Model-Based Availability Evaluation of Multi-Cloud Applications

BY

GIOVANNI PAOLO GIBILISCO
M.S., Politecnico di Milano, Milan, Italy, 2012
B.S., Politecnico di Milano, Milan, Italy, 2010

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 Buy, Chair and Advisor
Mark Grechanik
Pier Luca Lanzi, Politecnico di Milano

To my Family...

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

CHAPTER **PAGE**

LIST OF TABLES

LIST OF FIGURES

LIST OF FIGURES (Continued)

FIGURE **PAGE**

LIST OF ABBREVIATIONS

AMI	Amazon Machine Image
API	Application program interface
AWS	Amazon Web Services
DTMC	Discrete Time Markov Chain
EC2	Elastic Cloud Computing
IaaS	Infrastructure as a Service
IDE	Integrated Development Enviroment
LQN	Layered Queuing Network
MTTF	Mean Time To Failure
MTTR	Mean Time To Repair
PaaS	Platform as a Service
PCM	Palladio Component Model
QoS	Quality of Service
SaaS	Software as a Service
SLA	Service Level Agreement
UIC	University of Illinois at Chicago

LIST OF ABBREVIATIONS (Continued)

VM	Virtual Machine

SUMMARY

The emergence of Cloud computing architectures in last years has changed the way applications are delivered to users. The growing number of Cloud providers and companies that rely on this infrastructure is a sensible indicator of its popularity. Cloud Computing offers a cost effective solution to the problem of resource provisioning by giving developers access to a virtually infinite pool of resources in a matter of minutes. Usually Cloud resources are priced in a pay per use basis so Cloud users can maintain under control the costs of deploying their applications by utilizing only resources they need. The scaling capability of Cloud providers allows companies to change the size of their virtual IT infrastructure according to their needs.

One of the major problems faced by companies when deciding to move to a Cloud environment is the loss of control on the management of the IT infrastructure. Companies are worried of outages that can not be directly kept under control. In order to cope with this problem Cloud providers offer service level agreements with their users by explicitly quoting the availability that they guarantee to provide. Many Cloud providers offer a service level agreement availability value of 99.95%. Real data shows that the availability that Cloud users experience from their providers is much lower and in the order of 95%. Such a low value of availability can not be accepted by developers of critical applications that usually require a much higher value of availability.

In order to fulfill the availability requirement developers could replicate their application on multiple Clouds. This thesis targets applications whose components are redundantly deployed

on multiple Clouds at the same time. These applications are called Multi-Cloud applications. The aim of this thesis is to help developers to evaluate the availability impact of their design choices early during the development process.

We present here an extension to the Palladio-Bench modeling framework for the description of availability requirements of Multi-Cloud applications. Developers can annotate in the extended Palladio Component Model some characteristics of Cloud environments on which the application will be deployed. The annotated Palladio component model is then automatically translated into a Discrete Time Markov Chain that has been extended 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. Such analysis is achieved through a simulation tool implemented in Matlab capable of analyzing the model against user defined environment changes in order to retrieve availability measures for the entire system. Characteristics that define the simulated scenario includes application workload, components availability, costs and service rates.

Two different applications have been modeled and simulated using our approach in order to show how their availability varies with respect to different scenarios, like the sudden failure of a Cloud provider or a gradual degradation of its service. Experiments show that our model is capable of describing the most important aspects of the Cloud infrastructure and the simulation tool is capable of reproducing different failure scenarios.

SUMMARY (Continued)

This work has been developed in the context of the MODAClouds European project [1]

[1] www.modaclouds.eu

CHAPTER 1

INTRODUCTION

Cloud computing is both an emerging technology and an industrial reality. It offers application developers a completely new paradigm for building web based application and provide them to final users. Many companies already offer different kind of services and many more uses those services, sometimes even ignoring the fact that they are hosted on a cloud platform. Some very popular examples of these services are Dropbox [1], Gmail [2] and Office 365 [3]

This revolution in the way applications are provided to end-users has changed the way applications are built. Developers now need to use different tools and techniques in order to let their apps exploit the full potential of this environment. Companies have accepted this new technology and are starting to using it in order to provide higher quality services and save money.

The main advantage of cloud computing is its ability to change the size of the infrastructure that hosts an application according to user needs in matter of minutes [3]. Providers of the application are charged only for the resources that they use on a hourly base and have access to a potentially infinite pool of resources. One of factor that limits the diffusion of cloud technology

---

[1] www.dropbox.com

[2] mail.google.com

[3] office.microsoft.com

is the fact that at the moment no Cloud provider offer mechanisms or tools to guarantee the Quality of Service of deployed applications.

Another factor that makes cloud application development challenging is the choice of which cloud provider to use. Most of cloud provider offers proprietary APIs and different type of services. This heterogeneity makes hard moving from a cloud provider to another. If the development of the application has been done with a focus to a particular cloud provider moving the application to a new one might involve re-writing some application code, migrating large databases and manual re-deployment of the application. In some cases also a significant change in the architecture of the application is needed. These operations are usually very expensive.

The focus of this thesis is in the identification of proper modeling mechanisms that allow developers to asses availability properties of complex multi-Cloud applications. Availability is a non functional property of an application, it measures the portion of time in which the system responds correctly to user requests. Applications that require a high level of availability are called (business) critical applications. Usually when such an application fails for an extended period of time the provider of the application suffers a great loss. Companies running such critical applications are reticent to move to an infrastructure that is not completely under their control even if they could benefit of a much hiher degree of replication.

Cloud providers offer a service level agreement (SLA) that states the expected availability of the infrastructure the provide, if the deployed application respect some criteria. Usually the

provided availability of of 99.95% of up-time in a given year. Amazon EC2 SLA[1] states that: "AWS (Amazon Web Services) will use commercially reasonable efforts to make Amazon EC2 available with an Annual Uptime Percentage of at least 99.95% during the Service Year". If the availability requirement of the SLA is not met usually the provider of the cloud service grant some free usage quotas to damaged users. Again on Amazon the user "will be eligible to receive a Service Credit", which means, to run its application for free for a period of time that depends on the size of the occurred failure.

Windows Azure SLA[2] offers 99.95% availability of internet connectivity and 99.9% of uptime of users virtual machines evaluated on a monthly basis.

A study conducted by Bitcurrent [1] shows that real availability values of cloud providers are much lower. For example the average availability of Amazon European region for the period of time of the study was 96.32%, Windows Azure service offers an even lower availability value of 95.39%.

Since critical applications needs a very high level of availability moving them to a cloud environment is a very risky choice. In order to increase the availability of a system usually critical components are replicated. We may use the same strategy by replicating the whole critical application on multiple cloud providers. This strategy works because failures between cloud providers are independent.

---

[1] http://aws.amazon.com/ec2-sla/

[2] http://www.microsoft.com/en-us/download/details.aspx?id=24434

This thesis has been developed in the context of *MODAClouds*[1], a European community project that aims to ease this choice of committing to a single cloud provider by uniforming the way developers access Cloud resources allowing applications and companies to freely move from one Cloud to another or even use mixed solutions.

1.1 Thesis objectives

The objective of this thesis is to contribute to the development of quality of service analysis of component based applications in the context of Multi-Cloud architectures, with a special focus on availability requirements.

To reach this goal, a model capable of describing availability requirements of Multi-Cloud applications has been developed. We have extended the already existing integrated modeling environment Palladio Bench to model Multi-Cloud applications using our novel paradigm. Finally, we implemented a tool to create simulated environments and test applications behaviors on different scenarios. We present here two different use cases that show different aspects of Cloud infrastructures that can be modeled and simulated: a web system scenario with two single-region Clouds, and a multi-region scenario.

1.2 Structure of the thesis

This thesis starts by introducing some background concepts in Chapter 2. In particular the Chapter introduces the concept of non functional requirements and of availability. It also introduces the Discrete Time Markov Chain is introduced a model useful to analyze availability

[1] http://www.modaClouds.eu

of component based software systems. It then proceeds by analyzing the most important characteristics of the Cloud computing environment.

Chapter 3 introduce some tools that can be used in order to evaluate some performance metrics of software systems and a tool called Palladio Bench that offers an integrated development environment to design model based application and asses their performance with a static analysis or a simulation based approach. None of these tools offers specific support to applications deployed in a Cloud environment.

Chapter 4 Shows the extension proposed in this thesis for the DTMC model. In particular how these model has been specified and extended to incorporate some Cloud specific concept.

Chapter 5 shows the two tools that have been implemented during this thesis. The first one is an extension to the Palladio Bench IDE that enables application developers to automatically generate models described in Chapter 4. The second tool is a simulation engine capable of simulating the designed application under user define working conditions.

Chapter 6 shows the results of two applications modeled by means of the extended Palladio Bench and evaluated by the simulation engine.

Chapter 7 make some considerations on the obtained results, highlights some possible future works and concludes this thesis.

CHAPTER 2

BACKGROUND

In this chapter we will provide some definitions and the minimum background knowledge required to better understand our work and to build a common lexicon, since for some terms there might not be well-established meanings.

Section 2.1 presents the cloud computing environment by showing some of its complexity and introducing some of the main features that makes it so attractive. Section 2.2 introduces the subject of non functional requirements. Section 2.3 introduces the popular model of DTMC and extends it with rewards in order to model a cost function. Sections 2.4 and 2.5 shows some of the main problems that affects the cloud computing environment. Section 2.6 explains the important autoscaling feature of cloud provider that, along with low costs, makes cloud computing one of the most attractive environment to run application. Sections 2.7, 2.8 and 2.9 respectively give an overview on Infrastructure, Platform and Software as a service. These are the three main fashions in which cloud computing has been described in literature.

2.1 Cloud Computing

Cloud computing is a technology born by the idea of companies with a strong IT infrastructure of renting some of their computing capacity when it was not needed by the company itself. Some big companies like Amazon which core business is based on the internet have to carefully build an infrastructure capable of coping with the changing number of users of their

system. Amazon noted that its online store was overwhelmed of users in the period before the Christmas holidays. In order to cope with this huge amount of users Amazon decided to build huge data-centers. This very expensive infrastructure was fully used during very short period of time and underutilized for most of the time. In order to reduce costs Amazon developed EC2, a mechanism that allows the elastic acquisition of resources. Amazon then decided to make it public and rent some of the computing capacity that was not used.

The main advantages of using a cloud infrastructure from a developer's point of view is very low, or in many cases null, start-up costs. In order to build and test an application on some of the major cloud providers (e.g. Amazon, Azure and Google app engine) users are granted a free usage quota. Another advantage is the fact that the size of the infrastructure hosting the application can be increased and decreased in matter of minutes. This allow application developers to overcome the problem of rapidly chaining workloads while avoiding over-provisioning at the same time.

Since cloud computing is a relatively new technology no strong standardization has been done. In this section we will show some of the major cloud providers highlighting similarities and differences among them. We will see that there is a hughe number of different cloud services. Some of them are almost identical but uses different APIs some other offer similar functionality but in very unique ways. The aim of this section is to make the user aware of the difficulties in writing portable cloud applications. The section ends with a brief overview of some of research work in progress to ease this migration process.

Authors of[1,4] claims that Cloud computing may refer to application and services delivered over the Internet as well as hardware and software that provide those services in the data centers. Many different kinds of Cloud services has appeared over the last few years such as Software, Platform, Infrastructure, Storage, Data, API and much more. Understanding the difference among this services and choosing the best combination of them is already a hard choice, but this is not the only hard task. This services can be offered in many different ways such as *private*, *public* or *hybric* Clouds.

Private Clouds are data-centers hosted by the same company that uses them, they are not available for public use and the exploitation of cloud technology is done in order to ease the maintenance of the IT infrastructure. *Public Clouds* are available to the public on a pay-as-you-go basis. The physical data-centers in which the computation takes place is managed by the cloud provider while application developers rent some of the computational power. *Hybrid Clouds* is a mixture of the two previous solutions. In this case part of the IT infrastructure is hosted by the provider of the application, usually the most critical part or a part that process sensitive data, while another part of the infrastructure is built using public cloud technologies.

One of the most common categorization of cloud computing technologies is this one:

- **Infrastructure-as-a-Service (IaaS)**: The provider allow users to rent some of the hardware it maintains. The way users access to this hardware resources is through virtual machines (VMs), users are allowed to choose froma pool of virtual machines or to upload custom VMs. The cloud provider is then only responsible for hardware maintenance and management of the virtualization middleware. Managing the operating system, applying

updates, configuring the application stack is responsibility of the cloud user. The usual pricing model for this kind of service is based on the size of the hardware resources acquired and the time of usage. Examples of this kind of services are: *Amazon EC2* [5], *Rackspace Cloud* [6] and *Terremark's cloud* [7].

- **Platform-as-a-Service (PaaS)**: It is usually built on top of a IaaS solution, it offers users a higher level of abstraction since some parts of the software management, like OS management, is done by the cloud provider. Cloud users only need to provide their application code to the system and the choice of appropriate hardware resources and their management is delegate to the cloud provider. This allow developers to put more effort in building better application rather than maintaining them. One of the main limitation of this kind of services is the fact that they only support a limited number of programming languages. Examples of this service are: *Salesforce's Force.com cloud* [8] and *Google's App Engine* [9].

- **Software-as-a-Service (SaaS)**: is the most abstract level of service. Providers of this kind of service offer users complete applications, that are built on top of PaaS or IaaS solutions, and allow users to interact with them usually using a browser interface or programmatically through APIs. Usually limited functionalities are offered for free and monthly fees are charged to users that needs more advanced functionalities. Examples of this services are: *Google applications* [10], *Netsuite* [11], *Freshbooks* [12] and *Hotmail* [13] *Dropbox*.

Deciding which level of control is needed and therefore which Cloud service to use is a very important factor in the development of the application. IaaS models give a very high control on the resources used to host the application but the complexity of configuring and managing these resources could be very high. Some applications with particular behavior like very CPU-intensive tasks or highly parallelism could benefit from particular hardware resources that can only be accessed using this kind of technology, Amazon for instance offers machines with dedicated GPU to perform parallel computation. A drawback of IaaS is the fact that the application administrator has to manage the scalabilty of hardware resources. It has to identify changes in the workload and react accordingly. In a PaaS environment both VMs and scalability are managed by the provider. Usually PaaS providers have a set of VMs in a steady state than can be put in production in a matter of seconds. Users are not charged for this VMs until they get to use them, the effect of this mechanism from the cloud user perspective is an environment that reacts quicker to changes at a lower price. A possible drawback of this kind of services is the fact that usually a VM hosts code of many different applications so multitenancy problems could occur. To avoid this situation most of cloud providers implement artificial upper bounds called "governors" [1].

Another issue for cloud users is usually storage. In order to move a legacy application to the cloud or to build a new one usually a big amount of data has to be moved to a database hosted by the cloud provider. Each provider offer different kind of services, for example *Google's Bigtable* is very fast in retrieving data[1] but is slow in writing operations, *Amazon Simple Storage Service (Amazon S3)*[14] offers the possibility to choose in which data-center the database will

be store in order to improve locality of data and reduce overall latency. Such a geographical control is desirable also for some critical application for which the locality of data is subject to governmental laws.

2.2 Non-Functional Requirements

In this thesis we are taking care of those requirements that define how a system should be, not what the system should do in terms of functionality. Our interest is in the quality of service of an application. These kind of requirements are called non-functional and have not to be neglected since in some scenarios, especially for critical applications, a system should not just work *sometimes*, or *eventually* give the result, but there are strong quality constraints that have to be satisfied. A detailed description of these quality measures can be found in [15]. In [16] they are defined as:

- *Usability*, which is highly related to the user experience and the ease in using the application.
- *Reliability*, which can be defined as the probability that a functional unit will perform its required function for a specified interval of time under stated conditions. The most common reliability parameter is the *mean time to failure* (MTTF).
- *Maintainability*, that is the ease with which a product can be maintained. Its basic measure is the *mean time to repair* (MTTR).
- *Availability* is also a very important non-functional requirement, 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 $\frac{uptime}{uptime+downtime}$, or else, identically, as $\frac{MTTF}{MTTF+MTTR}$.

2.3 The Discrete Time Markov Chain with Reward

Discrete Time Markov Chains (DTMC) are a useful formalism to describe systems from the reliability viewpoint. Authors of[17] describe DTMC as graphs. Nodes are used to represent *states* in which the system could be, edges are used to represent the possible *transitions* between different states. Each edge exiting a node is characterized by a number between 0 and 1 that represents the probability of going through that transition given that the system is in the state it starts from.

A state represent a possible configuration of a system or a possible point of its executions, a transition specify the possible evolution of the system from a state to other states. The DTMC contains also a particular state called initial state and a set of final states. Final states are usually, although it is not mandatory, represented by absorbing states without exiting transitions, or with just one self loop transition with probability 1. When a system enters a final state its execution is terminated.

Formally, a DTMC is a tuple $(S, s_0, \mathbf{P}, L)$ where:

- S is a finite set of states
- s_0 is the *initial state*
- $\mathbf{P} : S \times S \to [0, 1]$ is a stochastic matrix (i.e. $\forall s_i \in S \ \sum_{s_j \in S} \mathbf{P}(s_i, s_j) = 1$)

- $L : s \to 2^{AP}$ is a labeling function that marks every state s_i with the Atomic Propositions (AP) that are true in s_i.

In order to make the DTMC more expressive states and transitions can be marked with rewards. Rewards are numbers that can be used to model costs or benefits. Whenever the system enters in a state s the reward attached to that state is added to the total reward amount.

This models have been extensively used to deal with component based systems. Cloud based systems can be seen as a part of component based system in which a component represent a computational unit that can be hosted in-house or on a cloud provider. Transitions can then be used to model the flow of a user request between the different component of a system. In order to model failures of the critical components we need to add a particular final state called failure state. All requests that flow through a broken component will reach this state and can be counted. The probability attached to the transition that goes from a state to a failure state can be set measuring the success rate of that single component. In this contexts rewards can be used to measure the cost of using a service.

Availability is expressed in DTMC as a reach ability property. It is a formula constraining the probability of reaching a final state, in this case the success state. Given that S_R is an

absorbing state, the vector $\bar{x}$ whose entries x_i correspond to the probabilities of reaching s_R from state s_i is computed as solution of the linear equation system in variables $\{x_i | s_i \in S\}$:

$$x_i = \begin{cases} 1 & \text{if } s_i = s_R \\ 0 & \text{if } s_i \neq s_R \text{ is absorbing} \\ \sum_{s_j \in S} \mathbf{P}(s_i, s_j) \cdot x_j & \text{otherwise} \end{cases} \tag{2.1}$$

thus the item x_0 corresponds to the probability of reaching state s_R from the initial state. If we consider s_R as the success state then the formula give us the availability of the system.

2.4 Availability in the cloud

Moving to the Cloud means to rely on the infrastructure of the provider. Developers of critical applications usually require very high level of availability and usually do not want to loose the control over their infrastructure in order to reduce costs. As seen in Section2.1 most of cloud providers offer 99.95% of availability in their SLAs. Real data shows that the actual value of availability of these providers is much lower. Table I shows the values of availability of major cloud providers as in[1].

Some of the problems that affect the cloud providers are due to the size and complexity of their infrastructures. Usually cloud providers manage multiple data-centers worldwide in different geographical regions, copies of users data are stored for backup and synchronized automatically from the infrastructure. Complex virtualization mechanisms are put in place in order to allow multiple users to share common resources. A failure in the management of

TABLE I: AVAILABILITIES OF CLOUD PROVIDERS FROM[1]

GoGrid	96.33%
Google App Engine	93.05%
Joyent	94.87%
Rackspace CloudServer	96.33%
Windows Azure	95.39%
EC2 APAC	95.61%
EC2 EU	96.32%
EC2 US-East	96.42%
EC2 US-West	95.80%

these systems does not only affect the cloud provider but all the companies that provide their applications using its infrastructure. Some big failure of could providers are reported here:

- Amazon S3 Availability Event happened on july 20th, 2008. It lasted 8 hours and affected US and EU data centers.[1]
- Gmail suffered of a major outage on february 24th, 2009. For near two and a half hours many users could nor reach their mailing accounts.[1]

---

[1]http://status.aws.amazon.com/s3-20080720.html

[1]http://googleblog.blogspot.it/2009/02/current-gmail-outage.html

- Amazon suffer of an outage of the Relational Database service on april 21th, 2011. It affected many popular sites like Foursquare, HootSuite, Quora and Reddit that suffered of bad availability and increased responsetime. [2]
- Hotmail suffered of an outage on December 31th, 2010. Lasting more than three days it left empty in-boxes to many users. [3]

The size and complexity of cloud providers infrastructures makes it hard to fix such problems in a short time, even if the failure is found very quickly. In order to compensate the loss of service usually cloud provider grant some free usage quotas to users that have been affected by the failure.

2.5 Cloud Portability

As stated in [18] one of the main challenges for the long term success of cloud computing paradigm is to avoid the vendor lock-in that is currently happening among cloud providers.In order to do that we need to abstract the programmatic differences among providers, develop a way to move applications from local servers to cloud servers or to run in an hybrid context, unify communication between providers both at application level and data storage level and create a common management system capable of abstracting cloud providers architectural differences. This is a very difficult challenge, mainly because it requires a standardization effort of systems

---

[2] http://www.crn.com/news/cloud/229402004/amazon-ec2-goes-dark-in-morning-cloud-outage.htm

[3] http://www.crn.com/news/cloud/228901610/microsoft-windows-live-hotmail-back-after-e-mails-inboxes-disappear.htm

that are already in place, as explained in [19]. This thesis aims at analyzing the behavior of an application, developed on a such a unified environment, that exploits multiple cloud providers. For this reason portability and interoperability features are taken as prerequisite for out work. In particular this features can be divided into three levels:

- Programming level: Applications can be moved from one cloud provider to another without the need of re-writing code or reconfiguring the application manually. Since we are dealing with runtime adaptation of the application this is a basic prerequisite. This is not an easy task because it does not only involve the adoption of a common programming language, java is currently supported by almost all cloud providers, but also the development of standardized libraries and interfaces to access data, the definition of a common ontology of cloud resources and APIs to use them.
- Monitoring level: Monitoring of QoS properties of different components of an application is crucial, so standardized metrics and monitoring tools across different cloud providers are necessary. This involves the ability to retrieve metrics both on the utilization of cloud resources (e.g. CPU of VMs) and of quality of service provided by those resources (e.g. availability). Another characteristic of cloud provider that should be standardized is the pricing model, since different cloud providers charge users based on different metrics (network usage, I/O accesses, CPU hours) it's very hard to keep track of all of these aspects of the application and predict exactly the cost of deploying on a provider with respect to another.

At the programming level there are many attempts to create a set of open APIs that aim to hide the differences between cloud provider specific APIs and give access to features like blob storage or queues that are common to many providers, but none of them has been capable of providing sufficient functionality and at the same time exploit each cloud provider peculiarities. Examples of this APIs are jClouds (Java), libcloud (Python), Cloud::Infrastructure (Perl), Simple Cloud (PHP) and Dasein Cloud (Java).

2.6 Scaling

The major characteristic of Cloud computing is its ability to change the size of the infrastructure according to user needs. This ability is usually called *scaling*. If a company that provides a service decides to host its own physical servers it usually acquires a pool of resources that is able to process a fixed amount of requests. In modern computing systems the incoming workload may change very quickly. Having an infrastructure whose size is fixed causes two type of problems. When the workload reaches a point in which the system is overloaded with requests some of them are rejected. This problem is called under provisioning and causes a loss in the availability of the system. In order to solve this problem it is necessary to acquire new hardware resources, configure and add the to the pool of working machines. This process is usually very long and expensive. It is very rare that after a burst of requests the workload stabilizes at the highest value, usually it returns to the previous level. If a company decides to purchase equipment in advance in order to be able to cope with bursts of requests it will incur in an opposite problem called over provisioning. From the point of view of availability over provisioning is not a problem because the system has always enough computational capacity

to serve all user requests. From the economical point of view over provisioning is a very serious issue. The capacity offered by cloud computing of modifying the size of the infrastructure according the incoming workload in matter of few minutes can effectively solve these problems.

Cloud provider offer different level of services at different prices. As an example in the context of IaaS, the one that let users control scaling mechanisms freely, users can choose between two kind of scaling policies:

- *Vertical scaling* consist in changing the performance of a single cloud resource (e.g. a VM). This process can be done without service disruption in some cases(e.g. adding a new HDD to a virtual machine), but requires the restart of the cloud resource in many other cases (e.g. adding a new virtual core or using a more powerful core). item *Horizontal scaling* consists in changing the number of replicas of a cloud resource. In the context of VMs when a scale out request is performed new virtual machines are booted up and added to the pool of running VMs.

It is not always easy to choose between vertical or horizontal scaling. Horizontal scaling is usually done when the system receives a very high number of requests, new VMs can take some of the incoming workload in order to reduce the load on the overall system. If the system is in a steady state in which machines are not saturated and the queues are not very populated then horizontal scaling will probably not affect the response time of the application. Vertical scaling on the other hand is capable of reducing the time needed to generate a response to a user request since it directly affect the processing time of each request when the system is under a light load.

The process of reducing the size of the infrastructure is called scale down or scale in. It is usually done in order to reduce costs when the system is in an under utilization state. While an action of scale out can not decrease the quality of the service, an action of scale in could reduce the size of the infrastructure to a point that it causes a loss of availability.

To distribute the incoming workload among this elastic pool of resourced cloud provider offer a load balancing service. This is a critical part of the system and its management is usually delegated entirely to the cloud provider. Cloud providers use very highly specialized hardware in order to perform load balancing effectively. Some providers also allow users to customize load balancing rules. A representative example of a load balancing service is shown in section 2.7.1.

Another service needed to effectively use the scaling capability of the cloud environment is *monitoring*. In order to identify when a scaling action is needed application administrators have to monitor the load of the cloud resources that are processing user requests. Monitoring can be done at many different levels. Application level monitoring can be done using instrumented code, system level monitoring is sometime offered from cloud provider in order to retrieve the usage level of virtual hardware components (e.g. the utilization of a virtual core). The most common metrics provided by all cloud providers are the network traffic and the CPU utilization. Some other metrics are very specific to the type of cloud resource that is used (e.g. number of queries on a cloud database).

2.7 Infrastructure-as-a-Service (IaaS)

This section presents some of the most popular cloud providers and the services they offer.

2.7.1 Amazon EC2

The IaaS platform offered by Amazon is one of the most mature and hosts many popular services like Dropbox or Reddit[1]. Users access cloud resource using virtual machines. Amazon offers a web based tool to manage the provisioning and configuration of virtual machines. APIs are also available to build custom management tools. Amazon allows users to:

- launch virtual machines from a predefined set (including major Linux distributions and Windows Server) or custom images by uploading an *Amazon Machine Image (AMI)*;
- configure security and network access to virtual machines;
- choose instances type for every virtual machine, as listed in TableII;
- choose the location of virtual machines between seven different regions, manage IP end-point and block storage attached;
- automatically mange load balancing between active machines
- build custom scaling rules
- integrate storage with the Amazon S3 service

Users are charged for the amount of resources they require on a hourly base. Pricing is applied to computing instances, data transfer and storage. Beside using pre-configured system images users are allowed to build custom images with all the software needed to run their applications. These images can then be uploaded in the Amazon cloud system building Amazon

---

[1]http://aws.amazon.com/solutions/case-studies/

Machine Images (AMI). Some pre-configured AMIs are made available by Amazon with many popular software already installed. Some of the most popular software available are:

- IBM DB2
- Oracle Database 11g
- MySQL Enterprise
- Microsoft SQL Server Standard 2005
- Apache HTTP
- IIS/Asp.Net

Beside the software configuration Amazon offers also multiple virtual hardware configuration for VMs. They differ in number and speed of cores, amount of RAM and storage. A particular kind of virtual machines with attached GPU is available to perform efficient parallel computation. Table II shows different instance types.

TABLE II: AMAZON EC2 INSTANCES TYPES

Type	**Subtype**	**Memory GB**	**Compute Power ECU**	**Storage GB**
Micro	Micro	0.613	up tp 2 ECU	EBS only
	Small	1.7	[1,1]	160
Standard	Medium	3.75	[2,1]	410
	Large	7.5	[4,2]	850

	Extra Large	15	[8,4]	1690
Second Generation	Extra Large	15	[13,4]	EBS only
	DoubleExtra Large	30	[26,8]	EBS only
High-Memory	Extra Large	17.1	[6.5,2]	420
	Double Extra Large	34.2	[13,4]	850
	Quadruple Extra Large	68.4	[26,8]	1690
High-CPU	Medium	1.7	[5,2]	350
	Extra Large	7	[20,4]	1690
Cluster-CPU	Quadruple XL	23	[33.5,2]	1690
	Eight XL	60.5	[88,4]	3370
Cluster-GPU	Quadruple XL	22	[33.5,2] + 2 NVIDIA Tesla M2050 GPU	1690
High I/O	Quadruple XL	60.5	[35,16]	1024 SSD

Different pricing policies can be chosen by users depending on their needs. The pricing options offered by Amazon are:

- **On-Demand**: Users are charged only of the resources they use on a hourly base without lont term commitment or any advanced payment;
- **Reserved**: Users make an advanced payment in order to get a discount on the hourly price of instances. Usually the commitment of the advanced payment grants discounts for one or three years;
- **Spot**: Users can make a bid to acquire computing instances and, if the bid exceeds the price set by Amazon they acquire the resources. If the price exceeds the user bid resources are deallocated. This pricing policy allows users to pay less for extra resources but does not ensure the continuity of service. On the other hand when Amazon needs more resources it can raise the price of this VMs in order to exceed some of the users bids and free up resources.

Amazon also allows users to build an hybrid cloud using a *Virtual Private Network (VPN)*. In order to do so Amazon offers a set of isolated virtual machines that run on hardware dedicate to a particular user, this service is called *Virtual Private Cloud*. Existing IT infrastructure can be connected directly to the Virtual Private Cloud. The VPN connection is priced as well on an hourly basis.

Amazon also offer a IaaS level storage service called Simple Storage Service (S3). It can be used as remote storage or attached to VMs. Objects stored in S3 are redundantly copied in different facilities in the region selected by the user. Copies are kept synchronized automatically by the management system that uses operation like PUT and COPY to write synchronously

on all the available copies of user data. Periodic checks on the integrity of data is performed via checksum. amazon offers two types of storage services:

- **Standard Storage**: Ensures high replication of user data and provides 11-nines durability and 4-nines availability of users objects over a year. This critical storage has been designed so that even the concurrent failure of two facilities does not cause a loss of data. A versioning service is also available.
- **Reduced Redundancy Storage (RRS)**: Is a less redundant service that provides 4-nines durability and 4-nines availability. It can support the loss of a single facility without losing user data.

In order to add or remove resources to their infrastructure users can create *autoscaling groups*. These groups are composed of homogeneous VMs that are used to process user requests. Load balancing of incoming traffic is done automatically by the Elastic Load Balancing service that splits equally the traffic among all the available VMs. A monitoring service periodically checks the health of VMs in an autoscaling group and when a VM does not respond to the monitoring request it is marked ad broken and removed from the autoscaling group. The monitoring service constantly pings broken VMs in order to add them back to the autoscaling group as soon as the problem is solved. The monitoring service operates at an operating system level so it is not able to recognize application errors that may cause the VM to fail in replying to user requests.

2.7.2 Rackspace Cloud

A service similar to Amazon Ec2 offered by Rackspace is called *Cloud Servers*. As in Ec2 Rackspace offers a web based tool manage VMs. The main difference with EC2 is thet fact that Rackspace offers the support of a specialized team that can help users to deploy and manage their instances.

Another key difference is the fact that Rackspace machine images are persistent. If a machine is rebooted all files are kept as they were before like in a normal computing system while in an Amazon image when a machine is terminated all changes are lost and the new machine is launched from the scratch image that was created by the user or selected among the preconfigured AMIs. Rackspace also offers a persistent public IP address for each VM while Amazon uses dynamic private IP addresses under a NAT. Other differences between the two providers can be found in [20]

Rackspace offers also different solutions storage:

- *Cloud Files* is an object storage solution that uses the Akamai CDN [21] to distribute contents over the web and uses triple replication.
- *Cloud Database* is built to offer a High-performance MySQL database.
- *Block Storage* is a low level storage solution that can be attached to VMs, it offers SSD or SATA disks.
- *Backup* offers file level backup for servers in the Cloud.

2.7.3 Terremark Cloud Computing

Terremark is another IaaS provider that offer services similar to those presented in previous sections. In particular Terremark offers two IaaS:

- *vCloud* is an easy to use cloud service designed for fast development of cloud solutions.
- *Enterprise Cloud* is a more complex service designed to host enterprise-wide applications with fine tuned control on scaling, performance and security.

Both these tools are accessible via a web interface and through APIs. Terremark uses VMWare virtualization technology to manage their datacenters, as Rackspace it offer persistent VMs. A particular feature of this cloud provider is its ability to change the characteristics of user's VMs, like number of cores or amount of RAM without rebooting the machine.

Terremark does not offer a separate storage service, it provides VMs with one or more HDDs with variable sizes. Proprietary servers can be placed in Terremark's colocation service in order to form an hybrid cloud. It is also possible to connect existing enterprise IT infrastructure to its Cloud.

2.8 Platform-as-a-Service (Paas)

This section presents some PaaS environments.

2.8.1 Google App Engine

Google App Engine is the PaaS offered by Google. It allow users to write applications and run them on the hosted infrastructure without the need of managing server provisioning and configuration.

Google App Engine supports three programming languages: Java, Python and Go. The runtime environment of App Engine allows developers to use standard Java technologies like JVM and servlets or any language with a JVM-based interpreter like JavaScript or Ruby. Both Java and Python runtime environment are built in order to minimize interference among different applications that run on the same system.

As in IaaS payment is manage with a pay-as-you-go policy. There is no advance or recurring cost and users are billed for storage and bandwidth they use monthly. The paying system also allows the set up of a maximum monthly budget in order to avoid unexpected highly usage. The service also offers a free tier of 500MB of storage and an amount of processing power that can serve near 5 million page views per month.

Google's PaaS APIs allow access to many other products like Gmail to send automatically e-mails, spreadsheets and Google docs. There is also a persistent storage service, atomatic management of scaling and load balancing, queue for task and a scheduled event manager.

A virtualization environment called Sandbox ensure that applications run in a safe way by allowing them only limited access to the underlying operating system. This allows App Engine to manage efficiently the scaling of applications moving them between shared and reserved servers. The sand box also isolate application in a way that they are independent on the underlying hardware and operating system so that they can be moved to faster machines when it is needed. It does enforce some quality parameters to the application, as an example it allows only application that can respond to user requests in at most 30 seconds. Also the communication

capabilities are limited only to http/https ports. Internal communication between processes is manage using a task queue.

A peculiar characteristic of Google App Engine is its distributed data storage service. It is built in order to follow the growth of Web Servers and accommodate the increasing amount of data generated. Data objects are represented by *entities* that are described by a type attribute and a set of properties. The transaction engine can retrieve entities of a type filtered and sorted by properties. Data objects are schemaless and their structure is enforced by the code. Goggle App Engine supports Java JDP/JPA and Python datastore interfaces in order to enforce as much structure as needed withing the application. One of they key point of this storage system is that it provides consistency and a optimistic concurrency control. Actions that modify data can be grouped in a single transaction that is automatically repeated by the system if other processes are trying to update the same value.

2.8.2 Microsoft's Windows Azure Platform

Microsoft provides a PaaS called *Windows Azure Platform*. It is composed by different Cloud technologies that provide different services:

- **Windows Azure** is the main component that provides a windows based environment to run applications and store data;
- **SQL Azure** is a database service that can be used by cloud applications;
- **Windows Azure platform AppFabric** can be use to connect different components of the application.

Windows Azure allows developers to build their application in any language of the .NET Framework or in many other languages like Java or PHP. The data storage service is always accessed using RESTful APIs. It allows to store *binary large objects* (blobs), store and retrieve messages in queues in order to allow communications between application components and access the SQL database. Application developers can manage the services they use through a web based tool or via APIs.

2.9 Software-as-a-Service (SaaS)

This section introduces some of the most famous SaaS.

2.9.1 Google applications

Google applications is a well known SaaS provided by Google. The most known application in Gmail, a browser based mailing system. It also offers a complete suite of web based software for business most of which have a free counterpart. Among the software offer we can find:

- *Google Calendar* a calendar application that allows to synchronize calendars between devices, share calendars and manage notification of events.
- *Docs* offers a complete suite of office automation software. It is composed by a word processor, spreadsheets and a tool to build presentations.
- *Sites* is a web hosting service that allow easy creation of websites.

Google services can be accessed without charge for personal use, paid versions offer more advanced functionality. The *Google applications Marketplace* allows users to publish their google based web application that are hosted on Google App Engine.

2.9.2 Rackspace

Rackspace offers a web hosting application called *Rackspace Sites*. It hosts Wordpress, Joomla and Drupal servers in order to easily build and maintain websites.

2.9.3 Microsoft

Microsoft offers its *Business Productivity Online Suite*, a commercial suite of software that includes Exchange services, calendar, contacts management, SharePoint, Communications and Live Meeting. There is also a web based version of the well known Office suite called Office 365.

CHAPTER 3

EXISTING TOOLS AND METHODOLOGIES

We provide now an overview of some tools that are used to deal with the problems introduced in previous sections. Section 3.1 presents Palladio, a tool used to model an application in details, from a class diagram representing it's logical structure to an allocation diagram that represents its deployment onto physical machines. This tool can perform some transformation to the model of the application described by the developer team in order to build different models to evaluate non functional properties. Section 3.4 presents some other approach to the problem of evaluating QoS properties on component based software applications with a particular focuse on performance metrics like the response time.

3.1 Palladio-Bench

Palladio is an IDE based on Eclipse Modeling Framework developed and supported by Karlsruhe Institute of Technology (KIT), FZI Research Center for Information Technology, and University of Paderborn. As stated in [22] it provides different tools for each developer role allowing them to build separate diagrams describing some characteristics of the system to be. The tool then automatically integrates all these diagrams and generates models of the entire system to analyze some QoS properties at design time. In this section we will shortly describe basic procedures to model a system in Palladio-Bench and clarify its limitations in modeling a dynamic application in the Cloud, which is the subject of this thesis.

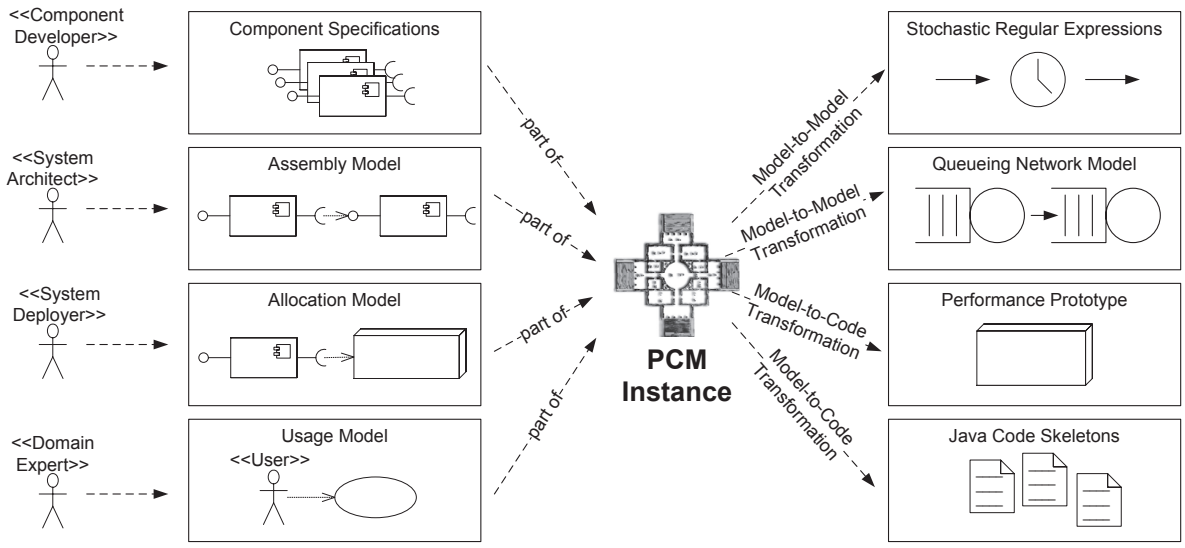


Figure 1: Palladio Component Model - Roles

One of the key point of the Palladio suite is its ability to clearly separate development roles, as shown in Figure 1 Palladio supports the design of the application by automating multiple steps each one performed by a different role in the development team, these roles are:

- Component Developer
- Software Architect
- System Deployer

- Domain Expert

3.1.1 Palladio Component Model

Palladio Component Model, presented in[22], is the core of Palladio-Bench, it is composed of four models that describe different aspects of the system and a usage model that describes users' behavior. The four system models are:

- Component Repository
- System Diagram
- Execution Environment
- Component Allocation

The *Component repository* diagram describes all the components of the software and their interfaces. It is built by *component developers* which specify required and provided features for their components. A component is the basic element of the application, it offers some functionalities and it may require some other functionalities to work, a simple example of a component could be the code of an application that replies to users' requests. This application may need to interact with a data base. In such case the component would require that another component implement a common database interface.

Component repository can include composite components, which represent subsystems, and additional information like failure state specifications, whose meaning will be explained later on in this section. This diagram can be divided in two main layers, the upper one represents interfaces, components and their provided/required relations, the lower one represents effects of

the implementation of provided interfaces by components. A diagram that specifies the behavior of a component while executing a certain function is called a Service Effect Specification (SEFF). A SEFF diagram consists of a chain of actions from a starting point to an ending one. To build this diagram the component developer can choose from many kind of predefined actions, the two most important are internal processing or call to an external service. Other actions include control like branches or loops. Internal actions are used to represent some processing that occurs inside the component, processing actions can be annotated with a failure type description with an attached probability. This attribute represent the possibility that something in the processing of the internal action goes wrong, this kind of failure refers to a software failure, not the failure of the hardware on which the component runs. Another important parameter that component developers can specify is the resource consumption. This parameter models the expected required use of hardware components from the functionality implemented by the module, it can specify the amount of resources required in terms of CPU and HDD. These annotations are used by Palladio when generating different models for prediction of QoS measures. For example the failure probability of an internal action is used to build a DTMC model for availability analysis while the resource consumption is used when building performance models. External actions are used when developing a SEFF to model calls to external services, when adding an external call action the developer is supported by Palladio that let users choose which external action to call within the pool of functionalities defined by the interfaces required by the component. An example of a very simple repository diagram with two interfaces and three components is shown in Figure 2

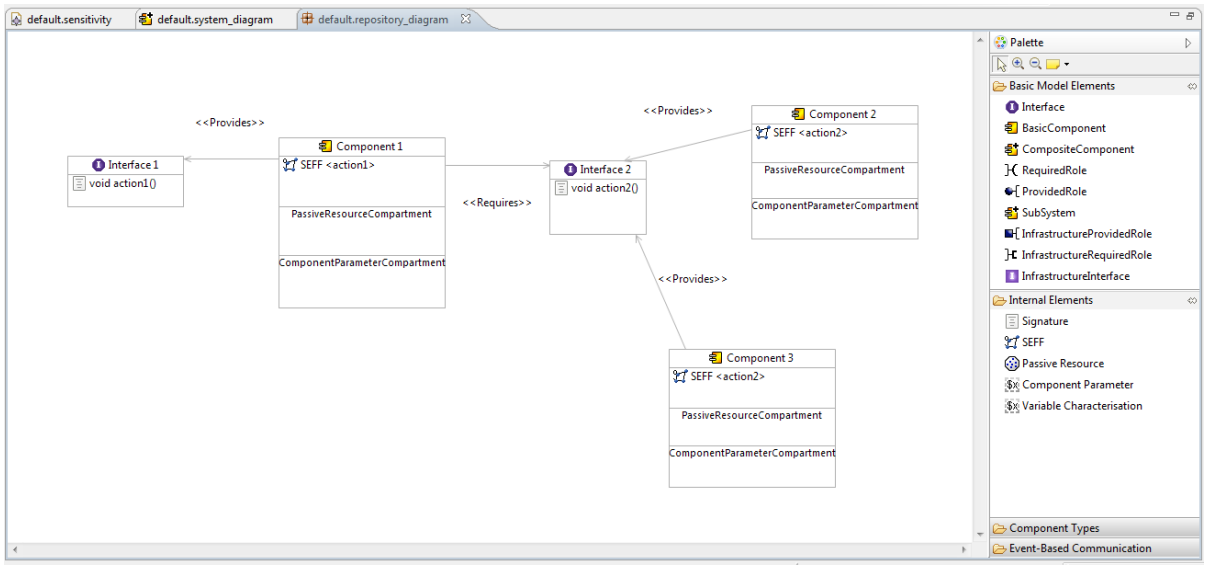


Figure 2: PCM - Repository diagram

The *System Diagram* is built by *software architects* which compose instances of the components from the repository into an architecture of the system. The system diagram can be specified after the repository diagram has been defined, this is due to the fact that components in the repository diagram represents classes while components in the system diagram represent instances of those classes.

Information about how a functionality is implemented is not useful when connecting components, the only information needed in order to connect two components is their required and provided interfaces. Software architects define assembly contexts for each component that will be used in the system and connect the required and provided interfaces of components defying the structure of the system. This diagram can also specify a provided role for the entire system

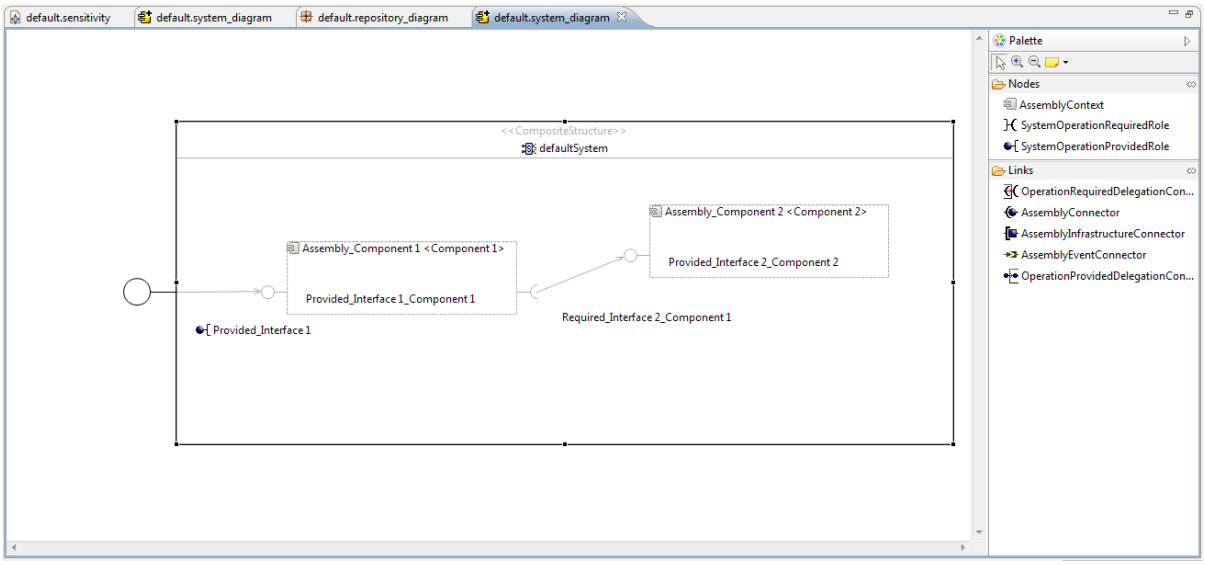


Figure 3: PCM-System diagram

which is the service that end users are actually going to call. An example of a system diagram is shown in Figure 3

The *execution environment* is defined by *system deployers* with a resource environment diagram which models the physical structure of the system by means of Resource Containers, with processing power, storage resources, and links. This diagram is used to model the environment on which the application will be deployed. An example of a resource environment is shown in Figure 4. In this diagram system deployers can also specify MTTF and MTBF of components.

The linking between the resource environment diagram and the system diagram is specified by the *component allocation* diagram that specifies which instance of each allocated component is deployed on each physical machine. In the very simple case of Figure 5 the execution environ-

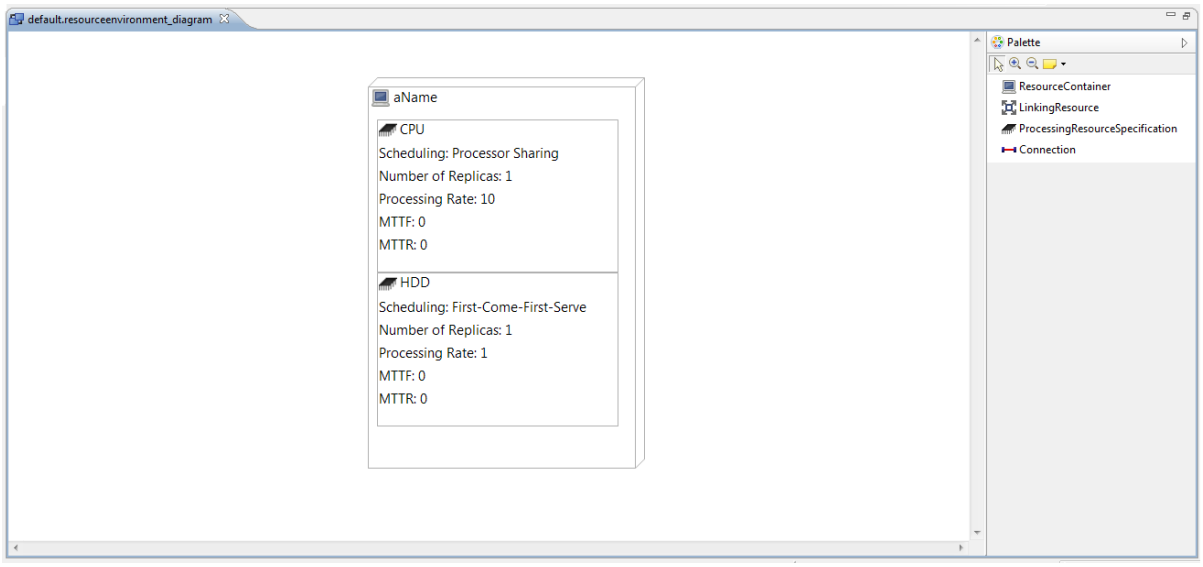


Figure 4: PCM-Resource diagram

ment specified in Figure 4 consists of a single resource container with a CPU and an HDD so the components specified in Figure 3 are allocated on this machine. More complex environments can include multiple machines networked together or machines with multiple copies of the same resource. Merging the SEFF diagram, the system diagram, the resource environment diagram with this diagram Palladio can derive actual resource usage in terms of CPU seconds or time to access the HDD for each function of each component.

Palladio let the developers team specify also a *usage model* diagram in order to model the behavior of the users of the system. This diagram is usually built by the *domain expert*. This diagram is used to generate model for performance prediction based on Layered Queuing

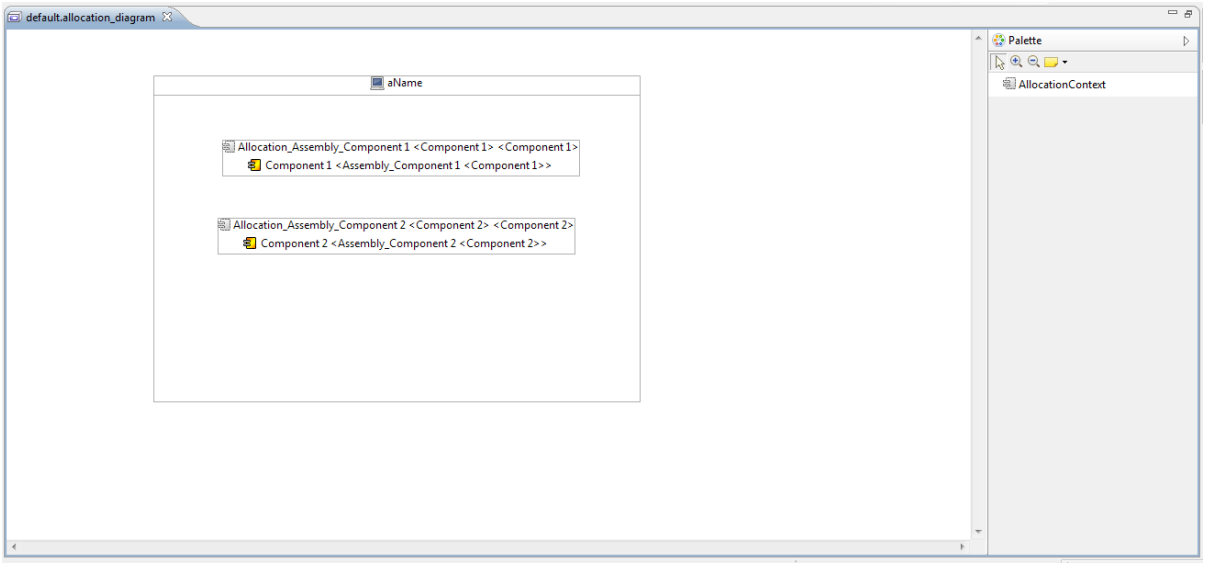


Figure 5: PCM-Allocation diagram

Networks since we are dealing with DTMC models this diagram will not be discussed any further.

Palladio offers some great features to develop a system so can be really useful when dealing with complex systems but it also has some limitations when dealing with the Cloud environment. Currently Palladio let system deployers create only server like entities in the resource environment diagram with resources like CPU, HDD and Network links. Since this simple entity is not suited to model Cloud systems (e.g. dynamic computing resource allocation and Cloud performance variability) which are much more complex we chose not to use it. The lack of Cloud entities for the deployment diagram can be associated to the lack of standardization in the Cloud environment shown in Section 2.1. Since we are dealing with availability measures

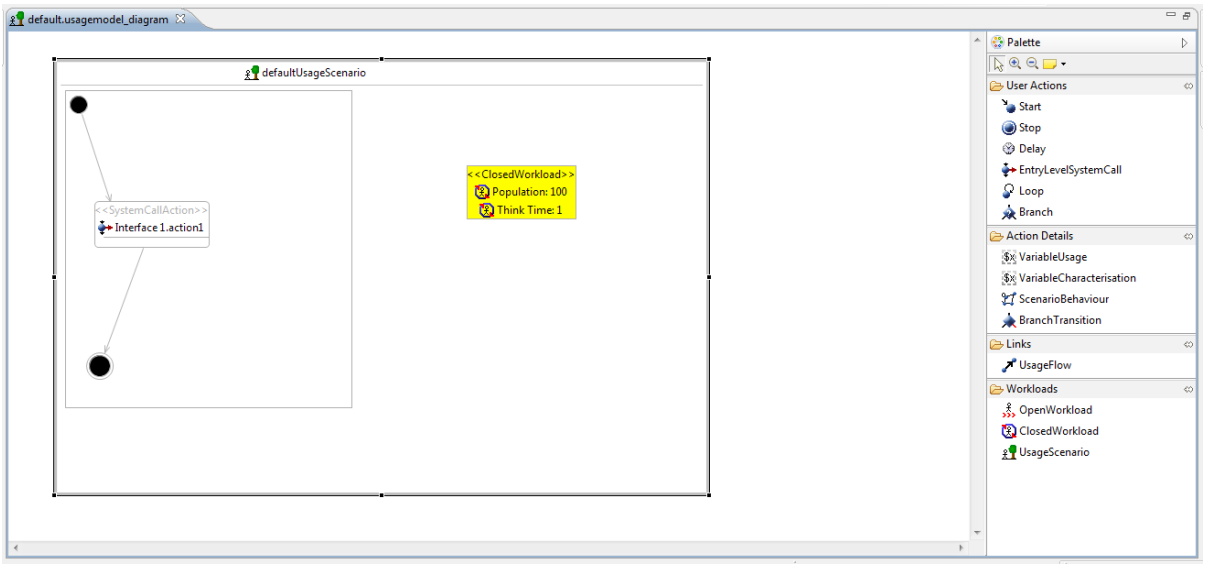


Figure 6: PCM-Usage diagram

our interest in processing power of machines is limited to the case of requests rejection due to an overload of the machine. We decided for simplicity, to model this aspect in another way by associating this information with the failure type description. Using this approach one does not need to add resource consumption specification in SEFF diagrams but just a failure probability. This kind of specification allowed me to separate the failure description inserted by the domain expert which model the failure of a service due to some external reasons to the failures due to the overloading of the machine its service run on which is managed with a queuing theory method explained in Section 5.1.

Another limitation encountered working with Palladio is the fact that each interface connector in the system diagram can be connected to a single providing component instance. This

has been done in order to avoid ambiguity that may arise by connecting more instances of the same components or, more in general, of components implementing the same interface, without explicitly deciding when to use one or the other. This feature can be implemented by specifying in the repository diagram an interface for each copy of the component one wants to connect. Then we can choose to have a component implementing in a similar way all these interfaces and replicate it inside the system diagram, or to have multiple components implementing each one a single interface and instantiate them just once. This approach moves the semantic choice of which service to call in case of multiple similar services into the SEFF diagram, which is much more expressive in terms of conditions on user input data. Also the system diagram is more readable and easy to build because when an instance of a component which requires multiple interfaces is created the number of components providing that interface is unambiguous. The drawback of this approach is the fact that if one wants to add two components providing the same functionalities to another component he/her have to build two identical interfaces and, if there are many components of this kind in the system, the deriving representation becomes large and not very easy to read.

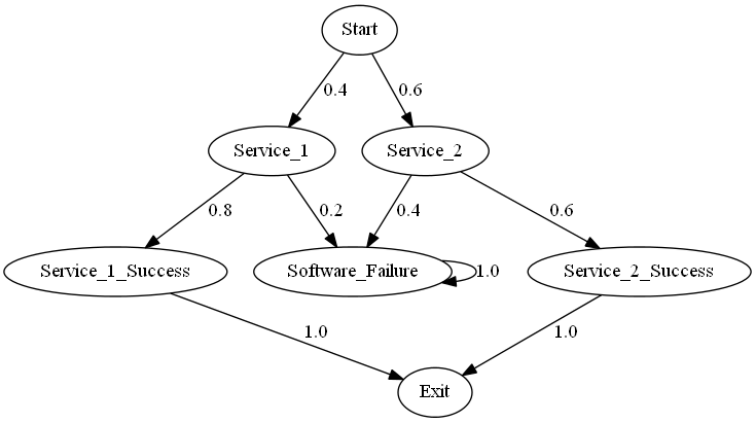
3.1.2 PCM transformations

The Palladio Component Model (PCM) defined by the diagrams of Section 3.1 are used by Palladio Bench as a starting point form different transformations. Depending on what the user is interested in Palladio Bench can transform the PCM model into different models, the most used are Layered Queuing Networks (LQN) and Regular Expressions. Both these models are used to derive performance measures from the model, in particular the LQN model can be

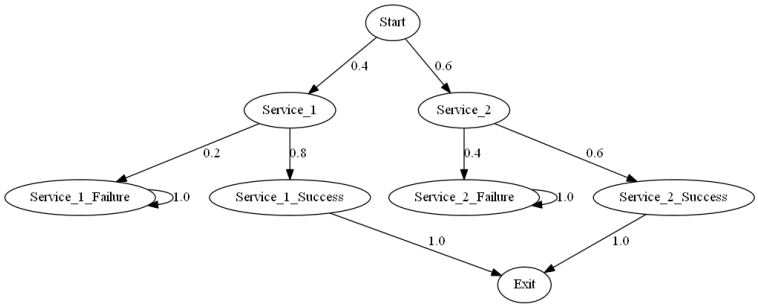
solved analytically or with a simulation tool integrated in Palladio. Even if not integrated in the final release there is also a transformation engine that allows to derive DTMC models from PCMs. Since this thesis deals with availability requirements this last transformation engine is the one that will be used for our purposes. The effect on the DTMC of using a single software failure type or multiple failure types during system design can be seen in Figure 7. In 7(a) the general software failure type has been used so the generated Markov model has a single failure type with many incoming arcs, while in 7(b) two software failure types has been declared. Using multiple failure types give more information about the failing component in the final analysis.

Other components like probabilistic branches and loops can be inserted in the SEFF diagram these structures are then transformed in different ways into the Markov chain. In particular probabilistic branches are translated as in Figure 8, since in Palladio it's not possible to define a probability for remaining in a loop but only a fixed number of iterations, the transformation of loops involve the loop unrolling procedure, the final outcome is shown in Figure 9.

By transforming the specified model into a DTMC, Palladio is capable of calculating the probability that the system ends in a success state and show the effect of the failure of each service specified with a failure type on the overall failure probability. In this way system developers can find major points of failure and focus their attention in reducing their probability of failure. The main limitation of the analysis performed by Palladio is the fact that it is a static analysis of the system. In order to overcome this limitation Palladio allows developers to specify a sensitivity file in which one can define some characteristics of the system as parameters and provide a range in which they can vary. An example of a parameter could be the probability of



(a) Single failure type



(b) Multiple failure types

Figure 7: PCM - Failure types

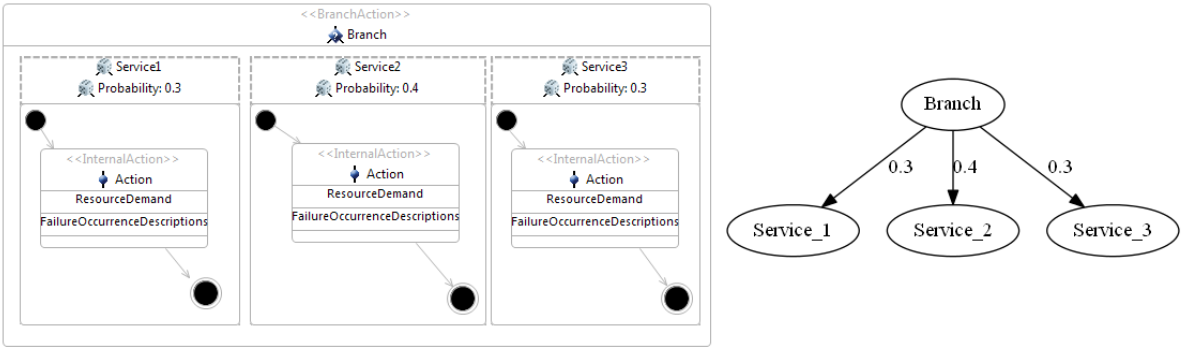


Figure 8: Branch conversion

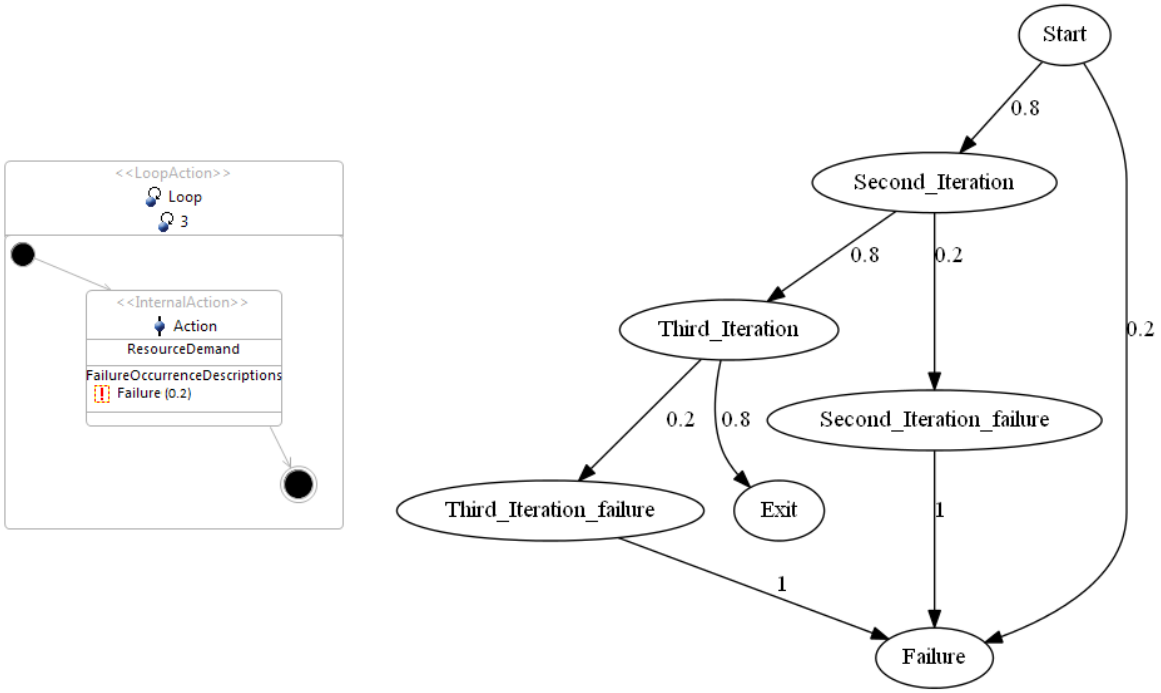


Figure 9: Loop conversion

failure of a system or the probability of taking a branch in a SEFF diagram. This sensitivity files are used by Palladio to run several iterations of the system evaluation by modifying one parameter at a time in order to build a final report. This approach is quite easy to use for small systems in which few variables can change. To model a complex system like the one in our use case described in Chapter 6 in which many parameters change over time a more versatile environment is necessary.

3.2 The Descartes Meta-Model

The Descartes Meta-Model (DMM)[2] is a modeling language developbed to specify QoS aspects of IT systems. Figure 10 shows the four sub models that compose the DMM.

The **Resource Landscape** meta-model is capable of modeling resources inside a datacenter. It allows users to model both physical and logical resources at different levels of abstraction. Detailed resource landscape models are used at runtime to detect environmental changes and trigger the adaptation.

The **Application Architecture** meta-model allow users to define component based systems. Each component can be modeled by means of its execution, calls to external services and some other aspects that regard the execution environment. Dependencies between parameters of the model can be described in a probabilistic way.

The **Adaptation Points** meta-model express the free space of the adaptation process. It describe the boundaries in which the adaptation can be performed.

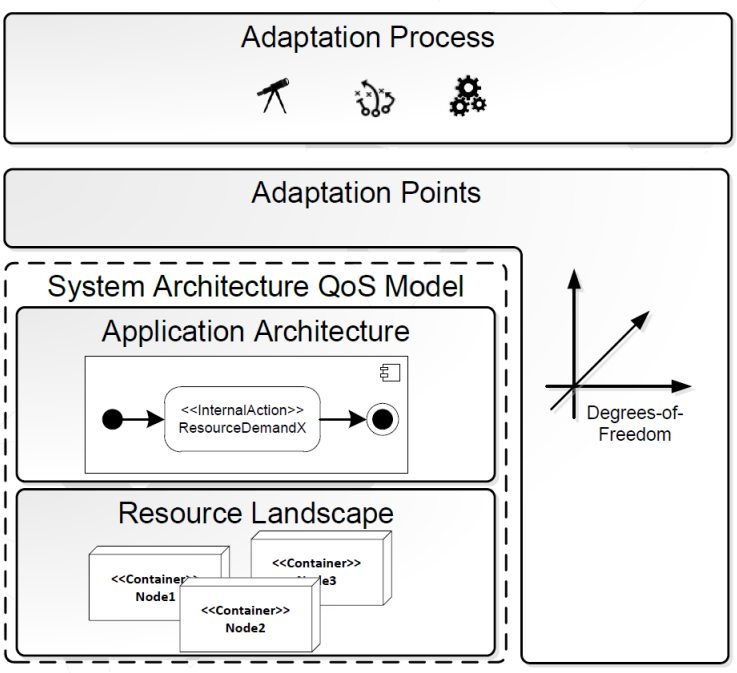


Figure 10: Descartes meta-model sub components [2]

The **Adaptation Process** meta-model describes how the adaptation is performed in terms of *Adaptation Strategies*, that are high level sequences of adaptation actions, and *Adaptation Tactics and Actions* that specify at a lower level how the adaptation is performed in the system

3.3 Model solving tools

Models described in this section are built to support the design of applications while allowing developers to easily insert information regarding the performance of internal components. This models are not suited for direct analysis of performance properties. An common approach is to automatically derive performance models from these models then use tools specialized in the

TABLE III: CHARACTERISTICS OF PRESENTED TOOLS

Tool	Domain	Constraints
Palladio	Enterprise systems	not specified
Descartes	Datacenters and Virtualized systems	specified as triggers for adaptation

analysis of these models to derive performance metrics. This section presents two class of tools, simulation tools and analytic tools.

3.3.1 Simulation tools

Simulation tools allows very accurate estimation of non-functional properties of the modeled system. One of the major advantages of these tools is their ability to estimate not only the mean values of many performance parameters, but also their probability distributions. This kind of methods allow also to avoid some of the limitation imposed by the assumptions made by analytic solvers, e.g. exponential distribution of service time or inter arrival times. The main disadvantage of this kind of methods is the fact that the system has to be modeled in details and the time needed for the simulation grows rapidly with the dimension of the system under study and the desired accuracy.

SimuCom[23] is a simulation tool implemented in the Palladio framework. It takes as input a PCM instance described in Section 3.1.1 and performs a model-2-code transformation to generate a prototype of the system. Components behaviors described in the instance of the PCM are used to derive the code for the behavior of the component mapping control flows to their corresponding Java constructs. External calls are mapped to direct methods

invocation. PCM allows the specification of parameter, e.g. number of loop iterations, according to probability mass functions. The transformation uses a random number generator in order to derive the real value of the parameter according to the user defined distribution. The simulation tool instantiates a class implementing a queue and a scheduling policy for all the processing resources defined in the PCM. Currently only FIFO and processor sharing policies are available but more complex and realistic policies are planned for future releases of the tool. In order to simulate the impact of parameter passing between components, the framework simulates also a stack frame. PCM allows developers to specify arbitrary distributions for resource demands of software components and arrival rates of workloads. The simulation takes as input a workload specified in the PCM instance and generate traffic according to it. The workload can be open or closed. The simulation tool performs several iterations of the specified workload until the resulting probability mass function does not change significantly. The simulation tool updates results while the simulation is still running so that the system architects can stop it if the reached precision is sufficient for the goal of the analysis. The main advantage of a simulation approach to the evaluation of the availability of the system is that it can emulate many environment. On the contrary an evaluation based on an analytic solution of a queueing network model can not deal with G/G/1 or G/G/n queues.

LQNSIM[24] is a simulation tool for LQN models. It is distributed along with the LQNS tool that will be presented in Section 3.3.2. The simulation tool can be used when the developer wants to model some aspects of the system that the analytic solver tool is not capable of handling. An example could be a system in which nodes use a completely fair scheduling

TABLE IV: ANALYSIS PERFORMED BY LQNS E LQNSIM

Measure	Measure description
Mean Delay (Variance) for a Rendezvous	It is the (variance of) queueing time for a request from a client to a server. It does not include (the variance of) the time the customer spends at the server.
Mean Delay for a Send-No-Reply Request	It is the time the request spends in queue and in service in phase one at the destination.
Mean Delay for a Join	It is the maximum of the sum of the service times for each branch of a fork. The variance of the join time is also computed.
Service Time	The service time is the total time a phase or activity uses for processing in the model.
Throughput and Utilization per Phase	Throughput by entry and activity, and utilization by phase and activity. The utilization is the task utilization, i.e., the reciprocal of the service time for the task.
Utilization and Waiting per Phase for Processor	Processor utilization and the queuing times for every entry and activity running on the processor.

policy. The simulator is also capable of using different distributions for the service time of requests while the analytic solver uses only an exponential distribution.

The list of features that it is capable of analyzing, along with a short description, can be found in [25] and is reported in Table IV for convenience.

3.3.2 Analytic Solvers

Analytic solvers offer a way to solve the performance model analytically. Usually these solvers give mean values of the parameters under analysis. The main advantage of this kind of solvers is that they can perform the analysis of large systems quickly.

LQNS[24] is an analytic solver for LQN models. It is capable of deriving measures listed in Table IV performing a mean-value queuing approximations to solve the queues.

Many tools have been implemented to analyze Petri-Net models. The IT department of informatics of the Universität Hamburg maintains a web page[1] that lists major tools for the analysis of this kind of models. As an example, a quite popular solver for this kind of model is the **TimeNET** suite[26]. It includes a graphical editor to build models, different kind of analysis (and simulation) and it supports a wide variety of Petri Nets models like extended deterministic and stochastic Petri nets, colored stochastic Petri nets and stochastic UML state charts. Another interesting tool that offers a GUI to create SPN model instances is called SPNP[27]. It automatically generates and solve Markov reward models from the provided SRN. It can be used to analyze also non-Markovian SPN and fluid SPN. This tool also implements a discrete-event simulation engine.

SMCSolver[28] is a solver for the most important classes of Markov Chains. It implements algorithms for the solution of Quasi-Birth-Death processes, M/G/1 and G/M/1 queues. It provides a set of Matlab functions and a Fortran package that includes a graphical interface for the selection of the algorithm and the visualization of the resulting matrix.

SHARPE[29] (Symbolic Hierarchical Automated Reliability and Performance Evaluator) is a tool that provides a specification language and a GUI for specifying different performance, reliability and performability models. It also provides solution methods for the specified models

---

[1]http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/quick.html

in order to analyze their behavior. The metrics analyzed by the tool depends on the kind of model used for the specification of the application. The list of supported models is:

- Markov Chains (irreducible, acyclic and phase-type)
- Semi-Markov Chains (acyclic and irreducible)
- Reliability Block Diagrams
- Fault Trees
- Single-Chain Product-Form queuing Networks
- Multiple-Chain Product-Form queuing Networks
- Generalized Stochastic Petri Nets
- Series-Parallel Graphs

The Software Reliability Estimation and Prediction Tool (**SREPT**)[30] is a framework that comprehends two main approaches to the prediction of the reliability of an application. The first one is a black box approach that consider the whole application without looking at its internal structure. The other approach is architecture-based. It takes as input a model representing the structure of the application, in the form of a DTMC, CTMC, SMP or DAG, and a specification of the failure behavior of internal components, either seen as reliability or failure rate. The tool is capable of integrating the information given in the two models in order to build a *composite model*. This new model can be solved to derive reliability measures and other performance measures such as time to completion.

Platform Independent Petri net Editor 2 (PIPE 2) [31] is a tool for creating and analyzing generalized stochastic Petri networks. It provides a graphical framework for modeling Petri nets and supports the Petri Net Markup Language to import instances developed with other tools. The analysis capabilities of PIPE 2 are based on its plugin structure. The modules for structural analysis contained in the standard package are:

- *Model Classification Module*, used to classify the GSPN into a more specific class of models
- *Model Comparison Module*, to define criteria for comparing two GSPNs and apply those criteria to two instances of the model
- *Incidence and Marking Module*, to derive the incidence matrices and the initial marking from the model
- *Reachability Graph Module*, to build the rechability graph

Performance analysis can be conducted using the *Simulation Module* that performs a Monte Carlo simulation in order to retrieve the average number of tokens per place and mean transitions throughput. The tool also integrates two analyzers, DNAmaca and SMARTA, to retrieve state and count measures from the steady state distribution or the probability density and cumulative distribution functions for the time needed to complete a predefined passage on the specified model. In addition the *GSPN Analysis Module* is capable of deriving analytically the distribution of tokens on places and the mean throughput of timed transitions.

The language used to express performance queries is called Performance Trees, the tool offers a graphical editor to build such trees and visualize query results directly on them.

3.4 Performance evaluation of component-based software systems

Several approaches to prediction of QoS, with particular attention to performance measures, are reviewed in [32]. In particular the approach in [33] called Component-Based Software Performance Engineering uses UML with the SPT profile to model applications and perform automatic transformations to derive execution graphs and queuing networks. Transformed models can be analyzed by the RAQS solver, whose features are presented in [34], in order to obtain response time and throughput values of the application.

A similar approach is presented in [35], the main difference is the fact that the language used to design the application is called Component-Based Modeling Language, which is an extensions to the Layered Queuing Network (LQN) model. This language is not standardized and widely used as UML but it is closer to LQN models which are used for performance prediction. This affinity between the two languages helps to reduce ambiguity. The CBML model can be specified according to an XML schema. Since CBML is an extension to the LQN its models can be directly analyzed and solved by LQN solvers without the need of transformations.

Another interesting approach to the problem of performance prediction is presented in [36]. This method aims at ease transformations between different component system models in order to derive specific models to analyze different aspects of an application. KLAPER defines a minimal set of modeling constructs useful for performance evaluation. Since the main goal of this project is not of providing an environment to application designers but rather a model transformation tool no graphical editors for KLAPER have been developed, component devel-

opers are encouraged not to model their applications directly in KLAPER but rather to build transformations from their modeling language to KLAPER models.

Authors of[37] propose a framework for the assessment of availability of Component-based applications based on Cloud Services. Applications ad modeled in Systems Modeling Language (SysML) which is an extension of UML. The model of the application is composed of three kind of diagrams:

- Internal Block Diagrams (IBDs) describe the structure of the application and the connections among components.
- State Machine Diagrams (STMs) express the behavior of components and the transitions between states of the application.
- Activity Diagrams (ADs) are used to model maintenance operations that affect the behavior of the application.
- SySML Allocation diagrams are used to represent general cross-association of elements. Examples of these associations are *operation* that links a maintenance operation described in an AD to a specific block in an IBD.

The model of the application is then transformed into a Stochastic Reward Network (SRN) that can be used to perform analysis on the availability of the modeled application. The kind of analysis that can be performed with this framework use as input the expected failure probability of each component of the system and some parameters that specify the behavior of predefined reactions to component failures. This tool takes into account also the impact

of scheduled maintenance operations but is not suited to model an environment in which the incoming workload affect the availability of some components of the system that may change dynamically.

CHAPTER 4

MODEL EXTENSIONS

In this chapter we present an extension to the classical DTMC model that allows it to represent some peculiar aspects typical of Cloud computing. An instance of this DTMC model can be used to describe an application deployed on multiple Clouds or even in a hybrid environment. The model can be useful to perform design time analysis of the behavior of the application in different working scenarios and to conduct analysis similar to the one described in Section 3.1. These kinds of analysis can be used by system developers to take design decisions regarding the structure of the application.

4.1 Overview of the solution

The solution we propose in this thesis is shown in Figure 11. It is composed by two main parts:

- A *model* that stores information on the structure of the application and on the characteristic of the environment in which it is deployed.
- A *simulation engine* that uses information about the incoming workload and the Cloud infrastructure to derive the expected availability of the system along with other performance measures.

The model is derived automatically from an instance of a Palladio PCM tool via an extension of Palladio that we have developed, described in Section 5.2.

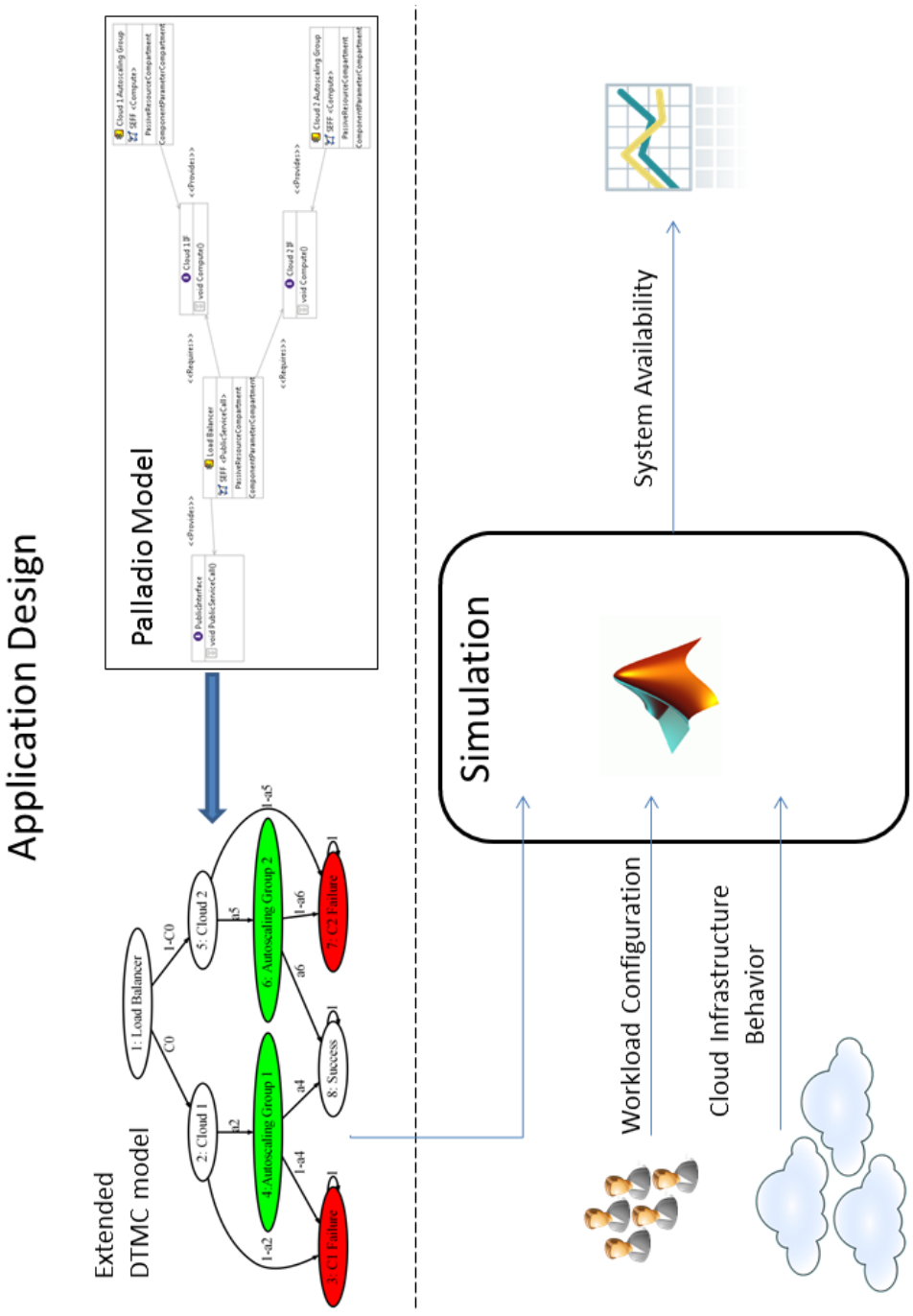


Figure 11: Overview of the solution

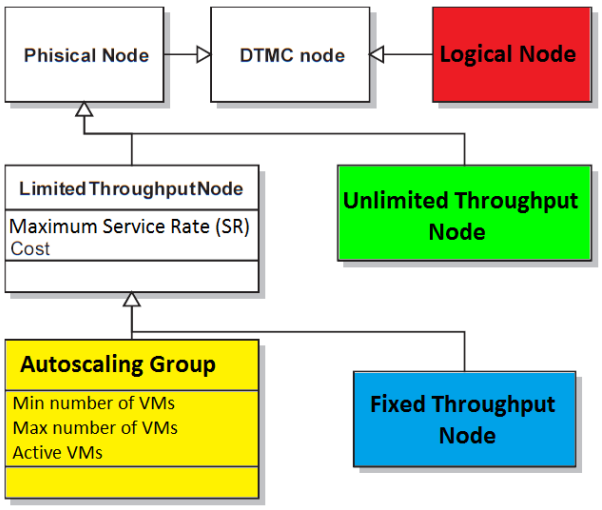


Figure 12: Type of nodes

4.2 The Model

The DTMC model presented in Section 2.3 is frequently used for availability analysis but it is not suitable for representing some peculiarities of the Cloud environment. Our goal is to build a model that is capable of describing a multi-Cloud application with particular focus on the estimation of availability measures.

As stated in [38] usually models are built using numerical estimates of various parameters made by domain experts. These parameters usually change over time and precise estimates are seldom correct, in order to cope with this problem authors suggest to keep models alive at runtime and dynamically update estimated parameters via monitoring of the real system. We developed our model with this focus so that it could be used to estimate applications availability also at runtime.

As first step in the specialization of the model we divided the nodes of the classical DTMC as shown in Figure 12. The main distinction is between *Logical* and *Physical* nodes. The rationale behind this distinction is the following:

- *Locigal nodes* are those nodes that do not represent any component in the system that perform computation. The most common logical nodes are those that represent success or failure states. Logical nodes could be introduced in the chain in order to make the instance of the model easier to read. An example of such a node could be a node added to represent the point in which requests enter the system.
- *Physical nodes* are those that represent components of the application or of the infrastructure that can be mapped onto a physical resource. This node type can be used to model load balancers as well as servers, VMs, databases, network links or any other component that actively perform some kind of computation over the requests flowing in the system.

When modeling an application our main focus should be in specifying properties of physical nodes as these nodes represent components that may act as bottlenecks for the system. Since these nodes are so important for the evaluation of the availability of the entire application we decided to specify them further by diving them in:

- *Unlimited Throughput Nodes.* These nodes represent physical resources that perform some very light computation on the requests traversing them. Designers could use these nodes to model components that are supposed not to cause bottlenecks and whose management is not in control of the application. This type of node could represent the load balancer

of a Cloud provider that distributes requests within an autoscaling group. Usually this component is provided by the Cloud provider with very efficient specialized hardware that is managed automatically.

- *Limited Throughput Nodes.* This node represent a component that actually perform some heavy computation on incoming requests. We can say that this node is the heart of the application because it implements the logic of the application that process the requests. Since this node is responsible of processing requests its limited processing capacity could not be sufficient to cope with the incoming traffic. When a node is overloaded by requests it start to reject some of them. A peculiarity of this node is the fact that it has an outgoing arch to a failure node whose probability depends both on the workload entering the node and the processing power of the node.

Since Limited Throughput Nodes correspond to physical resources that perform computation on requests two parameters have been added to these nodes. The first is the *Maximum Service Rate* and represent the maximum number of requests that can be processed by the node each second. The other parameter is the cost of using the node. Physical resource performing computation are associated with a cost in the infrastructure. One of the main difference between the Cloud environment and other computing environments is the fact that the processing power of the application components can change according to the application developers will. This feature of the Cloud is called *auto scaling* and is usually implemented by using an homogeneous pool of virtual machines behind a load balancer. The system manager can define rules that, when triggered, modify the number of virtual machines in the pool adding or removing

computational power. In order to model this behavior the limited throughput node has been further distinguished into two separate nodes:

- *Autoscaling Group*. This node represent a pool of VMs that is capable of change size according to application needs as presented in 2.1. The maximum processing capacity of this kind of nodes is given by the number of Active VMs multiplied the Maximum Service Rate.
- *Fixed*. This node is used to model classical servers whose computing capacity is fixed. This node has been introduced in order to let the user model hybrid environments in which part of the application is deployed on a Cloud infrastructure while other components (e.g. a database) is deployed on a in house infrastructure.

Three additional parameters are associated to autoscaling groups. *Minumum number of VMs* and *Maximum number of VMs* represent boundaries to the number of VM that can be run simultaneously on the resource modeled by the node. A lower bound on the number of VMs could be used if, for example, building an application that requires high availability, the designer decides that on each region of a Cloud provider there should be at least two machines always running. On the other hand the maximum number of running instances is used to model a resource cap that some providers may impose.

As in[39] we extended the classical DTMC model by adding *control variables*, with labels starting with letter 'C', and *measured availability*, starting with letter 'a' as labels to transitions. Measured availability represent factors external to the application that come from the infrastructure used. This factors may influence the behavior of the application and could lead

to degradation of the availability of the system. Examples of this factors are blackout or outages due to middleware management of data centers, which may lead to world wide outages, or failure of an in-house computing resource. Control variables represent alternative choices, made according to certain probabilities. This probabilities define the rate at which requests are routed among connected nods. The values of this variables can be set by application deployers in order to define how their applications will use different Cloud providers.

A common extension to the DTMC model, discussed in Section 2.3, is the definition of *rewards*, or, in our case, *costs*. In our model rewards are attached to states and model the cost generated by a request traversing the node. Recalling the distinction of nodes just presented, one can note that only computing resources represent nodes with a positive cost while logical nodes have cost equal to zero. This is due to the fact that they are not mapped, as a first approximation, to any physical resource consumption that leads to an increase in the cost of the system.

Pricing is usually given in instance hour. The user is charged for every machine for the entire hour, even if one machine is turned off before the end of the hour. In our solution, we decided to assume per second billing pay for simplicity. Per hour billing pay is left to future work.

So we ask the developer to annotate the nominal cost of using the resource modeled by the node. Instance pricing is usually constant and retrievable on the provider web site. Though, we took into consideration the fact that prices could change, like for the Amazon spot instances. APIs are usually provided by the Cloud provider to read current costs.

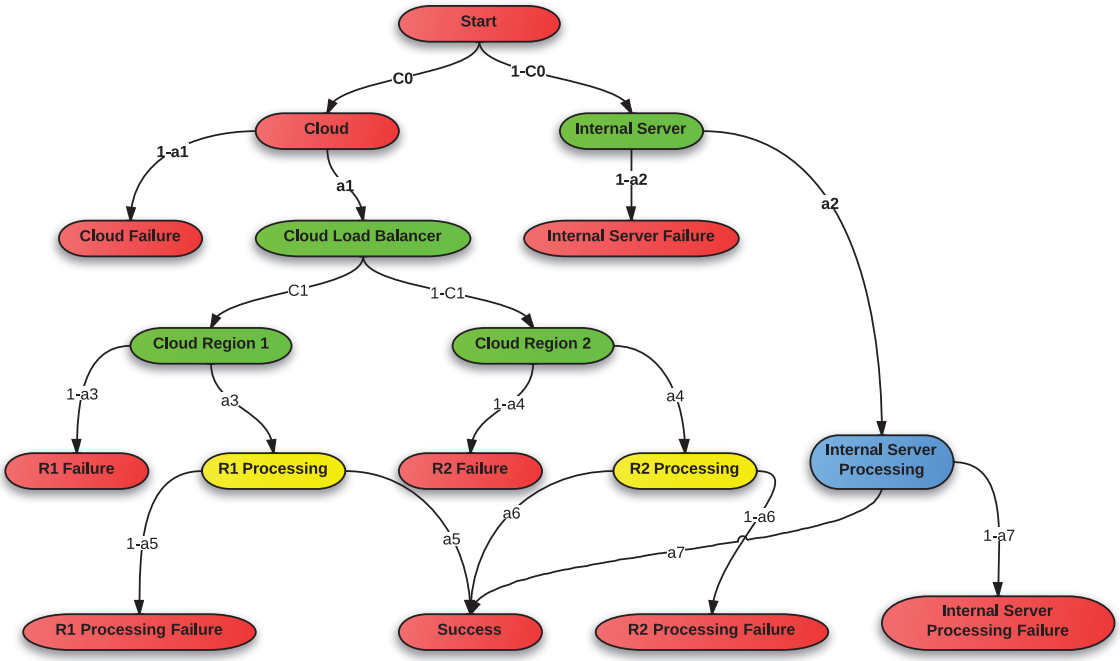


Figure 13: Instance of the model

An example of a complete model is reported in Figure 13. The type of node is shown by the color of the node.

The chain start with a logical node that represents the entry point of requests in the application. Requests are directed either to the Cloud platform or to an internal server for processing. The internal server is composed by an unlimited throughput node that represents the network infrastructure and a fixed capacity node. Both these nodes could reject requests for different reasons. The availability value *a2* represent the portion of requests that are correctly delivered to the processing node. The blue *Internal Server Processing* node represent an in house server with fixed computational capacity. It can reject requests due to many factors. One of the main cause of the reduction of the availability of this node is the fact that the incoming workload could be greater than the actual processing capacity of the node. Rejected requests flow through arc labeled *1-a7* to a failure state.

Requests directed to the Cloud infrastructure through *C0* enter node *Cloud*. This logical node represent the entry point in the Cloud infrastructure. It has been introduced to model the scenario in which the whole Cloud infrastructure fails, represented by the arc to *Cloud Failure* node. *Cloud Load Balancer* node represents a load balancer controlled by the system administrator, as shown by arc *C1*. The system administrator can define the portion of requests that go to each of the two regions of the selected Cloud provider.

Both nodes *Cloud Region 1* and *Cloud Region 2* represent entry points of autoscaling groups. These nodes are in charge of distributing requests to available VMs of the corresponging au-

toscaling group. The nodes represent the middleware infrastructure of the Cloud provider that manages VMs.

Requests successfully dispatched to autoscaling groups enter nodes *R1 Processing* and *R2 processing*. These nodes are similar to the *Internal Server Processing* node with the only difference that their computational power can changed according to some policy defined by the system administrator. The portion of requests successfully processed by these nodes goes directly to the final success state.

The parameter that we wish to estimate from this model of the application is the portion of requests arriving to the final success state with respect to the total incoming requests.

CHAPTER 5

TOOL

This Chapter introduces the two tools developed in this thesis. The first tool presented in Section 5.1 is a simulation engine that takes as input an instance of the model described in Chapter 4 and some specification of the working condition that the user is interested in. The simulation engine gives as output the availability of the system during the simulated period, the total cost of using the system and some other performance information. The second tool presented in Section 5.2 is an extension to the Palladio Bench IDE that can be used to derive models useful to analyze the availability of modeled applications.

5.1 Simulation

The model described in Section 4.2 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 we 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 on open workload in which requests enter the system, flow through the chain of the model presented in Section 4.2 and exit the system.

We built our simulation engine 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 groups. As explained in 2.7.1 load balancing among instances of the same autoscaling group is done equally. 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 tuned in order to simulate different scenarios and the outcome of the simulation.

5.1.1 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 in 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 situations 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 of Figure 14. 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 and generates the incoming traffic according to it. The function used to specify the arrival rate of Figure 14 is:

Listing 5.1: Matlab function used to generate the bimodal distribution

```
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))));
```

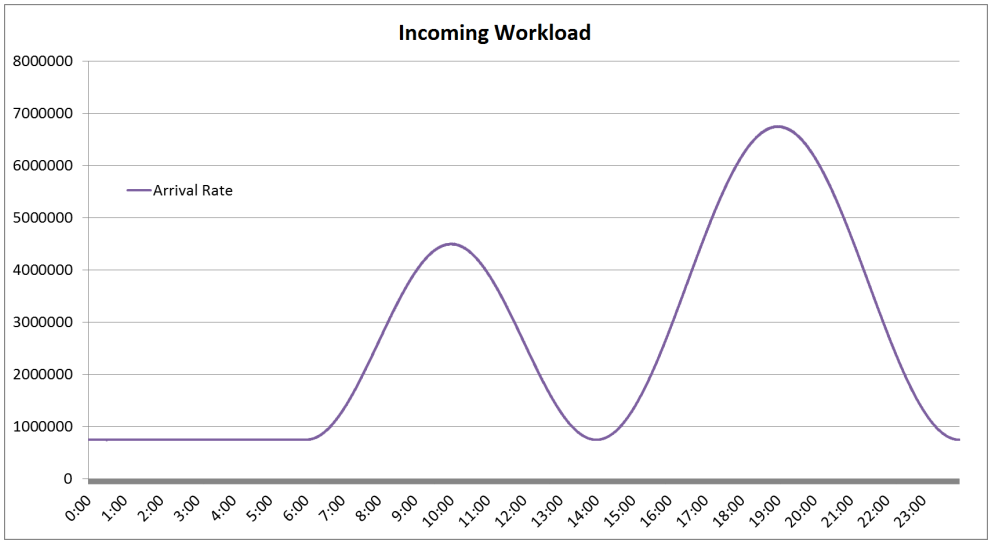


Figure 14: Bimodal distribution of requests

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 the user defined function is used. More information about realistic traffic generation can be found in [40]

5.1.2 Infrastructural parameters

The model described in Section 4.2 include 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 [3] this value depends on the Cloud provider, 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 [41].

5.1.3 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 for the simulation. These values are used to define the number of steps for the entire simulation.

The goal of the simulator is to evaluate the design of the application modeled by the development team. In order to evaluate more precisely the behavior of the application in the Cloud environment it is necessary to define a policy to manage the scaling capability of the underlying infrastructure. In order to do that we 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 VMs 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 lode of nodes and the frequency with which the control policy is applied.

The user can vary these parameters in order to simulate different scenarios. For example if the user wants to test how its application reacts to a peak of requests he may put the desired shape of the incoming workload and leave other parameters unchanged. Another example could

be testing how service rate variation during time of the day affects the application. In order to simulate this scenario the user can adjust the maximum node capacity parameter. Some of the scenarios that we have simulated are described in Chapter 6

5.1.4 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 4.2 nodes capable of autoscaling model group of VMs in the Cloud, their maximum processing capacity is given by

$$\text{number of VMs} \times \text{VM maximum service rate}$$

By using this formula we are exploiting the fact that VMs in the same node have the same processing capacity. This assumption 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 scale out (or down) 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. Examples of the output of the simulation can be seen in chapter 6.

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[40], 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 out and the timeout for the scale out of the node has expired
9. historical data is saved
10. the autoscale controller checks if any scale out or scale in process has to be performed according to the user defined rules

Here it follows the while cycle used to simulate in matlab one step of requests processed by the system.

```
arrivals = poissrnd(arrival_rate(t) * seconds_per_step);
workload = zeros(1,n_nodes);
incoming_workload = input_node * arrivals;
outgoing_workload = zeros(1,n_nodes);
failures = zeros(1,n_nodes);
successes = zeros(1,n_nodes);
service_time = zeros(1,n_nodes);
time_left = seconds_per_step * ones(1,n_nodes);
while any(incoming_workload ~= outgoing_workload)
        workload = workload + incoming_workload;
        to_do = incoming_workload;
```

```
        service_time_required = gamrnd(floor(to_do), 1./(running_machines .*
              service_rate(t))) + mod(to_do,1) .* 1./(running_machines .* service_rate(t
              ));
        outgoing_workload = min(1, time_left ./ service_time_required) .* to_do;
        timed_out_reqs = to_do - outgoing_workload;
        failed_from_ext_problems_reqs = outgoing_workload .* sum(
              dtmc_matrix_no_failure_loops(:,failure_nodes),2)';
        failures = failures + timed_out_reqs + failed_from_ext_problems_reqs;
        successes = successes + outgoing_workload - failed_from_ext_problems_reqs;
        outgoing_workload(success_node|failure_nodes)=0;
        incoming_workload = outgoing_workload * dtmc_matrix + timed_out_reqs *
              dtmc_matrix_to_failure_nodes;
        time_left = max(0, time_left - service_time_required);
        service_time = min(seconds_per_step, service_time + service_time_required);
    end

    cpu_load = service_time ./ seconds_per_step;
    availability = ones(1,n_nodes);
    availability(workload~=0) = min(1, successes(workload~=0) ./ workload(workload~=0));
    availability_values = num2cell(availability);
    system_availability = system_availability_function(ctrl_values{:}, availability_values
          {:});
```

5.2 Palladio Extension

In order to exploit the simplicity of modeling a software system offered by Palladio we decided to extend it by allowing the generation of an instance of the model introduced in Section 4.2. In order to do so, we extended Palladio Bench by implementing a post processing phase that is executed after the generation of the DTMC model by Palladio. This post processing

phase transforms the DTMC and annotates it by adding the parameters introduced in Section 4.2.

We decided to reuse many of the features already available in Palladio and integrate our code by reusing its structures. One of the features that we used is the *sensitivity file*. A sensitivity file is an XML file that can be generated in Palladio in order to modify some parameters of the model while performing its evaluation. In a sensitivity file system, designers can change some of the numerical values introduced in the model in order to easily perform multiple evaluations of it and compare different design choices. Examples of parameters that can be specified in a sensitivity file are the failure probabilities of each failure type and the branching probabilities of branch actions in SEFF diagrams.

Figure 15 shows a simple repository composed of four components: a web server, that processes incoming requests and uses some external processing to produce the result, a load balancer component, that is responsible of distributing incoming traffic, and two components modeling some service on two different Cloud providers.

Let assume that we are now interested only in the impact of Cloud failure on this simple architecture. Therefore, we model internal processing action of Cloud providers with a failure type description by utilizing SEFF diagrams, depicted in Figures 16(a) and 16(b). The load balancer SEFF diagram is shown in Figure 16(c) we can imagine that system developer does not have control on the availabilities of Cloud provider but only on the probabilities on the load balancer.

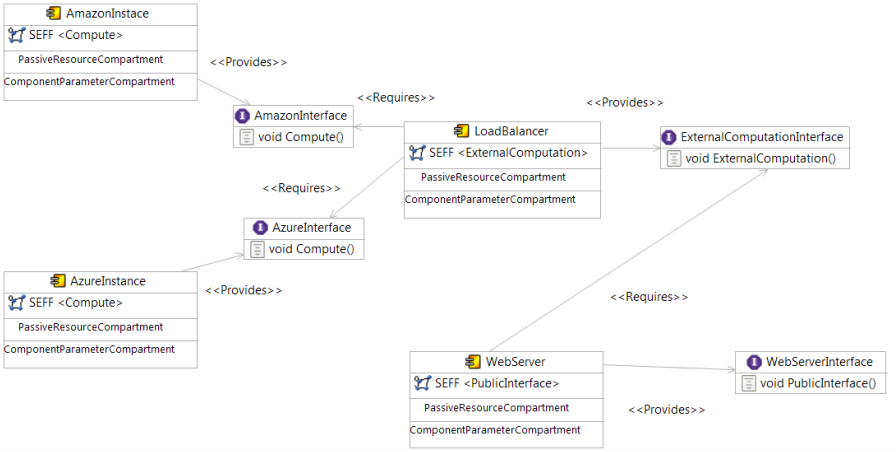
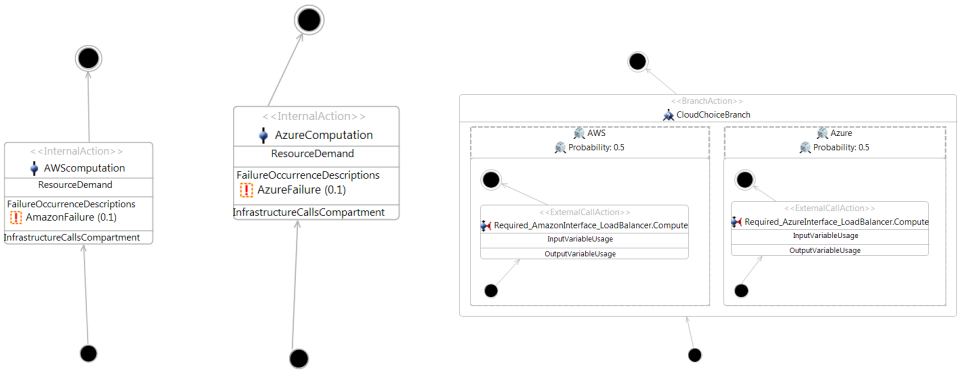


Figure 15: Example Repository



(a) (b) (c)

Figure 16: SEFF diagrams

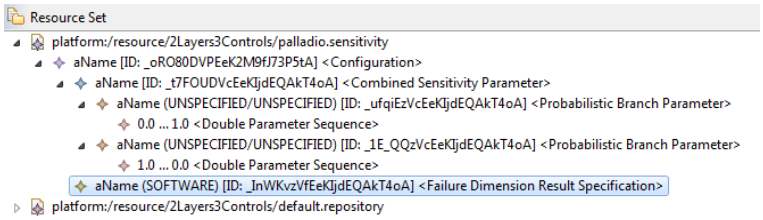


Figure 17: Sensitivity file example

If we model this system in Palladio we can run its evaluation tool based on DTMC and discover that the expected availability of the system is 0.85. This result was expected because if we simply analyze the diagrams presented we can see that the application uses equally both Clouds which have availability values of 0.9 and 0.8 respectively, the result of this single evaluation is not very helpful to developers who have to decide the best values for their load balancer. By specifying in a sensitivity file like the one in Figure 17 a variation of the parameters for the load balancer, Palladio is able to run several iterations of the evaluation of the system by modifying the specified values.

The result of this analysis is stored in a log and can be viewed in table V, this table is much more useful because it shows how the choice of the value for the load balancer variable affects the final availability of the system. In this toy example the best choice of using only the Cloud with the higher availability was clear, but the purpose of this example is to show how the sensitivity analysis work, not to model any complex real case.

TABLE V: RESULT OF A SENSITIVITY RUN

Branch Name	**Branch Probability**	**Success Probability**
Azure	0	0.8
	0.2	0.82
	0.2	0.84
	0.2	0.86
	0.2	0.88
	1	0.9

The sensitivity analysis is useful if the number of changing parameters is small, otherwise the output produced is too detailed to be used by developers. In our work we reused the structure of the sensitivity analysis mainly because the graphical tool for building sensitivity files is well integrated in Palladio and the resulting XML is easy to parse with common parsers like the javax.xml.parsers.DocumentBuilder.

Reusing this file to query the user for information used to annotate the model made also simpler the mapping between attributes of the model and elements of Palladio.

Since Palladio transformations give as a result a static model in which all transitions have a fixed probability we had to keep track of the failure types defined by the user and mark them as measured availabilities. We also kept track of branches whose probability had been marked as control variables in the sensitivity file in order to mark them as control variables also in the model. The sensitivity file is structured as in Figure 18 in this example we can see that the user has specified four failure type parameters which will be marked as measured availabilities and three probabilistic branch parameters.

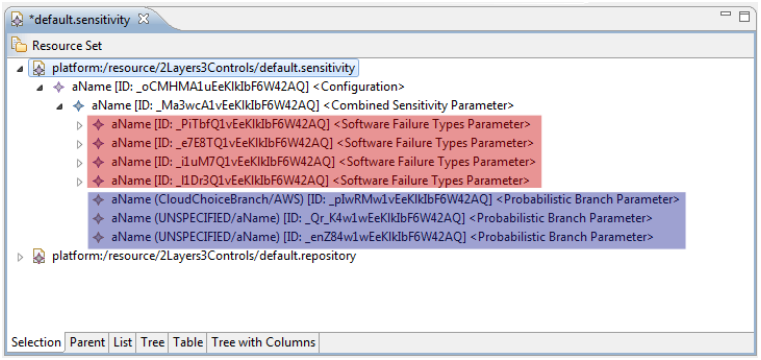


Figure 18: Complete Sensitivity File

At this point the user would specify the range in which parameters can vary but, since we are interested in more attributes for each node, we require the user to specify a *string parameter sequence* as a child of each software failure type. In this parameter the user can insert a number of strings to specify each of the attributes described in Section 4.2.

In order to obtain the final model we exploited the transformation engine already built in Palladio to obtain a DTMC which is then transformed and refined until it meets our needs. Even if the modeled application is very simple the DTMC resulting from the transformation done by Palladio is huge. Palladio offers natively the possibility to reduce this chain but what it practically does is solving the chain by calculating all the failure and success probabilities (one failure probability for each specified failure type) and build a new very compressed DTMC with one start state directly connected to the success state and to all failure states annotated with their probabilities. This small matrix does not contain enough information on the structure of the application so is useless if we want to perform a different kind of analysis. For this reason

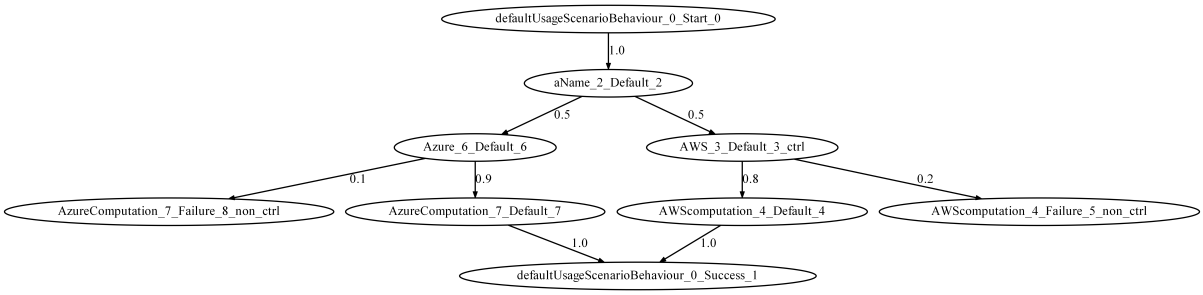
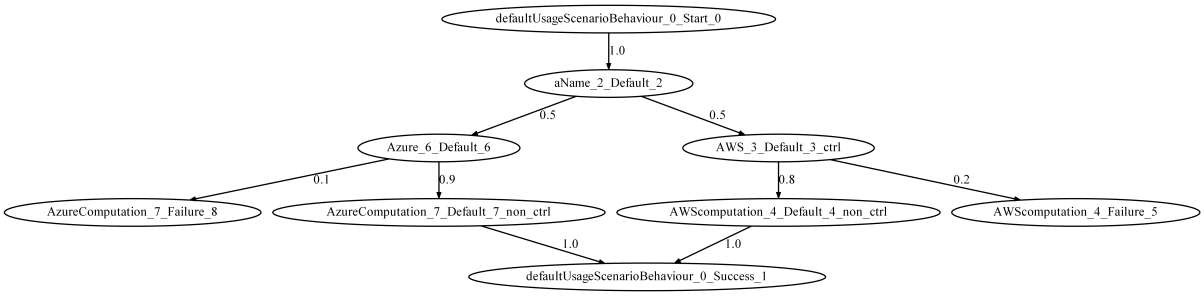


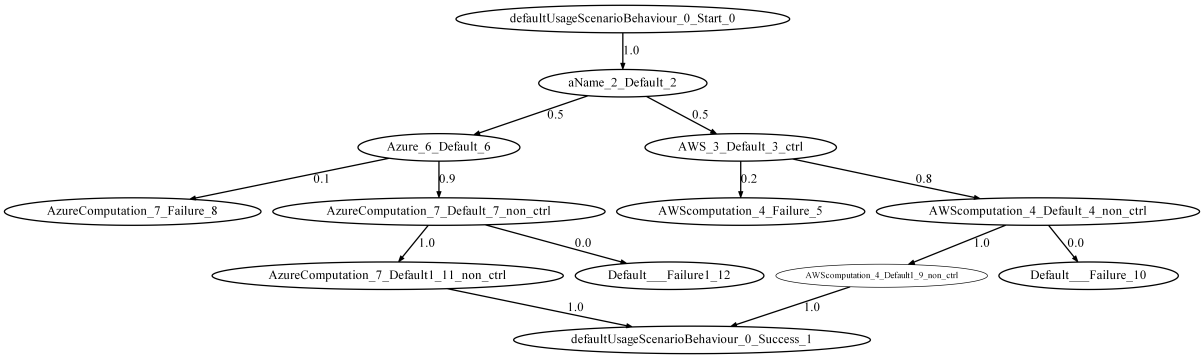
Figure 19: First step of the transformation

we decided to skip the chain reduction offered by Palladio and implement an ad-hoc reduction function which simply eliminates all the transitions that have probability one. This reduction is very simple from the logical point of view but helps to heavily reduce the size of the final chain and prepare it for further transformations. So, for example, the result of applying this simple transformation step to the Palladio model depicted in Figure 15, can be seen in Figure 19. For this example, the web server and the load balancer controller are set to logic nodes since, as we said in Section 4.2, we do not see them as possible point of failures of our architecture and we want to focuse more on the backend.

The next step of the transformation is to move labels of non controlled variables from failure states to corresponding success states, this is done in order to simplify the process of the successive function which is expands those states by adding a failure state for each of them which corresponds to failing requests due to the limited processing capabilities of these nodes. The output of steps two and three can be seen respectivly in Figure 20(a) and Figure 20(b).



(a)



(b)

Figure 20: Second and third steps of the transformation

Steps four and five are dedicated to the generation of measured availability variables, in order to do so we need to label as non controlled all the states having as incoming transition only transitions that do not represent control variables or measured availabilities. Step four does this by labeling corresponding states and step five moves the labels from states to the corresponding transitions. The output of these steps is shown in Figures 21(a) and 21(b).

The last modification that we need to perform on the DTMC is to add self loops with probability one to all final success or failure states in order to make them absorbing states for

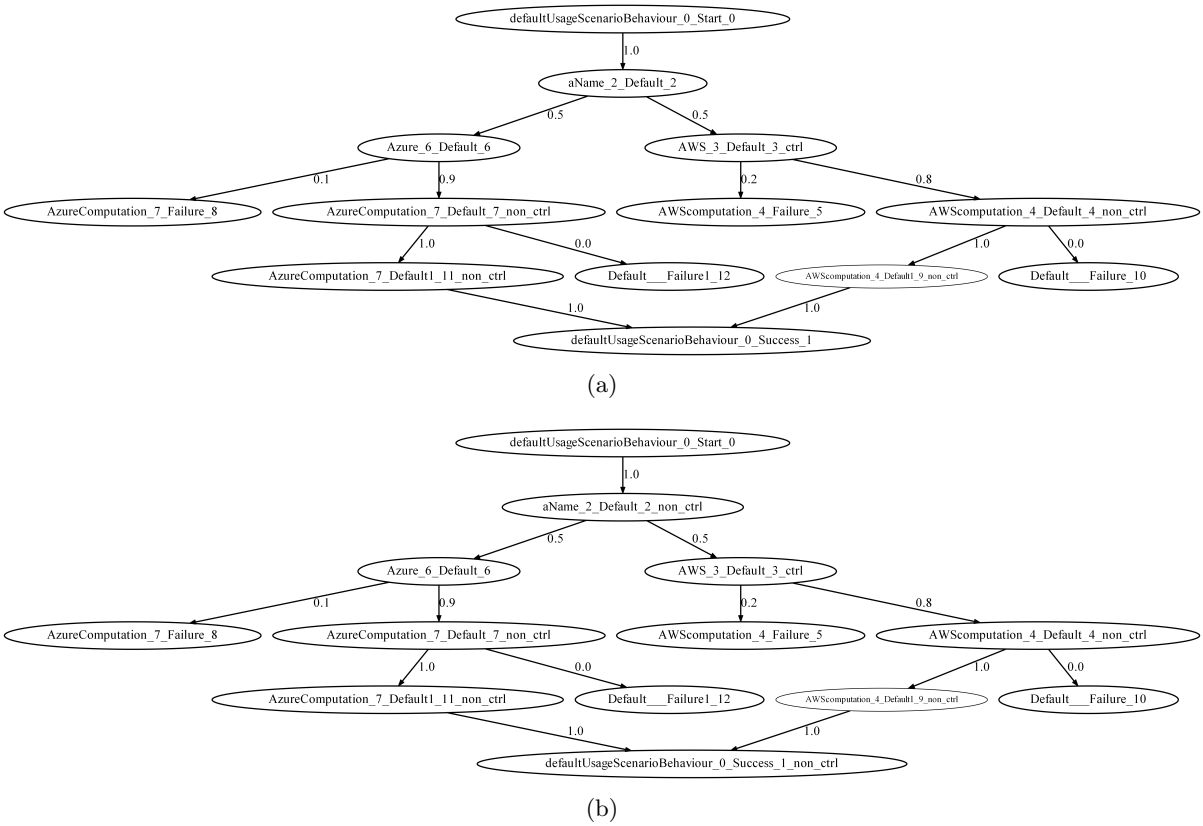


Figure 21: Fourth and fifth steps of the transformation

requests flowing in the system. This is done in the last step which gives as output the model in Figure 22.

As introduced in Chapter 1 we have developed a simulation engine capable of deriving availability measures from applications described using this model against user defined scenarios. The code for the simulator, that will be described in Section 5.1, is composed by some matlab files with some tokens in correspondence with fields that describe the model, simulation parameters or user inputs. After generating the final DTMC model, the tool parses these template

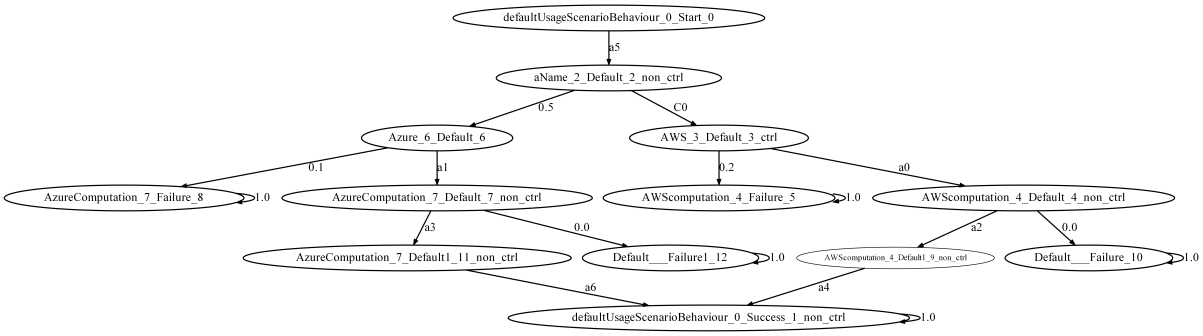


Figure 22: Final result of the transformation

files and writes in the appropriate sections information like the matrix of the DTMC system and all the parameters needed for the simulation.

CHAPTER 6

EXPERIMENTAL ANALYSIS

We now present two use cases that have been used to evaluate our approach. These use cases simulate the behavior of an application deployed on multiple clouds against different working conditions. The simulation technique used for this tests is introduced in Section 5.1. Each use case represent a different deployment configuration of a Web application that has been simulated against different scenarios. In particular the use case of Section 6.1 shows the replication of the application on two independent cloud providers. The use case of Section 6.2 enrich the previous one by replicating the application also on a sub-region of one of the two Clouds.

6.1 A Web System Use Case

The purpose of this use case is to show the capability of the model and the tool to specify to describe and analyze many different kind of failures or working conditions that may affect a Cloud environment.

The Palladio repository model of the application is shown in Figure 23. It shows two components representing different autoscaling groups, one for each cloud provider, and a load balancer component. From this Palladio description of the application our tool automatically derived the DTMC model shown in Figure 24 where the two autoscaling groups are represented by the green nodes and the load balancer by an initial node with two outgoing arcs. Probabilities

associated with these arcs have been fed by the application developer in the Palladio models and, for this example, are both equal to 0.5.

Nodes 2 and 5 represent respectively entry points of Cloud provider 1 and 2. A request in one of these states can either reach the corresponding auto scaling group and be processed by the application or fail and reach one of the two final failure states 3 and 7. Probabilities of reaching one of the two autoscaling group from their respective cloud entry points are represented by $a2$ and $a5$. In our scenario those values represent the availability of cloud providers or, more precisely, that part of the availability that can not be related to the application but only to the cloud infrastructure. Events like a power outage, a connectivity issue or effects of the multitenancy can affect these values.

Avilability issues due to the failing of processing requests that reach an autoscaling group is modeled by the probability associated to the arc that exits the aoutoscaling group and enters the corresponding failure state. In particular these probabilities are represented by $a4$ and $a6$. These probabilities are affected by software bugs contained in the application and by the limited computational capacity of VMs that host it.

In order to show the capability of the tool to simulate different working conditions we modeled three different scenarios reported in Sections 6.1.1, 6.1.2 and 6.1.3. Each scenario shows a variation of a common configuration of the system that is reported in Table VI. These parameters have been kept consistent in all the scenarios since they model static aspects of the cloud environment such as the cost of using a VM or their maximum service rate. Also the behavior of the application, in terms of scaling policies, have been kept consistent.

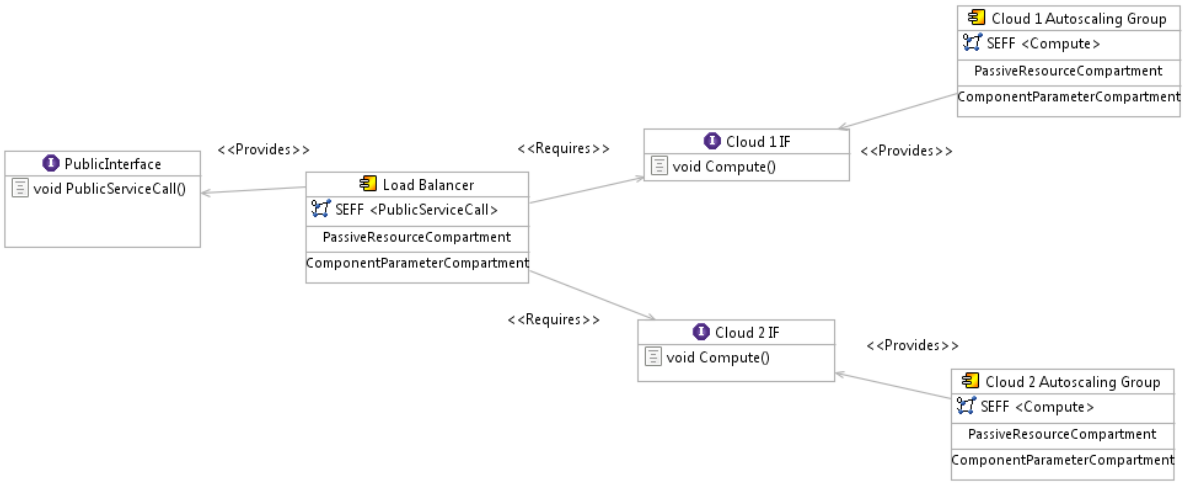


Figure 23: Palladio model of the first usecase

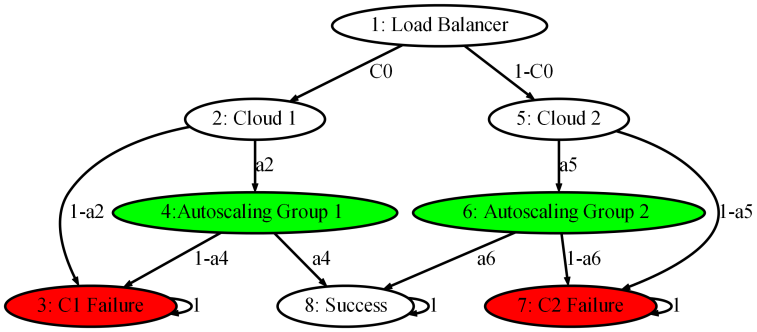


Figure 24: DTMC model representation of the Multi-Cloud application.

TABLE VI: PARAMETERS OF SCENARIO 1

	Cloud 1	**Cloud 2**
Cost per VM	0.30$/hr	0.50$/hr
VM startup time	100 s	100 s
VM nominal SR	$10,000\ \frac{\text{reqs}}{\text{s}}$	$15,000\ \frac{\text{reqs}}{\text{s}}$

TABLE VII: SCALING POLICIES

Scaling policy		
CPU scale out	100%	3×
	90%	2×
	85%	1.5×
	80%	1.1×
CPU scale in	20%	$1/2\times$
	40%	$1/1.6\times$
	50%	$1/1.3\times$
	60%	$1/1.1\times$

In this use case Cloud provider 1 offer cheap virtual machines with a quite low processing capacity while Cloud 2 offers more expensive and powerful machines.

The policy for managing the autoscaling of the application has been designed by the application developer and is reported in Table VII. The central column represent the values that trigger the scaling action while the right column shows the multiplication factor that is applied to the number of currently running VMs.

6.1.1 Scenario 1

The first scenario we introduce allow us to show the effect of a system failure at the cloud entry point. The simulation spans four hour of usage of the system during which the arrival rate is constant at $1e6$ requests per second.

This scenario simulate a total failure between 00:30 and 01:00 and a gradual degradation of service between 01:30 and 02:30. The first failure can be due to a loss of connectivity to the Cloud datacenter, a power outage or a major failure of the middleware that manages the datacenter. The gradual and partial degradation of the QoS of the cloud may be an effect of the multitenancy tipical of the Cloud. If the datacenter is overloaded with requests that has to be served by different applications the middleware system, and the load balancer in particular, could discard some requests that will not reach the autoscaling group.

As shown by Figure 25 the availability of Cloud 1 is not affected by these failures.

One of the outputs of the simulation is shown in Figure 26. It is the overall availability of the entire application. We can recognize three main working conditions:

- *Start-up.* The Scenario starts with the application deployed on 2 VMs in the autoscaling group of Cloud 1 and 1 VM in the autoscaling group of Cloud 2. The incoming number of request is of $1e6$ requests per seconds which is split by the load balancer to $5e5$ requests per seconds to each autoscaling group. Since the number of VMs that the application is using is not enough to serve all the requests the availability of the system is very low. The scaling policy needs about 5 minutes to increase the number of VMs to 53 for cloud 1 and 77 for Cloud 2 which are enough to serve all the requests and have 100% availability.

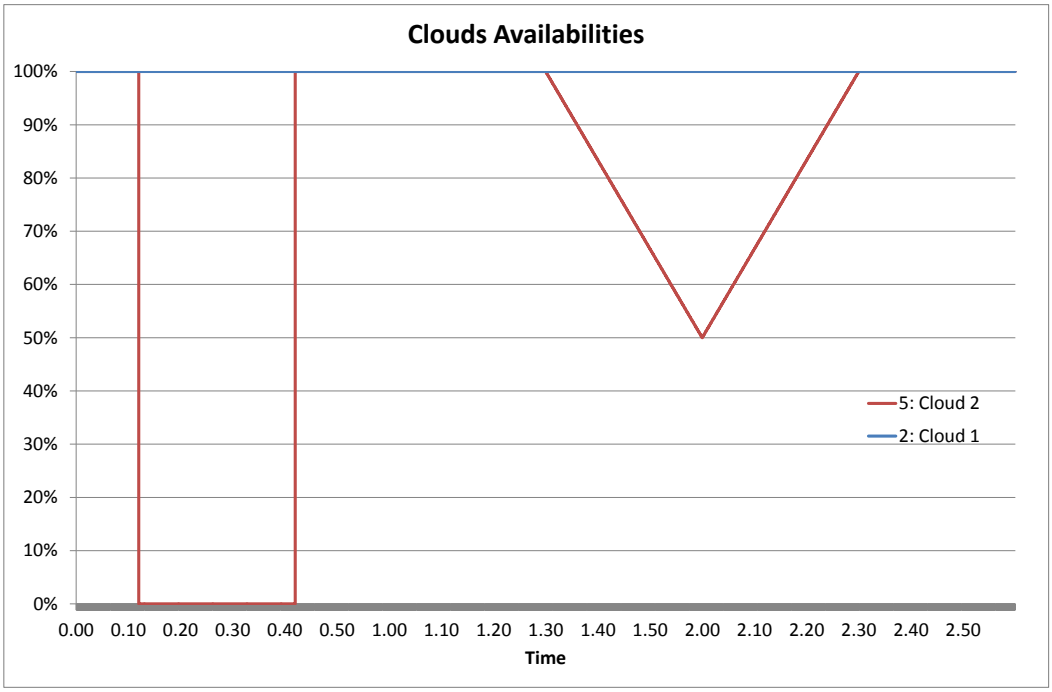


Figure 25: Cloud provider availability in Scenario 1

- *First failure.* The first failure of Cloud 2 happens at minute 15. Since it is a total blackout no requests are capable of reaching the autoscaling group and all of them flow directly in to the final failure state. The application developer set the load balancer to forward 50% of the incoming requests to this cloud so the overall availability of the system is halved. Since no requests reach the autoscaling group the number of VMs inside it is reduced to its minimum, which in this case is 1. When the issue that caused the failure is resolved at minute 45 requests starts flowing again into the autoscaling group that enters in a second start up phase. 5 minutes later the number of VMs of this autoscaling group is restored and the availability of the application goes back to 100%.
- *Second failure.* The secon failure starts at time 01:30. The main difference with respect to the previous failure is the fact that now Cloud 2 shows a gradual degradation of its availability which stops at 50% and then raises back to 100%. The effect of this degradation is that the overall availability of the application is gradually reduced to 75% and then brought back to 100%. Since Cloud 2 is serving only half of the incoming requests the effect of its loss of availability over the availability of the entire system is reduced.

The number of virtual machines used for each Cloud provider during the simulation is shown in Figure 27.

During the *Start-up* phase both autoscaling group increase the number of running VMs in order to fulfill all the incoming requests. In particular Cloud 1 keeps adding VMs to the autoscaling group until it reaches 108 VMs. This is a very strong reaction to the incoming

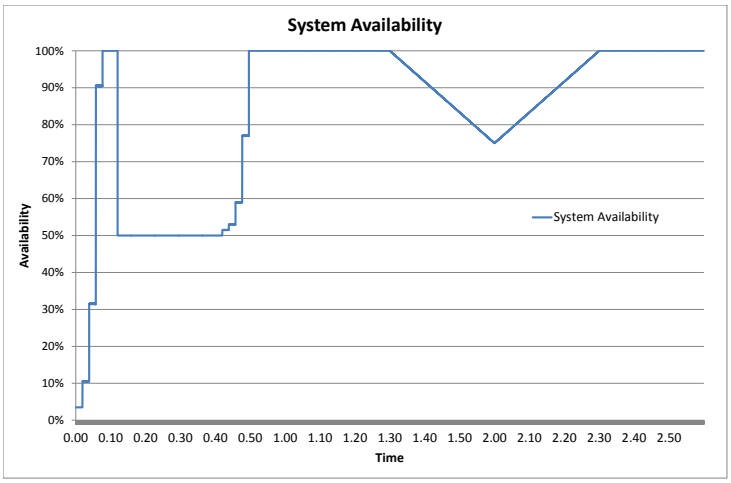


Figure 26: Overall availability of the application of Scenario 1

workload and few minutes later the number of VMs is reduced to 77. This strong reaction is due to the fact that at minute 4 Cloud 1 hosts 54 VMs but, as shown in Figure 28, the CPU load is over 90%. The scaling policy defined in VI tells the autoscaling group to double the number of VM. When the new machines starts to serve requests the average CPU load is reduced to 42%. Again the scaling policy tells the autoscaling group to recduce the number of VMs. The behavior of Cloud 2 during *Start-up* is similar to the one of Cloud 1 but enters a stable configuration with 53VMs.

During the *first failure* no requests reach the autoscaling group of Cloud 2 so the load of the CPU of those machines goes to zero. Again the scaling policy comes into play by reducing the number of virtual machines. Suddenly the availability of the Cloud provider is restored to

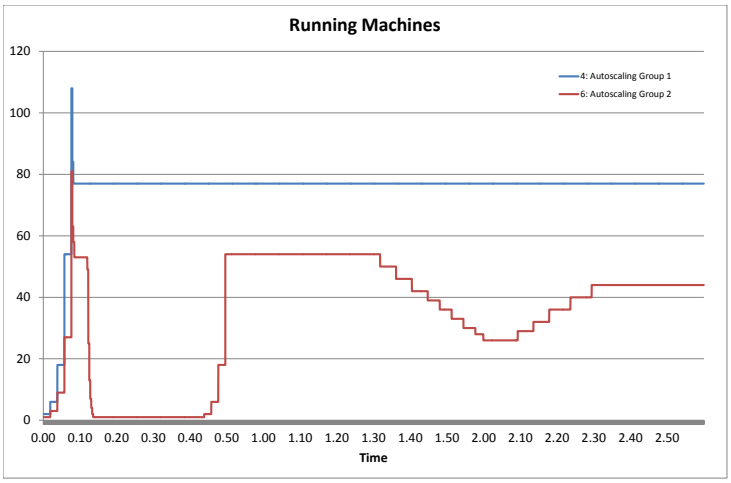


Figure 27: Active VMs on Scenario 1

100% and the requests start reaching the autoscaling group which now contains just a single VM. The system then enters another *Start-up* phase that is just limited to Cloud 2.

During the *second failure* the autoscaling group of Cloud 2 sees the number of incoming requests gradually decreasing. Figure 28 shows how the average CPU load of VMs of this autoscaling group decreases until it reaches 60%. Then a scale in is performed and the number of VMs is reduced. This leads to an increase of the CPU load of the remaining VMs but since the number of incoming requests is still reducing the CPU load is reduced again. When the availability of the cloud provider is restored a gradual scale up phase is performed.

The cost of using the system calculated by the tool at the end of the simulation is of 117,98$ and the overall availability during the simulated period is of 83%

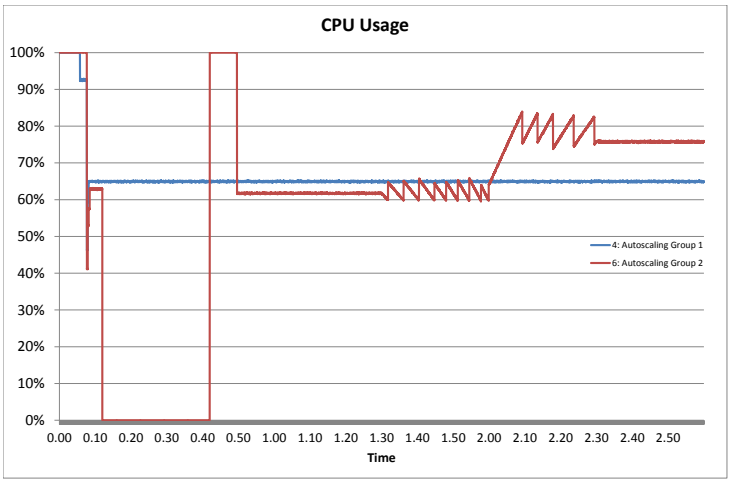


Figure 28: CPU usage of Cloud 1 and Cloud 2 in Scenario 1

6.1.2 Scenario 2

This scenario allows us show what happens to the system when the availability of the cliud provider is full but the service rate of the virtual machines is reduced.

This kind of problems are typical of the Cloud environment since Cloud providers try to pack VMs of different users in the same physical machine in order to reduce power usage of their datacenters.

The base setup of this scenario is similare to the one presented in Section 6.1.1 but the availability of both Cloud provider 1 and Cloud provider 2 are kept constant at 100%.

Again we simulated two different kind of failures for this scenario that are reported in Figure 29:

- *Sudden degradation.* A sudden loss of service rate of VMs of Cloud 1 happend betweeb 00:10 and 00:40. The Service rate is suddenly reduced by 80%. This may be caused by an application of another user of the Cloud environment accessing a critical resource for the application like HDD in a very intensive way.
- *Gradual degradation.* A gradual degradation of the service rate of the VMs composing the autoscaling group of Cloud 2 is simulated between 01:30 and 02:00. This scenario simulate the interference of an application of another user that receives a peack of requests and gradually starts asking for more resources. If the scaling policy of the other users' application is not capable of reacting to the increasing number of requests by adding more VMs and maintaining the same CPU usage level, that application will stress the physical CPU of the machine more heavily.

Figure 30 shows the CPU usage of VMs of both Cloud 1 and Cloud 2 hosting the autoscaling groups of the application. Since the service rate of Cloud 1 during the first failure is reduced requests processed by that autoscaling group will require much more CPU time so the CPU load is suddenly increased from 65% to 100%. This sudden increase makes the system discard some requests and causes a loss of availability shown in Figure 32. The main difference between this loss of availability and those presented in Scenario 1 is the fact that the scaling capability of the cloud environment can be exploited to reduce or even fill completely this loss.

Figure 24 shows the Markov chain representation of the system. From this representation we can see that he failing requests of Scenario 1 flow directly from state 5 to state 7 while in this Scenario they all go through state 4. Since the application developer has control over the

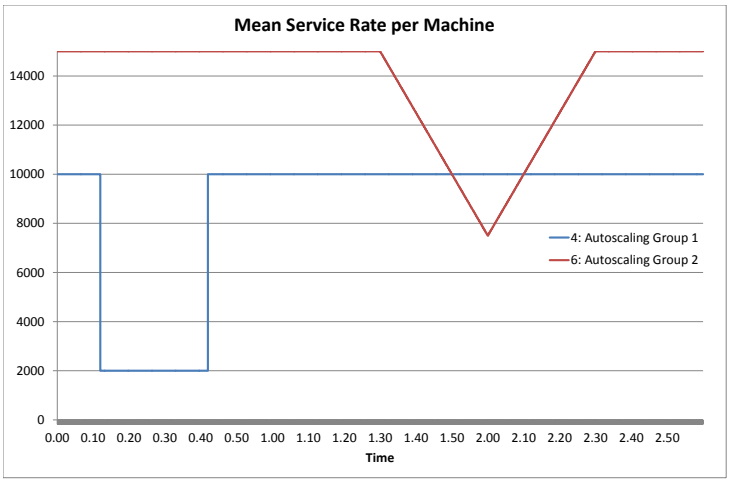


Figure 29: Changes in the Maximum Service Rate of VMs in Scenario 2

number of VMs in the autoscaling group represented by state 4 it has partial control over the probability of going from that state to the final success state. This control is applied by the scaling policies.

Figure 31 shows how the scaling policy defined by the application developer is used to reduce this loss of availability by increasing the number of VMs. In particular then the number of VMs is increase at time 00:15 the average CPU load of the autoscalign group is reduced from 100% to 64%. At the end of the failure the CPU load of the autoscaling group is reduced and the number of VMs is reduced as well in order to bring back the utilization near 65% and reduce costs.

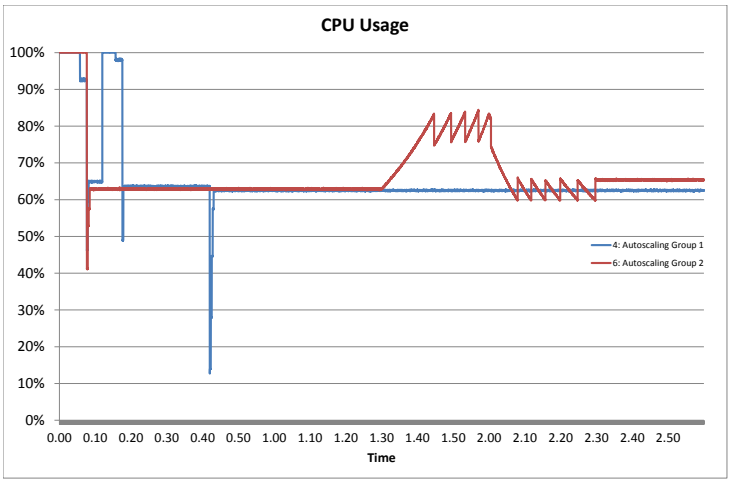


Figure 30: CPU utilizations of Scenario 2

Figure 32 shows that the speed of change in the service rate of the VMs from 100% to only 20% caused a loss of availability that the autosclaing policy was capable of restoring only after some amount of time. It is interesting to notice that the figure shows no degradation of availability in the second half of the simulation. This is due to the fact that the second failure happens in a gradual way.

A gradual reduction of the service rate of VMs in the autoscaling group of Cloud 2 causes a gradual increase in their CPU loads that, thanks to the scaling policy defined by the application developer, never reaches 100% and never reject any user request.

The simulation shows that the availability of the entire system for the simulated period is of 96% and the total cost is 193.26$

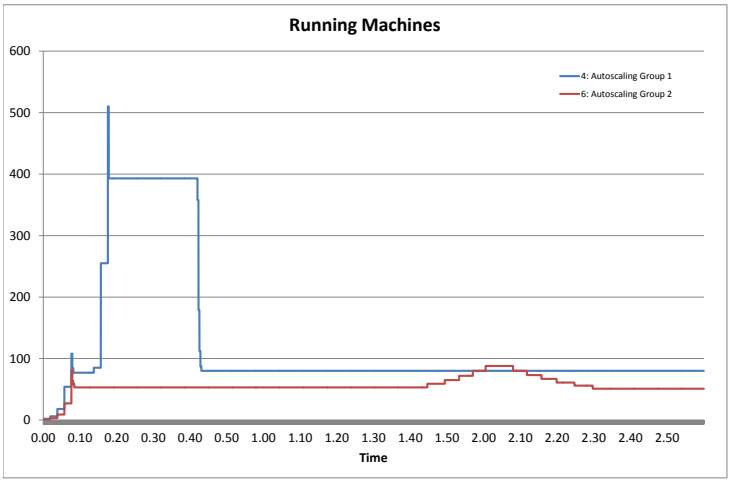


Figure 31: Number of VMs of Scenario 2

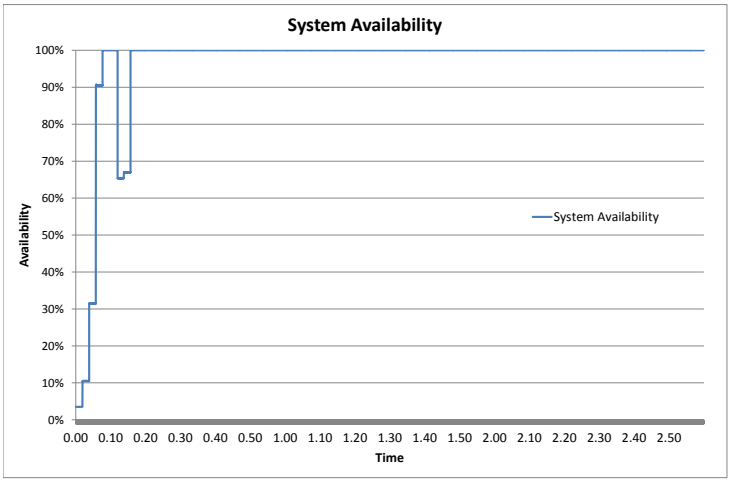


Figure 32: Overall availability of the system in Scenario 2

6.1.3 Scenario 3

This last Scenario for the first use case allow us to show the capability of the simulation tool to deal with variable workloads. Leaving all parameters of Table VI untouched we changed the number of requests that enter the system as shown in Figure 33.

We used a bi-modal distribution to generate the workload since it represents a common pattern that can be found in traces of web applications since it models. The distribution is made of a base number of requests which is $1e6\frac{req}{s}$ and two peaks centered respectively at 00:45 and at 02:00. The first peak spans 30 minutes and has a maximum of $6e6\frac{req}{s}$ while the second lasts for 1 hour reaching a maximum of $9e6\frac{req}{s}$.

Figure 34 shows the CPU utilization of the two autoscaling groups. Similarly to the two previous scenarios the first minutes of the simulation are used to reach a stable configuration in which the system is capable of serving all users requests and no scaling action is taking place. At minute 00:30 the workload starts increasing and the CPU utilization of both autoscaling groups grows as well reaching very rapidly 100%. The scaling policy defined in Table VI reacts to the increased CPU utilization by adding up more VMs to the autoscaling group.

As shown in Figures 35 and 34 the first reaction of the scaling policy to the incremented workload is not enough to keep the utilization level in the safety range specified by the scaling policy so after few minutes more VMs are added to the autoscaling group. As more requests come into the system and the CPU utilization surpasses the triggers defined in the scaling policy more VMs are added.

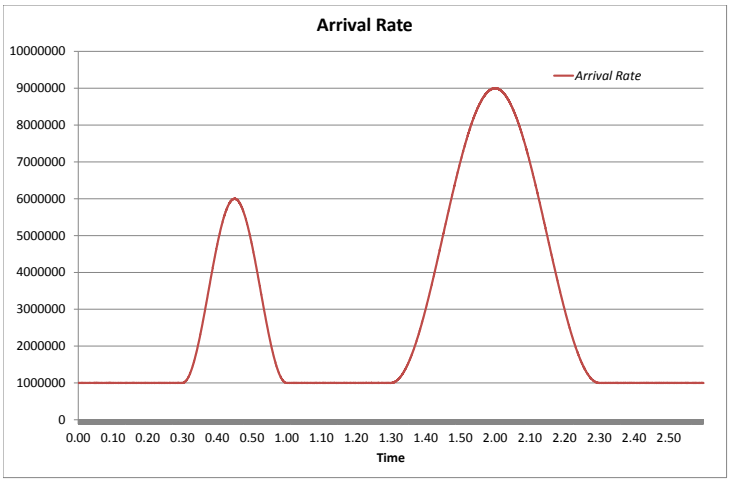


Figure 33: Bimodal distribution of incoming requests of Scenario 3

In the second part of the peak the number of requests starts decreasing and so does the CPU utilization of the autoscaling group. Scaling policies then reduce the number of VMs in order to keep the utilization level over 60%. At the end of the first peak of requests the system goes back to a stable configuration.

At time 01:45 the workload starts to increase again but at a lower rate than the first peak. Even if the maximum number of requests for the second peak is higher than the one of the first peak the scaling policy is capable of increasing the number of VMs so that the CPU utilization of both autoscaling groups reaches 100% only for a very short time.

The effects of the scaling policy and the workload on the availability is shown in Figure 36. During the first peak of requests the system suffers a severe loss of availability because the slope

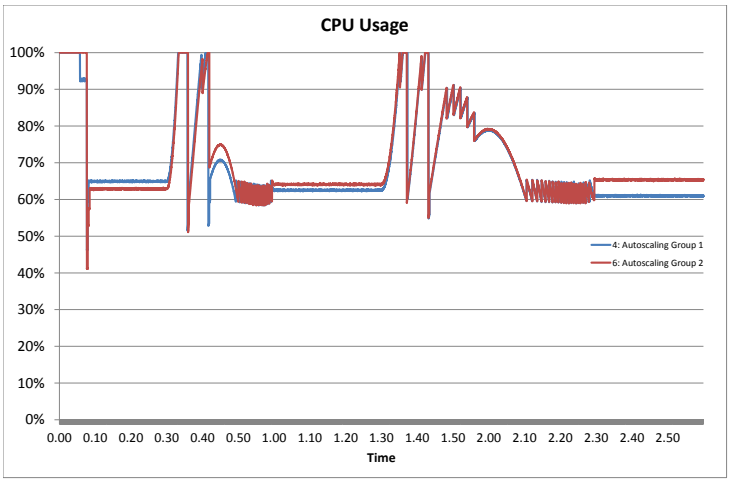


Figure 34: CPU utilization of autoscaling groups of Cloud 1 and 2 of Scenario 3

of the peak is very steep and the scaling policy can not compensate rapidly the new incoming requests by adding enough machines. During the second peak the availability of the system is less affected even if the maximum amount of requests is higher because the slope is lower.

The results of the simulation shows that the overall availability of the system is 98.68% and the cost is 361.78$

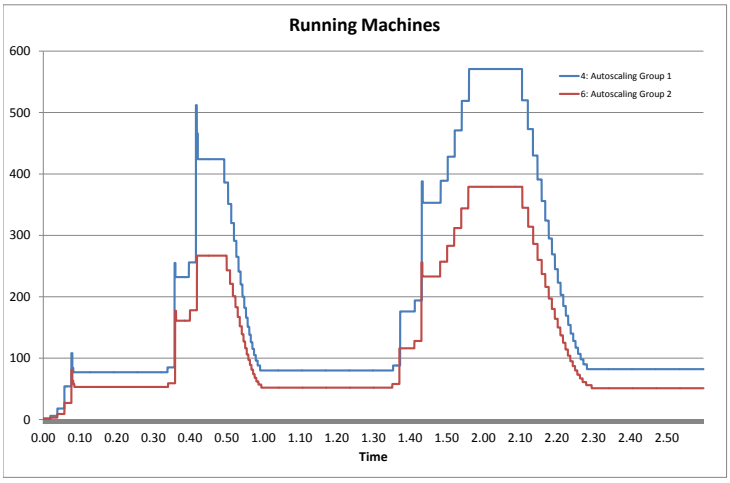


Figure 35: Number of VMs for Scenario 3

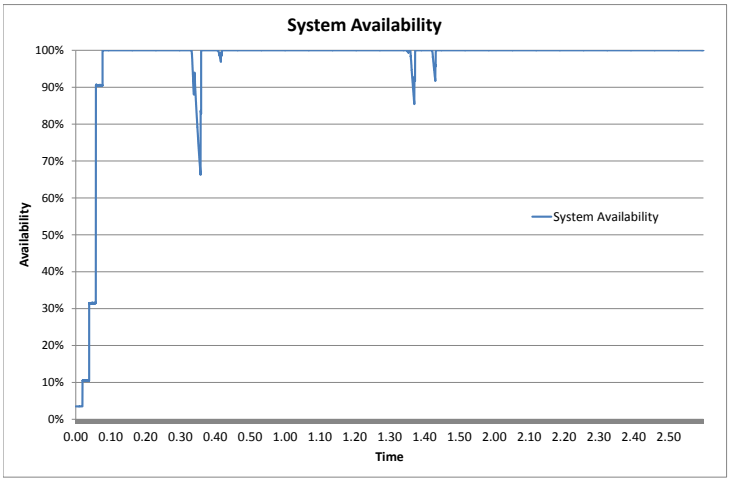


Figure 36: Overall availability of the system of Scenario 3

6.2 A Multi-Region Use Case

The second use case allow us to mix some of the different working conditions introduced with the use case of Section 6.1. In order to analyze a more challenging scenario we expanded the model used in the previous use case by splitting one of the two Cloud providers in two regions. The deployment of the application as a Palladio diagram is shown in Figure 37. Requests entering the system are directed by the first load balancer to one of two Cloud providers. Requests that are directed to Cloud 1 are then forwarded either to region 1 or region 2 by a second load balancer internal to the Cloud. The mapping from the Palladio model to the DTMC is shown in Figure 38.

Table VIII shows performance parameters for this use case.

Table IX shows the scaling policy defined by the application developer.

The scenario in this use case simulates 24 hours of usage of an application deployed on three Cloud providers. In order to make the scenario more realistic all the three aspects introduced

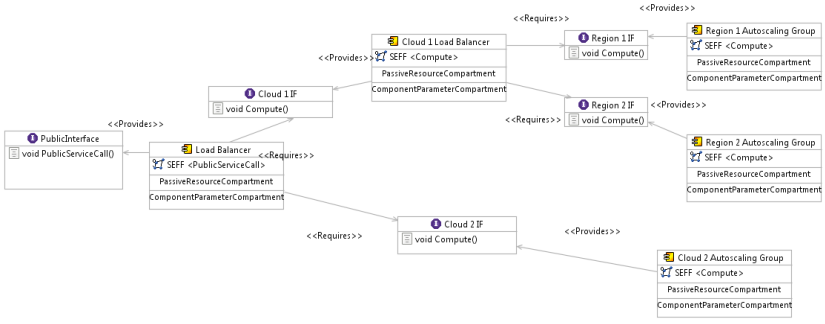


Figure 37: Application deployment of Use Case 2

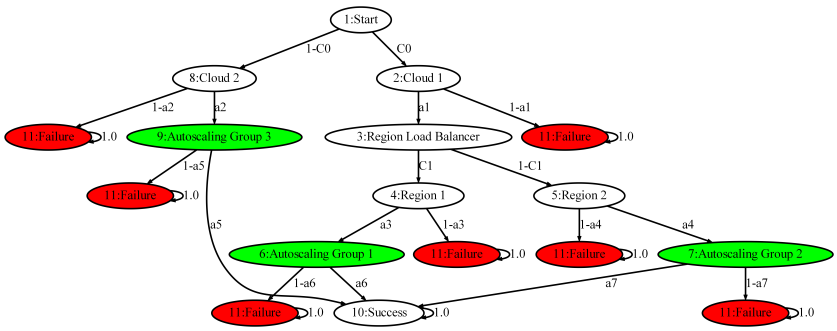


Figure 38: DTMC represneting the deployment of the application in Use Case 2

TABLE VIII: PERFORMANCE PARAMETERS OF USE CASE 2

	Cloud 1 (R1)	**Cloud 1 (R2)**	**Cloud 2**
Cost per VM	0.5$/hr	0.5$/hr	0.3$/hr
VM startup time	80 s	100 s	50 s
VM nominal SR	$10,000\ \frac{\text{reqs}}{\text{s}}$	$15,000\ \frac{\text{reqs}}{\text{s}}$	$10,000\ \frac{\text{reqs}}{\text{s}}$

TABLE IX: SCALING POLICIES OF USE CASE 2

CPU scale out policy	100%	3×
	90%	2×
	85%	1.5×
	80%	1.1×
CPU scale in policy	20%	$1/2\times$
	40%	$1/1.6\times$
	50%	$1/1.3\times$
	60%	$1/1.1\times$

in Section 6.1, dynamic arrival rate, variable Cloud availability and service rates, have been introduced in this scenario.

During the simulated period the incoming workload of the application is characterized by a bimodal distribution with two peaks occurring at time 10:00 and 16:00 as shown in Figure 40. This is a common pattern in real systems that are usually more stressed at daytime with peaks during working hours.

The three Clouds are characterized by different service rate profiles, in particular:

- *Cloud 1 - Region 1* shows a degradation of its service rate at night, especially between 23:00 and 06:00. This aspect simulates the influence of other application running in the same region performing batch processing or a middleware system update that is usually done at night to limit service disruption.
- *Cloud 1 - Region 2* shows the nominal service rate of $1.5e4\frac{req}{s}$
- *Cloud 2* shows the nominal service rate of $1e4\frac{req}{s}$ for most of the time but has a small degradation around time 13:00 due to a peak of requests in other applications running on the same datacenter.

The three Clouds show the following availability profiles:

- *Cloud 1 - Region 1* has a total blackout from 14:00 to 16:00. This evet may be caused by a lack of connectivity to that provider.
- *Cloud 1 - Region 2* shows a slight degradation of service between 09:30 and 11:00.
- *Cloud 2* is fully available all the time.

In addition to the failures described above another limitation has been introduced in this scenario. The total number of VMs that Cloud 2 can provide to run the application has been capped to 100. This is a common situation because if a user decide to provide its application in a Cloud environment he would probably make a contract in order to get discount on certain amount of VMs. We can think about Cloud 2 as a particular type of VMs for which we have such kind of contract.

Figure 41 shows the availability of the entire system during the simulation. We can see that the system has two main losses of availability.

The first loss happens near time 10:00. This loss is due to a combination of factors. First of all the increasing number of requests makes the number of needed VMs grow. When Cloud 2 reaches his maximum number of available VMs, as shown in Figure 39 it starts to reject part of the requests that are sent to it. In addition to this failure the availability of region 2 of Cloud 1 decreases. When the number of requests decreases and the availability of Cloud 2 is restored the system goes back in a state where all received requests are fulfilled.

The second loss of availability happens between time 14:00 and 17:00. The sudden loss of availability is due to the blackout of region 1 of Cloud 1 that was processing slightly more than 30% of all the requests. After this sudden loss of availability there is another gradual degradation that is again due to the fact that Cloud 2 has reached the maximum number of available VMs and can not scale out further.

If we look at Figures 39 and 42 we see that the average CPU load and the number of machines of region 1 in Cloud 1 before 05:00 and after 23:00 is higher than during their corresponding

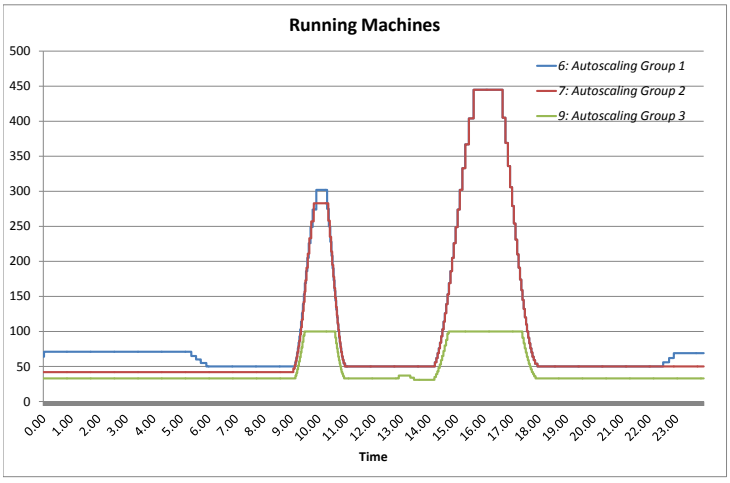


Figure 39: Number of VMs of use case 2

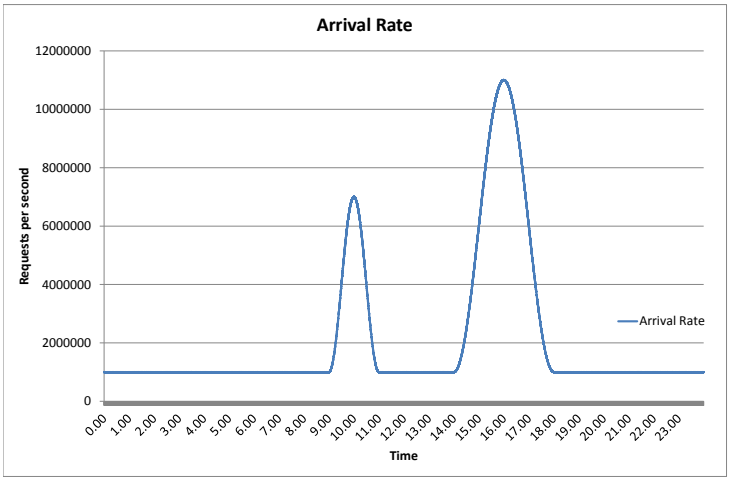


Figure 40: Arrival rate of use case 2

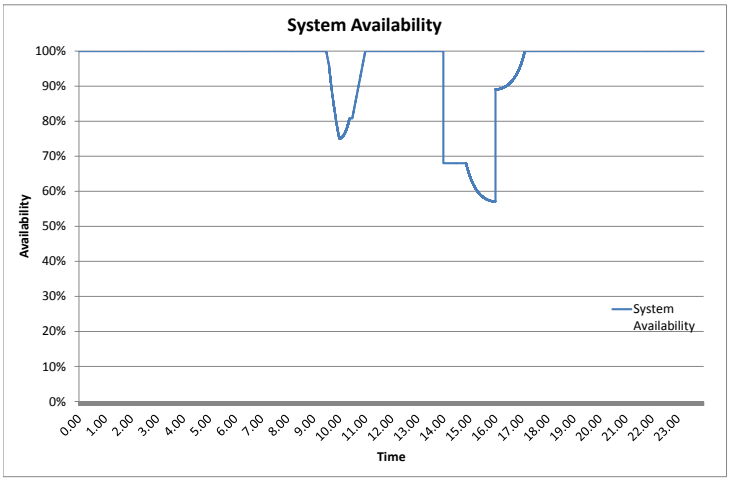


Figure 41: System availability of use case 2

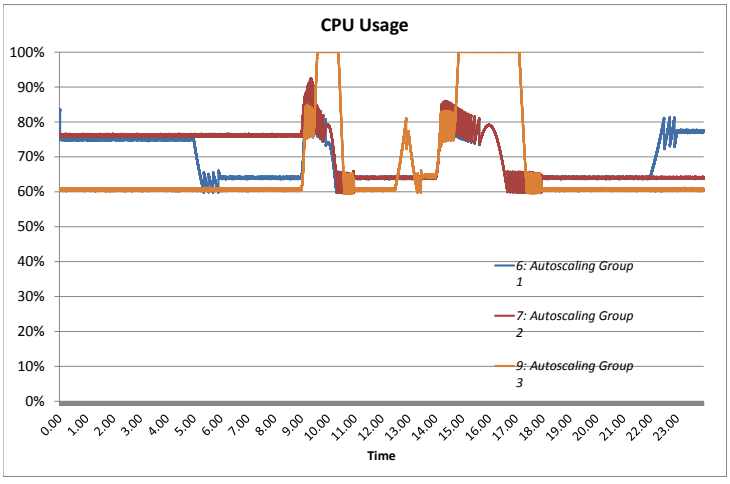


Figure 42: Average CPU load of use case 2

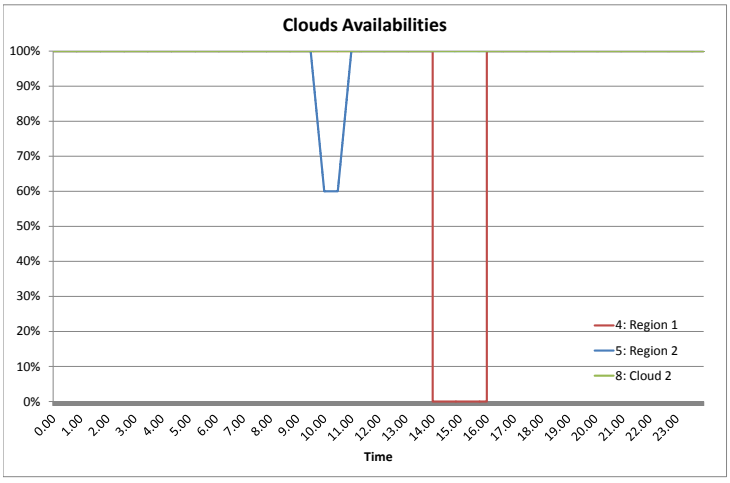


Figure 43: Cloud provider availability of use case 2

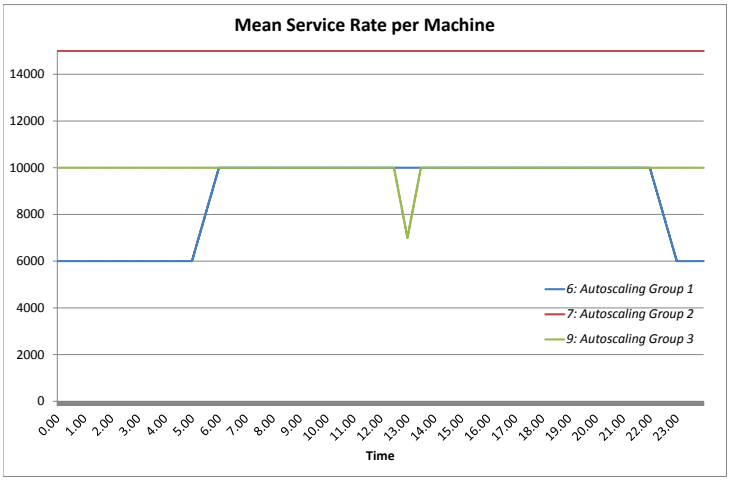


Figure 44: Maximum VMs service rate of use case 2

values in the steady state. This is due to the fact that region 1 shows a lower service rate during that time. The same happens to Cloud 2 near time 13:00. The fact that this behavior does not affect Figure 41 shows that the control policies defined by the application administrator are able to cope with this kind of failures.

6.3 Results analysis

Scenarios presented in this Chapter cover a variety of different failures and working conditions that could happen in a real Cloud environment. These results show that the simulation tool is capable of reproducing combinations of different factors affecting the availability of an application deployed in a Cloud enviroment.

All the simulation presented in this Chapters have been performed on a laptop with an Intel Core i7 CPU (Q740) running at 1.73GHz and 8GBs of RAM. The time needed to simulate scenarios presented in Section 6.1 never exceeded 15 minutes and the time needed to simulate the 24 hours scenario of Section 6.2 was of 1 hour and 33 minutes. The time needed to simulate the system could also be reduced by changing a parameter in the simulation tool that define the number of steps for the simulation. The simulations presented here have been run with the default value of 1 that produces 1 simulation step for each simulated second. A less accurate simulation of the 24 hours use case that still presents all of the system behaviors presented in Section 6.2 have been produced in less than 5 minutes by using a simulation step every 20 seconds.

The flexibility of the simulation tool makes it useful to application developer to build and test their control policies according to expected failure scenarios.

CHAPTER 7

CONCLUSIONS

In this thesis we approached the evaluation of model based application provided in a Cloud environment.

First, we extended the state of the art by augmenting the model used to describe a service oriented application from the availability viewpoint to cope with Multi-Cloud applications. In particular, we proposed to model each state of the DTMC as a resource with a processing capacity. Each state can model a component with fixed capacity or a scalable one. Scalable nodes are used to model autoscaling groups of a generic Cloud provider. Therefore, we introduced the concept of virtual machine with its cost per hour and service rate inside the model.

We then developed a simulation tool in Matlab capable of testing an application against different working conditions. We extended the already existing modeling integrated environment Palladio Bench to allow developers to model their Multi-Cloud application and generate models compliant to the simulation tool. The simulation tools allows the specification of scaling control policies so that Cloud application administrators can test at the same time the architectural choices and the dynamic behavior of the system.

Both the model and simulation tool has been built in order to be extended by allowing the addition of more complex control logic both from the scaling point of view and for the request distribution. Future improvements could aid the developer in the specification of complex control policies in order to build more failure tolerant Cloud applications.

As a future research both model and simulation could be improved by providing more realistic descriptions and features, according to the current solutions offered by Cloud providers like hourly price, on spot instances and other type of contracts. An interesting evolution could be the definition of a database of real world working conditions derived from historical logs of data centers.

Another interesting topic to investigate for future research is how the scaling policy affects the availability of the system. The author of[42] proposes an autoscale controller that utilizes the simulation system presented in this thesis to control the deployment of the configuration in terms of autoscaling actions and traffic redirection at the load balancer level in order to guarantee availability requirements.

CITED LITERATURE

[1] Bitcurrent: Cloud performance from the end user. Technical report, http://www.bitcurrent.com/, 2010.

[2] Brosig, F., Huber, N., and Kounev, S.: Descartes Meta-Model (DMM). Technical report, Karlsruhe Institute of Technology (KIT), 2012. To be published.

[3] Mao, M. and Humphrey, M.: A performance study on the vm startup time in the cloud. pages 423 –430, June 2012.

[4] Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., and Zaharia, M.: A view of cloud computing. Communications of the ACM, 53(4):50–58, April 2010.

[5] Amazonec2. http://aws.amazon.com/ec2/. Accessed date 07/10/2013.

[6] Rackspace. http://www.rackspace.com/cloud/. Accessed date 07/10/2013.

[7] Terramark. http://www.terremark.com/services/infrastructure-cloud-services/enterprise-cloud.aspx. Accessed date 07/10/2013.

[8] Salesforce. http://www.force.com/. Accessed date 07/10/2013.

[9] Appengine. https://developers.google.com/appengine/. Accessed date 07/10/2013.

[10] Googleapps. http://www.google.it/intl/it/enterprise/apps/business/. Accessed date 07/10/2013.

[11] Netsuite. http://www.netsuite.com/portal/home.shtml. Accessed date 07/10/2013.

[12] Freshbooks. http://www.freshbooks.com/. Accessed date 07/10/2013.

[13] Hotmail. http://it.msn.com/. Accessed date 07/10/2013.

[14] Amazons3. http://aws.amazon.com/s3/. Accessed date 07/10/2013.

[15] Avizienis, A., Laprie, J. C., Randell, B., and Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. IEEE Trans. on Dependable and Secure Computing, 1(1):11–33, 2004.

[16] Danilo Ardagna, M. C.: D51 - analysis of the state of the art and scope definition. Public deliverable, 2013.

[17] Baier, C. and Katoen, J.-P.: Principle of Model Checking. April 2008.

[18] Petcu, D.: Portability and interoperability between clouds: Challenges and case study. 6994:62–74, 2011.

[19] Hogan, M. D., Liu, F., Sokol, A. W., and Jin, T.: Cloud computing standards roadmap. 2011.

[20] Rackspace. http://www.rackspace.com/cloud/public/servers/compare/. Accessed date 07/10/2013.

[21] Akamai. http://www.akamai.com/. Accessed date 07/10/2013.

[22] Becker, S., Koziolek, H., and Reussner, R.: The palladio component model for model-driven performance prediction. J. Syst. Softw., 82(1):3–22, January 2009.

[23] Becker, S., Koziolek, H., and Reussner, R.: The palladio component model for model-driven performance prediction. Journal of Systems and Software, 82(1):3 – 22, 2009. Special Issue: Software Performance - Modeling and Analysis.

[24] Franks, G., Maly, P., Woodside, M., Petriu, D., , and Hubbard, A.: Layered Queueing Network Solver and Simulator User Manual. Real-Time and Distributed Systems Lab, Carleton Univ, Canada, 2009.

[25] Cortellessa, V., Marco, A. D., and Inverardi, P.: Model-Based Software Performance Analysis. Springer, 2011.

[26] Zimmermann, A.: Modeling and evaluation of stochastic petri nets with timenet 4.1. In Performance Evaluation Methodologies and Tools (VALUETOOLS), 2012 6th International Conference on, pages 54 –63, Oct. 2012.

[27] Hirel, C., Tuffin, B., and Trivedi, K. S.: Spnp: Stochastic petri nets. version 6.0. In Proceedings of the 11th International Conference on Computer Performance Evaluation: Modelling Techniques and Tools, TOOLS '00, pages 354–357, London, UK, UK, 2000. Springer-Verlag.

[28] Bini, D. A., Meini, B., Steffé, S., and Van Houdt, B.: Structured markov chains solver: software tools. In Proceeding from the 2006 workshop on Tools for solving structured Markov chains, SMCtools '06, New York, NY, USA, 2006. ACM.

[29] Hirel, C., Sahner, R. A., Zang, X., and Trivedi, K. S.: Reliability and performability modeling using sharpe 2000. In Proceedings of the 11th International Conference on Computer Performance Evaluation: Modelling Techniques and Tools, TOOLS '00, pages 345–349, London, UK, UK, 2000. Springer-Verlag.

[30] Trivedi, K.: Srept: a tool for software reliability estimation and prediction. In Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on, page 546, 2002.

[31] Pipe. http://pipe2.sourceforge.net/index.html. Accessed date 07/10/2013.

[32] Koziolek, H.: Performance evaluation of component-based software systems: A survey. Performance Evaluation, 67(8):634 – 658, 2010. Special Issue on Software and Performance.

[33] Bertolino, A. and Mirandola, R.: Cb-spe tool: Putting component-based performance engineering into practice. In PROC. 7TH INTERNATIONAL SYMPOSIUM ON COMPONENT-BASED SOFTWARE ENGINEERING (CBSE 2004, pages 233–248. Springer, 2004.

[34] Kamath, M., Sivaramakrishnan, S., and Shirhatti, G.: RAQS: A software package to support instruction and research in queueing systems. In Proceedings of the 4th Industrial Engineering Research Conference, IIE, Norcross, GA., pages 944–953, 1995.

[35] Wu, X. and Woodside, M.: Performance modeling from software components. SIGSOFT Softw. Eng. Notes, 29(1):290–301, January 2004.

[36] Grassi, V., Mirandola, R., and Sabetta, A.: From design to analysis models: a kernel language for performance and reliability analysis of component-based systems. In Proceedings of the 5th international workshop on Software and performance, WOSP '05, pages 25–36, New York, NY, USA, 2005. ACM.

[37] Machida, F., Andrade, E., Kim, D. S., and Trivedi, K.: Candy: Component-based availability modeling framework for cloud service management using sysml. In Reliable Distributed Systems (SRDS), 2011 30th IEEE Symposium on, pages 209 –218, Oct. 2011.

[38] Epifani, I., Ghezzi, C., Mirandola, R., and Tamburrelli, G.: Model evolution by run-time parameter adaptation. In Proceedings of the 31st International Conference on Software Engineering, ICSE '09, pages 111–121, Washington, DC, USA, 2009. IEEE Computer Society.

[39] Filieri, A., Ghezzi, C., Leva, A., and Maggio, M.: Self-adaptive software meets control theory: A preliminary approach supporting reliability requirements. pages 283 –292, Nov. 2011.

[40] Almeida, J. M., Almeida, V. A. F., Ardagna, D., Cunha, I. S., Francalanci, C., and Trubian, M.: Joint admission control and resource allocation in virtualized servers. J. Parallel Distrib. Comput., 2010.

[41] Schad, J., Dittrich, J., and Quiané-Ruiz, J.-A.: Runtime measurements in the cloud: observing, analyzing, and reducing variance. Proc. VLDB Endow., 3(1-2):460–471, September 2010.

[42] Marco, M.: Model Based Control for Multi-Cloud Applications. Master's thesis, UIC, Chicago, 2013.

[43] Number, O. M. G. D. and Files, A.: Uml profile for marte : Modeling and analysis of real-time embedded systems. Engineering, 15(November):738, 2009.

[44] Reussner, R., Becker, S., Happe, J., Koziolek, H., Krogmann, K., and Kuperberg, M.: The Palladio component model. Karlsruhe, 2007. http://digbib.ubka.uni-karlsruhe.de/volltexte/1000007341.

VITA

NAME: Giovanni Paolo Gibilisco

EDUCATION: Bachelor of Science in **Engineering of Computing Systems**, **Politecnico di Milano**, 2009

Master of Science in **Engineering of Computing Systems**, **Politecnico di Milano**, 2012

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

PUBLICATIONS: Model Based Control for Multi-cloud Applications - MiSE 2013