Model Based Control for Multi-Cloud Applications

BY

MARCO MIGLIERINA
B.S., Politecnico di Milano, Milan, Italy, 2010
M.S., Politecnico di Milano, Milan, Italy, 2012

THESIS
Submitted as partial fulfillment of the requirements for the degree of Master of Science in Computer Science in the Graduate College of the University of Illinois at Chicago, 2013

Chicago, Illinois

Defense Committee:

Ugo A. Buy, Chair and Advisor
Rigel Gjomemo
Marco D. Santambrogio, Politecnico di Milano
To my Family and Eleonora
ACKNOWLEDGMENTS

I first would like to thank Prof. Elisabetta Di Nitto and Prof. Danilo Ardagna as my advisor and co-advisor respectively for the thesis at Politecnico di Milano.

I would also like to thank Prof. Carlo Ghezzi, Prof. Alberto Leva and Dr. Antonio Filieri for valuable discussions.

MM
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>INTRODUCTION</td>
</tr>
<tr>
<td>1.1</td>
<td>Thesis Objective</td>
</tr>
<tr>
<td>1.2</td>
<td>Structure of the Thesis</td>
</tr>
<tr>
<td>2</td>
<td>BACKGROUND</td>
</tr>
<tr>
<td>2.1</td>
<td>Availability</td>
</tr>
<tr>
<td>2.2</td>
<td>The Discrete Time Markov Chain with Reward</td>
</tr>
<tr>
<td>2.3</td>
<td>Control Theory</td>
</tr>
<tr>
<td>3</td>
<td>STATE OF THE ART</td>
</tr>
<tr>
<td>3.1</td>
<td>Cloud Computing Overview</td>
</tr>
<tr>
<td>3.1.1</td>
<td>Portability</td>
</tr>
<tr>
<td>3.2</td>
<td>Model Based Control</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Self-Adaptive Software Meets Control Theory</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Availability Model for Multi-Cloud Applications</td>
</tr>
<tr>
<td>3.2.3</td>
<td>A Multi-Cloud Simulator</td>
</tr>
<tr>
<td>3.2.3.1</td>
<td>Workload</td>
</tr>
<tr>
<td>3.2.3.2</td>
<td>Infrastructural parameters</td>
</tr>
<tr>
<td>3.2.3.3</td>
<td>Simulation parameters</td>
</tr>
<tr>
<td>3.2.3.4</td>
<td>Simulation Engine</td>
</tr>
<tr>
<td>3.3</td>
<td>Final Considerations on the State of the Art</td>
</tr>
<tr>
<td>4</td>
<td>THE CONTROLLER FOR MULTI-CLOUD APPLICATIONS SELF-ADAPTATION</td>
</tr>
<tr>
<td>4.1</td>
<td>Overview of the approach</td>
</tr>
<tr>
<td>4.2</td>
<td>The autoscaling controller</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Objective</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Monitoring</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Control</td>
</tr>
<tr>
<td>4.2.4</td>
<td>Timing</td>
</tr>
<tr>
<td>4.3</td>
<td>The load balancer controller</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Objective</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Monitoring</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Control</td>
</tr>
<tr>
<td>4.3.4</td>
<td>Timing</td>
</tr>
<tr>
<td>5</td>
<td>EVALUATION</td>
</tr>
</tbody>
</table>
# TABLE OF CONTENTS (Continued)

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>A Web System Scenario</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Scenario 1</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Scenario 2</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Scenario 3</td>
</tr>
<tr>
<td>5.2</td>
<td>A Multi-Region Scenario</td>
</tr>
<tr>
<td>5.3</td>
<td>A Smart City Scenario</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Application Model</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Filtering Part</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Process Model</td>
</tr>
<tr>
<td>5.3.3.1</td>
<td>Reasoning</td>
</tr>
<tr>
<td>5.4</td>
<td>Results analysis</td>
</tr>
<tr>
<td>6</td>
<td>CONCLUSIONS</td>
</tr>
<tr>
<td></td>
<td>CITED LITERATURE</td>
</tr>
<tr>
<td></td>
<td>VITA</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>TABLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>52</td>
</tr>
<tr>
<td>II</td>
<td>55</td>
</tr>
<tr>
<td>III</td>
<td>59</td>
</tr>
<tr>
<td>IV</td>
<td>59</td>
</tr>
<tr>
<td>V</td>
<td>74</td>
</tr>
<tr>
<td>VI</td>
<td>84</td>
</tr>
</tbody>
</table>

I. SIMULATION PARAMETERS OF THE WEB SYSTEM SCENARIOS
II. CONTROLLED VS NON-CONTROLLED RESULTS OF THE WEB SYSTEM SCENARIO 1
III. CONTROLLED VS NON-CONTROLLED RESULTS OF THE WEB SYSTEM SCENARIO 2
IV. SIMULATION PARAMETERS OF THE MULTI-REGION SCENARIO
V. SIMULATION PARAMETERS
VI. SMART CITY SCENARIO RESULTS
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>DESCRIPTION</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Concept of the feedback loop to control the dynamic behavior of the system</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>An image filtering service, a representative example</td>
<td>17</td>
</tr>
<tr>
<td>3</td>
<td>DTMC model for the image filtering service</td>
<td>18</td>
</tr>
<tr>
<td>4</td>
<td>Reliability of the system: set point (dashed) and achieved value (solid)</td>
<td>21</td>
</tr>
<tr>
<td>5</td>
<td>Control variables of the system: $c_{1a}$ dashed, $c_{1b}$ solid and $c_5$ dashed dotted</td>
<td>22</td>
</tr>
<tr>
<td>6</td>
<td>Ontology of nodes in the extended DTMC model</td>
<td>23</td>
</tr>
<tr>
<td>7</td>
<td>Model of a two Clouds example</td>
<td>25</td>
</tr>
<tr>
<td>8</td>
<td>Bimodal distribution of requests</td>
<td>27</td>
</tr>
<tr>
<td>9</td>
<td>Overview of the approach</td>
<td>35</td>
</tr>
<tr>
<td>10</td>
<td>Model of the Web System use case</td>
<td>49</td>
</tr>
<tr>
<td>11</td>
<td>DTMC model representation of the Web System use case. Green nodes represent autoscaling groups, red nodes represent failure states</td>
<td>50</td>
</tr>
<tr>
<td>12</td>
<td>Availability of the system of the Web System Scenario 1</td>
<td>53</td>
</tr>
<tr>
<td>13</td>
<td>Number of running VMs of the Web System Scenario 1</td>
<td>54</td>
</tr>
<tr>
<td>14</td>
<td>Maximum service rate of VMs of the Web System Scenario 2</td>
<td>56</td>
</tr>
<tr>
<td>15</td>
<td>Running Machines of the Web System Scenario 2</td>
<td>57</td>
</tr>
<tr>
<td>16</td>
<td>Cpu Usage of the Web System Scenario 2</td>
<td>58</td>
</tr>
<tr>
<td>17</td>
<td>Clouds’ availabilities of the Web System Scenario 3</td>
<td>60</td>
</tr>
<tr>
<td>18</td>
<td>System Availability of the Web System Scenario 3</td>
<td>61</td>
</tr>
<tr>
<td>19</td>
<td>Control variable values of the Web System Scenario 3</td>
<td>62</td>
</tr>
<tr>
<td>20</td>
<td>Running Machines of the Web System Scenario 3</td>
<td>63</td>
</tr>
<tr>
<td>21</td>
<td>Model for the Multi-Region Scenario</td>
<td>64</td>
</tr>
<tr>
<td>22</td>
<td>DTMC model for the Multi-Region Scenario</td>
<td>65</td>
</tr>
<tr>
<td>23</td>
<td>System Availability Setpoint for the Multi-Region Scenario</td>
<td>66</td>
</tr>
<tr>
<td>24</td>
<td>Clouds’ Availabilities of the Multi-Region Scenario</td>
<td>67</td>
</tr>
<tr>
<td>25</td>
<td>System Availability of the Multi-Region Scenario</td>
<td>68</td>
</tr>
<tr>
<td>26</td>
<td>Number of running machines of the Multi-Region Scenario</td>
<td>69</td>
</tr>
<tr>
<td>27</td>
<td>Control variables values</td>
<td>70</td>
</tr>
<tr>
<td>28</td>
<td>Structure of the Smart City Management Application</td>
<td>71</td>
</tr>
<tr>
<td>29</td>
<td>DTMC model of the filtering part of the smart city usecase</td>
<td>72</td>
</tr>
<tr>
<td>30</td>
<td>Model of the filtering module of the smart city emergency system application</td>
<td>73</td>
</tr>
<tr>
<td>31</td>
<td>Simulated arrival rate during 24h period at the Smart City Emergency System Application</td>
<td>75</td>
</tr>
<tr>
<td>FIGURE</td>
<td>Description</td>
<td>PAGE</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>32</td>
<td>Cloud 4 service rate during 24h period at the Smart City Emergency System Application</td>
<td>76</td>
</tr>
<tr>
<td>33</td>
<td>Cloud 1 availability during 24h period at the Smart City Emergency System Application</td>
<td>77</td>
</tr>
<tr>
<td>34</td>
<td>Cloud 3 availability during 24h period at the Smart City Emergency System Application</td>
<td>78</td>
</tr>
<tr>
<td>35</td>
<td>System Availability using exclusively Cloud 1 for the Smart City Emergency System Application</td>
<td>79</td>
</tr>
<tr>
<td>36</td>
<td>System Availability using exclusively Cloud 2 for the Smart City Emergency System Application</td>
<td>80</td>
</tr>
<tr>
<td>37</td>
<td>System Availability using exclusively Cloud 3 for the Smart City Emergency System Application</td>
<td>81</td>
</tr>
<tr>
<td>38</td>
<td>System Availability using exclusively Cloud 4 for the Smart City Emergency System Application</td>
<td>82</td>
</tr>
<tr>
<td>39</td>
<td>System Availability of the Smart City Emergency System with load balancer enabled</td>
<td>83</td>
</tr>
<tr>
<td>40</td>
<td>System Availability of the Smart City Emergency System with load balancer enabled</td>
<td>84</td>
</tr>
<tr>
<td>41</td>
<td>Availability of the controlled system with set point to 5-nines</td>
<td>85</td>
</tr>
</tbody>
</table>
# LIST OF ABBREVIATIONS

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACS</td>
<td>Cloud Security Alliance</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>AR</td>
<td>Arrival Rate</td>
</tr>
<tr>
<td>DTMC</td>
<td>Discrete Time Markov Chain</td>
</tr>
<tr>
<td>EC2</td>
<td>Elastic Compute Cloud</td>
</tr>
<tr>
<td>IT</td>
<td>Information Technology</td>
</tr>
<tr>
<td>LTN</td>
<td>Limited Throughput Node</td>
</tr>
<tr>
<td>MODAClouds</td>
<td>MOdel-Driven Approach for design and execution of applications on multiple Clouds</td>
</tr>
<tr>
<td>MTTF</td>
<td>Mean Time To Fail</td>
</tr>
<tr>
<td>MTTR</td>
<td>Mean Time To Repair</td>
</tr>
<tr>
<td>NIST</td>
<td>National Institute of Standards and Technology</td>
</tr>
<tr>
<td>QoS</td>
<td>Quality of Service</td>
</tr>
<tr>
<td>SLA</td>
<td>Service Level Agreement</td>
</tr>
<tr>
<td>SME</td>
<td>Small and Medium Enterprises</td>
</tr>
<tr>
<td>SOA</td>
<td>Service Oriented Architecture</td>
</tr>
</tbody>
</table>
**LIST OF ABBREVIATIONS** (Continued)

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SR</td>
<td>Maximum Service Rate</td>
</tr>
<tr>
<td>UIC</td>
<td>University of Illinois at Chicago</td>
</tr>
<tr>
<td>UTN</td>
<td>Unlimited Throughput Node</td>
</tr>
<tr>
<td>VM</td>
<td>Virtual Machine</td>
</tr>
</tbody>
</table>
SUMMARY

Cloud Computing is assuming a relevant role in the world of web applications and web services. The main advantage Cloud technologies are bringing is the concept of utility computing. SME do not need to cope with large up-front investments in hardware resources, since they can just acquire resources on a pay-per-use basis from any of the several Cloud providers coming into market in the recent years. Moreover, these technologies allow the possibility of completely delegating to the Cloud Provider intensive tasks, such as the management and the maintenance of the Cloud infrastructure.

However, the usage of Cloud systems can lead to relevant issues, which mainly derive from the lack of technology standards and from the intrinsic characteristics of such geographically distributed systems. For example, we can mention the lock-in effect related to the portability of Cloud applications, the problem of data location and data security, the lack of interoperability between different Cloud systems, the problem of performance and cost estimation.

What is currently discouraging companies from moving to a Cloud environment is the lack of exclusive access to the IT infrastructure. The quality of service is not under companies direct control. It is Cloud providers responsibility to guarantee a certain level of performances and availability. Since hosting hardware resources are shared among Cloud users, the variability of this quality is very high and this cannot be accepted by several companies, especially those dealing with critical applications.
SUMMARY (Continued)

In order to cope with this instability of quality of service, the only solution seems to be deployment of the same application on multiple Clouds and traffic redirection where the performances are best at a given time.

In this thesis we propose a controller able to guarantee the desired availability by simultaneously managing several instances of the application on multiple Cloud providers. Controller decisions are based on a model which is able to describe the Multi-Cloud application. The model is kept alive at runtime, by means of continuous parameter estimation. The controller iteratively infers from the updated model if requirements are satisfied and takes decisions on both workload distribution among the various Cloud providers and virtual machines scaling, minimizing costs.

The approach was tested in simulated environments against common usage scenarios. Evaluation shows that our controller is capable of minimizing the cost of running the application while respecting availability requirements.
CHAPTER 1

INTRODUCTION

Cloud Computing is a new paradigm that is becoming very popular in the last few years. The main innovation Cloud technologies is bringing is the concept of utility computing, that is, the possibility of easily acquiring hardware resources on a pay-per-use basis through the Internet and then releasing it when not needed anymore. Several SMEs are deciding to migrate their entire IT, or part of it, to the Cloud for both economical and flexibility reasons. Even developers of researchers requiring huge computational power for their experiments, maybe for really short periods of time, they are now extremely facilitated both from the economical and technical point of view.

However, the main drawback of completely relying on a Cloud solution, today, is the lack of exclusive access to the IT infrastructure. Some companies, especially those whose core business is strictly dependent on the quality of service delivered to theirs customers, are still reluctant on moving to an environment where resources are shared and the service may vary unpredictably.

A possible solution to this problem is to deploy the same application on multiple Clouds (i.e., offered by different providers), distributing the workload among them, so to continuously guarantee the average QoS requirements, and dynamically allocate resources, satisfying the portion of traffic delivered to each Cloud. Such applications, which are able to run on different Cloud providers, are called multi-Cloud applications.
At present, there are many providers that offer Cloud services and, since this is a quite a new and profitable market, more providers are appearing. Each provider offers specific APIs and programming / design paradigms. Thus, developing an application written for one provider, for a different one, would, in many cases, imply re-writing part of the code, dealing with large databases syncing problems and manually re-deploying applications.

Our thesis has been developed in the context of the MODAClouds\(^1\) (MOdel-Driven Approach for design and execution of applications on multiple Clouds) project which is an European community project whose main goal is “to provide methods, a decision support system, an open source IDE and run-time environment for the high-level design, early prototyping, semi-automatic code generation, and automatic deployment of applications on multi-Clouds with guaranteed QoS”.

In\(^{[1]}\) MODAClouds partners list some of the main challenges system developers and operators have to face:

1. **Vendor Lock-in.** The risk of technological lock-in is a major concern for Cloud customers\(^{[2]}\).

   Thus, portability of applications and data between Clouds, as well as reversibility (moving applications and data from Cloud to non-Cloud environments) should be addressed.

2. **Risk Management.** There are several concerns when selecting a Cloud technology such as payment models, security, legal and contractual, quality and integration with the enterprises architecture and culture. Thus, proper tools to support such choice could

\(^{1}\)http://www.modaclouds.eu
be beneficial and limit serious financial consequences. However, while risk management has been well coded for traditional IT systems\cite{9}, at present only embryonic tools and decision support methods exist to support selecting and binding to a specific target Cloud, or taking a decision to move from Cloud to Cloud in case requirements or services change. Late binding\cite{4} to a specific target Cloud can decrease project failure risks, but at the moment it is not supported neither at the design nor at the run-time level.

3. Quality Assurance. Cloud performance can vary at any point in time. Elasticity may not ramp at desired speeds. Unavailability problems exist even when 99.9% up-time is advertised (\textit{e.g.}, Amazon EC2 and Microsoft Office 365 outages in 2011). Given the criticality of many business applications, analytical techniques are needed to predict QoS and to reason on software systems properties at design-time, but also run-time mechanisms and policies able to provide end-to-end quality.

In this thesis we decided to focus only on the last challenge, while not considering any portability issue or any risk analysis.

1.1 Thesis Objective

The objective of this thesis is to contribute to the development of self-adaptive software systems in the context of multi-Cloud applications, focusing our attention on availability requirements and cost minimization.

To reach this goal, we exploit a model to describe availability requirements of Multi-Cloud applications defined in\cite{5} by another partner of the European project, and we define a two layer controller able to manage both in-Cloud configuration policies and traffic routing through
different Cloud providers, keeping the model alive at run-time. The controller’s objective is to guarantee high availability, while minimizing costs. In\cite{5}, the author extended the already existing integrated modeling environment Palladio Bench to model multi-Cloud applications and implemented a tool to create simulated environments to test decisions taken at design time in different scenarios. In this thesis we extended this simulation tool so to keep the model coherent at runtime and implementing our controller to dynamically adapt the system and cope with availability requirements, minimizing costs.

We evaluated our approach using the simulation tool through three different use cases: a web system scenario with two single-region Clouds, a multi-region scenario, and a smart city scenario.

Our work starts from the assumption that the application is already able to migrate from Cloud provider to Cloud provider. From what we already said, this is a pretty relevant assumption but as we will see in Section 3.1.1 there are many active projects dealing with this problem, besides MODAClouds.

The proposed solution was also presented in\cite{6} and the paper was accepted for a full presentation in the MiSE Workshop at ICSE’13.

1.2 Structure of the Thesis

Chapter 2 gives some background definitions of the theoretical aspects required for this work. Chapter 3 gives an overview of the major Cloud providers (Section 3.1), showing some of the similarities and differences among Cloud providers, underlining the possibility and the challenges, especially in terms of availability and portability. Section 3.2 introduces the models
that are going to be used in our approach and the starting point for the self-adaptive approach by means of extensive descriptions of two works ([5,7]).

Chapter 4 shows the innovative contribution of this thesis. It first presents an overview of the approach we used. Then it introduces our extension to the controller in order to cope with a multi-Cloud environment and to perform adaptation on both in-Cloud scalability and Clouds orchestration.

Chapter 5 introduces the three use cases that have been tested for the evaluation of the control approach. The first is based on a 4 hours simulation of a two tires application that makes use of two Cloud providers in a single-region scenario, showing the peculiarities and the behavior of our approach on a simple case. The second is based on a 6 hour simulation of the same application on a multi-region scenario. The third describes a much more complex application that controls the emergency response system of a smart city. These use cases are introduced by stating their requirements and their simulated workload, the environmental conditions in which they run and their architecture. These applications are simulated and the results of the availability and costs obtained by the proposed control system are discussed.

Chapter 6 summarizes this thesis contribution and provides an overall evaluation of our approach, pointing out some future work that we consider worth to be investigated.
CHAPTER 2

BACKGROUND

In this chapter we are going to focus on some concepts that are required in order to have a common knowledge and vocabulary in this thesis. In Section 2.1 we will define availability, which is the non-functional requirement which we will focus on in this work. Section 2.2 describe the mathematical formalism which support our modeling approach. Finally, Section 2.3 will give some basis of control theory.

2.1 Availability

Availability is a non-functional requirement of major concern, especially when dealing with critical applications, since it measures the probability that a system is in a functioning condition at a given time. It can be measured as \( \frac{\text{uptime}}{\text{uptime} + \text{downtime}} \), or else, identically, as \( \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}} \).

When dealing with services, however, the developer is not aware of the entire history of the components he is relying upon. Therefore, availability is often equalized to the reliability on-demand, which expresses the probability that a certain operation is completed successfully and is usually estimated by computing the number of successful requests over the total number of requests arriving at the system over a time window.

Availability in the Cloud, as we will see in Section 3.1, is totally delegated to Cloud providers, which are usually promising Software Level Agreements (SLAs) that are seldom satisfied as can
be seen in the history of Cloud failures (e.g., Amazon EC2\textsuperscript{1} and Microsoft Office 365\textsuperscript{2} outages in 2011).

2.2 The Discrete Time Markov Chain with Reward

Discrete Time Markov Chain (DTMC) is a mathematical model, representing states and transactions between states with attached probabilities, mostly used to describe stochastic processes. Once a random process is described by a DTMC, model checking tools such as PRISM\textsuperscript{3} can reason about the satisfaction of certain properties. DTMCs can model any stochastic process satisfying the Markov property, \textit{i.e.}, the future states only depend on the current state and not from the sequence of previous states. The process is characterized by a set of states and a transition matrix which represent the probability of moving from one state to another. One of the properties that may be verified is the reachability, which is the probability of reaching a certain state from another.

This formalism may be used to represents service invocation that may fail with a certain probability as in\textsuperscript{[7]}. In addition, if invoking a certain service has a cost that we want to consider, a cost may be attached to each state, or symmetrically a reward. This extension to the model

\textsuperscript{1}http://www.crn.com/news/cloud/229402004/amazon-ec2-goes-dark-in-morning-cloud-outage.htm


\textsuperscript{3}http://www.prismmodelchecker.org
Figure 1: Concept of the feedback loop to control the dynamic behavior of the system. Source: [http://en.wikipedia.org/wiki/Control_theory](http://en.wikipedia.org/wiki/Control_theory)

is called Discrete Markov Chain with Reward and it is the model that we are going to use in this thesis to describe cloud services with the cost implied in using a certain service.

## 2.3 Control Theory

Control theory is an interdisciplinary branch of engineering and mathematics that deals with the behavior of dynamical systems with inputs. The external input of a system is called the reference. When one or more output variables of a system need to follow a certain reference over time, a controller manipulates the inputs of a system to obtain the desired effect at the output of the system. Controllers can be of two kinds. *Open-loop* controllers take decisions based only on the current state of the system and its model. *Closed-loop* controllers, on the other hand, consider also feedback, *i.e.*, the output of the system, measured by some sensor. The former has an obvious limitation in not being aware of how the system is actually behaving to the inputted data. Errors or changes unpredicted when building the model may lead to completely different results from what expected. The concept of the feedback loop is shown in Figure 1. The main advantages of closed-loop over open-loop controllers are disturbance rejection and guaranteed performance even if the model does not perfectly fit the real system.
CHAPTER 3

STATE OF THE ART

In this chapter we will go through the state of the art both from the technological point of view, with reference to Cloud commercial offers, and from the point of view of ongoing research on software development, modeling, automation and QoS requirements satisfaction on the Cloud. Section 3.1 presents the cloud computing environment by showing some of its complexity and introducing some of the main features that makes it so attractive. Section 3.2 presents previous works used as starting point for this work and tools that were then extended to test our approach. In Section 3.3 we will give some final considerations on the state of the art.

3.1 Cloud Computing Overview

In 1961, John McCarthy was the first to suggest the idea of selling computing like a utility, such as water and electricity, during a speech at MIT. Even though it looked like a brilliant idea, the technology was not ready yet. The idea behind this certainly looks back to old mainframes, where the expensive and gigantic hardware was shared among institutions and organizations, who could access the computing capacity by means of dumb terminals. What really led McCarthy’s idea to what is now called Cloud Computing is certainly the advent of virtualization, which really made the management and partitioning of resources much easier, and the great improvement in Internet connections offered to all users. One of the first and
most important Cloud providers is Amazon, which, after realizing he was only using 10% of its computing capacity on average, decided to provide Cloud computing to external customer and launched Amazon Web Services (AWS) in 2006. Since then, a huge variety of heterogeneous Cloud services entered the scene, with still lack of standards, offering storage, infrastructures, platforms and many other resources at different level of abstraction, using different payment models and different ways of accessing the service.

Among this multiplicity of actors and offers, cloud computing services are usually classified into the following three categories:

- **Infrastructure-as-a-Service (IaaS):** the provider offers a complete virtual machine. The user can choose among several operating systems and is in charge of managing the entire software stack. One example is Amazon EC2\(^8\).

- **Platform-as-a-Service (PaaS):** The provider offers a framework on which the code is directly executed. The user is not aware of the underlying software and does not have to bother about any operating system management. Examples are Salesforce’s Force.com Cloud\(^9\) and Google’s App Engine\(^{10}\).

- **Software-as-a-Service (SaaS):** The provider offers an application as a service. This Cloud solution is usually directly addressed to final users, rather than developers. Examples are Google mailing service (GMail) or the storage service Dropbox.

As we can see, the developer is nowadays required to find himself the best trade-off among different factors according to its needs.
The main feature of Cloud computing that we are going to exploit in this thesis is *scalability*. On-premise solutions have a fixed pool of resources that can be used to deliver the service and meeting users’ demand. Before Cloud computing, companies had to rely on expected workloads in advance in order to do upfront investment in resource, this often leading to over-provisioning or under-provisioning, and therefore, loss of money. Scalability is the possibility of acquiring and releasing resources on-demand, almost instantaneously, according to the current demand. Cloud users can, for instance, ask for a new virtual machine, and have it up and running in a matter of minutes and they would only pay for the time they use it, commonly according to a pay-by-the-hour policy.

Scalability is a feature that is managed by the user in case of IaaS solutions. PaaS solutions usually keep the scalability hidden to the Cloud user, the user should only think about the application and the provider is the one managing resource allocation and deallocation.

There are mainly two kind of scaling: *vertical scaling* and *horizontal scaling*. Vertical scaling consists in adding resources to the single VM, for example adding CPU cores or memory. This would be really usefully if the VM could be enhanced dynamically, but current Cloud solutions do not offer this kind of mechanism. Vertical scaling, or *scale up*, is done by starting up a more powerful brand new VM, with a copy of the application, and then switching the less powerful one off. Horizontal scaling consists in adding new and identical virtual machines to the system, typically serving each almost the same amount of traffic by means of a load balancer which distributes the load among them. In this thesis we are going to exploit the horizontal scaling for adapting the system to the incoming workload.
3.1.1 Portability

The National Institute of Standards and Technology (NIST) defines portability as "the ability of prospective cloud computing customers to move their data or applications across multiple cloud environments at low cost and minimal disruption."[11]

According to the Cloud Security Alliance (CSA)[12] there exist different requirements of portability among the three models of Cloud services:

- In the **IaaS** case, portability is the ability to easily migrate a virtual machine and its relative data from one provider to another.
- In the **PaaS** case, portability is the ability to deploy the same application on different platforms.
- In the **SaaS** case, portability is the ability of extracting data from the original provider and import them in a new one.

Some of the main reasons why a Cloud customer should decide to migrate his whole infrastructure, or part of it, to another are:

1. SLAs offered by the provider are not satisfied.
2. Costs increase.
3. Competitors offer services missing in the current Cloud service provider.

Therefore, in the Cloud Computing market, as well as all other markets, customers should be free to choose, monitor and change their provider if needed. Nowadays, however, due to
missing standards and cross-compatibility between Clouds, costs are usually too high and Cloud customers are usually locked (*lock-in* phenomenon).

In our thesis we will not focus on Cloud portability. However, we now take a short overview of ongoing works on this field, so to show that assumptions in this thesis on Cloud portability are based on problems that research community are actually coping with.

The most distinguished work is certainly the one relative to the IEEE work group named **P2301**, Guide for Cloud Portability and Interoperability Profiles[13]. They are in charge of tracing common directives for APIs, file formats and computational conventions.

**VMware** and **Google**, leading actors of the Cloud market for a long time, agreed on a partnership[14] with the aim of leveraging on their respective Cloud offers simplifying portability among them.

**SimpleCloud**[15] is another interesting project aiming at increasing PHP applications portability by means of adapters. The project is open-source and conceived by ZendTechnologies in collaboration with *IBM, Microsoft, Rackspace, Nirvanix* and *GoGrid*.

**mOSAIC**[16] (Open-Source API and Platform for Multiple Clouds) is an FP7-ICT project which is developing a platform that promotes an open-source Cloud API and a platform targeted for developing multi-Cloud oriented applications.

**Cloud4SOA**[17] is another FP7-ICT project, coordinated by Atos Origin, aiming at strengthening three fundamental and complementary paradigms, Cloud computing, Service Oriented Architectures (SOAs) and lightweight semantics, with the objective of proposing a reference architecture that allows interoperability among different Cloud vendors.
**OCCI**[^18] (Open Cloud Computing Interface) is a protocol and API for all kinds of management tasks. OCCI was originally initiated to create a remote management API for IaaS model based Services, allowing for the development of interoperable tools for common tasks including deployment, autonomic scaling and monitoring. It has since evolved into a flexible API with a strong focus on integration, portability, interoperability and innovation while still offering a high degree of extensibility. The current release of the Open Cloud Computing Interface is suitable to serve many other models in addition to IaaS, including e.g. PaaS and SaaS.

Lastly, **MODAClouds**[^19], the European project that was already cited in Chapter 1 as the context in which this work was conducted, is aiming at avoiding vendor lock-in problems supporting the development of Cloud enabled Future Internet applications.

### 3.2 Model Based Control

Several approaches to the general problem of modeling, analyzing and enforcing QoS requirements of software systems have been presented in the literature.[^20] provides an overview of these approaches. Most of them target the design phase. Some, like the one presented in[^21], focus on the modeling of QoS requirements and in supporting developers in the creation of documentation. Others (see[^22]) help developers by optimizing in an automatic or semi automatic way the architecture of an application. As stated in[^23], modern software systems are increasingly embedded in an open world that is constantly evolving, because of changing in the requirements, in the surrounding environment, and in the way people interact with them. The platform itself on which software runs may change over time, as we move towards cloud computing (Section 3.1). For these reasons, a developer cannot guarantee requirements
satisfaction just from an analysis conducted at design time. The assumptions made at development
time can change in ways developers did not think of. Often, changes in the application
cannot be handled off-line, but require the software to self-react by adapting its behavior
dynamically, to continue to ensure the desired QoS. The work in [23] advocates that future
software engineering research should focus on providing intelligent support to software at
run-time, breaking today’s rigid boundary between development-time and run-time. Models
should be kept alive at run-time so that software is able to evolve. In order to react in case
of changes in the environment, we need to equip our running software with some instruments
that are not strictly related to functional aspects of the application (what a system should
do), but rather to non-functional requirements, which define how a system should be (e.g.,
usable, available, reliable). Besides the model, we need a mechanism to actually change the
implementation of the software when the model is modified, we need monitors to retrieve data
useful to verify the requirements satisfaction, and we need a controller capable of automatically
evolve the model in case the preset objectives are no longer reached. We will now describe some
works that are starting points for this thesis (Section 3.2.1 and 3.2.2).

3.2.1 Self-Adaptive Software Meets Control Theory

In Section 2.3 we introduced a field which seldom deals with self-adaptive software. The
first examples coming to one’s mind when talking about control theory are its applications in
car’s cruise control or thermostat-controlled temperature regulators. However, control theory is
not bounded to any practical field, it is just a mathematical theory which deals with anything
that may be modeled as a dynamic system. In this Section we will present one of the first
works where a control theoretical approach was used to solve problems of self-adaptation in software system models \cite{7}. In the paper where this work was presented, the authors focused on systems where reliability requirements have to be guaranteed. The typical scenario the authors refer to is a service-oriented application that composes external services through a workflow. External services have their own failure profile, which is unpredictable, and the degree of freedom necessary to self-adaptation is given by the choice of the service, expressed by using probabilities. The application is formally modeled as a DTMC. The controller is any system that, properly coupled to the software system, makes it fulfill its requirements whenever they are feasible. Requirements can be strict constraints on the behavior (\textit{e.g.}, reliability equal to a certain value) or related to the optimization of certain metrics on the observed software executions (\textit{e.g.}, minimization of outsourcing costs or maximization of throughput). The claim this work support is that control theory provides a number of instruments that software engineers to satisfy non-functional requirements even in case of changes in the environment. In particular the authors claim the controller is able to provide:

1. A way to adapt the system in case of change in the requirements.
2. Robustness to fluctuations or sudden changes in the reliability of external services, that may vary around nominal values during normal execution. Actual values are supposed to be estimated on line through monitoring.
3. Robustness to accuracy error in measurement and monitoring.

Figure 2 shows the high level software model of the case study introduced in the paper. An image filtering service is composed by three different implementation of a beautifying
Figure 2: An image filtering service, a representative example from [7].

filter, where one of them is outsourced (External Filter). The DTMC model of the system is shown in Figure 3. The controller will be responsible of adapting the system acting on the control variables C1a, C1b and C5. Therefore, it is in charge of distributing the requests among the three different filters and of deciding whether re-iterating on the iterative filter. All the alternatives are assumed to be black-box services, whose failure rates are collected by run-time monitors that are then responsible of estimating the probability that an invocation to the service will fail. Starting from the DTMC model of Figure 3 and using the formalism described in Section 2.2, the authors write down the corresponding equation system as in Equation 3.1. By solving that system for s0 it is possible to obtain the closed formula
Equation 3.1 that describes the explicit dependency of reliability (s) on control variables (c) and measured reliabilities (r).

\[ s = r_0 \cdot r_6 \cdot \left( \frac{c_{1a} \cdot (-1 + c_5) \cdot r_2}{-1 + c_5 \cdot r_2} + c_{1b} \cdot r_3 + (1 - c_{1a} - c_{1b}) \cdot r_4 \right) \]  

(3.1)

Suppose that the adaptation mechanism acts at instants identified by an index k. Also, let the average duration of a step be significantly longer than the time scale of the controlled systems dynamics. This means that if at the beginning of a step the controller altered the transition probabilities of the DTMC, then at the end of the same step the effects of our actions can be measured. So the dynamic system of the software model would be

\[ s(k + 1) = f(r(k) + \Delta r(k), c(k)) \]  

(3.2)
where $s(k+1)$ is the application reliability at step $k+1$, $c(k)$ is the vector of control variables at step $k$, which is kept constant through the step, $r(k)$ is the vector of expected reliabilities at step $k$ (which are estimated via monitoring), and $\Delta r(k)$ accounts for any discrepancy between the real and expected reliabilities in step $k$. The form of function $f$ comes from the DTMC model as computed in Equation 3.1.

In a nutshell, the idea of feedback presented in [7] can be summarized as plugging the controlled system into a larger one where its input is made dependent on its measured output, possibly its state or an estimation of it in the case it cannot be measured, and on the desired behavior for the controlled system. Let $J(k) = f_j(c)$ be a cost function on the control variables $c(k)$, that can also be an uninformative one (such as a constant value) to indicate no preference among all the feasible solutions. In this case the problem is transformed in a satisfiability problem because the controller has just to find a feasible assignment to control variables and not an optimal one. The controller comes into play by solving the problem

$$
\min J(c)
$$

subject to the constraint

$$
||goal(k + 1) - \hat{s}(k + 1)|| \leq \alpha ||goal(k) - s(k)||
$$

(3.4)

$$
\forall c_i(k), \ 0 \leq c_i(k) \leq 1
$$
where $\alpha$ is a value in the range $(0, 1)$ that affects the convergence rate of the solution, that is in the next step we expect the absolute error to be reduced by a factor $\alpha$. $\hat{s}$ is the expected system reliability, computed as:

$$\hat{s}(k + 1) = f(\hat{r}(k), c(k))$$  \hspace{1cm} (3.5)$$

where $\hat{r}$ are the measured reliabilities, while control variables $c$ have to be set by the controller so to satisfy Equation 3.3 and Equation 3.4. \textit{goal} is the set-point, that is the desired reliability at each step. The set of constraints has to be extended with probabilistic constraints (the sum of outgoing transitions from each state has to be 1), as done for the control variables $c_i$.

For the proposed case study the control system acts minimizing

$$J(c) = (J_{1a}c_{1a} + J_{1b}c_{1b} + J_5c_5)^2$$  \hspace{1cm} (3.6)$$

where $J_{1a}, J_{1b}$ and $J_5$ are equal to one, therefore assuming that all costs are equal. Reliabilities $r_i$ vary according to the following functions

$$r_0 = 0.95 + 0.02stp(k - 25) - 0.20stp(k - 50) + 0.10stp(k - 75)$$

$$r_2 = 0.95 + 0.02stp(k - 20) - 0.20stp(k - 70) + 0.15stp(k - 85)$$

$$r_3 = 0.95 + 0.02stp(k - 15) - 0.97stp(k - 55) + 0.50stp(k - 65)$$  \hspace{1cm} (3.7)$$

$$r_4 = 0.95$$

$$r_6 = 0.95 + 0.05stp(k - 95)$$
Figure 4 shows the result of the simulation. The dashed line is the set point of the desired availability which is modified during the simulation, the solid line is the availability of the controlled system. From this figure we can see that the controller is capable of modifying the behaviour of the application in order to get the desired availability, it converges to the new set point in few time units and does not present oscillating behavior. Figure 5 shows the value assigned by the controller to control variables at each time unit. For time units between 55 and 65 a failure of node $r_3$ is injected, the controller reacts by changing the probability of using that node to 0 and raises other probabilities.

Figure 4: Reliability of the system: set point (dashed) and achieved value (solid).
3.2.2 Availability Model for Multi-Cloud Applications

We now introduce the model used in our work to describe availability properties of multi-Cloud applications. The model is an extension of the DTMC model described in 2.2 and in 3.2.1, and was first introduced in [5].

In [5] the authors propose an extension to the DTMC in order to model some peculiarities of the cloud environment. This model is used to analyze the availability characteristics of applications and their ability to fulfill the stated requirements.

The extended DTMC allows to model explicitly some cloud-specific concepts. Nodes may represent either Physical or Logical node (see Figure 6). Physical nodes correspond to some concrete resource, e.g., a physical server or a pool of virtual machines offered by a cloud provider. Logical nodes represent entities that are relevant for the DTMC analysis such as the success or
the failure state. In turn, physical nodes can either represent elements in the concrete system to which we want to associate a limit in the throughput and a cost, or elements, that, from the perspective of the model, have unlimited throughput and no cost associated. In the figure these are presented by *Limited Throughput Nodes (LTN)* and *Unlimited Throughput Nodes (UTN)*. In their work, for example, authors modeled load balancers as UTNs because they usually are managed by cloud providers in an automatic and transparent fashion. On the other hand, LTNs have a cost and a maximum service rate associated. They can be further classified in *Autoscaling Groups* or *Fixed Throughput Nodes*. The first ones represent entities capable of performing autoscaling, *i.e.*, changing dynamically their processing capacity, increasing the number of *Active VMs*. In this case node maximum service rate (SR) is given by *Active VMs*.
times VM maximum service rate (sr). Fixed Throughput Nodes may model nodes with fixed capacity such as local servers in a hybrid cloud.

Similar to the approach in [7], DTMC transition labels may model either control variables or measured availabilities. Measured availabilities represent factors originating from the infrastructure (e.g., cloud middleware failures, network delays) and are external to the application. In control theory terminology these factors are called disturbances and can be measured by monitors. Control variables define the probability of routing a certain request to either one node or another.

We saw in Section 2.2 that DTMC nodes and transitions can have a reward (or a cost) attached. In the extended DTMC model, costs are attached to nodes and model the cost generated by a request traversing that node. In a cloud environment, costs are usually due to the number of VMs instantiated, which is proportional to the number of requests that are sent to the node.

Every autoscaling node is labeled with a minimum and a maximum number of running instances. These two parameters represent, respectively, the minimum and maximum number of machines that can run simultaneously on the resource modeled by the node. These bounds are usually either set by the user or by the Cloud provider. The last parameter that the designer has to specify in order to build a complete instance of the model represents the availability constraint of the application which can be a fixed value or a function of time.

A simple two Clouds example modeled according to the extended DTMC model is depicted in Figure 7. Red boxes represent logical nodes. Green boxes model load balancers as physical
nodes with no costs and infinite processing capacities. Yellow boxes represent Autoscaling
groups. The application presented in this model is composed of two replicas of a service deployed
on top of two cloud providers represented by states 4 and 6. Requests entering the system are
directed to one of the two services by the load balancer according to probabilities C0 and 1
C0. When reaching one of the two cloud providers, represented in states 2 and 5, requests can
be lost because of a failure in the cloud infrastructure and go directly to failure states 3 and 7.
If a request reaches one of the services, it is processed. The processing of a request could lead
to a successful execution of the service, state 8, or to a failure due to the overloading of the
processing resource, represented by transitions to states 3 and 7. The table attached to node
4 represents the values of the parameters that characterize that specific node. For the sake of
simplicity only labels of the relevant nodes have been represented in the image.
3.2.3 **A Multi-Cloud Simulator**

The model described in Section 3.2.2 has been developed with the intent to describe availability related aspects of an application. In order to validate the design created by the development team authors of[5] built a simulation tool that is capable of evaluating the availability of the modeled system against many scenarios that may occur in the Cloud environment. The tool simulates an open workload in which requests enter the system, flow through the chain of the model presented in Section 3.2.2 and exit the system.

The simulation engine was built by looking at the infrastructure offered by Amazon Cloud. This infrastructure is quite common among Cloud providers. It has the concepts of regions, which are geographically separate data centers, availability zones, which are independent data centers in the same region, and autoscaling group. Load balancing among instances of the same autoscaling group is done uniformly. This factors has been taken in consideration while building the simulation system trying, at the same time, to generalize them.

The following sections describe parameters that can be varied in order to simulate different scenarios and the outcome of the simulation.

3.2.3.1 **Workload**

The *input workload* is the number of requests that enter the system per second. This parameter can be set to a fixed value for the entire simulation on order to asses the behavior of the modeled application in a steady state. It is quite rare that in a real case the workload is constant. In many situation the system manager is interested in evaluating how the system
reacts in case of a peak of requests. An example of a common workload is the one Figure 8. It presents two peaks of requests centered at two different hours of the day.

The simulation engine takes as an input a Matlab function that describes the workload, at each step of the simulation generates the incoming traffic according to it. The function used to specify the arrival rate of Figure 8 is:

Listing 3.1: Matlab function used to generate the bimodal distribution

```matlab
arrival_rate = 0.75e6*(1 + 2.5*rectpuls(t-(10*60*60),8*60*60).*\(1+\cos((t-(10*60*60)))\)) \*2*\pi/(8*60*60)) + 4*rectpuls(t-(19*60*60),10*60*60).*\(1+\cos((t-(19*60*60)))\)) \*2*\pi/(10*60*60));
```

At each step of the simulation the tool evaluates the specified function and scales it by the number of seconds considered in the simulation steps in order to get the mean arrival rate of
requests. It then generates the actual number of requests assuming the inter arrival times to be exponentially distributed. A Poissonian random number generator with mean given by a user defined function is used. More information about realistic traffic generation can be found in [24].

3.2.3.2 Infrastructural parameters

The model described in Section 3.2.2 includes some aspects of the infrastructure that is used to run the application. The simulation of the impact of infrastructural factors on the availability of the application requires the specification of some parameters specific to the modeled scenario. In particular the user can specify the availability of nodes in the Markov chain. This parameter can be used to model different scenarios like the complete or partial failure of a service due to factors external to the application.

Another parameter specific to the cloud infrastructure is the VM startup time. This parameter specify how many seconds a VM of a particular autoscaling group needs to boot up and perform initialization operations before being ready to serve requests. As shown in [25] this value depends on the cloud provider chosen, the operating system of the virtual machine, the computational power of the machine and many other factors. The simulator keeps track of scaling requests made by the autoscaling controller of each autoscaling group and uses timers set according to the user specified parameters to simulate the boot up of machines.

The maximum capacity parameter specifies the maximum number of request that a node can process each second. This value can be used to represent the computational capacity of autoscaling groups or of an in house server. The meaning of this parameter is different between
nodes representing an autoscaling group and fixed throughput nodes. If a node is of type Autoscaling Group the maximum number of request that the node is capable of processing is given by the number of requests multiplied by the number of active VMs. In a cloud environment physical resources are shared among VMs of many application providers and the actual performance of a VM can vary significantly. Examples of the variability of VM resources like CPU, memory or HDD are shown in [26].

3.2.3.3 Simulation parameters

Some other parameters can be used to tweak the simulation according to user needs. The Simulation time and time step size are used to specify the period of time that the system will simulate and the granularity of the discretization used in the simulation. These values are used to define the number of steps for the entire simulation.

It is then necessary to define a policy to manage the scaling capability of the underlying infrastructure. In order to do that they implemented a mechanism that is widely used on Amazon based on triggers on the CPU utilization. When the CPU utilization of a node in the model representing an autoscaling group exceeds a threshold the number of virtual machine in the node is adjusted by multiplying the number of currently available machines by a factor. The simulator let the user configure different thresholds and corresponding multiplication factors. Thresholds can be used both for increasing or decreasing the number of virtual machines in a node.
The last two parameters that have to be specified to completely define the control policy of autoscaling nodes are the width of the *time window* used to calculate the average value of the CPU load of nodes and the frequency with which the autoscaling policy is applied.

The user can vary these parameters in order to simulate different scenarios.

### 3.2.3.4 Simulation Engine

Every request entering the system is dispatched among nodes following the DTMC model. If a processing node is unavailable for a period of time, *i.e.*, its availability is set to zero, all requests going to that node are routed to the corresponding failing node. Nodes can also discard requests because of their limited computational capacity. This aspect is simulated using the maximum capacity parameter. Whenever a node is fed with more requests than those it can serve, exceeding requests are routed to its corresponding failure node. The number of requests that a node can satisfy can be fixed in case of non-scaling nodes or change. As explained in Section 3.2.2, nodes capable of autoscaling model groups of VMs in the Cloud, their maximum processing capacity is given by

\[
\text{number of VMs} \times \text{VM maximum service rate}.
\]

The assumption that all VMs have the same processing power is quite usual in real solutions for performance reasons, since load balancing is usually homogeneous. Anyway, this aspect can be taken into account while designing the model by splitting the node into two sub nodes with different processing capacity and costs. Requests flowing through an autoscaling node
may trigger a rule and start the scaling process. The simulation engine takes into consideration
scaling actions that may be taken by a policy defined by the application developer and changes
the number of VMs in the corresponding node only after a startup time defined by the user.

The simulation tool runs the simulation algorithm according to the parameters defined by
the user and shows the total availability of the system, the availability of each node and the
total costs.

For each step \( k \) of the simulation the engine performs the following operations:

1. loads the value of all parameters describing the state of the system environment at step \( k \)
2. the incoming traffic is then iteratively distributed to all nodes of the DTMC model
   according to the transition matrix until all requests reach an absorbing node (success
   or failure state)
3. as described in\(^{[24]}\), a simple way to simulate a realistic service time is modeling its
distribution by means of exponential variables. So, for each node traversed by the requests,
the total service time needed to serve incoming workload is generated using a random
generator over the Gamma distribution

\[
\Gamma \left( \text{number of reqs, } \frac{1}{\text{number of VMs } \times \text{ VM maximum service rate}} \right)
\]

In fact, the gamma distribution models sums of exponentially distributed random variables.

4. the amount of requests that fails due to timeout are computed by comparing the duration
   of the step and the total service time required
5. the average CPU usage is updated by comparing the total service time required by the node to process incoming requests and the duration of the step

6. the measured availability of each node is updated according to the success rate of the step

7. computes the availability of the system in the current step.

8. updates the number of running machines by checking if any node had requested a scale up and the timeout for the scale up of the node has expired

9. historical data is saved

10. the autoscale controller checks if any scale up or scale down process has to be performed according to the user defined rules

3.3 Final Considerations on the State of the Art

In Section 3.2.1 a work was introduced which deals with software self-adaptation based on QoS requirements using a control theoretical approach. The controller is able to adapt incoming traffic distribution based on feedback, but it does not consider nodes capacity and costs of using resources. Both these aspects are fundamental for the deployment on a Cloud environment in order to exploit its main peculiarities, that are, scalability and resources on demand. Moreover, in order to deal with scalability, the controller should also take into consideration another kind of control variable, besides the routing probabilities, that is, the number of nodes to be instantiated in each autoscaling group. IaaS providers usually allow users to define a scaling policy, however, these are usually limited and ad hoc controllers have to be implemented so to increase performances.
In Section 3.2.2 we discuss a model that we used in this thesis to model multi-Cloud application and is kept alive by feeding data from monitors and it is used to allow our proposed controller to take decisions. The tool described in Section 3.2.3 is reused in order to test our approach, using the simulator implemented in Matlab.
4.1 Overview of the approach

Our approach is based on control theory, a branch of engineering that deals with dynamic systems with inputs. In particular, we adopt a closed-loop controller approach that uses the model of the system and feedback from the measured output. As shown in Figure 9, we use the extended Discrete Time Markov Chain (DTMC), presented in Section 3.2.2, defined during design time, as our system model and data from a monitoring system as feedback. Using such information, our controller is able to make decisions on the actions to execute in order to fulfill the availability requirements defined at design time. Such decisions are made at runtime and have an impact on the actual configuration of the system. The controller keeps alive the model by updating its parameters with estimates based on the monitoring data. In order to generate the model we used the modeling tool Palladio extended in [5].

The controller we are going to define, is a dual layer controller. The first layer controller is responsible for managing one autoscaling group, controlling the number of running machines. Therefore, there are as many first layer controllers as the nodes modeling autoscaling groups. The second layer controller is a load balancer in charge of distributing the incoming traffic
among nodes. The cooperation between these two layers of controllers aims at guaranteeing system availability, while minimizing costs.

Both controllers work at discrete time, that is, monitoring data is aggregated and delivered from monitors at constant time intervals (steps).

4.2 The autoscaling controller

The first layer controller is in charge of performing adaptation at the node level of our DTMC model. As we saw in Section 3.1, PaaS solutions do not require the developer to specify scaling policies. In fact, the autoscaling is transparent and managed by the provider
automatically. Thus, the controller layer we are defining in this section is clearly only useful for those applications using at least an IaaS component. If the application is deployed on top of a PaaS system this layer of control is managed by the cloud provider and only the controller of Section 4.3 is necessary even though some modification may be required to estimate parameters such as the maximum service rate.

We are also assuming that the providers offer API’s to retrieve information about the CPU percentage utilization, the number of running machines, the status of machines (pending or running) and instances pricing, and API’s to turn machines on or off, which is quite a realistic assumption given the current providers’ offer (see Section 3.1).

4.2.1 Objective

The objective of this controller is to guarantee that there are enough resources to satisfy the workload dispatched to the node. More precisely, it has to manage the number of running machines in an autoscaling group so that the average percentage of CPU utilization is equal to a setpoint, which is a parameter that is chosen by the developer. This parameter has to be chosen wisely considering that keeping resources highly loaded will certainly reduce costs, since less running machines will be needed, but there will also be less safety margin in case of sudden increase of incoming workload and availability may be compromised.

4.2.2 Monitoring

Recalling Figure 1, what we need for a controller is a feedback loop. So, to begin with, we need data from “sensors” so that we can check how the system is behaving in response to controller’s decisions. First of all, we define a sliding observation window, which is the time span
(or number of steps) used to compute statistics from data collected by sensors. The statistics, which are all relative to the observation window, are the following:

- the *incoming workload*, that is the number of incoming requests to the node;
- the *successful requests*, that is the number of requests successfully processed by the node;
- the *average CPU load*, that is the average percentage of CPU utilization computed over all running machines in the node;
- the *number of running machines*.

From this data, the *success rate* is then estimated as

\[
\text{success rate} = \frac{\text{successful requests}}{\text{incoming workload}}
\]

The success rate will be our parameter of availability.

### 4.2.3 Control

Given estimated data from the monitoring system, we first need to find a control formula where the error observed between the desired behavior and the actual one, can be reduced (and asymptotically eliminated) at each control step acting on the control variables.

We can identify two main working conditions:

1. the number of machines is sufficient to satisfy the entire incoming traffic (average CPU usage ≤ 100%, availability = 100%);
2. the number of machines is not sufficient to satisfy the entire incoming traffic (average CPU usage = 100%, availability ≤ 100%).
In order to control the system we make use of the feedback loop methodology used in \cite{7} and presented in Section 3.2.1. The only control variable we have is the number of machines we want to have running at the next step, \( i.e., n(k + 1) \).

As for the first working condition, all incoming traffic is satisfied, therefore availability is 100\%, and we want to reach the CPU setpoint \( u \). Therefore, the controller should find \( n(k + 1) \) so to satisfy the following equation

\[
u(k + 1) - \hat{p}(k + 1|k) = \alpha(u(k) - p(k))
\] (4.1)

where \( \hat{p}(k + 1|k) \) is a function of \( n(k + 1) \) and represents the expected CPU usage at the next step. \( \alpha \) is a parameter in the range \((0, 1)\) and determines how fast is the convergence to the solution, that is, in the next step we expect the absolute error to be reduced by a factor \( \alpha \).

The relation between \( \hat{p}(k + 1|k) \) and \( n(k + 1) \) can easily be found given the equation

\[
p(k) = \frac{AR(k)}{SR(k)}
\] (4.2)

from \cite{27}, where \( SR \) and \( AR \) are the total maximum service rate and the total arrival rate respectively. We emphasize that Equation 4.2 holds only in the first working condition. Furthermore, we recall that \( SR \) is given by the contribution of all virtual machines having each a maximum service rate \( sr \):

\[
SR(k) = sr(k)n(k)
\] (4.3)
From Equation 4.2 and Equation 4.3 we get:

\[ p(k + 1) = \frac{AR(k + 1)}{sr(k + 1)n(k + 1)} \] (4.4)

We assume that time intervals are small enough to consider the maximum service rate of a machine and the arrival rate to remain constant. If this does not hold, prediction can be taken in consideration, but it is out of the scope of this thesis. Therefore our expected CPU utilization is:

\[ \hat{p}(k + 1|k) = \frac{AR(k)}{sr(k)n(k + 1)} \] (4.5)

Given Equation 4.3 and Equation 4.2, Equation 4.5 becomes:

\[ \hat{p}(k + 1|k) = p(k) \cdot \frac{n(k)}{n(k + 1)} \] (4.6)

Using this result with Equation 4.1 we analytically obtain the control formula to be used in the first working condition:

\[ n(k + 1) = \frac{n(k)p(k)}{u(k + 1) - \alpha(u(k) - p(k))} \] (4.7)

As for the second working condition, the incoming traffic is higher than the total service rate, therefore the total CPU usage is equal to 100% and we are not able to use control formula Equation 4.7 anymore. However, in this working condition it is easy to evaluate the maximum service rate \( SR \) since it is equal to the total throughput, i.e., the traffic actually satisfied over time. Using Equation 4.3 and Equation 4.4, setting \( p(k + 1) \) equal to our setpoint \( u(k + 1) \)
and given assumptions similar to the ones made before, after some algebra we obtain the exact number of machines $\bar{n}$ required to satisfy the incoming traffic:

$$\bar{n} = \frac{AR(k)n(k)}{SR(k)u(k + 1)}$$

(4.8)

In order to cope with noise we use a convergence rate to the desired setpoint, similarly to the previous case:

$$n(k + 1) - \bar{n} = \alpha(n(k) - \bar{n})$$

(4.9)

which gives the final control formula:

$$n(k + 1) = \alpha n(k) + (1 - \alpha)\bar{n}$$

(4.10)

We know from\textsuperscript{[7]} that exponential convergence to the setpoint is ensured for equations of the kind of Equation 4.1 and Equation 4.9 with rate $\alpha$.

Obviously the number of machines is an integer number, but we do not encounter any problem in rounding this number, unless we deal with a very small number of machines or with CPU utilization ranges too close to the set point. In these cases, there would be undesired behaviors, however not too difficult to cope with. In the case, for example, we are dealing with a small number of machines, we can use the ceiling of the decimal solution computed by the controller instead of rounding it. This way, if $n(k) = 1$ and the controller returns $n(k+1) = 1.25$, ...
the ceiling would make the node scale out, which is a preferred behavior when high availability
is required.

Once the controller computed this number, the controller is responsible of using the Cloud
provider’s APIs to turn off the exceeding machines or to launch new ones. As said before,
in case new machines are either launched or turned down, the controller enters the cool-down
state.

4.2.4 Timing

As stated in Section 3.1, launching new machines is a slow process, it may take minutes.
So we need to prevent the controller to take decisions while machines are turning on (pending
state). We will say that a node is stable whenever there are no machines in pending state. Also,
we want statistics from monitors to be estimated only from data observed after a scale out or
a scale down process happens. Therefore, we defined a cool-down state, which will inhibit the
controller as long as it is active. A node enters the cool-down state when a scaling process is
started (both scale out and down) and will exit from this state only after remaining in a stable
state for the entire duration of the observation window.

After exiting from a cool-down state, the controller will be allowed to take decisions, reacting
on statistics from monitors.

4.3 The load balancer controller

The second layer controller is responsible for setting the controllable variables on transitions
of the DTMC model, so to manage traffic distribution. The approach used for this layer
is a Cloud extension of the work in[7], where nodes capacity, scalability and costs were not considered.

4.3.1 Objective

This controller aims at distributing traffic among nodes guaranteeing the required availability, minimizing costs. As we said in Section 3.1 different providers offer different prices, that may change over time (e.g. Amazon hotspots). Furthermore, being the Cloud a shared infrastructure, performance can change over time as well. Therefore, at different time of the day may be more convenient one solution with respect to an other. Moreover, we also observed in Section 3.1 that the availability of a single Cloud region is very low, so the controller is responsible of reacting when an entire region fails, migrating incoming requests to the remaining available nodes.

4.3.2 Monitoring

At this level, we will need all the information already used by the first layer controller from each node, that is, the incoming workloads, the successful requests, the average CPU utilizations, the numbers of machines, plus, we are going to need the instance pricing of each node (cost per machine) and an estimate of the arrival rate at the input node of the system. From this data, aggregated parameters are estimated:

- the maximum service rate (SR), that is the number of requests processable by a node over time at 100% CPU utilization level, computed as

\[
\frac{\text{successful requests}}{\text{average CPU utilization}}
\]  

(4.11)
the cost per request, which measures the cost of a request traversing the node (see Reward Markov Chain in Section 2.2) at the desired CPU utilization. The number of incoming requests when all machines of one node are working at the desired CPU utilization $u$ is equal to $SR \times u$. It follows that the cost per request is computed as

$$\frac{\text{cost per machine per second} \times \text{number of machines}}{SR \times u}$$

(4.12)

The model is iteratively updated at runtime using monitored data.

### 4.3.3 Control

The setpoint defined by the user at this layer is the minimum success rate of the system. We decided to allow the developer to set a minimum because even though he would always like to have 100%, for some applications he might want to make a trade-off between costs and availability. So, for example, he might prefer that sometimes some requests fail, rather than migrating the application on a more expensive cloud which is actually guaranteeing 100% availability.

In this case, the problem cannot be solved analytically anymore. The load balancer controller is in charge of solving a non-linear constraint minimization problem. The control variables are, as said in Section 3.2.2, the attached probability of some arc of the DTMC model. The controller has to choose, among all feasible values, the ones that minimize a cost function.

Since we deal with probabilities, the first constraint is that controllable variables must be chosen in the range $(0, 1)$. Also, since we are dealing with a DTMC, the sum of the outgoing

•
arcs must be 1. This last constrain can be avoided allowing only two outgoing arcs on load
balancers and set the value of one of the arcs equal to one minus the other. If we want to have
a load balancer with three or more outgoing arcs, it is enough to put two or more binary load
balancers in cascade.

Then we need a constraint on the total availability, which has to be greater or equal to the
setpoint. To do this, we must obtain a formula that describes the explicit dependency of system
availability on control variables and measured nodes availabilities, like the one in the example
in Equation 3.1. First of all, given the transition matrix \( A \) of our DTMC model with self loops
removed (i.e., no 1’s on the diagonal), \( i \) is the row of the matrix relative to the input node, \( j \)
is the row of the matrix relative to the output node (i.e., the success state), we can write the
following dynamic system

\[ x^T(k + 1) = x^T(k)A + b^T \]  (4.13)

where \( x \) is a vector as long as the number of nodes, and \( b \) is the input vector, as long as \( x \), with
all 0’s except for the \( i \)th element which is 1. If \( b \) is constant the system is going to stabilize
and the values of \( x \) are going to be the work\l workload ratio arriving at each node:

\[ x^T = x^TA + b^T \]

\[ x^T(I - A) = b^T \]  (4.14)

\[ x^T = b^T(I - A)^{-1} \]
The $j$th element of $x$ is going to be the success rate as a function of the control variables and nodes availabilities, which will be used to estimate the availability. Since we are dealing with models whose structure is constant in time, the success rate function is always the same and can be computed at design time.

Now we can write the availability constraint function as

$$u(k + 1) - \hat{s}(k + 1|k) \leq \alpha \cdot \max (0, u(k) - s(k)) \quad (4.15)$$

where $u$ is the set point, $\hat{s}$ is the estimated availability, using the average availabilities of the nodes and letting $\hat{s}$ become a function only of the control variables. $\alpha$ is a parameter in the range $(0, 1)$ that will affect the convergence rate to the solution. Finally, $s$ is the system availability measured at step $k$. Using Equation 4.15 the controller is allowed to let $\hat{s}$ be greater than the set point $u$.

Now we define the cost function that has to be minimized. We already defined the cost of each node of our Reward DTMC model as the cost per request, which is estimated by the monitoring module. Nodes that are not autoscaling groups will clearly have cost equal to zero. The tentative cost function would be then

$$J_1 = x^T \cdot k \quad (4.16)$$
where $\mathbf{x}^T$ is the previously calculated *workload ratio* array that, once availabilities are substituted with the average availabilities measured for each node, depends only on the control variables. $\mathbf{k}$ is instead the vector containing the *cost per request* values, computed using Equation 4.12.

We said “tentative” cost function because there is still something missing. Let us suppose that all nodes are stable and healthy, that is, 100% availability. We are using one Cloud, and suddenly a second Cloud becomes more convenient. The minimization of the cost function $J_1$ would cause a sudden migration of requests from the first cloud to the second cloud, which will not be capable of satisfying the entire workload until the scaling process is complete. Consequently many requests will be lost, and availability will be consistently affected. Therefore, we also want to discourage the controller from overloading a node with more requests than the ones it is capable of processing. Whenever a migration for economic reasons is required, the workload has to be gently distributed on the cheaper node letting it the time to scale without overloading it, that is, without loosing requests and affecting availability. The cost function we finally defined is

$$J = \mathbf{x}^T \cdot \mathbf{k} + W \| \max(0, AR(k)\mathbf{x} - SR(k)) \|$$

(4.17)

where $W$ is a big number that has to be much bigger than the first member of the cost function. $AR$ is the average arrival rate to the input node of the system. $\mathbf{x}$ is the array of workload distribution on the node relative to the incoming workload to the input node, and depending on the control variables that will be chosen by the controller. $SR$ is the array of the estimated maximum service rate of each node.
This approach is a workaround to put a constraint to be considered only when the nodes availabilities are high. We want to avoid losses whenever the migration is only for economic reasons. When the availability constraint is not satisfied, because, for example, an entire autoscaling group failed, the controller will not find a minimal solution of $J$ without overloading a node, but in this case it is the desired behavior for the following reasons:

- All requests going to the failed node would be lost anyway
- The overloaded node will scale much faster in order to cope with the new workload since the availability $AR$ in Equation 4.8 will be very high.

### 4.3.4 Timing

The *autoscaling controller* and the *load balancer controller* work simultaneously. In order to prevent oscillations due to controllers coordination, it was sufficient to set a policy that inhibits the second layer controller whenever any Cloud is not able to satisfy the incoming traffic, giving time to the first layer controller to make it scale.
CHAPTER 5

EVALUATION

In this chapter we evaluate our approach deploying three example of multi-Cloud application on simulated IaaS Cloud environment. The simulator tool used was described in Section 3.2.3. The tool was extended with the implementation of our controller. Throughout all these use cases the $\alpha$ and $\beta$ parameters for the control algorithm have been initialized both to $1/3$ as default value.

The first example shown in Section 5.1 represents a simple web application deployed on two independent Clouds. The use case of Section 5.2 models again an application deployed on two independent Cloud providers one offering a single region model and the other offering two regions for the deployment and execution of applications. The last use case, described in Section 5.3 models a more complex application, that deals with the management of a smart city emergency system with high availability requirements. Section 5.4 makes some considerations on the results obtained and the behavior of the controller throughout the simulation.

5.1 A Web System Scenario

In this Section we consider a simple example to test different usage scenarios and how our approach is able to cope in case of simulated failures or changes in the domain. The main goal of this example is not to show a complex real world application but rather to test how the controller reacts to some specific scenario that may happen in the Cloud environment.
Figure 10 shows the model of the application. The application is composed by a load balancer that receives users’ requests and forwards them to one of the two Cloud providers on which the application is deployed. Figure 11 shows the DTMC model derived automatically by the extended Palladio tool developed in the work described in [5]. In this model we can see that the load balancer has been modeled by a node with two outgoing arcs whose probabilities is controlled by the control variable $C_0$.

The availabilities of the two Cloud providers are modeled respectively by $a_2$ and $a_5$. Failures of these two nodes are independent of the application and the resources directly related to it. They may model the entire Cloud failure or failure in the delivery of some requests due to network issues or software bugs of the Cloud management infrastructure.
Figure 11: DTMC model representation of the Web System use case. Green nodes represent autoscaling groups, red nodes represent failure states.

The failure of requests processed by autoscaling groups (represented by green nodes) due to their limited computing capabilities are modeled by arcs going from states 4 and 6 to the corresponding failure states according to $a_4$ and $a_6$. The availability of these nodes is dependent on the current allocated resources by the first layer controller. Finally, the success state is a logical state in which requests end, after being successfully processed by the system.
The transition matrix is the following:

\[
A = \begin{bmatrix}
0 & c_0 & 0 & 0 & 1 - c_0 & 0 & 0 & 0 \\
0 & 0 & 1 - a_2 & a_2 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 - a_4 & 0 & 0 & 0 & 0 & a_4 \\
0 & 0 & 0 & 0 & a_5 & 1 - a_5 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 - a_6 & a_6 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
\end{bmatrix}
\]

(5.1)

By means of Equation 4.14 we can obtain the work load ratio vector, whose 8th value (success state) corresponds to the system availability:

\[
s = a_5 \cdot a_6 \cdot (1 - c_0) + a_2 \cdot a_4 \cdot c_0
\]

(5.2)

We simulated three different scenarios against this application whose results are reposted in Sections 5.1.1, 5.1.2 and 5.1.3. In all of the three scenarios the parameters reported in Table I are kept consistent.

### 5.1.1 Scenario 1

The setpoint for the desired availability of the system has been initialized to 0.99 and kept constant during the simulation. This scenario simulates four hours of usage of the system in
TABLE I: SIMULATION PARAMETERS OF THE WEB SYSTEM SCENARIOS

<table>
<thead>
<tr>
<th></th>
<th>Cloud 1</th>
<th>Cloud 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cost per VM</td>
<td>0.30$/hr</td>
<td>0.50$/hr</td>
</tr>
<tr>
<td>VM startup time</td>
<td>100 s</td>
<td>100 s</td>
</tr>
<tr>
<td>VM nominal SR</td>
<td>10,000 reqs/s</td>
<td>10,000 reqs/s</td>
</tr>
<tr>
<td>CPU set point</td>
<td>80%</td>
<td>80%</td>
</tr>
<tr>
<td>Nominal cost per req</td>
<td>3.75E-5 /req</td>
<td>6.25E-5 /req</td>
</tr>
</tbody>
</table>

which the arrival rate is constant to 1e6 requests per second. Also the service rate of VMs is kept constant to its nominal value. The only parameter that changes dynamically in this scenario is the availability of Cloud 1. Cloud 2 shows a 100% availability for the entire simulation period while Cloud 1 experiences a failure between time 00:10 and 00:50. We can see from Table I that the cost of using Cloud 1 is lower than the cost of using Cloud 2 while VM maximum service rates are the same, so in standard conditions the system is expected to prefer the former.

Figure 12 shows the availability of the controlled system (in blue) and the desires set point (in red). From this Figure we can see that the failure of Cloud 1 at time 00:10 affects the system availability but the controller is capable of discovering this failure and react by routing traffic to the second Cloud provider. In this scenario the time needed to restore the desired system availability is of about 20 minutes.

Figure 13 shows the number of active VMs for Cloud providers. We can note that as soon as the controller sense the failure of Cloud 1 (at time 00:10) its number of active machines is 0 and the number of machine of Cloud 2 start raising.
This is due to the fact that the first layer controller reacts by moving the traffic from Cloud 1 to Cloud 2. The second layer controller, seeing such a huge traffic, aggressively increases the number of running VMs until the availability is back to desired value. On 00:50 Cloud 1 recovers from its failure so the controller starts to send some requests back to this one since it is the cheapest. The switching from Cloud 2 to Cloud 1 is done in order to reduce costs and happens much slowly than the first switch. If we look back to Figure 12 we can see that in this period availability is not affected.

The total results of the availability and cost of the system is shown in Table II. This table shows the total availability and cost of the system performing the same simulation using only

Figure 12: Availability of the system of the Web System Scenario 1
Cloud 1 first, then only Cloud 2 and finally by using both with both the autoscaling and load balancer control enabled.

This example shows the controller is capable of dealing with an unexpected complete failure of a Cloud provider and to switch between Cloud providers in order to reduce costs without affecting the availability of the system.

5.1.2 Scenario 2

The second scenario is quite similar to the one presented in Section 5.1.1. The length of the simulation is of four hours and the arrival rate is constant at $1e6$ requests per second. In this
scenario both Clouds’ availability are kept constant at 100% but the maximum service rate of machines is changed as depicted in Figure 14.

Figure 15 shows the scaling activity during the simulation. Figure 16 shows the average CPU utilization of VMs. The red line is the average CPU utilization of machines of the Autoscaling Group 1, the blue represent Autoscaling Group 2.

From these results we can observe that when the service rate starts to decrease the load on the CPU of Cloud 1 start to increase. When this value exceeds the maximum tolerated (around time 1:00 in Figure 16) the second layer controller increases the number of VMs in order to maintain the current CPU load near the desired one. Since the maximum service rate continue to decrease the this behavior is repeated several times.

After a certain point the maximum service rate of Cloud 1 falls behind a value that makes it inconvenient to use. This happens near 1:30 when the controller start to gradually move traffic from Cloud 1 to Cloud 2. The redirection of incoming requests causes the CPU usage of Cloud 1 stop growing and, when enough percentage of the incoming traffic is redirected to the more convenient Cloud 2, the CPU load on VMs on Cloud 1 starts to decrease. At the same time

### TABLE II: CONTROLLED VS NON-CONTROLLED RESULTS OF THE WEB SYSTEM SCENARIO 1

<table>
<thead>
<tr>
<th></th>
<th>Cloud 1</th>
<th>Cloud 2</th>
<th>Controlled</th>
</tr>
</thead>
<tbody>
<tr>
<td>C0 Availability</td>
<td>82.12%</td>
<td>100%</td>
<td>96.24%</td>
</tr>
<tr>
<td>Cost</td>
<td>125.88$</td>
<td>251.19$</td>
<td>178,11$</td>
</tr>
</tbody>
</table>
the new workload that enters Cloud 2 makes the CPU of its machines grow. The autoscaling controller of Cloud 2 manages the growth of incoming workload by scaling out the number of machines until the desired CPU load in reached (close to time 3:00).

From the results in Table III we can see that in all the presented cases the availability is 100% this is due to the fact that the first layer controller that manages autoscaling of nodes is capable of reacting to the gradual service degradation. On the other hand the cost of the controlled system is lower than the cost of both non-controlled ones. This is due to the fact that the first layer controller redirects requests on the Cloud that offers the same availability at the lowest price per request.
This scenario shows the fact that sometime in presence of gradual changes in the environment condition, the maximum service rate in this case, the second layer controller is capable of providing the desired availability. It also shows that the first layer controller can measure the effect of the degradation of machines performance and switch the application behavior in order to minimize costs.

5.1.3 Scenario 3

The last scenario for this use case is quite different and tests both the ability of the controller to react to changes in the availability of Cloud providers and to cope with changes in the setpoint. Like in the previous scenarios the simulation time is of 4 hours and the arrival rate
is constant. In this scenario the maximum service rate of VMs is kept constant to its nominal value shown in Table I. We changed Cloud 1 availability and the setpoint according to Figure 17 and Figure 18 (red line), respectively.

Figure 18 shows that the controller satisfies the availability constraint even if the set point is changed and react to these changes. Figure 19 shows how the control variable was changes so to obtain this result. The controller sends more requests to Cloud 2 when the desired availability is raised and more to the cheaper Cloud 1 when availability constraint is relaxed. We can see that from 3:00 on the desired system availability is 0.5 but the actual system availability is 0.8. This behavior is due to the fact that the system availability is requied to be greater or equal to
TABLE III: CONTROLLED VS NON-CONTROLLED RESULTS OF THE WEB SYSTEM

<table>
<thead>
<tr>
<th></th>
<th>Cloud 1</th>
<th>Cloud 2</th>
<th>Controlled</th>
</tr>
</thead>
<tbody>
<tr>
<td>Availability</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Cost</td>
<td>246.95$</td>
<td>251.19$</td>
<td>225.67$</td>
</tr>
</tbody>
</table>

SCENARIO 2

the setpoint. Figure 19 shows that from hour 3:00 all requests are migrated to Cloud 1. This is due to the fact that Cloud 1 is capable of providing the required availability at a lower cost. Figure 20 shows how the number of running machines in the two Clouds evolves overtime.

5.2 A Multi-Region Scenario

This use case models the system reported in Figure 21. It is composed by a first load balancer that splits requests among the two Cloud providers. Another load balancer inside Cloud 1 splits requests between Region 1 and Region 2. The DTMC model obtained is depicted in Figure 22.

The parameters set for this experiment are reported in Table IV. The cost of the first Cloud

TABLE IV: SIMULATION PARAMETERS OF THE MULTI-REGION SCENARIO

<table>
<thead>
<tr>
<th></th>
<th>Cloud 1 (R1)</th>
<th>Cloud 1 (R2)</th>
<th>Cloud 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cost per VM</td>
<td>0.30$/hr</td>
<td>0.30$/hr</td>
<td>0.45$/hr</td>
</tr>
<tr>
<td>VM startup time</td>
<td>100 s</td>
<td>100 s</td>
<td>100 s</td>
</tr>
<tr>
<td>VM nominal SR</td>
<td>10,000 req/s</td>
<td>10,000 req/s</td>
<td>10,000 req/s</td>
</tr>
<tr>
<td>CPU set point</td>
<td>80%</td>
<td>80%</td>
<td>80%</td>
</tr>
<tr>
<td>Nominal cost per req</td>
<td>3.75E-5 $/req</td>
<td>3.75E-5 $/req</td>
<td>5.62E-5 $/req</td>
</tr>
</tbody>
</table>
Figure 17: Clouds’ availabilities of the Web System Scenario 3

is varied during the simulation. The cost is set to 0.3$/hr at the beginning of the simulation and raised to 0.6$/hr at time 4:00. Costs are usually constant, but we want to be quite general in our approach, avoiding to bind to a specific Cloud provider. A Cloud provider like Amazon could change their prices in case spot instances are used, described in Section 3.1. Or else, a cloud provider may decide to change its pricing, after advising its customers, from a specific date.

The parameters we changed during the simulations are the setpoint that varies according to Figure 23. Availabilities of Cloud 1 and Region 1 are changed according to Figure 24. Moreover, the availability of Cloud 2 is kept constant to 99%, while region 2 is 100% available.
The duration of this simulation is 6 hours. The arrival rate is kept constant at 1e6 requests per second.

Figure 25 shows the total availability of the controlled system during the simulation. We can observe that the controller is capable of tracking the required availability and to recover it after the failure of region 1.

Figure 26 shows the running machines instantiated on the different regions over time.

Figure 27 shows how the control variables are changed by the controller during time.

When the set point is raised the system decides to send more resources to Cloud 2 that offers higher availability until time 2:00 when the availability of Cloud 1 is raised to 90%
Figure 19: Control variable values of the Web System Scenario 3 (Figure 24). The controller then slowly switches to use Cloud 1 only since it is cheaper and offers the required availability. At time 3:00 region 1 experiences an outage, due to bug in the Cloud hypervisor software introduced after an update of the system. The controller reacts by moving some requests to region 2 and redirecting some other requests to Cloud 2. At time 4:00 the cost of using Cloud 1 becomes higher than using Cloud 2. Therefore, the controller starts switching slowly to the cheapest Cloud.

5.3 A Smart City Scenario

We finally tested the validity of our approach in a challenging use case in the context of smart city management. The application we are considering deals with the management of
emergencies, it receives data from multiple sensors in the city, elaborates them, recognizes emergency situations and puts countermeasures in action. Examples of emergency situations are a fire in a building, a leak in a gas pipe or a car accident. Countermeasures comprehend alerting emergency teams, calculating optimal path for rescuers to the place of the emergency including traffic light control to evacuate certain zones and clear paths for rescuer squads.

Being a critical application the first and most important requirement is availability. In a deeply automated environment of a smart city the main response to emergency is given via its IT infrastructure, a failure in dealing with an emergency could result in severe damage to the city itself or even cause death.
Embedded sensors in buildings, on streets and on vehicles are already a reality. Once all these sensors are connected to the Internet the amount of data provided will be tremendous. Dealing with such a huge number of sensors involves processing of raw data on the order of TB/s that can vary over time of the day. This huge amount of data has to be cleaned from noise and aggregated. In order to process such a huge amount of data the infrastructure should be scalable. The last requirement of this application, quite obvious and popular this day, is to minimize costs of the IT infrastructure.

In order to fulfill these requirements the most reasonable choice these days is to exploit resources offered from Cloud computing providers. A Cloud platform like Amazon Web Services or Windows Azure can cope with the second requirement quite well and also help to reduce costs but cannot guarantee the availability we wish to have.
The availability requirement for our application is to have five nines, which means that we wish that our application runs for 99.999% of the time. If we consider a general provider with an availability of 95%, as shown in [28], we should use at least four different providers of that kind, assuming that provider failures are independent from each other. In fact, the probability that $n$ independent Cloud providers with an average availability of 95% fail simultaneously is $0.05^n$. We can calculate the minimum number of $n$ needed to fulfill the five nine requirement as shown:

\[
0.05^n < 0.00001
\]

\[
n > \log_{0.05}(0.00001) \quad (5.3)
\]

\[
n > 3.85
\]

So if we use at least four providers our availability requirement is fulfilled.
5.3.1 Application Model

The application is divided into three main layers. The first layer which take cares of collecting data from sensors, filtering, noise reduction and aggregation. The second layer receives aggregated data and update the process model which describes the dynamic state of the city. The third layer contains the reasoning module which is responsible of finding the best response to emergencies. In order to work properly this layer needs access to more information than the one deriving from aggregated data so it could instruct the first layer to reduce its aggregation policy or even to let some raw data pass directly to the reasoner. This structure is shown in Figure 28.
In order to adapt the behavior of the application to the environment conditions we thought of adding a middleware responsible for monitoring the healthiness of the application and taking decisions about design adaptation. We first focused on the filtering part since it’s the most computationally intensive and, consequently, the one with greater impact on costs.

5.3.2 Filtering Part

Given the huge amount of data to be processed it’s unreasonable to send it to all providers and make machines work redundantly to provide higher availability. The DTMC model of the system is shown in Figure 29. Nodes 4, 5, 6 and 7 represent the entry point of the four Cloud providers. Green nodes attached to these nodes represent the auto scaling group of VMs that
process requests in order to filter and clean data. Red nodes represent possible failures. Nodes with incoming arcs labeled $a_4$, $a_5$, $a_6$ and $a_7$ represent the availability of each Cloud provider.

5.3.3 Process Model

In the eventuality of a failure of a Cloud provider we may afford to lose some of the data from sensors but what we cannot lose is the state of the process model. Since it is the result of several hours of processing of incoming data it cannot be reconstructed instantaneously from new incoming requests. Therefore replication of this component on several Cloud providers is needed. Autoscaling should not be considered for this component since it will not require much computing power and the model cannot be distributed on several machines. So we just
have to deploy it on one highly reliable machine for each Cloud. Aggregated data coming from the filtering module is sent redundantly to all the running process models. If, for some reasons, a machine with the process model fails, it is excluded from the process model bucket in healthy state and a new machine with this role will be instantiated, its state will be updated using information from others machine in the bucket and finally added. In order to perform this simple behavior the controller should just check the liveness of these instances. Requests exiting from the filtering layer are replicated and sent to all the machines in a healthy state. Since our controller acts on the routing of requests among different Cloud it is not necessary in this layer. The maximum availability is easily guaranteed by the maximum degree of replication.
5.3.3.1 Reasoning

The reasoning module is supposed to be stateless, since its decisions are based on the information read from the process model and from the current data coming from filtering layer. However, we cannot ever afford to lose the reasoning module, since its failure cause the total system failure that would not be able to react to any emergency situation. For these reason a replication approach similar to the one for the process model should be applied. For performance reasons the reasoning module should retrieve information from the process model that run on the same Cloud provider to minimize latency. Whenever a Cloud provider looses its process model the controller should react by activating the reasoner module of another provider with
a process model working properly, similarly to the controller behavior of the modeling layer. Again since the availability constraints force us to replicate the model in all Cloud providers controlling this layers is not in the scope of this thesis.

Figure 30 shows the model of the Filtering module. Figure 29 shows the relative DTMC. The parameters used for simulating the scenario are reported in Table V. We can observe that VMs offered by Cloud providers 1 and 2 have the same performances and similar costs. Cloud providers 3 and 4 offers more performant VMs at higher costs. Though, as shown by the nominal cost per request (CpR), Cloud 1 and 2 offer more convenient machines.

The scenario simulates the usage of the system in a 24 hour period. The arrival rate is composed by a bimodal distribution shown in Figure 31 with two peaks at time 10:00 and 19:00. Maximum service rates for Cloud providers 1, 2 and 3 are kept constant to their nominal
Figure 29: DTMC model of the filtering part of the smart city use case
values while Cloud 4 experiences a degradation of its service rate between time 13:00 and 17:00 as shown in Figure 32. Also the availability of Cloud providers are changed in order to simulate different failure scenarios. In particular the availability of Cloud 2, the most expensive one, is kept constant at 100%. Cloud 4 experiences a total downtime between time 8:00 and 13:00 that could be caused by the Cloud provider lack of connectivity. Cloud 1 starts with a 95% availability, which is not enough to satisfy the 5 nines availability constraint but from time 10:00 on, its availability increases to 100% as shown in Figure 33. This scenario could happen if the workload of other users of the Cloud decrease and its overall architecture has a lighter load so the availability increases. The availability of Cloud 3 is shown in Figure 34. It starts
Like in the previous scenarios we first perform the same simulation disabling the load balancer controller and using only one cloud at a time. Figures 35, 36, 37 and 38 show how the system availability would be affected by the scenario described before.

We finally performed the same simulation activating the load balancer and the resulting system availability during the 24h is depicted in Figure 39. Figure 40 shows how the number of running machines varies with time on the four clouds.

We can observe that the controller uses only Cloud 4, which is the cheapest one, until it fails at time 8:00. Then, the controller uses Cloud 3 until the maximum number of available machines is reached. Since the workload keeps growing, the controller decides to use also Cloud 2 (time 10:00). At the same time the availability of Cloud 3 degrades to a point that the controller decides to switch some of its requests to Cloud 1 and shut down Cloud 3. Since from 10:00 the availability of Cloud 1 is enough to fulfill the availability requirement and it is cheaper than Cloud 2, the controller shuts down Cloud 2 and redirects all requests to Cloud 1. When the availability of Cloud 3 and Cloud 4 are set back to 100% the controller uses both Clouds

<table>
<thead>
<tr>
<th>TABLE V: SIMULATION PARAMETERS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cloud 1</td>
</tr>
<tr>
<td>---------</td>
</tr>
<tr>
<td>Cost per VM</td>
</tr>
<tr>
<td>VM startup time</td>
</tr>
<tr>
<td>VM nominal SR</td>
</tr>
<tr>
<td>CPU set point</td>
</tr>
</tbody>
</table>

from 100% and decreases to 95% between time 10:00 and 15:00.
instantiating their maximum number of available machines and using Clouds 1 and 2 only if the workload is too high for both Clouds 3 and 4 to serve.

Figure 41 shows how the control variables were varied by the load balancer controller.

We have shown that the controlled solution gives better results with respect to most of the single Cloud solutions in term of availability since it is capable of switching between Clouds when the availability value decreases. The only Cloud that performed better is Cloud 2 since it did not incurred in degradation of its availability during the simulation, Cloud 2 however turns out to be very expensive.
The overall results of the simulation are shown in Table VI. This Table shows that the controlled system does not only overcome non-controlled solutions in terms of availability but also in term of costs.

5.4 Results analysis

The simulation presented in this Chapter covers a variety of different user scenarios that could happen in a real Cloud environment. These simulations show that the controller is capable of adapting application behavior in case of changes in the environment in order to maintain...
or recover the desired availability. In particular we can identify two reasons that bring the controller to change the utilization of Cloud providers:

- The controller changes gradually the distribution of requests among Cloud provider for *economical reasons*;

- The controller reacts to *failure* or *degradation* of Cloud performances and modify the behavior of the application to restore availability.

The first kind of change can be observed in Figure 13 where the controller shuts down Cloud 2 and redirects all traffic to Cloud 1. Looking at Figure 12 we see that the availability of the system is not affected by this action of the controller.
The second kind of changes occurs in many simulated scenarios. The most challenging situation of this kind is the one shown in Figure 25 where a Cloud provider experiences a sudden complete outage. If, like in Section 5.2, the system is using that Cloud to process requests it experiences a sudden degradation of its availability. In this case the controller redirects all traffic going to that Cloud to both the other two Cloud providers in order to make them scale and restore the computing capacity. This sudden failure on a Cloud provider that the system is using is the most difficult scenario in which to satisfy the availability requirement.
The maximum time needed to bring back the availability of the system to the set point can be found by applying the equation used for error convergence in [7]. Assuming that the system converges when $e(k) \leq \epsilon$, then this happens when

$$k \geq \log_\beta \frac{\epsilon}{e(0)}$$  \hspace{1cm} (5.4)
where $e(0)$ is the initial error. By taking as initial error $e(0) = 1$, i.e., the worst case, $\beta = 1/3$ and setting $\epsilon = 0.01$ as the convergence tolerance, we obtain:

$$k \geq \log_{1/3} \frac{0.01}{1}$$

$$k \geq 4.2$$

So we need 5 iterations of the load balancer controller. After every iteration the load balancer will be inhibited (cool-down state) by the autoscaling controller on the overloaded
node until the entire traffic incoming to the node can be satisfied. The entire traffic is satisfied before reaching the setpoint computed in Equation 4.8. In fact, it happens when

$$n^* = \frac{AR(k)n(k)}{SR(k)}$$

(5.7)

Therefore, taking again the worst case, it happens when

$$k \geq \log_\alpha \frac{n - n^*}{\bar{n}}$$

(5.8)
Figure 38: System Availability using exclusively Cloud 4 for the Smart City Emergency System Application

which gives

$$k \geq \log_{1/3} 0.2 = 1.47$$  \hspace{1cm} (5.9)

Therefore, given $u = 0.8$, $\alpha = 1/3$ and $\beta = 1/3$, the autoscaling controller will exit the cool-down state after 2 iterations. Given that we considered the mean time for a VM machine to boot up to be 100s and observation windows of 60s, in the worst case our dual layer controller will converge in

$$t = 4 \times 2 \times 160s = 1280s$$  \hspace{1cm} (5.10)
that is, about 21 minutes. Based on $\alpha$ and $\beta$ parameters, the CPU setpoint and the availability setpoint, this time can be modified according to the system requirements and risk analyses. Reducing the convergence rate will result in a more reactive control, but the system will be more sensitive to noise. An extensive analysis should be performed on the traffic trend, so to optimize this choice.
Figure 40: System Availability of the Smart City Emergency System with load balancer enabled

<table>
<thead>
<tr>
<th></th>
<th>Cloud 1</th>
<th>Cloud 2</th>
<th>Cloud 3</th>
<th>Cloud 4</th>
<th>Controlled</th>
</tr>
</thead>
<tbody>
<tr>
<td>C0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Controlled</td>
</tr>
<tr>
<td>C1</td>
<td>1</td>
<td>0</td>
<td>*</td>
<td>*</td>
<td>Controlled</td>
</tr>
<tr>
<td>C2</td>
<td>*</td>
<td>*</td>
<td>1</td>
<td>0</td>
<td>Controlled</td>
</tr>
<tr>
<td>Availability</td>
<td>98.81%</td>
<td>100%</td>
<td>99.09%</td>
<td>72.63%</td>
<td>99.21%</td>
</tr>
<tr>
<td>Cost</td>
<td>2,755.47$</td>
<td>3,178.48$</td>
<td>2,359.91$</td>
<td>1,776.83$</td>
<td>2,318.43$</td>
</tr>
</tbody>
</table>
Figure 41: Availability of the controlled system with set point to 5-nines
CHAPTER 6

CONCLUSIONS

In this thesis we delved into the application of control theory to self-adaptive software in the context of Cloud environments.

We defined a two layer controller to manage both the autoscaling policy of single nodes and the load balancing at run-time, reasoning on the defined models, which is kept alive and continuously updated at run-time. The layer dealing with the autoscaling is responsible of ensuring that the number of machines are enough to cope with the incoming workload, maintaining a user defined CPU usage desired level. The second layer is instead responsible of distributing the incoming workload so to keep the availability of the entire system over a user defined threshold, minimizing costs. This models start form the works\cite{5,7} that have been extended in order to fit in the particular environment of Cloud Computing. We also expanded the controlling approach by adding costs and other kind of constraints specific to the cloud domain to the model.

Results during experimental evaluation, reveal that our approach can indeed be valuable since even when dealing with Clouds with average low availabilities, the controller is able to take decisions at run-time and distribute incoming workload to Clouds so to cope with the user defined availability requirement and so to minimize costs. It turned out to be a valuable approach even in the case where Clouds offer high availability but different (possibly varying) costs, since the controller is able to move the workload to the cheapest one.
Future research will should first focus on a deep analysis of the parameters setting, studying, for example, what the optimal choices of $\alpha$ and $\beta$ would be given a certain workload trend.

Different classes of requests should be considered as well, we are now only focusing on the assumption of identical requests.

The algorithm should also be improved so to avoid temporal drops of the availability, like in Figure 18, when the maximum capacity constraint cannot be satisfied. The drops are due to the two controllers decoupling. A fast scale out is here obtained by overloading the node, disregarding the temporal drop. A better integration between the two controllers may solve this issue by maximizing availability while fast scale out is performed.

A further important improvement to be investigated is the estimate of future parameters. In our solution, in fact, we used the average value in the observation window to estimate each future parameter. A Kalman Filter could be a valuable solution since it is an algorithm which operates recursively on streams of noisy input data to produce a statistically optimal estimate of the underlying system state.

Furthermore, both model and simulation could be improved by providing more realistic descriptions and features, according to the current solutions offered by cloud providers, and simulating different scenarios as close to real cases as possible.

Finally, the system should be tested on applications deployed on real infrastructures so to compare results from simulations with the more challenging environment that will be in use in industrial scenarios.


VITA

NAME: Marco Miglierina

EDUCATION: Bachelor of Science in Computer Science and Engineering,
Politecnico di Milano, 2009

Master of Science in Computer Science and Engineering,
Politecnico di Milano, 2012

Master of Science in Computer Science, University of Illinois at Chicago, 2013

PUBLICATIONS: Miglierina, M.; Gibilisco, G.; Ardagna, D.; Nitto, Di E.: Model
Based Control for Multi-cloud Applications, Modeling in Software
Engineering (MISE), 2013 ICSE Workshop on, 18-19 May 2013.
To appear.

Co-author of deliverables D5.1 and D6.1 of the MODAClouds
European project (www.modaclounds.eu), 2013.

GRANTS: Travel grant of $1,200.00 for the ICSE 2013 conference, held in
San Francisco, from SIGSOFT CAP fund, April 2013.