CEC/EEE 2007. The 9th IEEE International Conference on*, pages 551–558. (Cited on page 13.)

[Harold, 2002] Harold, E. R. (2002). *Processing Xml with Java*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. (Cited on page 55.)

[Institute, 2013] Institute, P. (2013). 2013 cost of data breach study: Global analysis. Technical report, Ponemon Institute LLC. Benchmark research sponsored by Symantec, Independently Conducted by Ponemon Institute LLC. (Cited on page 3.)

[Jiang et al., 2012] Jiang, L., Cao, J., Li, P., and Zhu, Q. (2012). A Mixed Multi-tenancy Data Model and Its Migration Approach for the SaaS Application. In *Services Computing Conference (APSCC), 2012 IEEE Asia-Pacific*, pages 295–300. IEEE. (Cited on pages 3, 20, 21 and 22.)

[Kao et al., 1996] Kao, M.-Y., Reif, J. H., and Tate, S. R. (1996). Searching in an Unknown Environment: An Optimal Randomized Algorithm for the Cow-Path Problem. *Information and Computation*, 131(1):63 – 79. (Cited on pages 42, 43 and 44.)

[Korf, 1987] Korf, R. E. (1987). Real-time heuristic search: First results. In *AAAI*, pages 133–138. (Cited on page 66.)

[Korf, 1990] Korf, R. E. (1990). Real-time heuristic search. *Artificial Intelligence*, 42(Issues 23):189 – 211. Computer Science Department, University of California, Los Angeles, CA 90024-1596, USA. (Cited on pages 45, 46, 48, 49, 50, 65 and 72.)

[Kumar, 2008] Kumar, E. (2008). *Artificial Intelligence*. I.K. International Publishing House Pvt. Limited. (Cited on page 46.)

[Liu et al., 2011] Liu, F., Tong, J., Mao, J., Bohn, R., Messina, J., Badger, L., and Leaf, D. (2011). NIST Cloud Computing Reference Architecture. *NIST special publication*, 500:292. (Cited on pages 7 and 10.)

[Luna Garcia et al., 2012] Luna Garcia, J., Langenberg, R., and Suri, N. (2012). Benchmarking cloud security level agreements using quantitative policy trees. In *Proceedings of the 2012 ACM Workshop on Cloud computing security workshop*, pages 103–112. (Cited on page 20.)

[Luštrek et al., 2006] Luštrek, M., Gams, M., and Bratko, I. (2006). Is real-valued minimax pathological? *Artificial Intelligence*, 170(6–7):620 – 642. (Cited on pages 48 and 66.)

[McCall and van der Meulen, 2011] McCall, T. and van der Meulen, R. (2011). Gartner Says Worldwide Software as a Service Revenue Is Forecast to Grow 21 Percent in 2011. (Cited on page 10.)

[Mell and Grance, 2011] Mell, P. and Grance, T. (2011). The nist definition of cloud computing. `http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf`. (Cited on pages 7, 8 and 9.)

[Mietzner et al., 2009] Mietzner, R., Metzger, A., Leymann, F., and Pohl, K. (2009). Variability modeling to support customization and deployment of multi-tenant-aware software as a service applications. In *Proceedings of the 2009 ICSE Workshop on Principles of Engineering Service Oriented Systems*, PESOS '09, pages 18–25, Washington, DC, USA. IEEE Computer Society. (Cited on page 21.)

[Motwani and Raghavan, 1995] Motwani, R. and Raghavan, P. (1995). *Randomized Algorithms*. Cambridge University Press. (Cited on page 43.)

[Nau, 1982] Nau, D. S. (1982). An investigation of the causes of pathology in games. *Artificial Intelligence*, 19(3):257 – 278. (Cited on page 66.)

[Neo Technology, Inc., 2013] Neo Technology, Inc. (2013). Neo4j - the world's leading graph database. `http://www.neo4j.org/`. Accessed: 2013-11-14. (Cited on page 40.)

[Nievergelt, 2000] Nievergelt, J. (2000). Exhaustive search, combinatorial optimization and enumeration: Exploring the potential of raw computing power. In *In SOFSEM 2000, number 1963 in LNCS*, pages 18–35. Springer. (Cited on page 34.)

[Ort and Mehta, 2003] Ort, E. and Mehta, B. (2003). Java architecture for xml binding (jaxb). http://www.oracle.com/technetwork/articles/javase/index-140168.html. Oracle Corporation. Accessed: 2013-10-21. (Cited on page 55.)

[Parkhill, 1966] Parkhill, D. (1966). *The Challenge of the Computer Utility*. Number p. 246 in The Challenge of the Computer Utility. Addison-Wesley Publishing Company. (Cited on page 6.)

[Pettey and van der Meulen, 2012a] Pettey, C. and van der Meulen, R. (2012a). Gartner Says Worldwide Cloud Services Market to Surpass $109 Billion in 2012. Gartner, http://www.gartner.com/newsroom/id/2163616. (Cited on pages 3 and 10.)

[Pettey and van der Meulen, 2012b] Pettey, C. and van der Meulen, R. (2012b). Gartner Says Worldwide Software-as-a-Service Revenue to Reach $14.5 Billion in 2012. Gartner, http://www.gartner.com/newsroom/id/1963815. (Cited on pages 3 and 10.)

[Piazza, 2009] Piazza, J. (2009). Computing Migration Strategies. Savvis Inc., http://www.savvis.com/en-us/info_center/documents/hos-20090513-external-whitepaper-computingmigrationstrategies.pdf. (Cited on page 2.)

[Plummer, 2009] Plummer, D. (2009). Experts Define Cloud Computing: Can we get a Little Definition in our definitions? http://blogs.gartner.com/daryl_plummer/2009/01/27/experts-define-cloud-computing-can-we-get-a-little-definition-in-our-definitions/. Accessed: 2013-10-05. (Cited on page 7.)

[Pohl et al., 2005] Pohl, K., Böckle, G., and Linden, F. J. v. d. (2005). *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA. (Cited on page 21.)

[Poole and Mackworth, 2010] Poole, D. and Mackworth, A. (2010). *Artificial Intelligence: Foundations of Computational Agents*. Artificial Intelligence: Foundations of Computational Agents. Cambridge University Press. (Cited on page 40.)

[Reinhardt, 2013] Reinhardt, M. (2013). Konzeption und Realisierung einer Mixed-Tenancy-fähigen Plattform zur Bereitstellung von SaaS-Applikationen. Master's thesis, Darmstadt University of Applied Sciences. (Cited on pages 2, 3, 15, 22, 54, 55 and 70.)

[Richardson and Ruby, 2007] Richardson, L. and Ruby, S. (2007). *Restful Web Services*. O'Reilly, first edition. (Cited on page 22.)

[Rosetta Code, 2013] Rosetta Code (2013). Levenshtein distance. http://rosettacode.org/wiki/Levenshtein_distance. Content available under GNU Free Documentation License 1.2. Accessed: 2013-10-22. (Cited on page XIV.)

[Ruehl et al., 2012] Ruehl, S., Andelfinger, U., Rausch, A., and Verclas, S. (2012). Toward Realization of Deployment Variability for Software-as-a-Service Applications. In *2012 IEEE 5th International Conference on Cloud Computing (CLOUD)*, pages 622–629. (Cited on pages 1, 3, 13, 22, 24, 54 and 70.)

[Ruehl, 2014] Ruehl, S. T. (2014). *Between Privacy Threats and Exploitation of Economies of Scale - Mixed-tenancy: a hybrid Approach between Single and Multi-tenancy*. PhD thesis, Clausthal University of Technology. (Cited on pages 15 and 16.)

[Ruehl and Andelfinger, 2011] Ruehl, S. T. and Andelfinger, U. (2011). Applying Software Product Lines to create customizable Software-as-a-Service Applications. In *Proceedings of the 15th International Software Product Line Conference, Volume 2*, SPLC '11, pages 16:1–16:4, New York, NY, USA. ACM. (Cited on pages 21, 22 and 70.)

[Ruehl et al., 2013] Ruehl, S. T., Wache, H., and Verclas, S. A. W. (2013). Capturing Customers' Requirements towards Mixed-tenancy Deployments of SaaS-Applications. In *2013 IEEE Sixth International Conference on Cloud Computing*, pages 462–469. IEEE. (Cited on pages 14, 22, 24, 54, 58, 70 and X.)

[Rupprecht, 2013] Rupprecht, M. (2013). Einführung von Mixed-Tenancy in eine existierende Enterprise SaaS Applikation durch automatisierte Portierung. Bachelor's Thesis, Darmstadt University of Applied Sciences. (Cited on page 22.)

[Russell and Norvig, 2010] Russell, S. and Norvig, P. (2010). *Artificial Intelligence: A Modern Approach*. Prentice Hall series in artificial intelligence. Prentice Hall. (Cited on pages 18, 19, 28 and 29.)

[Sengupta and Roychoudhury, 2011] Sengupta, B. and Roychoudhury, A. (2011). Engineering Multi-Tenant Software-as-a-Service systems. In *Proceedings of the 3rd International Workshop on Principles of Engineering Service-Oriented Systems*, PESOS '11, pages 15–21, New York, NY, USA. ACM. (Cited on pages 1 and 20.)

[Stentz, 1994] Stentz, A. (1994). Optimal and Efficient Path Planning for Partially-Known Environments. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA '94)*, volume 4, pages 3310 – 3317. (Cited on page 52.)

[Stentz, 1995] Stentz, A. (1995). The focussed dˆ* algorithm for real-time replanning. In *IJCAI*, volume 95, pages 1652–1659. (Cited on page 52.)

[Sun et al., 2008] Sun, W., Zhang, X., Guo, C. J., Sun, P., and Su, H. (2008). Software as a Service: Configuration and Customization Perspectives. In *Congress on Services Part II, 2008. SERVICES-2. IEEE*, pages 18–25. (Cited on page 20.)

[Tavcar, 2012] Tavcar, A. (2012). Analysis of a single-agent search. *Informatica: An International Journal of Computing and Informatics*, 36(2):177–183. (Cited on pages 48 and 66.)

[Thomson et al., 2008] Thomson, B., Bruckner, J., and Bruckner, A. (2008). *Elementary Real Analysis*. Number v. 1 in Elementary Real Analysis. www.classicalrealanalysis.com. (Cited on page 46.)

[Transvive, 2011] Transvive (2011). Transvive - Mainframe Migration Strategy. Transvive Inc., `http://www.platformmodernization.org/transvive/Lists/ResearchPapers/Attachments/1/Transvive-MainframeMigrationStrategy-WP.pdf`. (Cited on page 2.)

[Verma, 2011] Verma, G. (2011). Software as a Service. In *Pragyaan: Journal of Information Technology*, volume 9 of *1*, pages 32–37. Institute of Management Studies Dehradun. (Cited on pages 10 and 13.)

[Weisstein, 2013a] Weisstein, E. W. (2013a). Graph. From MathWorld - A Wolfram Web Resource. `http://mathworld.wolfram.com/Graph.html`. Accessed: 2013-10-07. (Cited on pages 17 and 18.)

[Weisstein, 2013b] Weisstein, E. W. (2013b). Partition. From MathWorld - A Wolfram Web Resource `http://mathworld.wolfram.com/Partition.html`. Accessed: 2013-11-16. (Cited on page 37.)

[Zhang et al., 2012] Zhang, Y., Juels, A., Reiter, M. K., and Ristenpart, T. (2012). Cross-VM side channels and their use to extract private keys. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 305–316, New York, NY, USA. ACM. (Cited on page 3.)

Appendices

A

Deployment Models

The following has been extracted from [Ruehl et al., 2013].

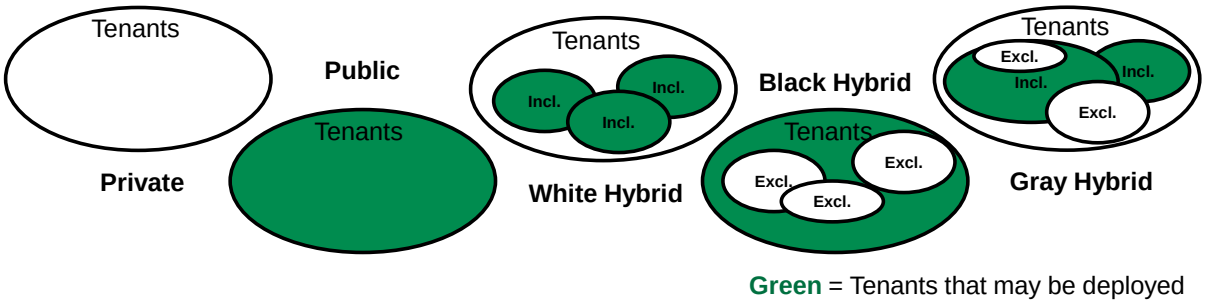


Figure A.1.: Deployment Models [Ruehl et al., 2013]

Private

Following this deployment model, an Deployment Unit is deployed specifically for a single tenant. This specific unit is not shared by multiple tenants. If applied to the AC Deployment Level, the entire application stops being a pure MT application since this deployed AC does not need to support Mixed Tenancy (MT) anymore. In fact, it would be possible that a customer demands that all ACs are deployed following the private model. This would lead to a single-tenancy deployment of a MT application. If another model would be chosen for lower Deployment Levels, it would still be possible for the operator to do some resource usage optimization.

Public

If this deployment model is chosen, a free number of tenants will share the same instance. The tenants have no influence on with whom they are deployed. As this is the easiest way for the operator to deploy the application, it usually is also the cheapest for the customer. For customers, this may also be the right way for ACs that do not handle sensitive data.

White Hybrid

This deployment model allows customers to specify choose with which specific tenants or groups of tenants they feel comfortable to share units. An application of this might be that a company demands that individual departments (belonging

to the same company group) share an instance, but any external tenants are not allowed to use the same instance. Furthermore, this deployment model allows to manage collaboration if applied to the AC Deployment Level. It can be used to specify which specific customers want to share the same AC instance that allows collaboration, for example by deactivating isolation of a MT component. However, this is not further elaborated in this work.

Black Hybrid

In this deployment model customers can specify with which other tenants they do not want to be deployed. A real customer demand that can be fulfilled using this model is that a customer demands not to be deployed with competitors (without specifically specifying who they are) or they demand not to be deployed with another specific customer.

Gray Hybrid

This deployment model allows customers to do both, specify with whom they want to be deployed and with whom they do not wish to be deployed. This model allows customers the most liberty to describe their constraints towards the deployment.

The model allows customers to specify, for example, that they want to share a unit only with tenants from a specific geographic region (e.g. Asia) but not with tenants from a particular industry (e.g. the ones they operate in).

B

Deployment Configuration Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified">
  <xs:element name="DeploymentConfiguration">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Server" maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="ACI" maxOccurs="unbounded">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="Tenant" maxOccurs="unbounded">
                      <xs:complexType>
                        <xs:sequence />
                        <xs:attribute name="id" type="xs:integer" />
                      </xs:complexType>
                    </xs:element>
                  </xs:sequence>
                  <xs:attribute name="ac" type="xs:string" />
                </xs:complexType>
              </xs:element>
            </xs:sequence>
            <xs:attribute name="ramsize" type="xs:integer" />
            <xs:attribute name="cpucores" type="xs:integer" />
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Listing B.1: Deployment Configuration Schema

C

Deployment Configuration: A concrete example

```xml
<?xml version="1.0" encoding="UTF-8"?>
<DeploymentConfiguration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="DeploymentConfiguration.xsd">
  <Server ramsize="1024" cpucores="2">
    <ACI ac="a">
      <Tenant id="0" />
    </ACI>
    <ACI ac="b">
      <Tenant id="0" />
      <Tenant id="4" />
    </ACI>
  </Server>
  <Server ramsize="512" cpucores="1">
    <ACI ac="a">
      <Tenant id="2" />
    </ACI>
    <ACI ac="b">
      <Tenant id="2" />
      <Tenant id="1" />
    </ACI>
  </Server>
</DeploymentConfiguration>
```

Listing C.1: Example of a Deployment Configuration

D

An Implementation of the Levenhstein Distance

```
String s1 = sourceConfiguration.toString().toLowerCase();
String s2 = targetConfiguration.toString().toLowerCase();
int[] costs = new int[s2.length() + 1];
for (int i = 0; i <= s1.length(); i++) {
  int lastValue = i;
  for (int j = 0; j <= s2.length(); j++) {
    if (i == 0) {
      costs[j] = j;
    } else {
      if (j > 0) {
        int newValue = costs[j - 1];
        if (s1.charAt(i - 1) != s2.charAt(j - 1)) {
          newValue = Math.min(Math.min(newValue, lastValue), costs[j])
   + 1;
        }
        costs[j - 1] = lastValue; lastValue = newValue;
      }
    }
  }
  if (i > 0) { costs[s2.length()] = lastValue;}
}
return costs[s2.length()];
```

Listing D.1: Levenhstein Distance [Rosetta Code, 2013]