POLITECNICO DI MILANO

Facoltà di Ingegneria dell'Informazione
Corso di Laurea Specialistica in Ingegneria Informatica
Dipartimento di Elettronica e Informazione

Model Based Control for Multi-Cloud Applications

Relatore: Ing. Elisabetta DI NITTO
Correlatore: Ing. Danilo ARDAGNA

Tesi di laurea di
Giovanni Paolo GIBILISCO - Matr. 755066
Marco MIGLIERINA - Matr. 754848

Anno Accademico 2011/2012

to Silvia, my family and all who sustained me in these years
Giovanni Paolo

to Eleonora, my family, and my roomies
Marco

Contents

List of Figures

List of Tables

Abstract

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 their 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 application that usually require a much higher value of availability. In order to fulfill this requirement one could decide to replicate the deployment of its application on multiple clouds and use only the one that works best at a given time.

In this thesis we propose a model for the high level description of availability requirements of Multi-Cloud applications and a controller able to guarantee the desired availability. In order to automatically generate the defined model, used for control, we developed an extension to the Palladio Bench modeling software. The controller monitors the state of the system at runtime, updates the model and intervenes both in the machine scaling management and in the distribution of requests within clouds. The overall goal of the control system is to minimize costs, while satisfying the availability

requirements.

In order to test our control approach against different usage scenarios we have implemented a simulation engine. Tests on the control system against common usage scenario shows that our controller is capable of minimizing the cost of running the application while respecting the availability requirement. It is also capable to recover quickly from different kind of cloud or network infrastructure failures.

Sommario

Negli ultimi anni, l'utilizzo di architetture cloud ha cambiato il modo in cui le applicazioni sono distribuite agli utenti. Il crescente numero di fornitori di servizi cloud e di compagnie che utilizzano tali servizi é un importante indice della popolaritá di questo tipo di architettura. Il cloud computing offre una soluzione efficace al problema dell'approvvigionamento di risorse IT fornendo su richiesta risorse virtualmente illimitate nel giro di alcuni minuti. Tipicamente le risorse messe a disposizione dai fornitori di servizi cloud sono offerte mediante una politica *pay-as-you-go*, in questo modo gli utenti di servizi cloud possono mantenere sotto controllo i costi infrastrutturali delle loro applicazioni modificandone la struttura in base alle loro esigenze.

Uno dei maggiori problemi affrontati da parte delle compagnie nella scelta di utilizzare una piattaforma di questo tipo é la perdita di controllo sulla gestione dell'infrastruttura. Le aziende che utilizzano i servizi cloud sono preoccupate dalla possibilitá di interruzioni di servizio che non possono essere gestite direttamente. Per far fronte a questo problema i fornitori di servizi cloud offrono accordi a livello di servizio in cui viene specificato il valore di disponibilitá del sistema che si impegnano a fornire.

La maggior parte dei fornitori offre un valore di availability pari al 99.95%. Dati reali mostrano che il valore di disponibilitá sperimentato da gli utenti del cloud é spesso inferiore, e si attesta sul valore di 95%. Un valore cosí basso non puó essere accettato dagli sviluppatori di applicazioni critiche che, tipicamente, richiedono una availability molto alta. Per soddisfare tale vincolo di availability si potrebbe decidere di replicare l'applicazione su diversi fornitori di servizi cloud e scegliere in ogni momento quale servizio utilizzare in base alla sua attuale availability.

In questa tesi proponiamo un modello per la descirizione di requisiti di availability di applicazioni replicate su piú cloud. Proponiamo inoltre un controllore capace di garantire il livello di availability richiesto dall'applicazione. Al fine di generare automaticamente una istanza di tale modello abbiamo esteso il software di modellazione Palladio Bench. Il controllore monitora lo stato del sistema durante il suo funzionamento, aggiorna il modello ed in-

terviene sia a livello di scalabilitá delle risorse cloud, sia nella distribuzione delle richieste entranti nel sistema tra piú cloud. L'obiettivo del controllore é minimizzare i costi delle risorse utilizzate e, allo stesso tempo, soddifare il vincolo di availability.

Al fine di verificare il comportamento del nostro approccio di controllo in diversi scenari di utilizzo abbiamo implementato uno strumento di simulazione. Le prove di controllo effettuate su diversi scenari di utilizzo mostrano che il controllore é capace di minimizzare il costo di utilizzo delle risorse cloud e rispettare il vincolo di availability dell'applicazione. Inoltre, il controllore é in grado di ripristinare il livello di servizio desiderato a fronte di diversi tipi di fallimento della piattaforma cloud o dell'infrastruttura di rete.

Chapter 1

Introduction

The advent of cloud computing has offered developers a new way of building services and offering them to the public. This new appealing paradigm has been widely accepted by the community of both developers and of companies which are deciding to move the services they offer into a cloud environment for economical reasons. With this new practice, system administrators can acquire resources in a much more flexible, scalable and rapid way than before. Cloud providers let users pay only for the resources they use and give them the possibility to acquire a potentially infinite pool of resources in matter of minutes [4]. Though, no cloud provider offers a native mechanism to guarantee the Quality of Service (QoS) required by specific application domains.

At present, there are many providers that offer cloud services and, since this is a quite new and profitable market, more providers are appearing. The choice of which cloud provider to trust is not an easy one. Each provider offers specific APIs and programming / design paradigms. Thus, moving from a provider to another involve, in many cases, re-writing part of the code, moving large databases from a technology to the other and manually re-deploy applications.

Our research interest is to identify proper modeling mechanisms that allow us to keep availability of a cloud-based application under control. Availability is a non functional property of an application which measure the portion of time in which the system behaves correctly. The usual way to achieve a high availability is replication of critical components or components that are more subject to failure, this is a quite common and successful practice but it is not very effective from the economic point of view, because it involves the acquisition of backup resources which are left unused for most of the time and exploited only in case of a failure of the primary system.

Reducing operative costs is something that companies always try to do,

on the other hand they also need a reliable architecture on which running their applications. In many contexts a period of downtime of the system generate losses that can not be balanced by the savings due to the utilization of a cheaper architecture. Examples of this are mission critical applications. If a company decides to move its services to the cloud for economical reasons it accepts the fact that it loses some control on the system on which its service run.

Users could decide to make this choice by looking at service level agreements (SLA) offered by cloud providers and choosing the one that provides them the highest availability. If we look at cloud providers' SLA we can observe that many of them offer 99.95% of availability. As an example, 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" and if this availability value is not met for some reasons depending 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] guarantees to provide 99.95% availability of internet connectivity to users virtual machines and 99.9% of uptime of virtual machine instances evaluated on a monlty basis. Both these providers require the user to deploy at least two machines in separate regions (or availability zones in case of Amazon) in order for their SLA to take effect.

These data alone are not representative of the real behavior of the cloud environment because are just nominal agreement values. In order to decide if an application can be safely moved into the cloud a company should evaluate the economical loss in case of a failure of cloud provider services using some more realistic data. In [3], a study has been conducted to analyze availability of cloud providers. These data shows that the availability values experienced by users of a cloud based service is much less than the one declared by SLAs. 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%.

Looking at this data it is clear that running mission critical applications in this kind of environment is a risky decision. In order to run applications with high availability requirements on the cloud, users could exploit the fact that usually cloud failures are independent of each other. Users can deploy their applications on the cloud provider that offers the highest availability at a given time and switch to another one if its availability falls behind a

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

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

certain value or if it is more convienient from an economical point of view.

Our thesis has been developed in the context of the *MODAClouds* project which is an European community project that aims to ease this commitment choice 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. A very attractive solution of the availability problem is the use of the flexibility of resources offered by cloud, this new approach could help companies that run highly available application to save money required for the provisioning of backup resources.

1.1 Thesis objectives

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

To reach this goal, we defined a model to describe availability requirements of Multi-Cloud applications and a two layer controller able to manage both in-cloud configuration policies and traffic routing through different cloud providers, keeping the model alive at runtime. The controller's objective is to guarantee high availability, while reducing costs. 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 to test our controller on different scenarios. We evaluated our approach through three different use cases: a web system scenario with two single-region clouds, and a multi-region scenario, and a smart city scenario.

Our work starts from the assumption that the application is already able to migrate from cloud provider to cloud provider. From what we already said, this is a pretty relevant assumption but as we will see in Section 2 there are many active projects dealing with it.

1.2 Structure of the thesis

Chapter 2 gives an overview about non functional requirements like availability, the Discrete Time Markov Chain (DTMC) models usually adopted for availability evaluations. It then proceeds by introducing some of the characteristics of the major cloud providers. This section does not aim at showing a complete list of cloud computing offers but at helping the reader to understand similarities and differences between cloud providers in order

to underline the possibilities offered by cloud computing and the challenges of application portability and control in this environment, the section ends with a small survey on some of the main approaches that try to solve the problem of portability between clouds.

Chapter 3 shows state of the art tools that have been exploited in order to build our control system, it introduces Palladio, a tool for designing applications capable of deriving a DTMC model from different diagrams built at design time, and some control techniques that have been used as a basis to build our controller.

Chapter 4 shows the innovative contribution of this thesis, it introduces an extension to the DTMC model which allows to annotate some properties peculiar to the cloud environment that are used later on for the simulation and control of the system. It describes additional properties of nodes of the DTMC, introduces control variables and characterizes parameters specific to the modeled case study that are necessary to initialize the control system. The second part of the chapter shows the extension we proposed to the controller introduced in Chapter 3 in order to deal with our new model and to perform adaptation on both in-cloud scalability and clouds orchestration.

Chapter 5 goes through the implementation of the extensions to Palladio, developed in Java, and the implementation of the simulation tool developed in Matlab.

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

Chapter 7 summarizes this thesis contribution and provides an overall evaluation of our approach, pointing out some future work that we consider worth to be investigated.

Chapter 2

Background

In this chapter we will provide some definitions and the minimum background knowledge required to better understand our work and in order to have 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 reward 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 Sofware 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 an emerging technology born by the idea of big IT companies of renting some of their computing capacity when it was not needed by the company itself. For example Amazon developed EC2 for internal purpose of reducing maintenance costs of its worldwide infrastructure, later on it made publicly available its service and started the business of cloud computing. The main advantages from a developer's point of view of cloud computing are its very low start-up costs, the fact that there is no effort required to manage hardware on which his application runs and the immense computing capacity cloud can offer. We now give a brief overview of some of

the major cloud providers in order to show to the reader the wide spectrum of offers that are available as cloud services, the similarity and the difference between cloud providers. This section should make the reader aware of the difficulties in choosing the best cloud provider for its needs and the challenges in provide portability of applications between cloud providers. This section ends with an overview of the work in progress to ease this process of migration. *Cloud computing*, as stated in [5, 3], refers to both applications delivered as services over the Internet, hardware and systems software in the data centers that provide those services. Cloud computing is very new and dynamic field of IT, it emerged in the last few years and a clear schema of all its aspects has not been developed yet. This is mainly due to the variety of objects offered *as a service* from cloud providers, among them we can count Software, Infrastructure, Platform, Storage, Data, API and much more. Other than the variety of services offered by cloud providers there are other factors that increase the complexity, for example the way these services are offered. *Clouds* can be *private* or *public*: By *private cloud* we refer to internal data centers of an organization which cannot be accessed by third parties; vice versa *public clouds* are publicly available on a *pay-as-you-go* basis. A third type of cloud has emerged with the name of *hybrid* cloud because it's made of a composition of a private IT infrastructure (e.g. a private cloud or a DBMS) and a public cloud. An example could be an *Hybrid Web Hosting* where Web servers are hosted in the cloud, while the database servers are situated in the internal network of a company, this is usually done in order to maintain critical data in a more controlled environment.

Among all others the main categorization that has been done for cloud services in this one:

- **Infrastructure-as-a-Service (IaaS)**: The provider let the user upload his own *virtual machine* or choose from a pre-configured set, in both cases the user is responsible for the operating system, the application stack and so on. The pricing methods for this service are based on the resource size of the user virtual machine(s) (e.g. CPU cores, RAM GBs). Examples are *Amazon EC2* [6], *Rackspace Cloud* [7] and *Terremark's cloud* [8].

- **Platform-as-a-Service (PaaS)**: The provider offers *code execution*. Operating system managment is done by the provider and user does not have to cope with security updates or failures of the OS. The user provide the application code (language support is usually limited) and the cloud platform takes care of its execution. Examples are *Salesforce's Force.com cloud* [9] and *Google's App Engine* [10].

- **Software-as-a-Service (SaaS)**: The provider offers an entire *application* as a service. The user can make use of the application by calling APIs or in other ways (e.g. Google apps, mailing servicies); usually a monthly per user fee is charged for the usage. Examples are *Google applications* [11], *Netsuite* [12], *Freshbooks* [13] and *Hotmail* [14].

Choosing the appropriate model for the user needs is a key factor for the success of user's application. For example the IaaS model gives a finer control on resources because it is possible to choose the operating system, the programming language, administration tools and so on, this finer control is good for CPU-intensive applications but requires a big effort to manage the system. For example scalability of the application in a IaaS context has to be managed by the system administrator which has to carefully choose the number and size of VMs needed and build strong rules to cope with traffic changes. In a PaaS model VMs management is delegated to the provider of the service so the user has only to build an application that is scalable in the sense that it may run on multiple instances but the management of the operating system of VMs and their scaling is managed by the provider, a clear advantage is the fact that usually each PaaS provider has some VMs in a steady state which can be used for autoscaling without waiting the usual boot up time that is needed in IaaS. A drawback of PaaS is the fact that many applications of different users may run on the same system so the performance of a user application may be subject to the load of other users' application. To avoid this situation most of cloud providers implement artificial upper bounds using the so called "governors" [3].

For cloud users storage could also be an issue because they need to get content into the cloud. Not all cloud storage system have similar characteristics: For example *Google's Bigtable* is very fast in retrieving data [3] but is slow in insert operations. A critical point that the user should take care of while choosing the right system is the latency: For example in *Amazon Simple Storage Service (Amazon S3)* [15] it is possible to choose from many different regions when storing data to minimize the time needed to retrieve them; moreover this choice impacts the performance of data transfer and the pricing to store and transfer data. An interesting option offered by *Amazon* is the "AWS Import/Export": When a user wants to transfer a large amount of data (in case of migration, backup, disaster recovery and so on) she/he can send to Amazon some portable storage devices and they will be uploaded directly into the *Amazon S3* storage system. This is useful when the Internet connection speed is not enough to transfer all data in a reasonable time.

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 functionalities. 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 [16], some of the most common are

- *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 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 $\frac{MTTF}{MTTF+MTTR}$. High availability systems, like the one presented in Section 6.3, usually require an availability of 0.99999, or, as commonly said, 5 nines availability (a term indicating the number of 9s after the decimal point).

2.3 The Discrete Time Markov Chain with Reward

Discrete Time Markov Chains (DTMC) are known as a useful formalism to describe systems from the reliability viewpoint and to support reasoning about it. In [17] DTMC are described as graphs where nodes represent *states*, and edges model *transitions*, i.e., state changes, with a probability attached to them. A state describes some information about a system at a certain moment of its behavior. Transitions specify how the system can evolve from one state to another. The successor state of state, say, s is

chosen according to a probability distribution. This probability distribution only depends on the current state s, and not on, e.g., the path fragment that led to state s from some initial state. Among all states, one is the *initial state*. Among all the other states, one or more represent the successful completion of the execution or the occurrence of a failure. Failure and success states are modeled as absorbing states, i.e. states with a self-loop transition labeled with probability 1. 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.

States (or transitions) of *Markov Chains* can be augmented with rewards, numbers that can be interpreted as bonuses, or dually as costs. The idea is that whenever a state s is chosen, the reward associated with s is earned.

These kinds of model well fit the kind of application we are going to deal with, that are cloud applications. In fact we can consider the components of our system, that can be in-house solutions, services offered by external providers (see *SaaS* in Section 2.9) or applications deployed on third parties platforms (see *PaaS* or *IaaS* in Section 2.8 and 2.7), as states of a DTMC and let transitions model the workflow through these services. We can then add failure states and attach probability of failure to each transition coming from nodes whose availability can either be known a priori or estimated from the success rate of the service modeled by the node. Some other transitions can instead model requests distribution among different alternative services. Finally, since external services would probably be with fee, nodes can be equipped with costs.

Once a system is modeled using a DTMC with rewards, reliability of the system can be expressed as reachability properties, i.e., as a relational formula constraining the probability of reaching certain states that represent failure situations, plus, costs can be used to compute the price of reaching a certain state through a certain path. 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

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

Table 2.1: Availabilities of cloud providers from [3]

$\{x_i | s_i \in S\}$:

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

thus the item x_0 corresponds to the probability of reaching state s_R from the initial state.

2.4 Availability in the cloud

One of the main concern of system developers is that their application satisfy certain availability constraints, using the cloud as production environment for an application frees the system administrator of the effort of maintaining the system and moves it to the cloud provider. Companies that use cloud services trust their cloud providers that the system on which they deploy their applications works correctly all the time. This is usually defined in the contract made with the provider by a service level agreement. Many cloud providers' SLA guarantee 99.95% of availability over a year. In reality data shows that their expertimented availabilities are much lower as shown in table

Availability problems in such big and complex infrastructures are not new, examples of cloud failures are:

- Amazon S3 Availability Event that happened on july 20, 2008 that lasted for 8 hours that affected US and EU data centers, costumers who relied on that services experienced downtime of their applications.[1]

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

- Gmail major outage of february 24, 2009 caused Gmail users not being able to access their e-mail account for about two and a half hours [2]
- Amazon Relational Database outage of april 21, 2011 affected response time and availability of many popular sites like Foursquare, HootSuite, Quora and Reddit. [3]
- Gmail failure on march 1, 2011 caused some users to loose access to their accounts and deletion of all emails for some hours. [4]
- Hotmail outage on december 31, 2010 lasted for more than three days leaving empty in-boxes for many users. [5]

Even if cloud provider maintenance teams can discover problems or outages quite fast, investigating their cause and providing efficient response in such a complex infrastructure usually requires a quite long of time. Usually cloud providers grant free computing hours as compensation for the outages of the affected clients.

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

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

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

[4] http://gmailblog.blogspot.it/2011/02/gmail-back-soon-for-everyone.html

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

- 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 is crucial for our control approach so standardized metrics and monitoring tools are necessary for any kind of control approach to work. 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

One of the most important features introduced by cloud computing is the concept of *scaling*. In classical computing systems a company owns a fixed pool of resources on which its applications run. If the utilization of applications grows the resources of the company may not be able to cope with the required computing power so the company has to acquire new resources and expand its pool. This is usually a very expansive operation for a company and has some drawbacks. First of all the old pool of resources when saturated starts to reject requests so the quality of the service offered by the company decreases dramatically, this is the main drawback of a static

architecture. The second drawback is the fact that even if the company acquire new resources and integrate them in the current architecture if the workload for its applications return on normal values the new resources will be useless until another peak of requests arrives. This approach of acquiring much more resources than the one necessary in a normal situation to cope with traffic peaks is called *over provisioning* and can deal to very high economical damages to companies. Cloud computing has the power to offer new resources with very low prices in matter of minutes from the request of them, this ability can effectively solve the problem of company resource saturation. It also offers the opportunity to pay only for used resources and deallocate them according to companies needs, this ability is very useful to reduce the problem of over provisioning. A group of resources that is able to scale is called *autoscaling group*.

Every IaaS provider offers different kind of VM as resources, this kind varies in processing power and prices, some more details on resource types will be given in Section 2.7.1. So the user of the cloud environment can choose between two kind of scaling:

- *Vertical scaling* consist in changing the processing power of VMs. Some providers offers ways to add dynamically resources (e.g. virtual CPU cores, RAM, disks) to a running VM, others does not allow this kind of mechanism but require the user to startup a new VM with more resources and migrate traffic from the old one to the new one when it is ready to serve requests.

- *Horizontal scaling* consist in changing the number of running virtual machines by adding or removing VMs according to user defined rules. When a scale up request is performed VMs have to boot up before start serving requests, the delay between the request of scale and the actual effect depend on the cloud provider. Some of them keep machines in a stand by state ready to scale up without charging the user.

When to prefer virtual scaling against horizontal scaling is not easy and depends heavily on application requirements. Horizontal scaling is quite easy to perform and manage. It mostly affects the number of requests that are served simultaneously while the average processing time of each request remains the same because the pool of resources is, generally, homogeneous. Vertical scale is quite different because it may affect processing time for each single request, so it may be useful to reduce response time of the application if for example incoming requests requires high processing power and pass queues quite fast. A combination of this scaling approaches can be done simply by requiring resources with a different size of the one already available.

Both of this kind of approaches can obviously be exploited to increase or decrease the pool of available resources, the decision of releasing some resource in order to reduce costs is also not an easy one, sometime it is even harder scaling up. It is quite obvious that if the system is rejecting requests new resources are needed and the number of new VMs can be decided with very complex policies that takes into consideration different facts or by a fixed number decided at design time. The same can be said for releasing resources but the under utilization of the system can not be seen just observing the number of successfully served requests. Another aspect that makes scale down even more difficult than scale up is the fact that if the response of scale up in front of a peak of request is too strong, i.e. to many new resources are allocated, the effect is not perceived by the users of the service whose requests are going to be served normally but if actions taken to reduce the size of the poll of resources in case of underutilization is too strong, i.e. too many machines are shut down, the remaining resources will be overloaded and requests will be rejected causing QoS degradation.

In order to perform successfully scale up and scale down two components are requested to cloud provider, the first is a common load balancing layer which routes incoming traffic among the autoscaling group. Every cloud provider offers this possibility, some of them also allow the user to customize load balancer rules. A representative example for the load balancing capabilities of cloud provider is the one of Amazon shown in section 2.7.1.

Another requisite to perform autoscaling is monitoring. In order to effectively exploit scaling capacity of cloud providers system administrators have to be able to monitor how their machines are responding to requests and how the workload affect their performance. Monitoring can be done at application level or at system level. Many cloud provider offer some system level monitoring of key performance metrics for their VMs. Metrics that are usually provided are incoming and outgoing network traffic and CPU utilization. Each cloud provider then offers some metrics like Disk Read/Write operations that can be common among resources or other that are specific to the kind of resource that is monitored (e.g. free storage space of Amazon RDS DB, cache hit or miss of Amazon Elastic Cache). Different cloud providers also offers different sizes of the monitoring interval.

2.7 Infrastructure-as-a-Service (IaaS)

2.7.1 Amazon EC2

Amazon's cloud system provides an IaaS model service to users granting them complete control over assigned virtual machines. Management is made available through a Web interface where it is possible to launch instances, deploy a custom application environment, manage network's access permissions, and run images using as many or few systems as desired. These operations are made available also using dedicated APIs provided by Amazon, so that users can embed them directly inside applications to perform automatic scaling or management operations. Amazon Iaas offers 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 Table 2.2;
- choose the location of virtual machines between seven different regions, manage IP endpoint and block storage attached;
- automatically mange load balancing between active machines
- build custom scaling rules
- integrate storage with the Amazon S3 service

The pricing model is a pay-as-you-go both for instances, data transfer and storage. When creating a custom *AMI*, users can include software based on their needs. Amazon offers a list of available software to choose from, like *IBM DB2*, *Oracle Database 11g*, *MySQL Enterprise*, *Microsoft SQL Server Standard 2005*, *Apache HTTP*, *IIS/Asp.Net* and many more. Amazon offers different VM type for any user needs, instances differs for RAM size, virtual CPU cores, Storage size and performance. Some VMs offers the possibility to perform parallel computation on GPU cards, this feature is very interesting for some application because it can really speed up the processing of data. Not every provider offer this service, so if the developer choses to use this solution he reduces the pool of provider that its application can run on. Table 2.2 shows the variety of instance types offered by Amazon.

Table 2.2: Amazon EC2 Instances Types

Type	**Subtype**	**Memory GB**	**Compute Power ECU**	**Storage GB**
Micro	Micro	0.613	up tp 2 ECU (short period)	EBS only
Standard	Small	1.7	[1,1]	160
	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

Amazon offers this instances with different payment options:

- **On-Demand**: The user pays only for the computing capacity he uses on a hourly basis with no long-term commitments;

- **Reserved**: The user pays a low one-time term (one or three years) payment for each instance and get a discount on the hourly usage fee;
- **Spot**: The user can bid on unused Amazon EC2's capacity and run instances as long as their bid exceeds the current Spot Price, which changes periodically based on supply and demand. This payment method allows users to acquire resources paying less compared to the on-demand instances but does not ensure the continuity of service.

Amazon EC2 is a public cloud but it offers the possibility to get a hybrid one using *Amazon Virtual Private Cloud* services. This way it is possible to connect the existing IT infrastructure to a set of isolated virtual machines via a *Virtual Private Network (VPN)* connection. The VPN is priced on an hourly basis.

Amazon offers a cloud storage service (Amazon S3) to use both as a remote storage and combined with EC2 service as a block level storage. Objects are redundantly stored on multiple devices across multiple facilities in an Amazon S3 Region chosen by the user; operations such PUT and COPY synchronously store data across multiple facilities in order to ensure redundancy immediately. Amazon will then periodically check storages using checksums and repair lost redundancy (checksums are used also to detect corruption of data packets when storing or retrieving data). There are two types of storage accessible to customers:

- **Standard Storage** (useful for mission-critical and primary data storage): Provide 11-nines durability and 4-nines availability of objects over a given year and is designed to sustain the concurrent loss of data in two facilities; only for this storage is available an optional versioning service;
- **Reduced Redundancy Storage (RRS)** (useful for non-critical and reproducible data): Provide 4-nines durability and 4-nines availability of objects over a given year and sustain the concurrent loss of data in a single facility.

Amazon implements automatically load balance among instances of an autoscaling group. Users can create autoscaling groups and add virtual machines to them, traffic entering the autoscaling group is equally distributed among VMs of the group. Amazon load balancer also automatically check the healthiness of each VM attached to the group and if a machine does not respond to its monitoring requests it is excluded from the working set of machines and its requests are redirected on other active machines. The load balancer keeps checking for the healthiness of VMs and if one of them start

to respond again, maybe after a reboot of the system or other maintenance action, its served again with requests. It is important to notice that this kind of monitoring is done at the operating system level not at the application level, if the applicative software of a machine incurs into a bug and stops answering requests but the machine is still active and responding to monitoring requests of the load balancer it will not be excluded from the working group and fed with requests. Amazon offers this service called *Elastic Load Balancing* inside availability zones, which are independent zones inside the same data center, and between availability zones of the same region.

2.7.2 Rackspace Cloud

The Rackspace Cloud service, called *Cloud Servers*, provide an IaaS model service granting users complete control over their virtual machines. Instances management is made accessible through a Web interface or APIs and available operations are similar to Amazon EC2 service.

Rackspace offer differs from Amazon EC2 mainly for the presence of a support team which can help VMs deployers to manage their instances. One of the main differences from the developer point of view between Amazon EC2 and Rackspace Cloud is the fact that EC2 images are not persistent, that means that when a running instance is shut down its state is lost. Rackspace's instances instead are presistent, that means that is a machine is shut down and then rebooted its state (e.g. attached storage, files) will be in the same state that they were when the machine was terminated. Another difference from the application point of view is the management of IP addresses. Rackspace offers a persistent public IP address for each instance while Amazon uses dynamic private IP addresses under a NAT. There are other minor differences between the two providers which are listed here [20]

Rackspace offers also many solutions for data storage.

- *Cloud Files* is an object storage solution to store files or media and deliver it over the Akamai CDN [21] and ensures data persistency with triple replication.
- *Cloud Database* offers a High-performance MySQL databases in the cloud.
- *Block Storage* offers a storage solution in which users can choose between SSD or SATA disks based on their I/O performance requirements
- *Backup* offers file level backup for servers in the cloud it implements a scheduled backup policy and ensures rapid recovery of data from backup in case of need.

2.7.3 Terremark Cloud Computing

The *Terremark Worldwide Inc.* cloud services are divided in two categories: *vCloud* and *Enterprise Cloud*. The first service is designed for small development teams and department needs: It offers a quick set up and a pay-as-you-go policy. Enterprise Cloud offers precise and dynamic allocation of computing resources with the scale, performance and security to handle enterprise-wide applications and is targeted to large organizations, IT executives and multi-site teams. As previous providers, the management is made available both through a Web interface and APIs. Terramark uses VMWare virtualization products and technology and offers also persistent virtual machines that will not be erased when the user shut them down, similarly to Rackspace. Terremark offers the possibility of increasing the size of the VM both in computing capacity or memory dynamically without restarting the tool.

Unlike previous providers, there is not a storage service to use in combination with the computing one: Every instance will be provided with one or more disks of variable sizes. Hybrid Cloud may be achieved by placing proprietary servers in Terremark's colocation service or by connecting the enterprise IT infrastructure to the cloud.

2.8 Platform-as-a-Service (Paas)

2.8.1 Google App Engine

Google App Engine is a PaaS offered by Google and lets users run Web applications on its infrastructure. The user develops and uploads her/his applications without taking care of servers administration. It is possible to bind a specific domain name or use one from *applicationspot.com*.

Google App Engine supports applications written in three different programming languages: Java, Python and Go. With App Engine's Java runtime environment, it is possible to build applications using standard Java technologies, including the JVM, Java servlets, and the Java programming language or any other language using a JVM-based interpreter or compiler, such as JavaScript or Ruby. App Engine also features a dedicated Python runtime environment, which includes a fast Python interpreter and the Python standard library. The Java and Python runtime environments are built to ensure that applications runs quickly, securely, and without interference from other applications on the system.

The payment policy is pay-as-you-go: There are no set-up or recurring costs, the user pays only for storage and bandwidth used every month. it is

possible to set a monthly budget and the system will put a resources usage cap to keep the used resources under that limit. Under 500MB of storage and an amount of CPU and bandwidth needed to serve around 5 million page views per month, the service is free of charge.

Some features of this service are: Dynamic Web serving, persistent storage (with queries, sorting and transaction), automatic scaling and load balancing, APIs for authenticating users and sending email using Google accounts, scheduled task for triggering events and task queues for performing work outside of the scope of a Web request.

Applications run in a secure environment, the Sandbox, that provides limited access to the underlying operating system. These limitations allow App Engine to distribute Web requests for the application across multiple servers, and start and stop them to meet traffic demands. The sandbox isolates applications in their own secure, reliable environment that is independent of the hardware, operating system and physical location of the Web server. It also restricts applications: response to requests are limited within 30 seconds, they have limited access to file system and listening ports (only common ones and http/https protocols). Inter process comunication is made available by task queues in which a process can put tasks that are retrieved by other processes that executes them.

Google App Engine provides a powerful distributed data storage service that features a query and transaction engine. As the distributed Web server grows with traffic, the distributed datastore grows with data. The App Engine datastore is not like a traditional relational database: Data objects, or "entities", have a type and a set of properties. Queries can retrieve entities of a given type filtered and sorted by properties values. Datastore entities are "schemaless". The structure of data entities is provided and enforced by the code of the application. The Java JDO/JPA interfaces and the Python datastore interface include features for applying and enforcing structure within application, which can also access the datastore directly to embrace as much or as little structure as it needs. The datastore is strongly consistent and uses optimistic concurrency control. An update of an entity occurs in a transaction that is retrieved a fixed number of times if other processes are trying to update the same entity simultaneously. The application can execute multiple datastore operations in a single transaction which either all succeed or fail, ensuring the data integrity. The datastore implements transactions across its distributed network using "entity groups". A transaction manipulates entities within a single group. Entities of the same group are stored together for execution and transactions efficiency. The application can assign entities to groups when the entities are created.

2.8.2 Microsoft's Windows Azure Platform

Microsoft's Windows Azure Platform is a PaaS provided by Microsoft composed by a group of cloud technologies, each providing a specific set of services to application developers. This service can be used both by applications running in the cloud and by on-premises applications. The components are:

- **Windows Azure**: Provides a Windows-based environment for running applications and storing data on servers in Microsoft data centers;
- **SQL Azure**: Provides data services in the cloud based on SQL Server;
- **Windows Azure platform AppFabric**: Provides cloud services for connecting applications running in the cloud or on premises.

Windows Azure is a platform for running Windows applications and storing their data in the cloud; it runs on a large number of machines, all located in Microsoft data centers and accessible via the Internet. Developers can build applications using the .NET Framework, unmanaged code, or other approaches. Those applications are written in ordinary Windows languages, such as C#, Visual Basic, C++, and Java. Developers can create Web applications, using technologies such as ASP.NET, Windows Communication Foundation (WCF) and PHP, applications that run as independent background processes or applications that combine the two. Both Windows Azure applications and on-premises applications can access the Windows Azure storage service, and both do it in the same way: Using a RESTful approach. This service allows storing binary large objects (blobs), provides queues for communication between components of Windows Azure applications, and even offers a form of tables with a simple query language. There is also a standard relational storage provided by *SQL Azure Database*. Customers can create accounts for running application, storing data or both; administration is made available through a Web interface and APIs.

Windows Azure Platform Appfabric is a service to address common infrastructure challenges in connecting distributed applications; it consists of two components: *Service Bus*, a way to expose endpoints (as URI) that can be accessed by other applications, whether on-premise or in the cloud, and *Access Control* which allows RESTful client applications to authenticate themselves and provide a server application with identity information. Users can administer this service via a Web interface. Developers can deploy applications written in .net, java, php, python or other languages. If the developer chooses to use .net as the language for its application he can exploit many

features integrated in Visual Studio to build, test, deploy and manage the application.

2.9 Software-as-a-Service (SaaS)

2.9.1 Google applications

Google applications is a SaaS offered by Google providing customizable versions of several Google products using an owned domain name. This service is mainly offered to companies and includes different Web applications such as *GMail for business*, *Calendar*, *Docs*, *Groups*, *Sites* and *Video*. There are different *application Editions* for every needs and fees: Standard (free up to 50 users with *GMail*, Calendar, Docs and Sites), Premiere (as Standard but with annual per user fee and adding more storage for emails, Video and Groups), Education, Government and Non-Profit. An optional add-on available to premier users is *Postini* useful for protecting, archiving and securing emails. Another option available to customers is *Google applications Marketplace*: It is a store where users can buy Web applications integrated with Google ones deployed using Google App Engine. There are several categories available ranging from administration tools, finance, customer relationship, document management, productivity, sales and marketing, etc.

2.9.2 Rackspace

Rackspace also offers *Rackspace Sites* a platform which lets web designers build and publish a site on its cloud platform in a very simple way using Wordpress, Joomla or Drupal

2.9.3 Microsoft

Microsoft offers its SaaS solution for business as *Microsoft Business Productivity Online Suite* which comprehends Exchange services for mail, calendar and contacts management, SharePoint services for collaboration, Communications and Live Meeting for communication and conference over chat, voice and video.

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 3.2 gives some basic knowledge about software system control methods based on applications models introducing the discrete time Markov chain model and providing some basic notion about control theory in general. Section 3.3 describes one of the first work coping with software self-adaptation by automatically modifying the model of the application using a control-theoretical approach. The model is kept alive at run-time through parameters estimation and requirement satisfaction is obtained solving a constraint optimization problem. Section 3.4 shows a control approach found in literature that manages the autoscaling behavior of a cloud provider. Authors take into account the differences of performance between VM instances in order to do scale up and the differences of processing needs of incoming requests in order to assign them to VM that can process them in the shortest 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

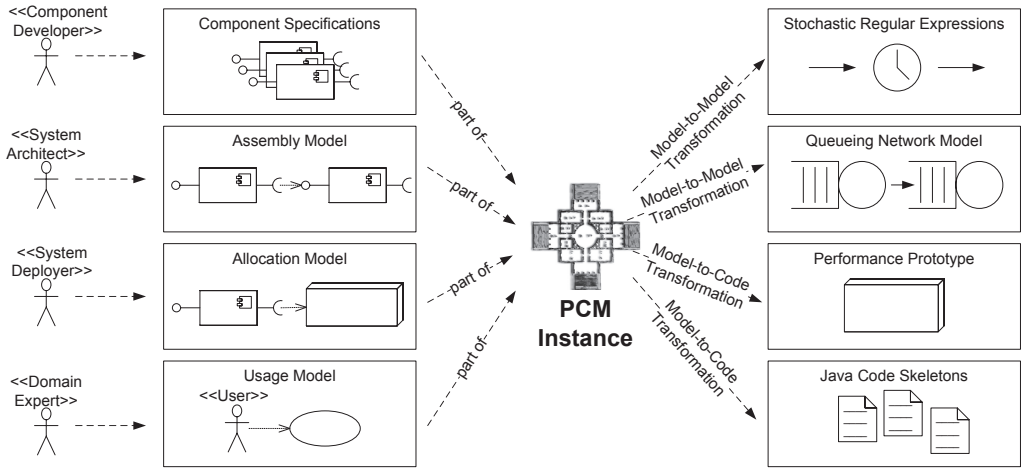


Figure 3.1: Palladio Component Model - Roles

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.

One of the key point of the Palladio suite is its ability to clearly separate development roles, as shown in Figure 3.1 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 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, if 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 informations 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 behaviour 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

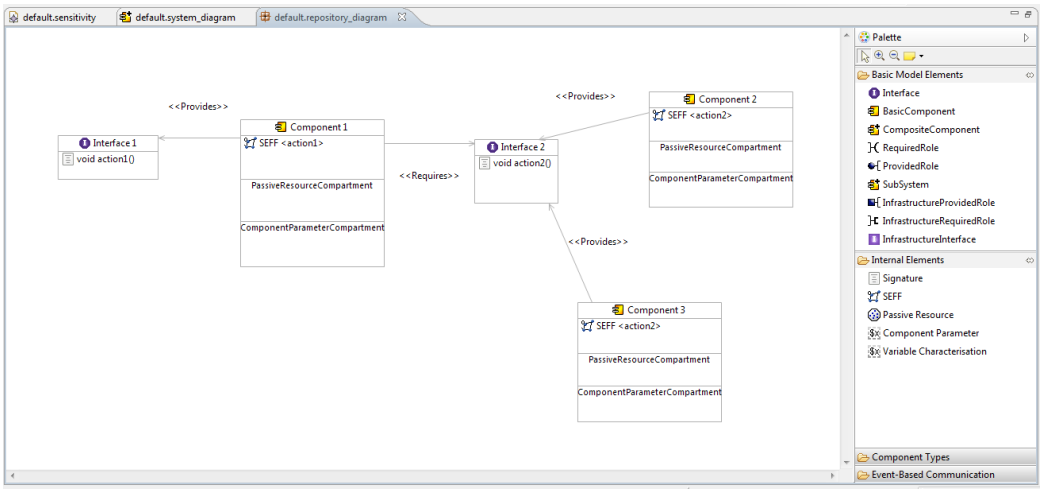


Figure 3.2: PCM - Repository diagram

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 3.2

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 has to be specified after the system 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 required 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 which is the service that end users are actually going to call. An example of a system diagram is shown in Figure 3.3

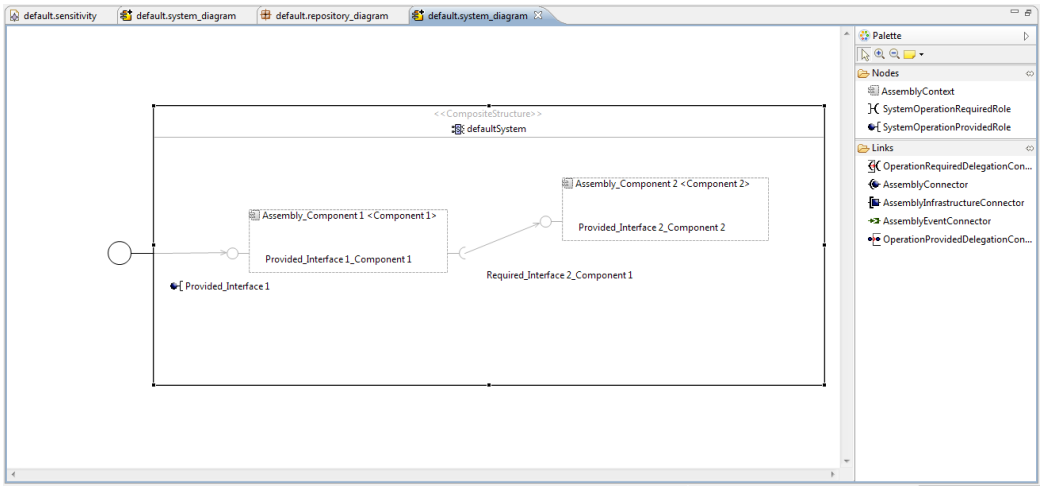


Figure 3.3: PCM-System diagram

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 3.4. 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 3.5 the execution environment specified in Figure 3.4 consists of a single resource container with a CPU and an HDD so the components specified in Figure 3.3 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 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

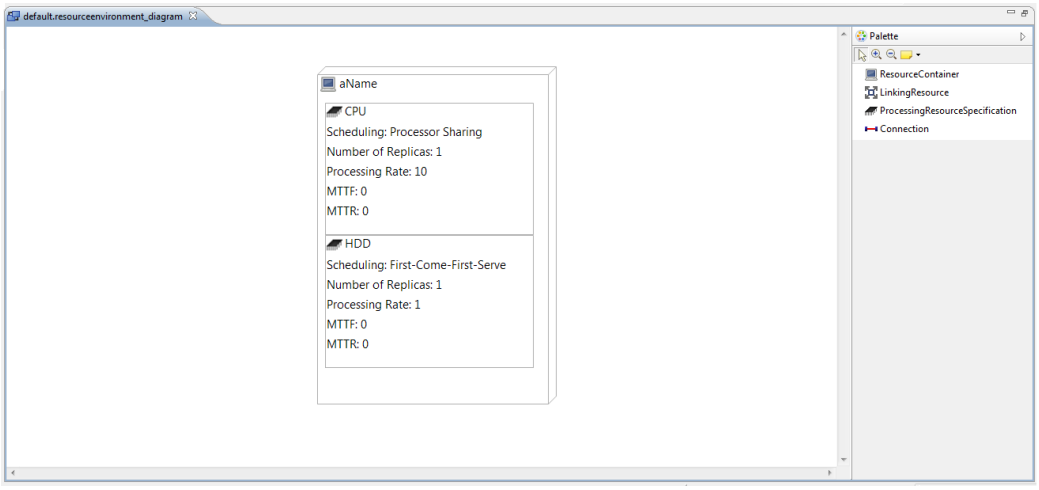


Figure 3.4: PCM-Resource diagram

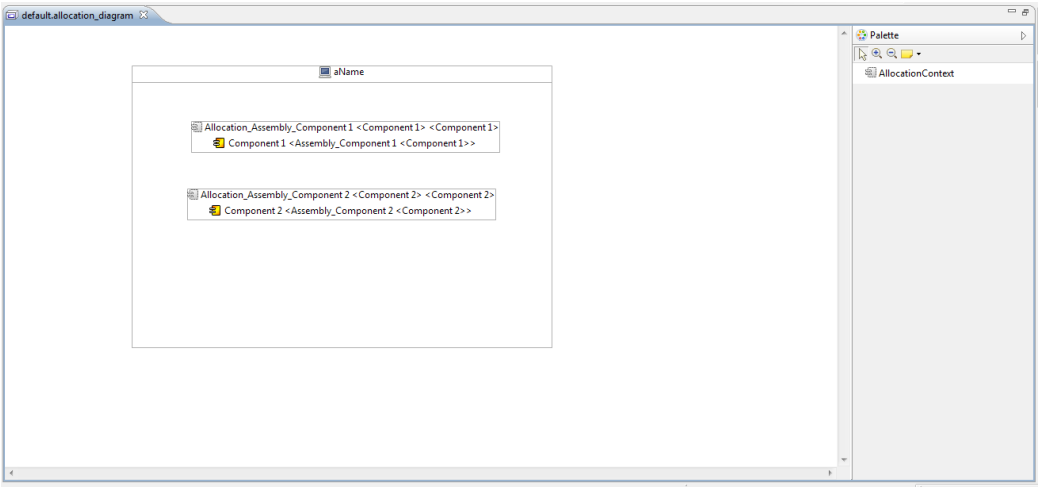


Figure 3.5: PCM-Allocation diagram

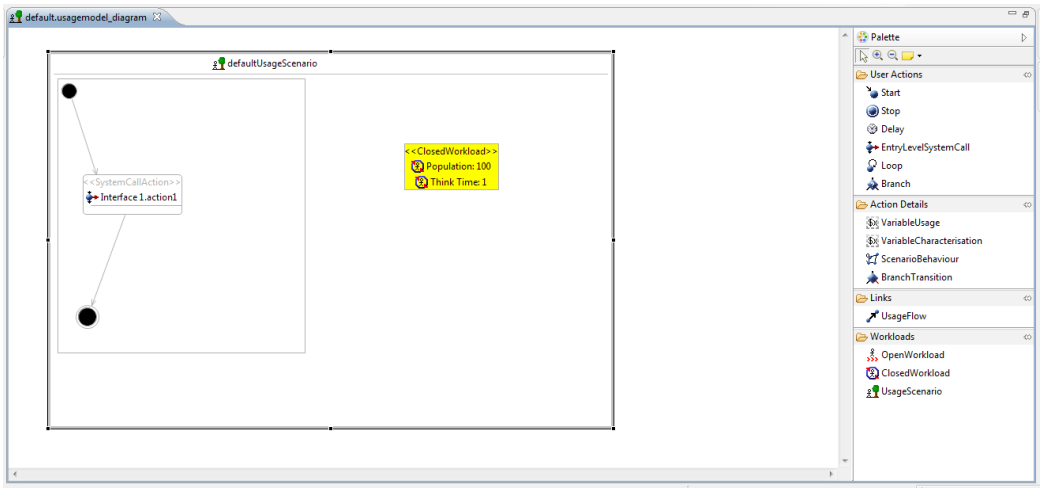


Figure 3.6: PCM-Usage diagram

when dealing with the cloud environment. Currently Palladio let system deployers create only server 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 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 we did not need to add resource consumption specification in SEFF diagrams but just a failure probability. Using this kind of specification allowed us 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 5.2.

Another limitation that we have encountered during our work 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 we want to connect. Then we can choose

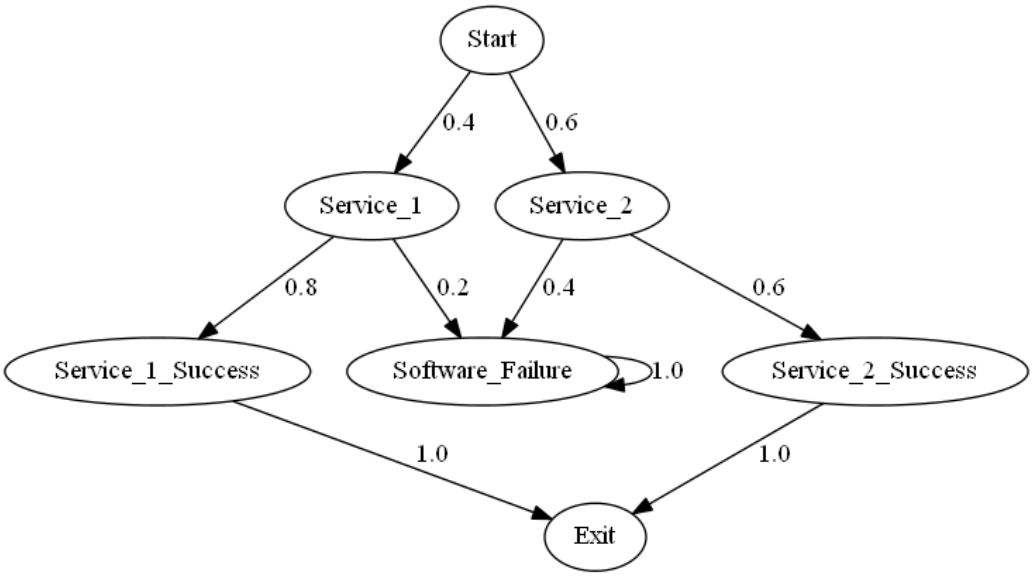
to have a component implementing in a similar way all these interfaces and replicate it inside the system diagram, or to have multiple components implementing each one a single interface and instantiate them just once. This approach moves the semantic choice of which service to call in case of multiple 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 we want to add two component providing the same functionalities to another component we have to build two identical interfaces and if there are many components of this kind in the system the deriving representation become large and not very easy to read.

3.1.2 PCM transformations

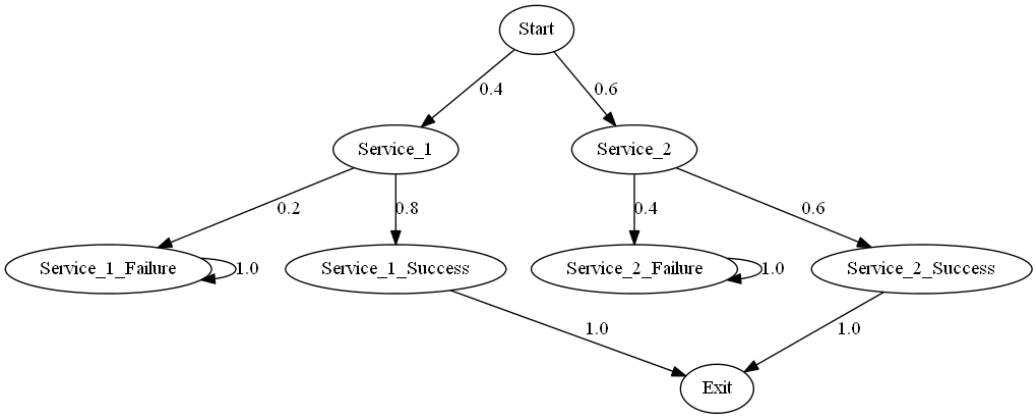
The Palladio Component Model (PCM) defined by the diagrams of Section 3.1 are used by Palladio Bench as a starting point form different transformations. Depending on what the user is interested in Palladio Bench can transform the PCM model into different models, the most used are Layered Queuing Networks (LQN) and Regular Expressions. Both these models are used to derive performance measures from the model, in particular the LQN model can be solve 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. The effect on the DTMC of using a single software failure type or multiple failure types during system design can be seen in Figure 3.7. In 3.7(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 3.7(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 3.8, 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 3.9.

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



(a) Single failure type



(b) Multiple failure types

Figure 3.7: PCM - Failure types

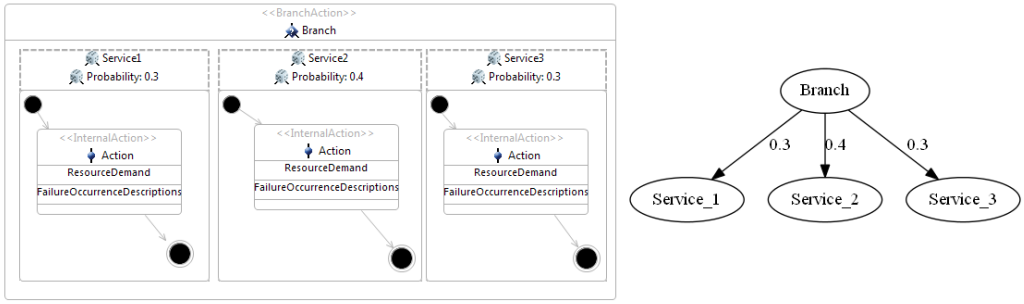


Figure 3.8: Branch conversion

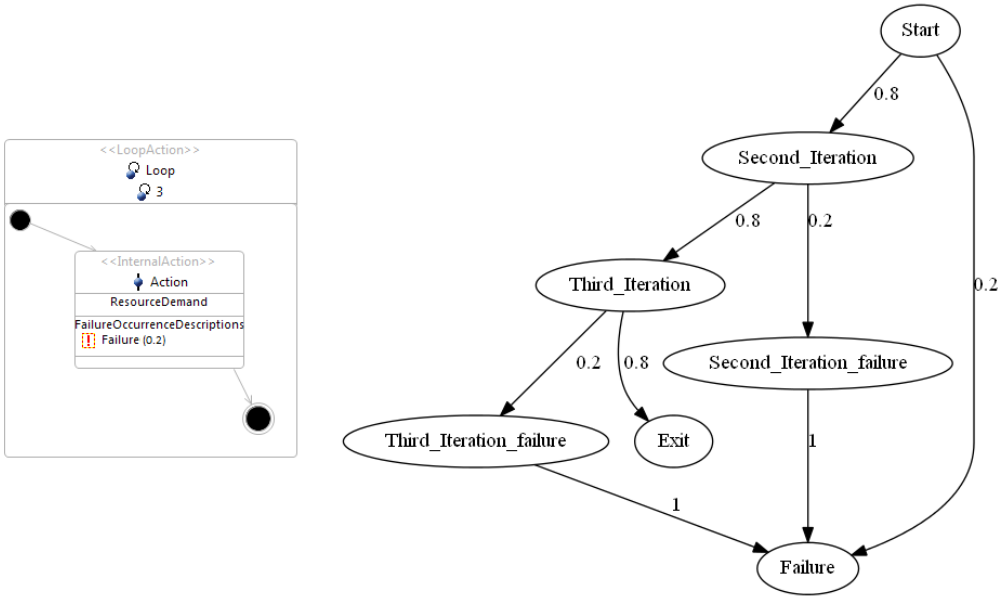


Figure 3.9: Loop conversion

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 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. Another limitation of Palladio is the fact that it is designed to perform an analysis on a fully determined system and not to optimize the behavior of the specified system with respect to decision variables, non controlled variables and a goal.

3.2 Model Based Control

As stated in [23], modern software systems are increasingly embedded in an open world that is constantly evolving, because of changing in the requirements, in the surrounding environment, and in the way people interact with them. The platform itself on which software runs may change over time, as we move towards cloud computing (see Section 2.1). For these reasons, a developer cannot guarantee requirements satisfaction just from an analysis conducted at design time. The assumptions made at development time can change in ways developers did not think of. Often, changes in the application cannot be handled off-line, but require the software to self-react by adapting its behavior dynamically, to continue to ensure the desired quality of service. The work in [23] advocates that future software engineering research should focus on providing intelligent support to software at run-time, breaking today's rigid boundary between development-time and run-time. Models should be kept alive at run-time so that software is able to evolve.

In order to react in case of changes in the environment, we need to equip our running software with some instruments that are not strictly related to functional aspects of the application, but rather to those non-functional requirements described in 2.2. Besides the model, we then need a mechanism to actually change the implementation of the software when the model is modified, we need monitors to retrieve data useful to verify the requirements

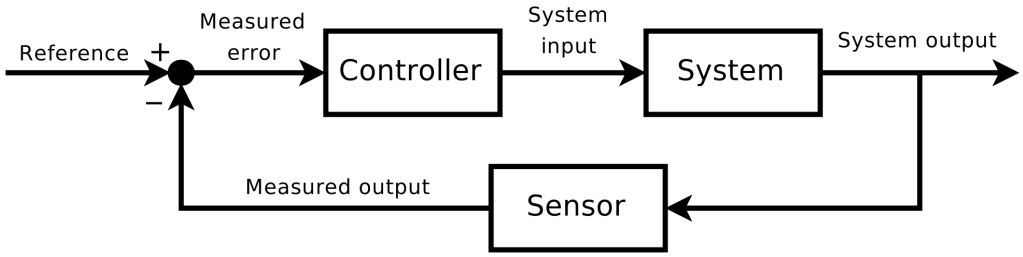


Figure 3.10: Concept of the feedback loop to control the dynamic behavior of the system. Source: `http://en.wikipedia.org/wiki/Control_theory`

satisfaction, and we need a controller capable of automatically evolve the model in case the preset objectives are no longer reached. We will now describe the model we are going to use and the control theory basis so to rely on a common background.

3.2.1 Control Theory

Control theory is an interdisciplinary branch of engineering and mathematics that deals with the behavior of dynamical systems with inputs. The external input of a system is called the reference. When one or more output variables of a system need to follow a certain reference over time, a controller manipulates the inputs to a system to obtain the desired effect on the output of the system. Controllers can be of two kinds. The ones that react using only the current state of the system and its model, which are called *open-loop controllers*, and the ones that use feedback, that is, the output of the system measured by some sensor, which are called *closed-loop controllers*. The obvious limitation in the first kind is that there is no information about how the system is actually reacting to the inputted data is observed. The concept of the feedback loop is shown in Figure 3.10. The main advantages of closed-loop over open-loop controllers are disturbance rejection and guaranteed performance even if the model does not perfectly fit the real system.

3.3 Self-Adaptive Software Meets Control Theory

In Section 3.2.1 we introduced a field which seldom deals with self-adaptive software. The first examples coming to one's mind when talking about control theory are its applications in car's cruise control or thermostat-controlled

temperature regulators. Though, control theory is not bounded to any practical field, it is just a mathematical theory which deals with anything that may be automated. Therefore, when talking about self-adaptive software we are actually dealing with a system whose requirements satisfaction needs to be automatically controlled. In this Section we will present one of the first works where a control theoretical approach was used to solve problems of self-adaptation in software system models [1].

In the paper where this work was presented, the authors focused on systems where reliability requirements have to be guaranteed. The typical scenario the authors refer to is a service-oriented application that composes external services through a workflow. External services have their own failure profile, which is unpredictable, and the degree of freedom necessary to self-adaptation is given by the choice of the service, expressed by using probabilities. The application is formally modeled as a DTMC (see Section 2.3). The controller is any system that, properly coupled to the software system, makes it fulfill its requirements whenever they are feasible. Requirements can be strict constraints on the behavior (e.g. reliability equal to a certain value) or related to the optimization of certain metrics on the observed software executions (e.g. minimization of outsourcing costs or maximization of throughput). The claim this work support is that control theory provides a number of instruments that software engineers to satisfy non-functional requirements even in case of changes in the environment. In particular the authors claim the controller is able to provide:

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

A Representative Example Figure 3.11 shows the high level software model of the case study introduced in the paper. An image filtering service is composed by three different implementation of a beautifying filter, where one of them is outsourced (External Filter). The DTMC model of the system is shown in Figure 3.12. The controller will be responsible of adapting the system acting on the control variables *C1a*, *C1b* and *C5*. Therefore, it is in charge of distributing the requests among the three different filters and of deciding whether re-iterating on the iterative filter. All the alternatives are

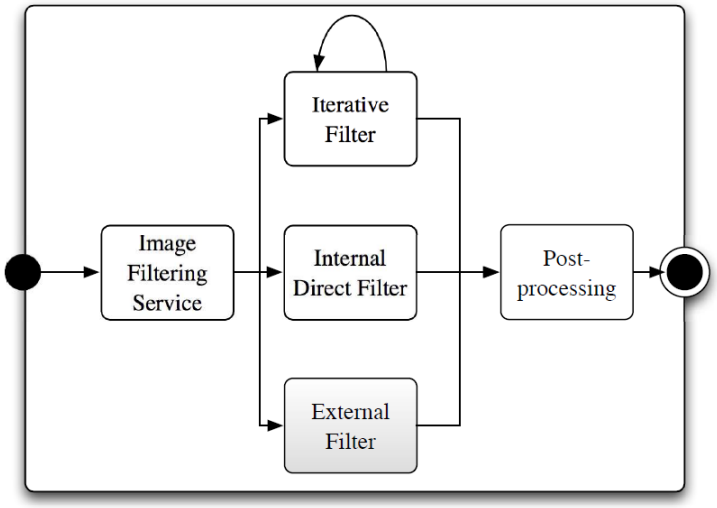


Figure 3.11: Schema of the software system. Source [1]

assumed to be black-box services, whose failure rates are collected by run-time monitors that are then responsible of estimating the probability that an invocation to the service will fail.

Starting from the DTMC model of Figure 3.12 and applying the approach described in Section 2.3 the authors write down the equation system as in Equation 2.1. By solving that system for s_0 it is possible to obtain the closed formula 3.7 that describes the explicit dependency of reliability (s) on control variables (c) and measured reliabilities (r).

$$s = r_0 \cdot r_6 \cdot \left(\frac{c_{1a} \cdot (-1 + c_5) \cdot r_2}{-1 + c_5 \cdot r_2} + c_{1b} \cdot r_3 + (1 - c_{1a} - c_{1b}) \cdot r_4 \right) \tag{3.1}$$

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

$$s(k+1) = f(r(k) + \Delta r(k), c(k)) \tag{3.2}$$

where $s(k+1)$ is the application reliability in step $k+1$, $c(k)$ are the control variables set for step k, which are kept constant through the step, $r(k)$ are

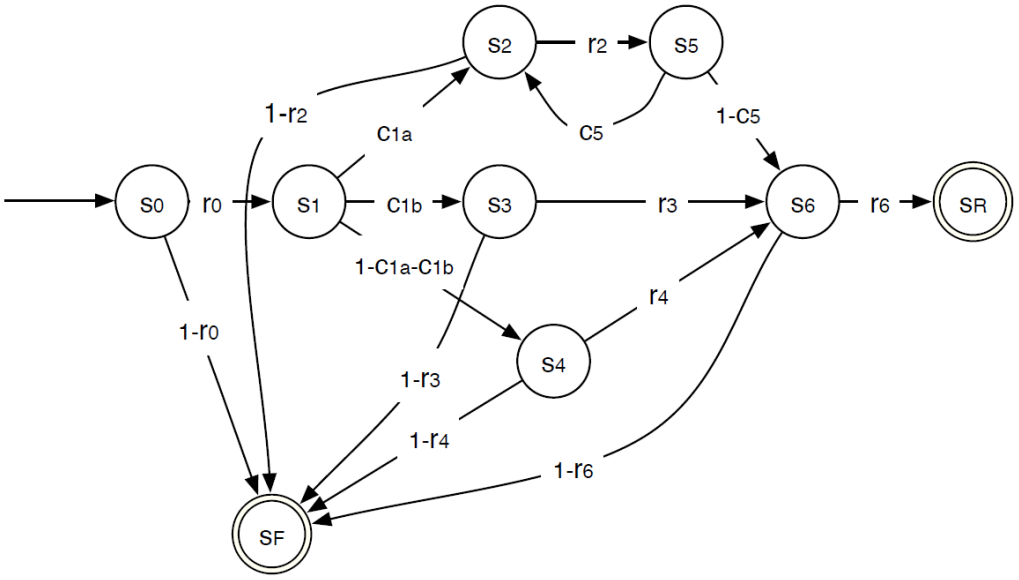


Figure 3.12: DTMC model for the example system. Source [1]

the expected reliabilities for step k (which are estimated via monitoring), and $\Delta r(k)$ accounts for any discrepancy between the real and expected reliabilities in step k. The form of function f comes from the DTMC model as computed in 3.7.

Controlling the System's Dynamics by Feedback In a nutshell, the idea of feedback presented in [1] can be summarized as plugging the controlled system into a larger one where its input is made dependent on its measured output, possibly its state or an estimation of it in the case it cannot be measured, and on the desired behavior for the controlled system. Let $J(k) = f_j(c)$ be a cost function on the control variables $c(k)$, that can also be an uninformative one (such as a constant value) to indicate no preference among all the feasible solutions. In this case the problem is transformed in a satisfiability problem because the controller has just to find a feasible assignment to control variables and not an optimal one. The controller comes into play by solving the problem

$$min\ J(c) \tag{3.3}$$

subject to the constraint

$$\begin{gathered} ||goal(k+1) - \hat{s}(k+1)|| \leq \alpha ||goal(k) - s(k)|| \\ \forall c_i(k),\ 0 \leq c_i(k) \leq 1 \end{gathered} \tag{3.4}$$

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

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

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

Experimental Evaluation For the proposed case study the control system acts minimizing

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

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

$$\begin{aligned}
r_0 &= 0.95 + 0.02stp(k-25) - 0.20stp(k-50) + 0.10stp(k-75) \\
r_2 &= 0.95 + 0.02stp(k-20) - 0.20stp(k-70) + 0.15stp(k-85) \\
r_3 &= 0.95 + 0.02stp(k-15) - 0.97stp(k-55) + 0.50stp(k-65) \\
r_4 &= 0.95 \\
r_6 &= 0.95 + 0.05stp(k-95)
\end{aligned} \tag{3.7}$$

Figure 3.13 shows the result of the simulation. The dashed line is the set point of the desired availability which is modified during the simulation, the solid line is the availability of the controlled system. From this figure we can see that the controller is capable of modifying the behaviour of the application in order to get the desired availability, it converges to the new set point in few time units and does not present oscillating behavior. Figure 3.14 shows the value assigned by the controller to control variables at each time unit. For time units between 55 and 65 a failure of node r_3 in injected, the controller reacts by changing the probability of using that node to 0 and raises other probabilities.

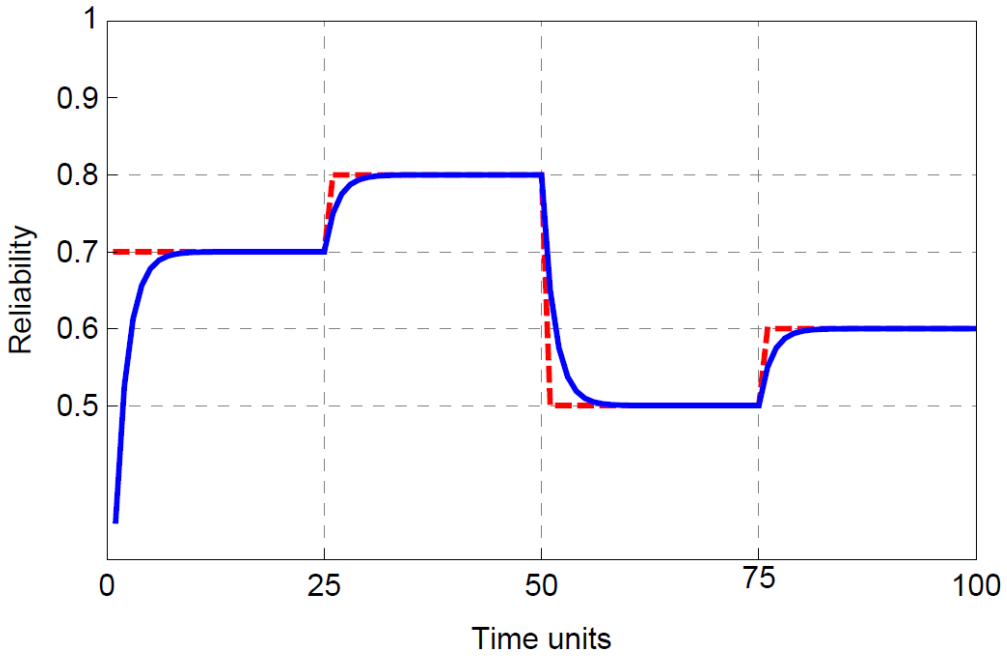


Figure 3.13: Reliability of the system: set point (dashed) and achieved value (solid).

3.4 Cloud Auto-scaling with Deadline and Budget Constraints

Another approach that can be adopted to manage auto scaling in a cloud environment is presented in [2] where authors build an integer programming problem from deadline and budget constrains and solve it to get scaling decisions. This article focus on modeling the incoming workload and the available processing resources by diving it them into sub classes. In particular cloud VMs are modeled into three subclasses in order to specify some special characteristics offered by machine of that type. The classes in which VMs are divided are: General, High CPU and High I/O. The workload is also divided into three classes that are: Mixed, CPU Intensive and I/O intensive. The goal of this control mechanism is to complete each job within a deadline that is assigned when the job enters the system. In order to make scaling decisions authors take into consideration the attributes shown in table 3.1.

The control system proposed in the article consists in a monitoring part that keeps track of application level performance measures like the average processing time of each job according to a given machine type. A decision

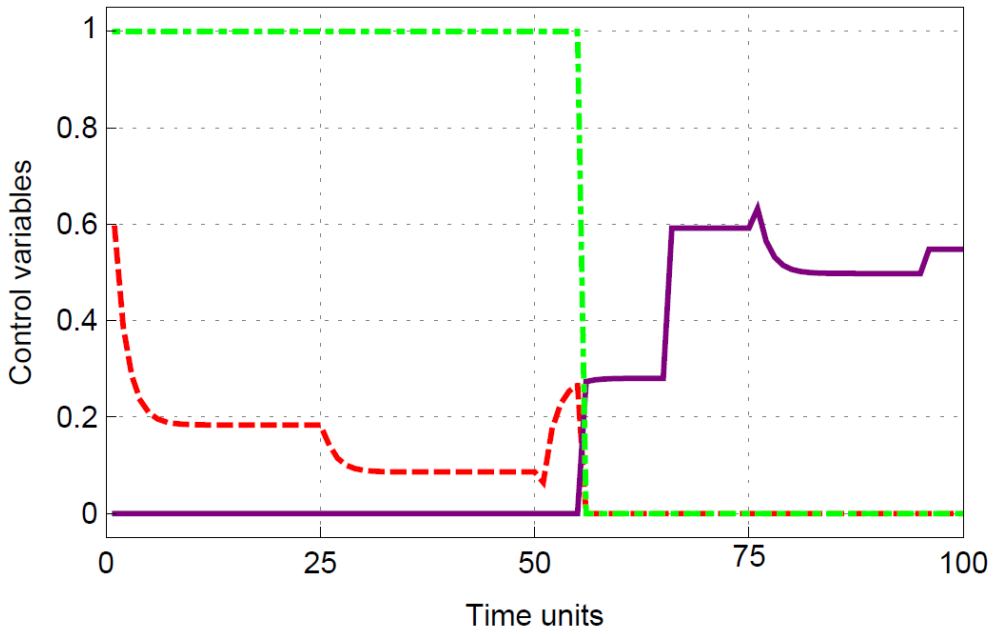


Figure 3.14: Control variables of the system: c_{1a} dashed, c_{1b} solid and c_5 dashed dotted.

J_j	jth class of job
n_j	Number of jobs of class j already in queue
V	VM type
I_i	ith instance (running or pending)
c_v	Cost per hour of a VM of type V
d_v	Average startup delay of VM of type V
s_i	Time spent in pending status for instance I
$t_{j,v}$	average processing time of job j on V
D	Deadline
C	Budget constraint
W	Workload
P	Computing Power

Table 3.1: Attribute used to take scaling decisions

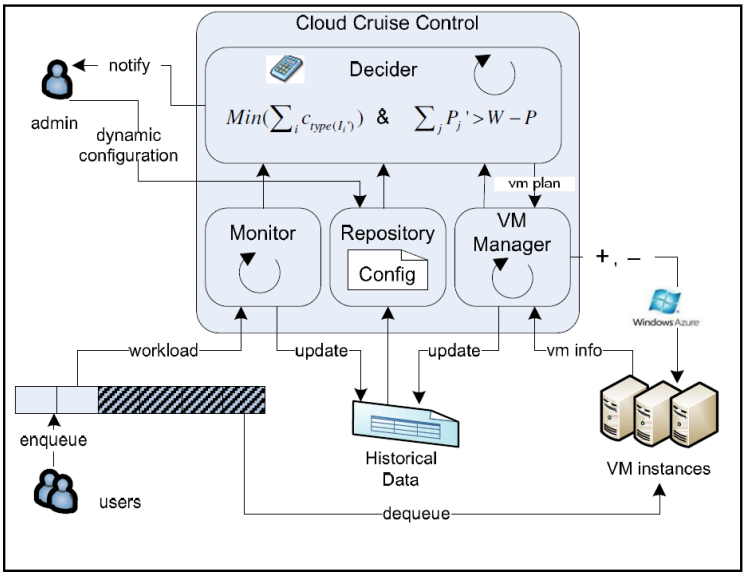


Figure 3.15: Structure of the controller in [2]

engine that take scaling decisions and a VM manager that perform the scale up or scale down action. This approach does not limit to horizontal or vertical scaling but tries to mix both by choosing a combination of instance types that is capable of processing all queued jobs within their deadline while mini zing costs. The scaling manager is executed at the arrival of each job, in order to determine if the increased workload (W) can be managed by the current computing power (P), and a few minutes before the end of a computing hour of each machine. This second activation of the controller is used to manage the scaledown process. Since billing happens on an hourly base at the each of each hour of uptime of each machine the controller has to check if that machine is needed in the next hour or can be safely deallocated to save budget resources. The structure of the controller is shown in Figure 3.15.

3.5 Cloud control approaches considerations

The work presented in Section 3.3 shows a control technique that uses monitoring to keep models created at design time alive with the running system. Authors use information from the updated model to automatically take adaptation actions. In particular the controller chooses how to route requests in some points of the application in order to fulfill an availability requirement. At a first approximation we can say that this controller acts as

a very smart load balancer for the system.

Section 3.4 shows a control approach aimed at managing efficiently the scaling ability of a cloud provider. They divide available resources and incoming processing requests by performance categories and try to find the best assignment of requests to resources when dealing with a scaling decision.

We considered both works when designing our controller by building two layers of control that work together. The first layer acts as a smart load balancer controller that monitors the system, updates the model and takes control decisions. The second acts as a scaling manager that takes scaling decisions by monitoring some parameters of its autoscaling group. Since we start from the hypothesis that resources of an autoscaling group are homogeneous we did not used the approach described in [2] to assign requests to machines.

Chapter 4

Model and Controller 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 instance 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. The main advantage that we are interested in is the possibility of keeping an instance of this model alive at runtime, update its value by monitoring the real application and take control decisions by perform some reasoning on the updated model. The controller described in Section 4.3 has been developed to update model parameters at runtime and assign values to control variables of the model in order to keep availability as close as possible to the set point defined by the user while reducing costs.

4.1 Overview of the solution

The solution we propose in this thesis is shown in Figure 4.1. It is composed by two main parts:

- A *model* that store information on the structure of the application and on the characteristic of the environment in which it is deployed.
- A *controller* that operates on the model in order to control the behavior of the system.

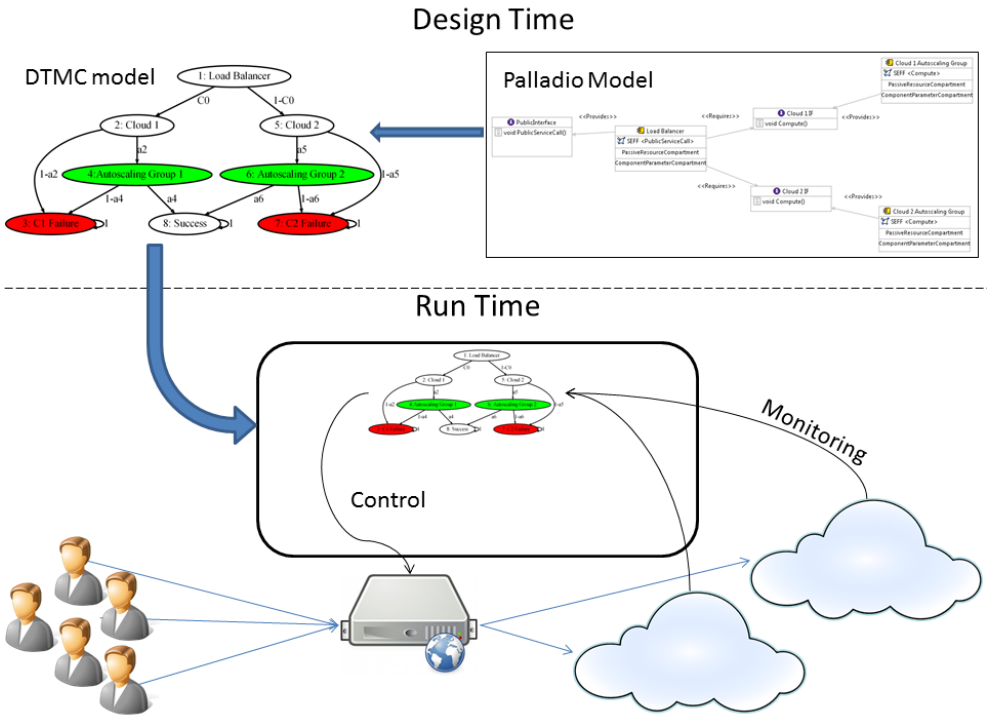


Figure 4.1: Overview of the solution

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 5.1. The controller uses the generated model by monitoring system parameters to update its parameters. It then calculates control variable values that are used to update the system while it runs.

In order to test the validity of our approach we implemented a tool capable of simulating the cloud environment as shown in the lower part of Figure 4.1. The tool is presented in Section 5.2 Users can specify some parameters of the simulated scenario like the incoming workload or the availabilities of cloud providers and the tool will simulate these behavior. The controller reads data from the simulated environment and apply control actions. The availability of the system, along with other parameters, is recorded during the simulation.

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.

In our model nodes represent two kind of entities:

- *Computing Resources* that can be a physical server of a company or a pool of VMs offered by a cloud provider. This elements have some peculiar characteristics that will be discussed deeper later.
- *Logical Nodes* like load balancers or other logical elements which do not perform any specific computation on the requests traversing the system. Logical nodes can be seen as particular computing nodes with infinite processing power.

The main difference between these two entities is the fact that the first one can represent a bottleneck of the system in which some requests are discarded because of the limited processing capacity, the second kind of node does not represent a bottleneck for the system but just distribute them among following nodes. Usually providers manage the autoscaling of these nodes in an automatic fashion that can't be controlled by the user.

As in [1] we extended the classical DTMC model by adding *control variables and measured availabilities* as labels to transitions. Measured availabilities 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. In control theory lexicon these factors are called disturbances and can be measured by monitors. 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. Augmenting the DTMC model with this two kind of variables makes it suited for control but it is still not enough to model other important aspects of the cloud.

A very important parameter that we added to our model is the one that represents the *scalability* of the entity represented by the node. This is a binary parameter, if it is true then the node represents an entity capable of performing autoscaling. This is a very important property of a node because it represents the fact that this node can change the amount of requests that it can process and introduces a new way for the control system to manage the execution of the application. Every node capable of scaling models an autoscaling group presented in Section 2.1 and is supposed to have its own load balancer, offered by the cloud provider, which automatically distributes incoming traffic across instances uniformly. The fact that a node can perform autoscaling or not heavily affects the usage of other parameters that will be presented later. A node representing a computing resource with scalability

parameter set to false can be used to represent a computing resource with fixed computing power, this is very useful if the user want to model an hybrid cloud architecture. In such a case in house servers are not capable of scale their computational power. If this parameter is true the maximum processing capacity of the node is given by $Numberofrunningmachines \times maximumprocessingpower$. This parameter is not used in logical nodes since it affects the processing power of nodes and such nodes are supposed to have infinite processing power.

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 that 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.

Though, we will not know how much a single request is going to influence the costs, we left the cost of our model a parameter that will be estimated by the controller at run-time. At run-time, in fact, we will have information like the current instances pricing, the number of machines and the service rate (or at least an estimate of them as we will explain in section 4.3) necessary to estimate the impact of sending a request to one cloud rather than to another. The cost per request is going to be computed with the following formula

$$\frac{\textit{cost per machine per second} \times \textit{number of machines}}{\textit{desired service rate}} \tag{4.1}$$

where the *desired service rate* is the estimated service rate of the entire node when working at the desired CPU capacity (see Section 4.3. This cost is, in fact, a measure of the convenience of using one cloud rather than another one. So, for example, suppose we have two clouds, cloud A and cloud B. They have the same pricing, but virtual machines of cloud A have a higher service rate. Then, cloud A will manage to serve more requests in the same interval of time, allowing to use less machines and, thus, to save money.

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, at design time 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. APIs are usually provided by the cloud provider to read current costs.

The next two parameters that will be presented are used only in autoscaling nodes since model features specific of the cloud environment. Each autoscaling node is labeled with a *minimum* and a *maximum number of running instances* this two parameters represent respectively the minimum and maximum number of machines that can run simultaneously on the resource modeled by the node. This parameter can be used if, for example, while 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. Without this parameter a controller that tries to minimize costs would be induced to shut down all providers except the most convenient one. On the other hand the maximum number of running instances is used to model a resource cap that the designer can set for some providers.

An example of a complete model is reported in Figure 4.2. Blue states are logical, those states are supposed to have attribute cost equals to zero and an infinite processing capacity. Green states are processing states that represent physical processing resources. Two of them are autoscaling states so they have a cost and a range in which the number of active machines can vary, while the other one is an internal processing server that is not capable of perform autoscaling so it has just a cost attribute. We can see that failure in this model can arise from two different events:

- logical state going to failure state representing the failure of the cloud provider or of one of its components (states "Cloud Failure", "R1 Failure" and "R2 Failure")
- failure due to computational resources bottlenecks (States "R1 Processing Failure", "R2 Processing Failure" and "Internal Server Failure")

4.3 The Controller

In Section 4.2 we augmented the classical DMTC model so to define a new model able to describe a Multi-Cloud application. This new model is supposed to be kept alive at run-time, so that whenever some controller modifies it, changes take effect on the actual implementation.

The controller we are going to define, is actually a dual layer controller. The first layer controller is responsible for managing one autoscaling group, controlling the number of running machines. So there are actually as many first layer controllers as the number of nodes modeling autoscaling groups.

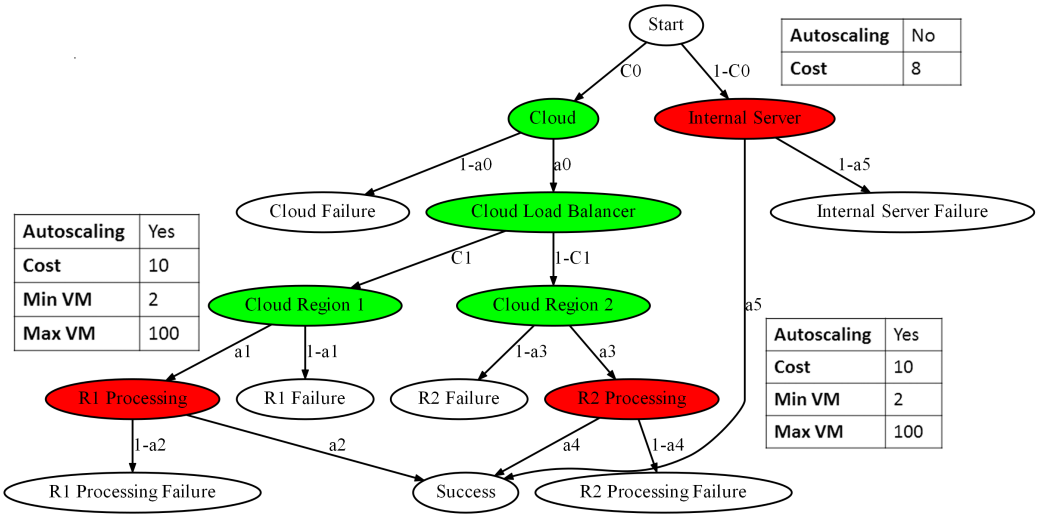


Figure 4.2: Instance of the model

The second layer controller is a sort of "smart" load balancer in charge of distributing requests among nodes. The cooperation between these two layers of controllers aims at guaranteeing system availability, while minimizing costs.

An important assumption is that they both work at discrete time, that is, sensorial data is aggregated and delivered from monitors every constant interval of time (step).

4.3.1 The autoscaling controller

The first layer controller is in charge of performing adaptation at the node level of our DTMC model. As we saw in Section 2.1, *PaaS* solutions do not require the developer to specify scale up or scale down policies. In fact, the autoscaling is transparent and managed by the provider automatically. Thus, the controller layer we are defining in this section is clearly only useful for those applications using at least an *IaaS* component. If the application is deployed on top of a PaaS system this layer of control is managed by the cloud provider and only the controller of Section 4.3.2 is necessary even though some modification may be required to estimate parameters like the maximum service rate.

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

current providers' offer (see Section 2.1).

Objective The objective of this controller is to keep the number of running machines so that the average percentage of CPU utilization is equal to the *desired cpu load*, a parameter that is chosen by the developer. This parameter has to be chosen wisely considering that keeping resources highly loaded will certainly reduce costs, since less running machines will be needed, but there will also be less safety margin in case of sudden increase of incoming workload and performance might decrease.

Monitoring Recalling Figure 3.10, what we need for a controller is a feedback loop. So, to begin with, we need data from "sensors" so that we can check how the system is behaving in response to controller's decisions. First of all, we define a *sliding observation window*, which is the time span (or number of steps) used to make statistics from data collected by sensors. The statistics, that are all relative to the observation window, are the following:

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

From this data, the success rate is then estimated as

$$\frac{\textit{successful requests}}{\textit{incoming workload}}$$

The success rate will be our parameter of availability.

This information could also be used to make predictions on data, for example estimating the next values of the CPU utilization or availability from their trends in the window, but all this will be future work (7).

Control As we said, the developer has to give the set point to the control system, that is, the desired average CPU utilization of the running machines. Since the scale up process is quite slow, we cannot afford to let the controller continuously take decisions and make the number of running machines change. We proposed a mixed approach to the controller intervention

timing. It has to be reactive, whenever the system is far from the desired state. The developer is in charge of setting, besides the set point, bounds to the CPU usage, that is a maximum and a minimum utilization levels. The system has to take action immediately whenever the CPU utilization overcomes these bounds, or whenever the performance degrade, that is, the success rate becomes smaller than 100%. We will see shortly that, according to our assumptions, a success rate smaller than 100% implies a CPU utilization of 100%.

In case the node is working inside CPU bounds, the controller is temporized. Thus, it is activated every constant interval of time, decided by the developer, paying particular attention to the fact that, as we will see, the controller on the first layer as higher priority with respect to the second layer controller. The smaller this time interval, the higher the probability of delaying the second layer controller intervention.

The temporized intervention is in charge of making the system approach the set point, otherwise the controller would only try to make the system work inside the bounds.

As we said, launching new machines is a slow process, it may take minutes as stated in Section 2.1. So we need to prevent the controller to take decisions while machines are turning on, or in *pending* state. We will say that a node is *stable* whenever there are no machines in *pending* state. Also, we want statistics from monitors to be estimated only from data observed after a scale up or a scale down process happens. Therefore, we defined a *cool-down* state, which will inhibit the controller as long as it is active. A node enters the *cool-down* state when a scaling process is started (both scale up and down) and will exit from this state only after remaining in a *stable* state for the entire duration of the *observation window*.

After exiting from a *cool-down* state, the controller will be allowed to take decisions, reacting on statistics from monitors, and timers for the temporized intervention are reset.

Given data from monitors, we first need to find a control formula where the error observed between the desired behavior and the actual one, can be reduced (and asymptotically eliminated) at each control step acting on the control variables. Our control variable in this case, is just the number of machines required. We need, therefore, to find a relation between the number of machines and the measured availability, and a relation between the number of machines and the average CPU utilization.

Let us start with an example to understand the assumptions that follows. Suppose that we have one node with 8 virtual machines. The arrival rate at the node is 1000 requests per second. The maximum service rate of each machine is 125. The node maximum service rate is therefore equal to the

arrival rate, which makes 100% of availability and 100% of average CPU usage. If our desired behavior is having a CPU utilization of 80% we will need 10 virtual machines. In fact, $\frac{1000}{10\times125} = 0.8$.

To be more precise, we are supposing the CPU utilization to be equal to $\frac{AR}{SR}$, where AR and SR are the arrival rate at the node and the node maximum service rate respectively. Therefore, we can write the following equations

$$CPU(k) = \frac{AR(k)}{SR(k)} \tag{4.2}$$

$$SR(k) = sr(k)n(k) \tag{4.3}$$

where sr is the maximum service rate of a machine, while n is the number of machines. It follows

$$CPU(k+1) = \frac{AR(k+1)}{sr(k+1)n(k+1)} \tag{4.4}$$

We suppose the time steps are small enough to consider the service rate of a machine and the arrival rate to remain constant. Otherwise, prediction can be taken in consideration, but it is out of this scope. Therefore Equation 4.4 becomes

$$CPU(k+1) = \frac{AR(k)}{sr(k)n(k+1)} \tag{4.5}$$

From 4.5 and 4.3 follows

$$CPU(k+1) = \frac{AR(k)}{SR(k)}\frac{n(k)}{n(k+1)} \tag{4.6}$$

Finally using 4.2 we get

$$CPU(k+1) = CPU(k)\frac{n(k)}{n(k+1)} \tag{4.7}$$

and therefore our desired number of machines can be computed as

$$n(k+1) = n(k)\frac{CPU(k)}{CPU(k+1)} \tag{4.8}$$

Let us now go back to our example. Suppose now that the maximum service rate of each machine is 10 requests per second. The node maximum service rate is therefore 100 requests per second, which makes 10% of availability. In order to satisfy 1000 requests per second we need our node to have at least maximum service rate of 1000 requests per second. Therefore we will need at least 100 virtual machines. This number can be easily computed

given the current number of machines n and the current availability a of the node, with the following formula

$$n(k+1) = \frac{n(k)}{a(k)} \tag{4.9}$$

To be more precise, we started from the assumption that the availability, that, as we said, is estimated through the success rate, is computed as

$$a(k) = \frac{SR(k)}{AR(k)} \tag{4.10}$$

Given the assumptions made for the previous case and through mathematical passages very similar to the ones just seen, we obtain

$$a(k+1) = a(k)\frac{n(k+1)}{n(k)} \tag{4.11}$$

and therefore our desired number of machines can be computed as

$$n(k+1) = n(k)\frac{a(k+1)}{a(k)} \tag{4.12}$$

From Equations 4.2, 4.10 and 4.3, we can describe the dependency of CPU usage and node availability on the number of running machines

$$CPU(n) = \frac{AR}{sr \cdot n},\ a(n) = \frac{sr \cdot n}{AR} \tag{4.13}$$

Figure 4.3 shows this dependency through an example.

We can finally resume our assumptions with the following working conditions:

- if the arrival rate is lower than the maximum service rate offered by the node, the availability is 100%, while CPU utilization decreases in indirect proportion to the number of machines.
- if the arrival rate is equal to the maximum service rate offered by the node, both CPU utilization and availability are 100%
- if the arrival rate is greater than the maximum service rate offered by the node, the CPU average utilization is 100%, while the availability will be lower than 100%, growing in direct proportion to the number of machines

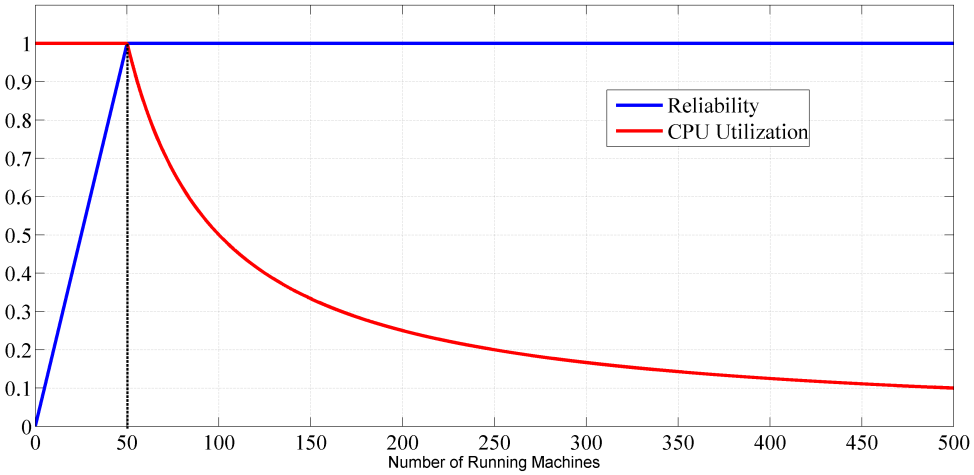


Figure 4.3: Dependency of the average CPU utilization and the availability of an autoscaling group on the number of running machines, where the arrival rate is 5000 requests per second and the maximum service rate of each machine is 100 requests per second.

$$\text{Availability} = \min\left(1, \frac{n}{50}\right), \ \text{CPU Utilization} = \min\left(1, \frac{50}{n}\right)$$

From these assumptions, we can identify two working modes, activating each a different control policy.

1. If availability is 100%, we want to make the CPU usage to converge to the set point. Similarly to the solution proposed in [1] for controlling a system through feedback loop, we make the autoscaling controller solve the following equation

$$u(k+1) - \hat{p}(k+1|k) = \beta(u(k) - p(k)) \tag{4.14}$$

where p is the CPU utilization and u is the desired CPU usage value. $\hat{p}(k+1|k)$ is the expected value of CPU usage at the next step, which, as seen in Equation 4.7, depends also on the number of machines at the next step. β is a parameter in the range $(0, 1)$ and determines how fast is the convergence to the solution, that is, in the next step we expect

the absolute error to be reduced by a factor β. Solving the equation, the analytical solution is

$$n(k+1) = \frac{n(k)p(k)}{u(k+1) - \beta(u(k) - p(k))} \tag{4.15}$$

2. If availability is not 100%, the following equation would converge to the number of machines needed to have the availability equal to the set point v

$$v(k+1) - \hat{a}(k+1|k) = \beta(v(k) - a(k)) \tag{4.16}$$

Since the objective of our controller is to have 100% availability and to have the desired CPU level we set $v(k) = 1$

$$1 - \hat{a}(k+1|k) = \beta(1 - a(k)) \tag{4.17}$$

β is again a parameter in the range $(0, 1)$ and determines how fast is the convergence to the solution, that is, in the next step we expect the absolute error to be reduced by a factor β. Substituting $\hat{a}(k+1|k)$ with the result obtained in Equation 4.11 we can compute the desired number of machines as

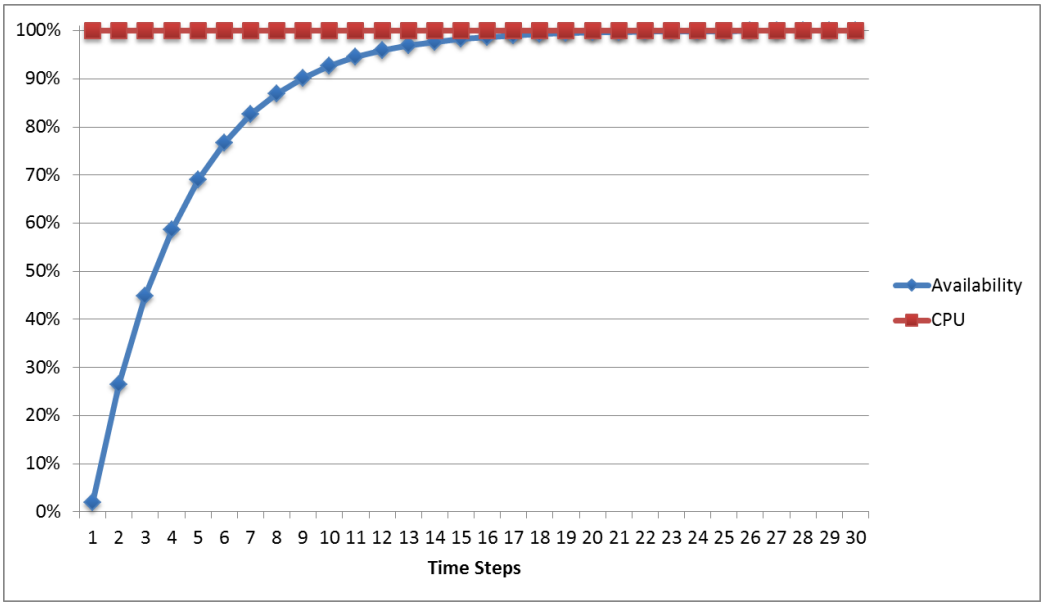
$$n(k+1) = \frac{(1 - \beta(1 - a(k)))n(k)}{a(k)} \tag{4.18}$$

Though, this result would converge asymptotically to a solution in the working point where both availability and CPU utilization are 100% (see Figure and 4.4). We prefer the solution not only to reach availability 100%, but also to reach fast the desired CPU usage level. So we decided to make two steps in one by using first Equation 4.18 and then Equation 4.15. We obtain the following formula

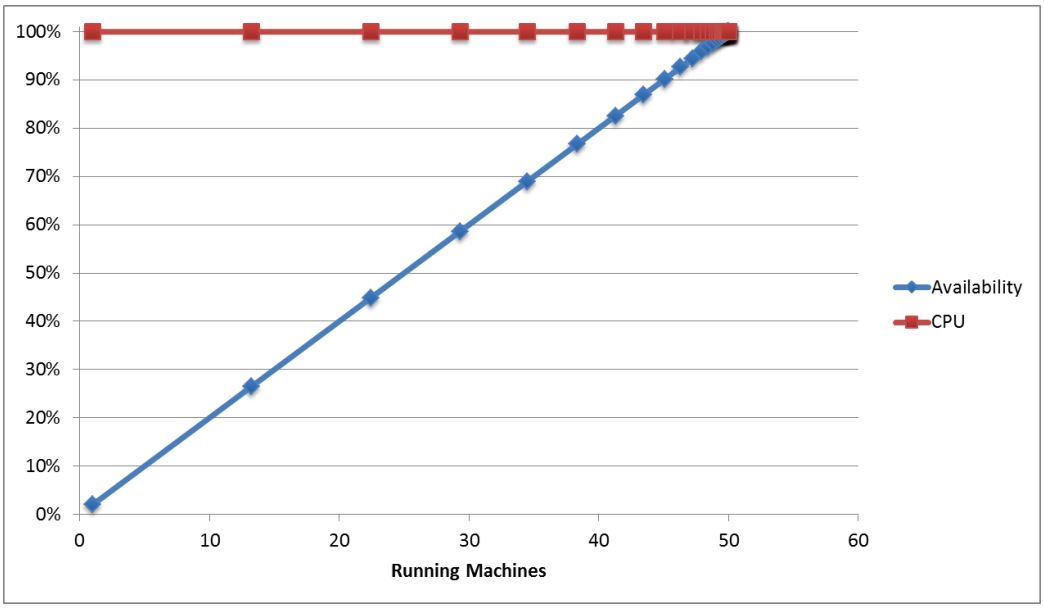
$$n(k+1) = \frac{(1 - \beta(1 - a(k)))p(k)n(k)}{a(k)(u(k+1) - \beta(u(k) - p(k)))} \tag{4.19}$$

This way we are certain that the controller will make the number of machines overcome the bound and get to desired CPU level. In Figures 4.5 and 4.6 we can observe the convergence of this equation.

Finally, we can notice that Equation 4.19 is identical to Equation 4.15 whenever $a(k) = 1$, therefore we can simply use the first equation in any working point. Figures 4.7 and 4.8 shows how convergence works in the scale down case, that is when the processor usage is very low and we want to turn off the spare machines.

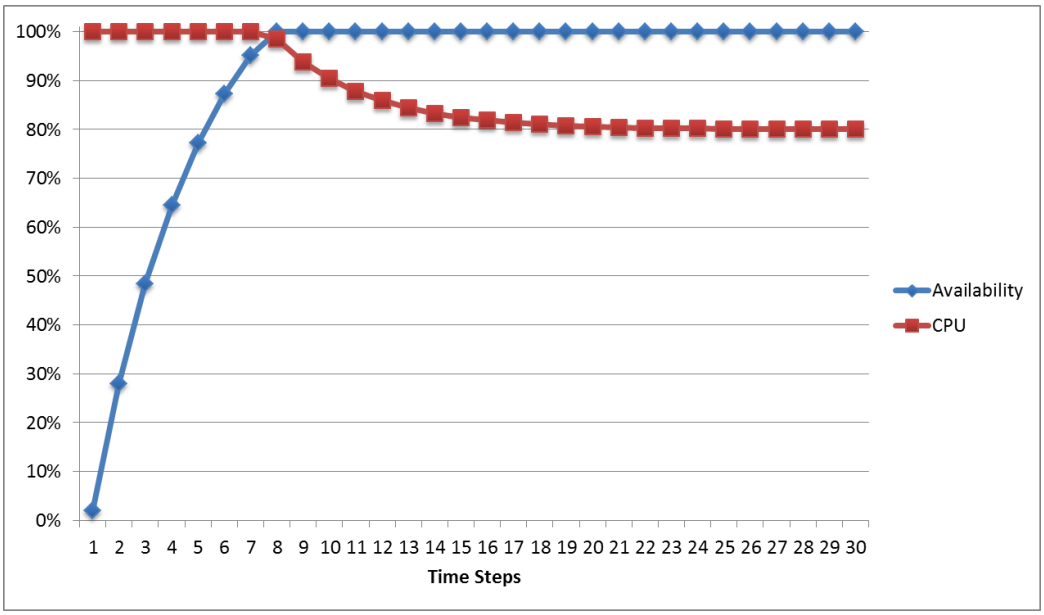


(a)

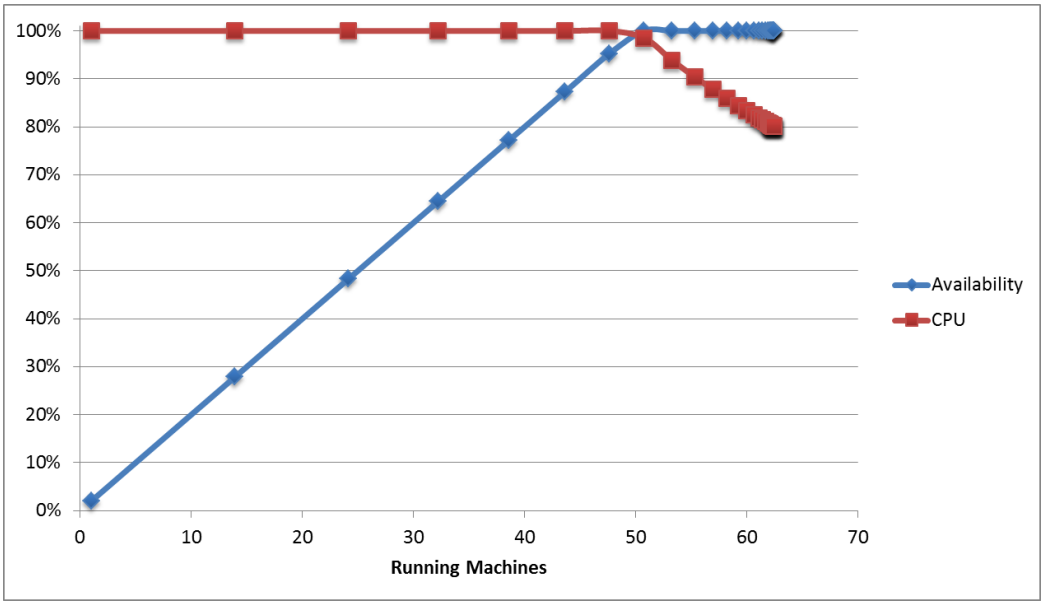


(b)

Figure 4.4: Convergence of Equation 4.18, starting from one only running machine, with an arrival rate of 5000 requests per second, a maximum service rate of each machine of 100 requests per second and a convergence rate $\beta = 0.75$.



(a)



(b)

Figure 4.5: Convergence of Equation 4.19, starting from one only running machine, with an arrival rate of 5000 requests per second, a maximum service rate of each machine of 100 requests per second and a convergence rate $\beta = 0.75$.

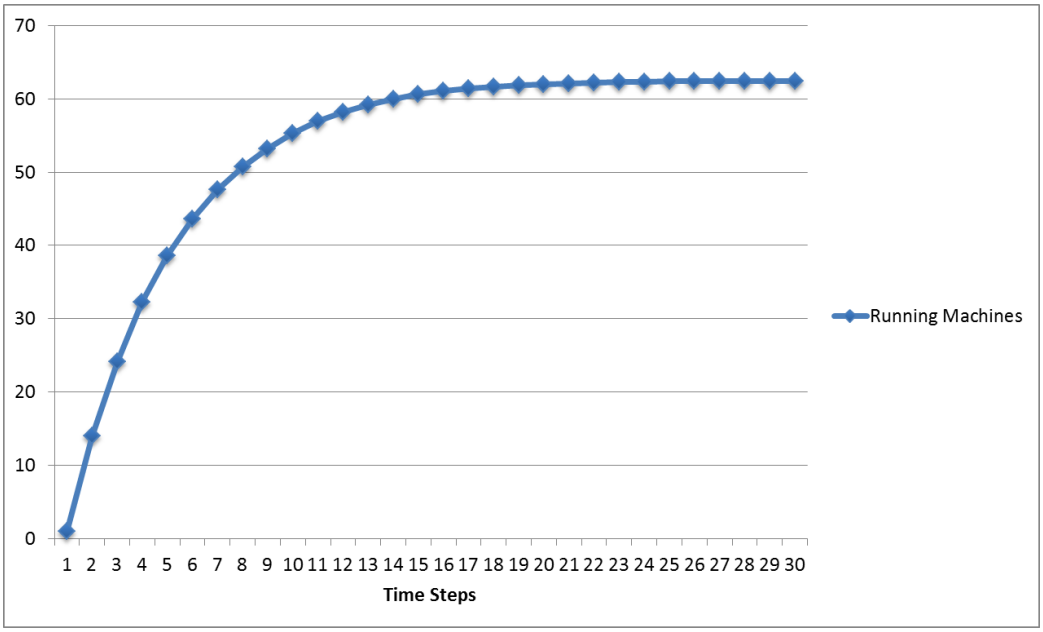
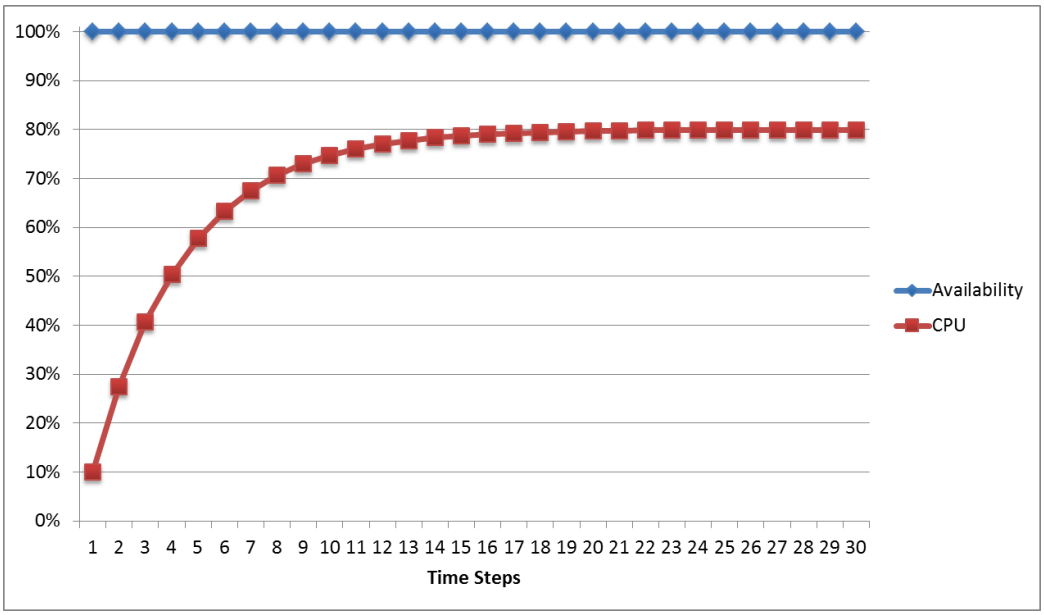
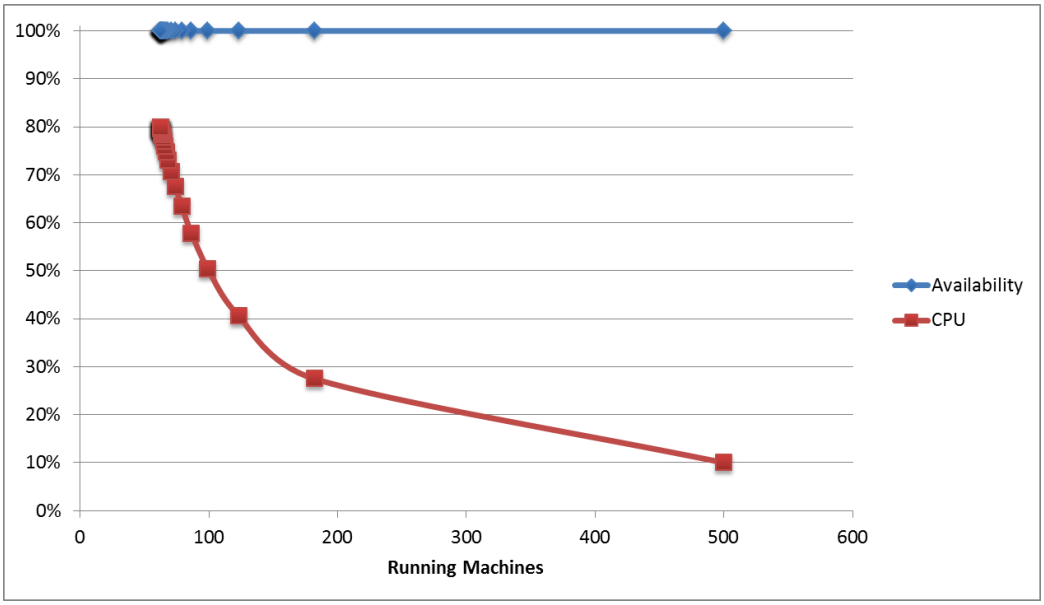


Figure 4.6: Convergence of Equation 4.19, starting from one only running machine, with an arrival rate of 5000 requests per second, a maximum service rate of each machine of 100 requests per second and a convergence rate $\beta = 0.75$.



(a)



(b)

Figure 4.7: Convergence of Equation 4.19, starting from 500 running machine, with an arrival rate of 5000 requests per second, a maximum service rate of each machine of 100 requests per second and a convergence rate $\beta = 0.75$.

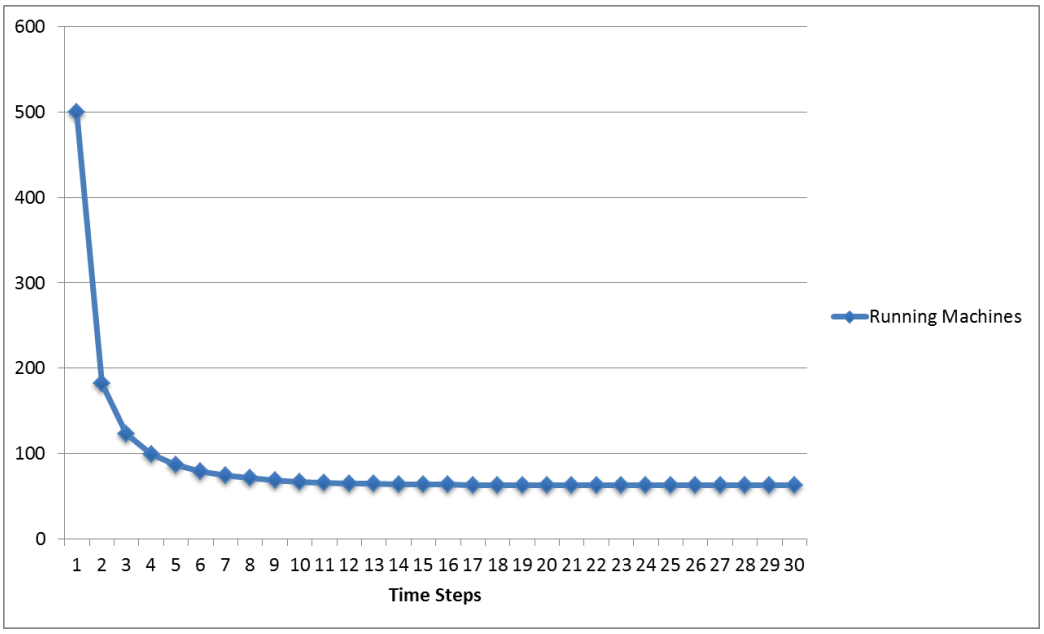


Figure 4.8: Convergence of Equation 4.19, starting from 500 running machine, with an arrival rate of 5000 requests per second, a maximum service rate of each machine of 100 requests per second and a convergence rate $\beta = 0.75$.

We might be interested in finding a bound in the number of steps required to reach convergence given a starting condition. From Section 3.3 we now that the error $u(k) - p(k)$ has an exponential decay. In fact, let $e(0)$ be the initial error $u(0) - p(0)$, then $e(k) = a^k e(0)$. If one assumes that the system converged when $e(k) \leq \epsilon$ then this happen when:

$$k \geq \log_\beta \frac{\epsilon}{e(0)} \tag{4.20}$$

.

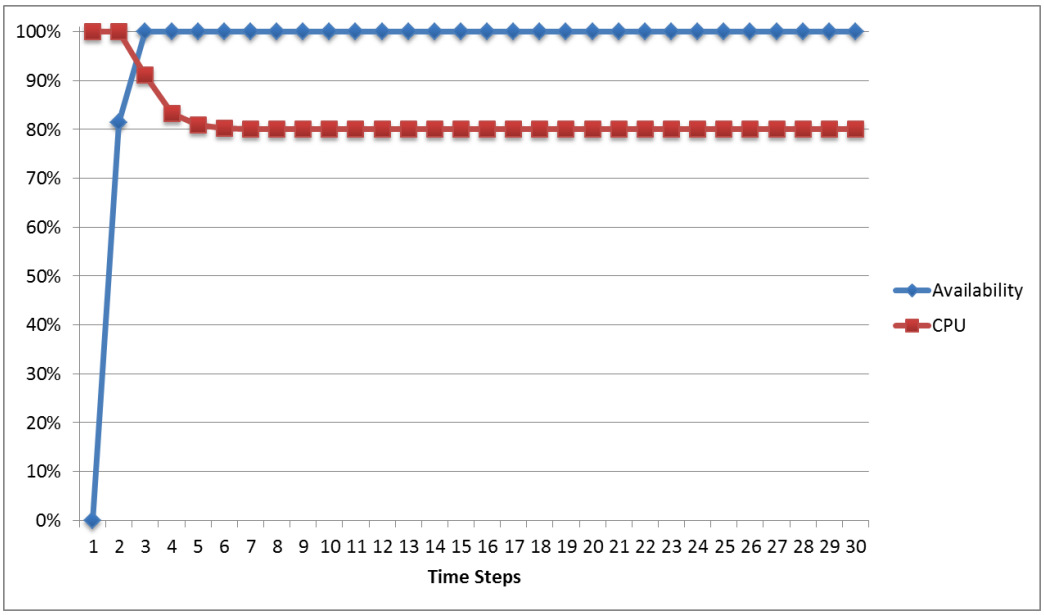
Though, working with formula 4.19, it is hard to find a minimum k for which we can consider the system converged analytically. We preferred to find a bound numerically, postponing a more rigorous formalization for the future work. First of all, considering equation 4.19 we notice that the convergence depends on the starting value of the availability, the initial number of running machines, the initial CPU usage, the desired CPU usage and the convergence rate β. As for the evaluation of our approach in Section 6, we are mainly interested in knowing how fast the controller will make the system work again after the failure of a cloud. Therefore, our priority is to check how long the scale up takes. Convergence was evaluated in the same working conditions used during our tests, that is:

- $\beta = 0.3$
- desired cpu usage $u = 0.8$
- cpu usage tolerance $t = 0.1$

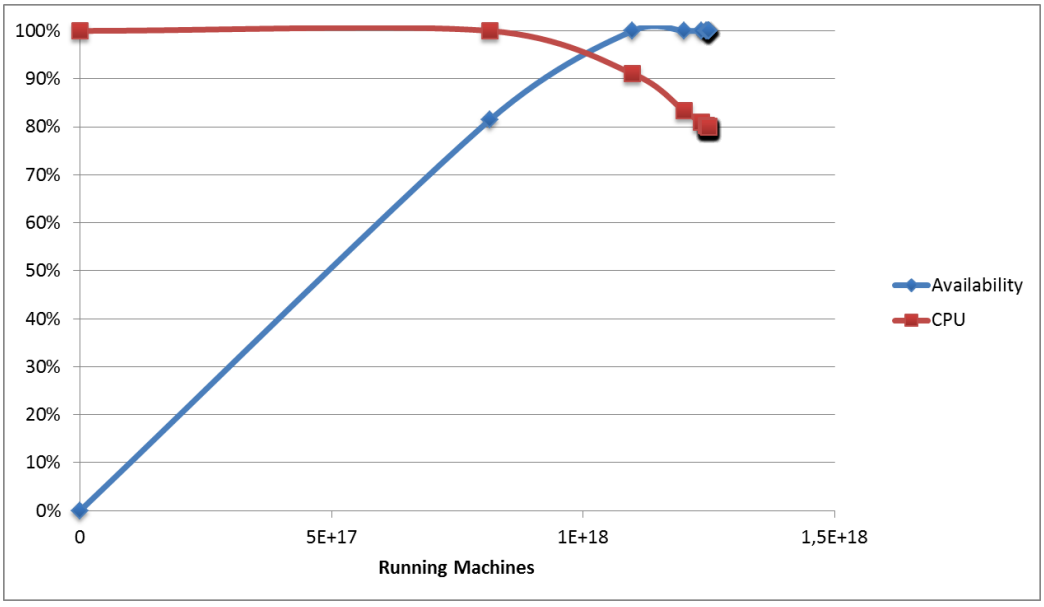
We then set the initial value of the running machines to 1 and availability very close to zero, setting a machine service rate equal to 100 and an arrival rate of $1e20$, so to consider the worst case scenario. Figure 4.9 captures the resulting convergence behavior, which tells us that the autoscaling group will reach the desired working state (i.e. $\|p(k) - u\| \leq t$) in 3 steps.

$a(k)$ and $p(k)$ on the left hand side of both the Equations 4.17 and 4.14 should actually be the predicted values at the next step, but, as for this thesis, we will just use the average value observed during the observation window. As we said, prediction is deferred to future work (see Chapter 7).

Obviously the number of machines is an integer number, but we do not encounter any problem in rounding this number, unless we deal with a very small number of machines or with CPU utilization ranges too close to the set point. In these cases, there would be undesired behaviors, however not too difficult to cope with. In the case, for example, we are dealing with a small number of machines, we can use the ceiling of the decimal solution



(a)



(b)

Figure 4.9: Convergence to the desired working condition, that is cpu usage between 70% and 90%, is reached in 3 steps.

computed by the controller instead of rounding it. This way, if $n(k) = 1$ and the controller returns $n(k) = 1.25$, the ceiling would make the node scale up, which is a preferred behavior when high availability is required.

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

4.3.2 The load balancer controller

The second layer controller is instead responsible of setting the controllable variables of the DTMC model. In order to work properly, this controller should work at different time scale with respect to the one managing autoscaling. In fact, the direct consequence of the load balancer decision may alter the amount of traffic going to the nodes, therefore the first layer controller will need some time to make the system stable back again. To avoid the risk of overloading a node, as we anticipated in Section 4.3.1, we made the first layer controller have a higher priority. The second layer controller is therefore inhibited by the first layer controller. Once every node's controller exits the *cool-down* state, the load balancer can restart its periodical control steps after having waited for its entire *observation window* to be fed by fresh data.

Objective This controller aims at distributing traffic among nodes guaranteeing availability and minimizing costs. As we said in Section 2.1 different providers offer different prices, that may change over time (e.g. Amazon hotspots). Furthermore, being the cloud a shared infrastructure, performance can change over time as well. Therefore, at different time of the day may be more convenient one solution with respect to the other.

Moreover, we also observed in Section 2.1 that the availability of a single cloud region is very low, so the controller is responsible of reacting when an entire region fails, migrating incoming requests to the remaining available nodes.

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

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

$$\frac{successful\ requests}{average\ CPU\ utilization}$$

- the *cost per request*, which measures the cost of a request traversing the node (see Reward Markov Chain in Section 2.3) at the desired CPU utilization and it is computed as

$$\frac{cost\ per\ machine\ per\ second \times number\ of\ machines}{service\ rate \times desired\ CPU\ utilization}$$

As already said, prediction could improve performance, but it is deferred to future work (7).

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

As we said, if any of the nodes is unstable, this controller is inhibited until all nodes are stable, that is until there is no more autoscaling process going on. Also in this case, we proposed a mixed approach between reactive and temporized control.

Obviously, whenever a failure occurs, we want to fix the availability of our system as soon as possible, so we decided to activate the controller whenever the average success rate of the system falls below the set point. Then, we decided not to be reactive on cost changes because they may change continuously, as for the Amazon hot spot instances (explained in Section 2.1), for example, and we do not want our controller to work no-stop, solving constrained minimization problems, which are quite computationally expensive, to have infinitive load balancing modifications. Therefore, we had the controller temporized. The control interval can be set by the developer based on some analysis on costs fluctuation rate, time constraints, or whatever application requirements are. The control timer is reset every time the controller takes a decision. In the case the timer runs out when the controller is inhibited by the first layer controller of some node, the controller will intervene as soon as the node exits the *cool-down* state.

As we anticipated, the controller is going to solve a constraint minimization problem. The control variables are, as we said in Section 4.2, the attached probability of some arc of the DTMC model. The controller has to choose, among all feasible values, the ones that minimize a cost function.

Since we deal with probabilities, the first constraint is that controllable variables must be chosen in the range $(0, 1)$. Also, since we are dealing with a DTMC, the sum of the outgoing arcs must be 1. This last constrain can be avoided allowing only two outgoing arcs on load balancers and set the value of one of the arcs equal to one minus the other. If we want to have a load balancer with three or more outgoing arcs, it is enough to put two or more binary load balancers in cascade.

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

$$\mathbf{x^T}(k+1) = \mathbf{x^T}(k)\mathbf{A} + \mathbf{b^T} \tag{4.21}$$

where $\mathbf{x}$ is a vector as long as the number of nodes, and b is the input vector, as long as $\mathbf{x}$, with all 0s except for the ith element which is 1. If $\mathbf{b}$ is constant the system is going to stabilize and the values of x are going to be the *workload ratio* arriving at each node:

$$\begin{aligned} \mathbf{x^T} &= \mathbf{x}\mathbf{A} + \mathbf{b^T} \\ \mathbf{x^T}(I - \mathbf{A}) &= \mathbf{b^T} \\ \mathbf{x^T} &= \mathbf{b^T}(I - \mathbf{A})^{-1} \end{aligned} \tag{4.22}$$

The jth element of $\mathbf{x}$ is going to be the success rate as a function of the control variables and nodes availabilities, which will be used to estimate the availability. Since we are dealing with models whose structure is constant in time, the success rate function is always the same and can be computed at design time.

Now we can write the availability constraint function as

$$u(k+1) - \hat{s}(k+1|k) \leq \alpha \cdot \max\left(0, u(k) - s(k)\right) \tag{4.23}$$

where u is the set point, $\hat{s}$ is the estimated availability, using the average availabilities of the nodes and letting $\hat{s}$ become a function only of the control

variables. α is a parameter in the range $(0, 1)$ that will affect the convergence rate to the solution. Finally, s is the system availability measured at step k. Using equation 4.23 the controller is allowed to let $\hat{s}$ be greater than the set point u.

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

$$J_1 = \mathbf{x^T} \cdot \mathbf{k} \tag{4.24}$$

where $\mathbf{x^T}$ is the previously calculated *workload ratio* array that, once availabilities are substituted with the average availabilities measured for each node, depends only on the control variables. $\mathbf{k}$ is instead the vector containing the *cost per request* values.

We said "tentative" cost function because there is still something missing. Let us suppose that all nodes are stable and healthy, that is, 100% availability. We are using one cloud, and suddenly a second cloud prices become more convenient. The minimization of the cost function J_1 would cause a sudden migration of requests from the first cloud to the second cloud, which will not be capable of satisfying the entire workload until the scaling process is complete. Consequently many requests will be lost, and availability will be consistently affected. In the future work (7), in the case we are dealing with nodes modeling *IaaS* autoscaling groups (see Section 2.1) we could think about a pre-instantiation of the machines before performing the migration. Though, we might also have *PaaS* which have their own scaling polocy. Or else, even with pre-instantiation option we will not be sure about the perfomance of the new cloud, and, consequently, about the exact number of machines needed in the new cloud before making it work at full capacity.

Therefore, we want somehow to discourage big changes on the control variables. One way would be to add a big weight to the increment:

$$J_2 = \mathbf{x^T} \cdot \mathbf{k} + M \left\| c(k+1) - c(k) \right\|^2 \tag{4.25}$$

where $c(k)$ is the vector with the old control values, while $c(k+1)$ is the vector with the new values, which would be left free for the controller to set it. M is a big number to be tuned, the bigger this weight the smaller the increment.

This option has still some issues on the tuning of parameter M, which is very sensitive to the use case, and difficult to set. Therefore we finally opted for the following solution

$$J = \mathbf{x^T} \cdot \mathbf{k} + W \left\| \max(0, AR(k)\mathbf{x} - \mathbf{SR}(k)) \right\| \tag{4.26}$$

where W is a big number, easier to tune than before since it is sufficient to have it much greater than the first member of the cost function. AR is the average arrival rate to the input node of the system. $\mathbf{x}$ is again the array of workload distribution on the node relative to the incoming workload to the input node, and depending on the control variables that will be chosen by the controller. $\mathbf{SR}$ is the array of the estimated maximum service rate of each node. The rationale beyond this cost function is to discourage the controller to load a node with more requests than the ones it is actually estimated to be capable of processing. Whenever a migration of requests for economic reason is required, the workload is gently distributed on the cheaper node letting it the time to scale without overloading it, that is, without loosing requests.

This approach is a workaround to put a constraint to be considered only when the nodes availabilities are high. We want to avoid losses whenever the migration is only for economic reasons. When the availability constraint is not satisfied, because, for example, an entire autoscaling group failed, the controller will not find a minimal solution of J without overloading a node, but in this case it is the desired behavior for the following reasons:

- All requests going to the failed node would be lost anyway
- The overloaded node will scale much faster in order to cope with the new workload since the availability a in the Equation 4.19 will be very low. Even in the case of a *PaaS* node, we expect the provider policy to react faster than the case of a gentle migration.

As we said, the second layer controller will not work until the first layer controller has stabilized the overloaded node. This way system oscillations are avoided.

Data starvation A critical aspect to deal with is *data starvation*. If a control variable is ever set exactly to 1 or 0, there will be nodes not receiving any request, causing the sensors on those nodes to fail in monitoring their effective healthiness. In order to cope with this problem we decided to put bounds to the values that control variables can have. This bounds are chosen so that every node, even the failed ones, are always fed with a very small workload. This quantity should be minimal with respect to the whole workload, so that the system availability is not compromised.

In Section 7 we investigate other alternatives that might be taken in consideration in the future to avoid loosing requests at all.

Chapter 5

Tool

This chapter presents two tools that have been developed in order to test the control approach presented in Chapter 4. The first is an extension to Palladio that allow us to easily design application and automatically derive the corresponding instance of the model (Section 5.1). The second is a simulation engine built in Matlab that receive as an input the model and a few more parameter about the use case and simulates it (Section 5.2).

5.1 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 5.1 shows a simple repository composed of four components: a web

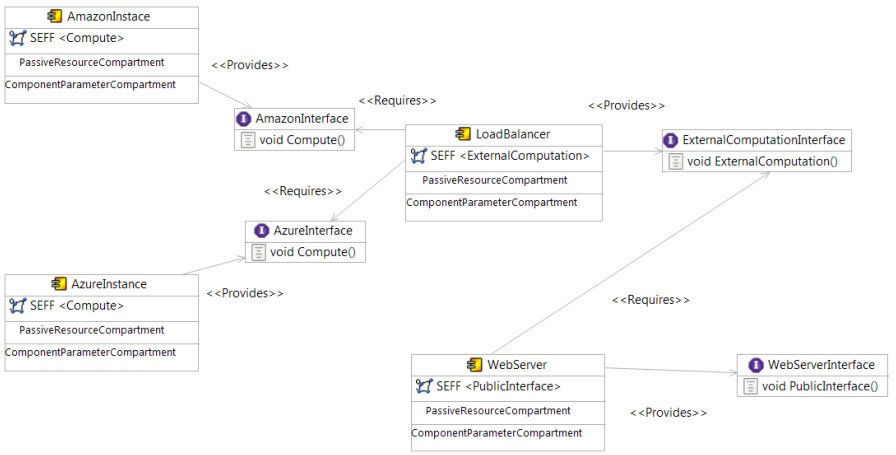
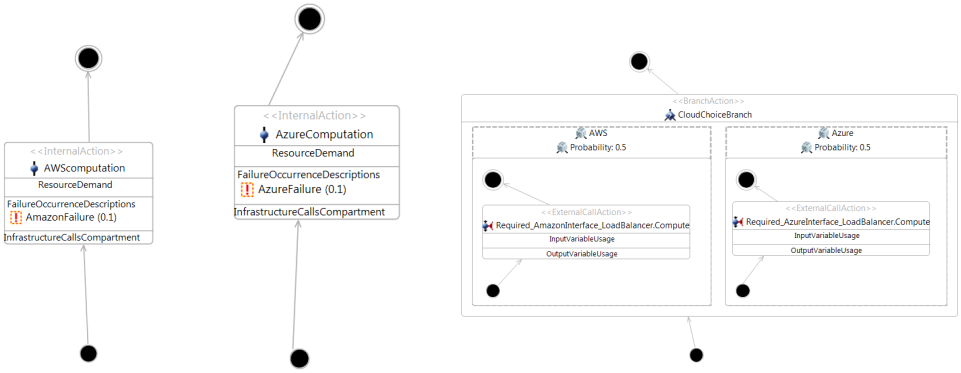


Figure 5.1: Example Repository



(a) (b) (c)

Figure 5.2: SEFF diagrams

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 Figure 5.2(a) and 5.2(b). The load balancer SEFF diagram is shown in Figure 5.2(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.

If we model this system in Palladio we can run its evaluation tool based on

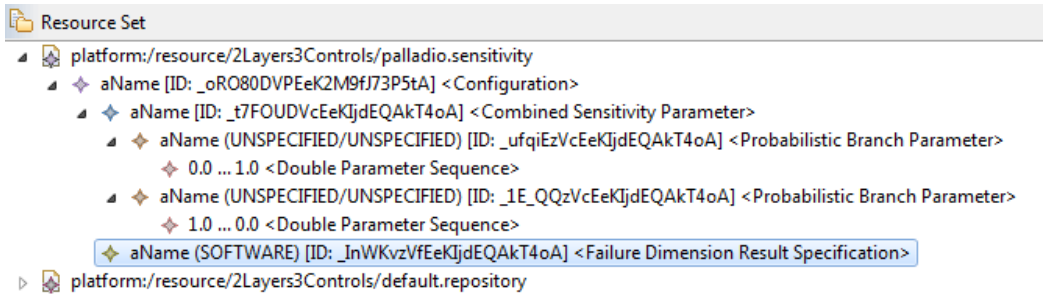


Figure 5.3: Sensitivity file example

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

Table 5.1: Result of a sensitivity run

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 5.3 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 5.1, 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.

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

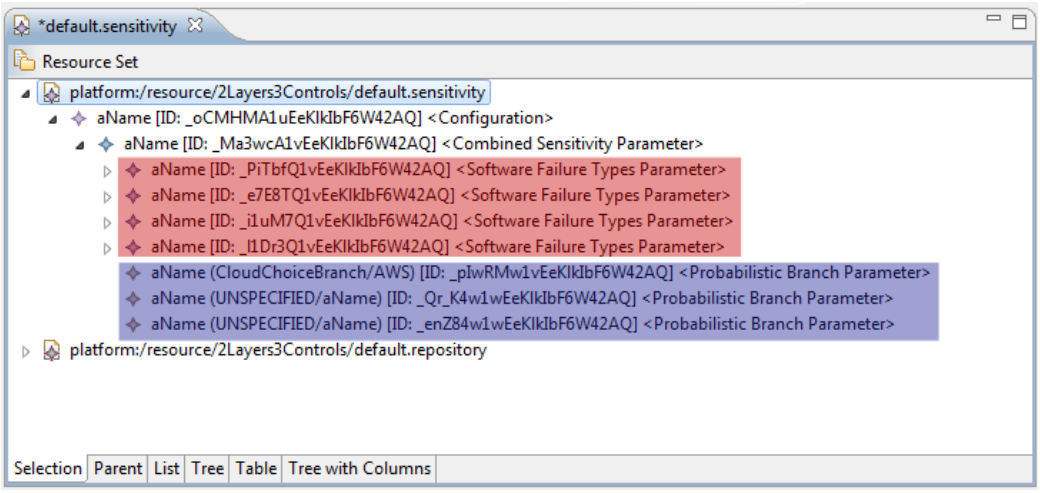


Figure 5.4: Complete Sensitivity File

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 5.4 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 that will be transformed in the model in control variables.

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

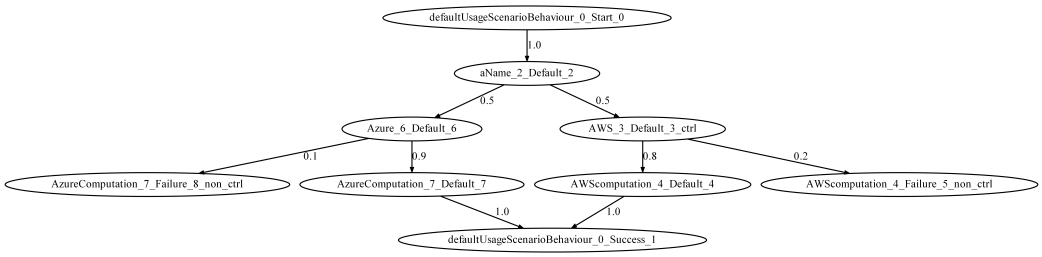


Figure 5.5: First step of the transformation

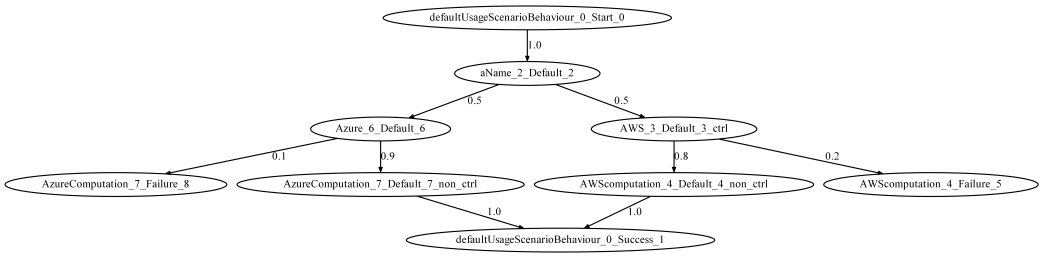
a new very compressed DTMC with one start state directly connected to the success state and to all failure states annotated with their probability. This small matrix does not contain enough information on the structure of the application so is useless for any control approach. For this reason we decided 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 5.1, can be seen in Figure 5.5. For this example, the web server and the load balancer controller are set to logic nodes, that is, as we said in Section 4.2.

The next step of the transformation is to move labels for non control variables from failure states from corresponding success state, this is done in order to simplify the process of the successive one which is to expand 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 5.6(a) and Figure 5.6(b).

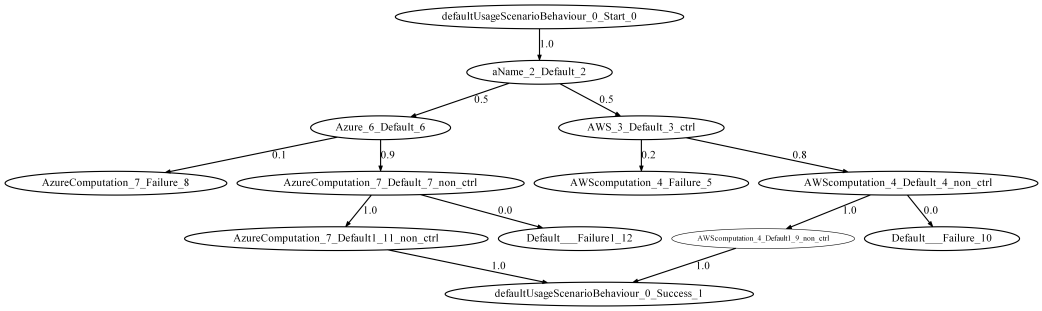
Steps four and five are dedicated to the generation of measured availability variables, in order to do so we need to label as non controlled all the states having as incoming transition only transitions that have not been already considered as control variables or measured availabilities. Step four does this by labeling corresponding states and step five moves the labels from state to the corresponding transitions. The output of these steps is shown in Figures 5.7(a) and 5.7(b).

The last modification that we need to do to the DTMC is adding self loops with probability one to all final success or failure states in order to make them absorbing states for requests flowing in the system. This is done in the last step which gives as output the model in Figure 5.8.

As introduced in Chapter 1 in order to verify the validity of the con-



(a)



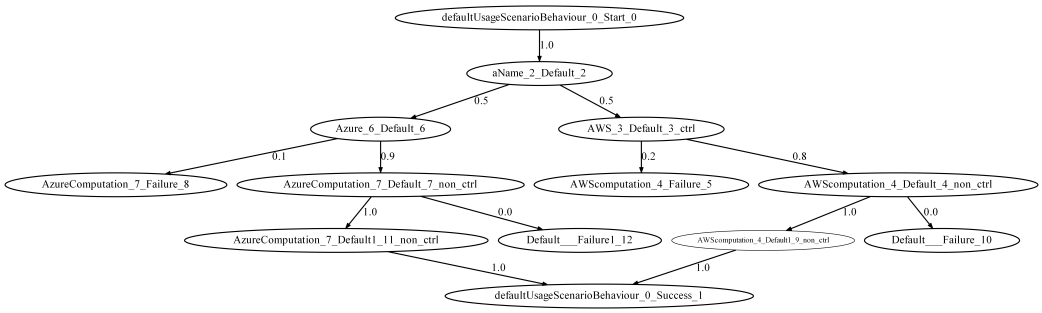
(b)

Figure 5.6: Second and third steps of the transformation

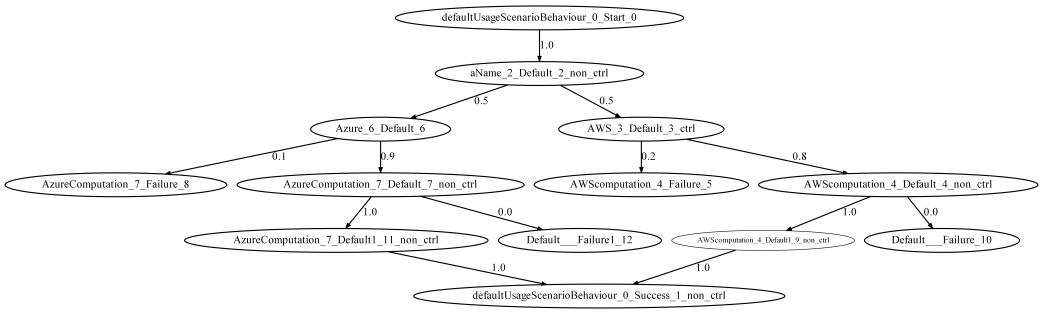
troller that we have developed we performed some simulations. The code for this simulation, that will be described in Section 5.2, is composed by some matlab files with some tokens in correspondence with fields that describe the model, simulation parameters or user inputs. After generating the final DTMC model, the tool parses these template files and writes in the appropriate sections information like the matrix of the DTMC system and all the parameters needed for the simulation.

5.2 Simulation

In order to validate the control approach presented in Section 4.3, we implemented a simulation algorithm based on the model presented in Section 4.2. 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 group. 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.



(a)



(b)

Figure 5.7: Fourth and fifth steps of the transformation

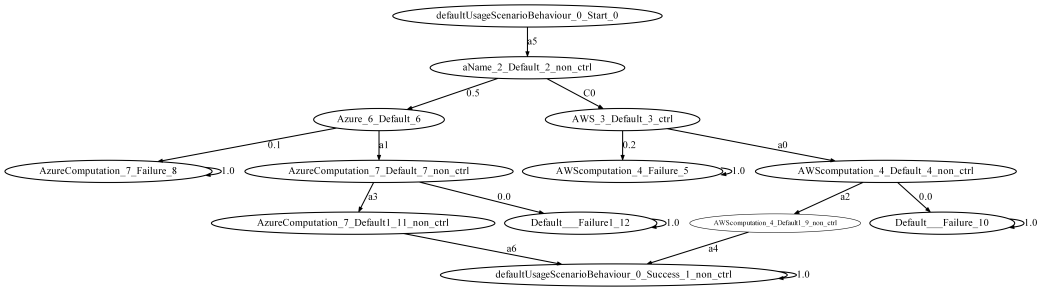


Figure 5.8: Final result of the transformation

During the simulation, the system evolves given input dynamics preset by the user. The variables that have to be predefined are:

- *The input workload*, that is, the number of request that the system has to process per step. This parameter is useful if we are interested in observing how the controller reacts to peaks of incoming requests or other fluctuations.

- *The availability* of each node. This parameter allows developer to sim-

ulate fault of a system, the fault can be a sudden death of a node or a degradation of its service.

- *The maximum capacity* of each node, that is, the number of requests that the node is able to process per step. This parameter has been added in order to simulate the fact that processing power of VMs can change dynamically. This is due to the fact that VMs use a shared infrastructure. It is common for cloud providers to run different VMs on the same physical machine, so it may happen that a VM of a user is affected by the behavior of other VMs. Cloud providers usually try to limit this behavior in various way but it is still present and it may affect QoS heavily. Usually large variations on the processing power of VMs is registered between daytime and night.

- *The startup time* of VMs. This parameter models the time that pass from the request of scale up the number of machines done by the controller and the time when machines start to serve requests. This parameters changes according to the cloud provider, the instance operating system and the instance size as shown in [4].

- *Simulation time* and *time step size* can be specified by the user in order to describe the time that he wants to simulate and the granularity of the discretization of this time. These values are then used to define the number of steps for the simulation.

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 with the 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

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 the one it can serve exceeding requests are routed to its failure state. 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 now using 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 up (or down) process. The simulation engine takes into consideration scaling actions requested by the controller 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 and the total costs. Examples of the output of the simulation can be seen in chapter 6.

The simulation is divided into steps, the user can choose the time duration that he wishes to simulate (e.g. a 24 hour scenario) and the granularity of the simulation steps. The number of steps is the given by

$$\left\lceil \frac{\text{simulated time}}{\text{seconds per step}} \right\rceil$$

For each step k the simulation engine performs the following operations

1. loads the value of all parameters describing the state of the system environment at step k

2. updates the transition matrix with control variables set by the controller in the previous step and with the availability values of each node.

3. generates a simple workload for the simulation, assuming the inter arrival times to be exponentially distributed. So, a Poissonian random number generator is used with mean given by a user defined function which specifies the arrival rate multiplied by the seconds in a step. More information about realistic traffic generation can be found in [24]

4. 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)

5. as described in [24], a simple way to simulate a realistic service time is modeling its distribution by means of exponential variables. So, for each node traversed by the requests, the total service time needed to serve incoming workload is generated using a random generator over the Gamma distribution

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

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

6. the amount of requests that fails due to timeout are computed by comparing the duration of the step and the total service time required

7. 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

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

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

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

11. historical data is saved to feed monitors with data for estimates

12. if any scale up timer runs out the pending machines are set to active and will be available for further computational power in the next step

13. the first layer controller, responsible for the autoscaling of machines, checks if any scale up or scale down process has to be performed

14. the second layer controller, responsible for setting the control variables of the DTMC model, checks if any change in the load balancing of requests among nodes have to be changed to satisfy the user defined goal at the minimum cost

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{:});
```

Chapter 6

Experimental Analysis

In this chapter we evaluate our approach by the means of three use cases through simulation of generic cloud providers in different scenarios. The simulation technique used for this tests is introduced in Section 5.2. Through out all these use cases the α and β parameters for the control algorithm have been initialized to 0.5 and 0.3 as default value. The first example shown in Section 6.1 represents a simple web application deployed on two independent clouds. The use case of Section 6.2 models again an application deployed on two independent cloud providers one offering a single region model and the other offering two regions for the deployment and execution of applications. The last use case, described in Section 6.3 models a more complex application, that deals with the management of a smart city emergency system. This application in particular has higher availability requirements than the other two and is deployed on top of four cloud providers. Section 6.4 makes some considerations on the behavior of the controller about the results obtained by the simulation.

6.1 A Web System Scenario

In this Section we consider a simple example to test different usage scenarios and how our approach is able to cope in case of simulated failures or changes in the domain. The main goal of this example is not to show a complex real world application but rather to test how the controller reacts to some specific scenario that may happen in the cloud environment.

Figure 6.1 shows the model of the application created by means of Palladio and our extension, explained in Section 5.1. The application is composed by a load balancer that receives users' requests and forwards them to the appropriate cloud provider.

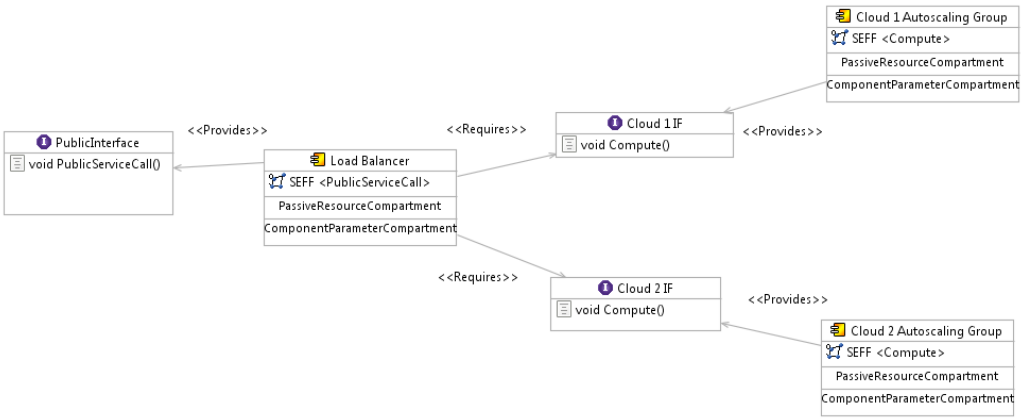


Figure 6.1: Palladio model of the first usecase

The load balancer is connected to two cloud providers on which the application is deployed. Figure 6.2 shows the DTMC model derived automatically by our tool introduced in Section 5.1. In this model we can see that the load balancer has been modeled by a node with two outgoing arch whose probabilities is controlled by the control variable $C0$.

The availabilities of the two cloud providers are modeled respectively by $a2$ and $a5$. Failures of these two nodes are independent of the application and the resources directly related to it. They may model the entire cloud failure or failure in the delivery of some requests due to network issues or software bugs of the cloud management infrastructure.

The failure of requests processed by autoscaling groups (represented by green nodes) due to their limited computing capabilities are modeled by arcs going from states 4 and 6 to the corresponding failure states according to $r4$ and $r6$. The availability of these nodes is dependent on the current allocated resources by the first layer controller. Finally, the *success* state is a logical state in which requests end, after being successfully processed by the system.

The transition matrix generated by the tool is the following:

$$\mathbf{A} = \begin{bmatrix} 0 & c_0 & 0 & 0 & 1-c_0 & 0 & 0 & 0 \\ 0 & 0 & 1-r_2 & r_2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1-r_4 & 0 & 0 & 0 & 0 & r_4 \\ 0 & 0 & 0 & 0 & 0 & r_5 & 1-r_5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1-r_6 & r_6 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{6.1}$$

By means of Equation 4.22 we can obtain the *workload ratio* vector, whose

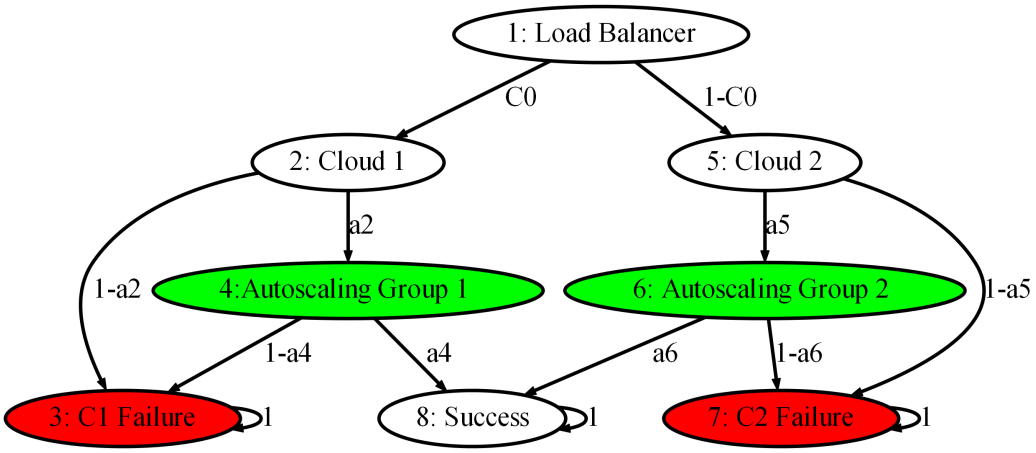


Figure 6.2: DTMC model representation of the Multi-Cloud application. Green nodes represent autoscaling groups, red nodes represent failure states.

	Cloud 1	**Cloud 2**
Cost per VM	0.30$/hr	0.50$/hr
VM startup time	100 s	100 s
VM nominal SR	$10,000\ \frac{\text{reqs}}{\text{s}}$	$10,000\ \frac{\text{reqs}}{\text{s}}$
CPU set point	80%	80%
CPU tolerance	10%	10%
Nominal cost per req	3.75E-5 $\frac{\$}{\text{req}}$	6.25E-5 $\frac{\$}{\text{req}}$

Table 6.1: Simulation parameters

8th value (success state) corresponds to the system availability:

$$s = r_5 \cdot r_6 \cdot (1 - c_0) + r_2 \cdot r_4 \cdot c_0 \tag{6.2}$$

We simulated three different scenarios against this application whose results are reposted in Sections 6.1.1, 6.1.2 and 6.1.3. In all of the three scenarios the parameters reported in Table 6.1 are kept consistent.

6.1.1 Scenario 1

The *set point* for the desired availability of the system has been initialized to 0.99 and kept constant during the simulation. This scenario simulates a four hours of usage of the system in which the *arrival rate* has been kept constant to 1*e*6 requests per second. Also the *service rate* of VMs has been kept constant its nominal value. The only parameter that changes dynamically in this scenario is the *availability* of cloud 1. Cloud 2 shows a 100% availability

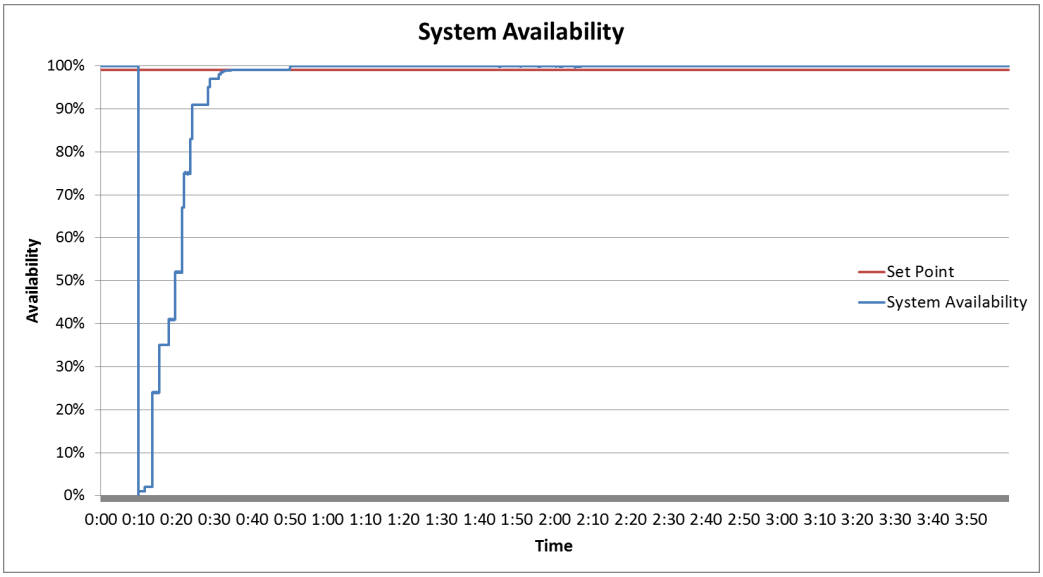


Figure 6.3: Availability of the system of Section 6.1.1

for the considered period while cloud 1 experience a failure between time 00:10 and 00:50. We can see that the cost of using cloud 1 is lower than the one of using cloud 2 while VM maximum service rates are the same, so in standard conditions the system is expected to prefere this provider over the other one.

Figure 6.3 shows the availability of the controlled system (in blue) and the desires set point (in red). From this Figure we can see that the failure of cloud 1 at time 00:10 affects the system availability but the controller is capable of discovering this failure and react by routing traffic to the second cloud provider. In this scenario the time needed to restore the desired system availability is of about 20 minutes.

Figure 6.4 shows the number of active VMs for cloud providers. We can note that as soon as the controller sense the failure of cloud 1 (at time 00:10) its number of active machines is 0 and the number of machine of cloud 2 start raising.

This is due to the fact that the first layer controller reacts by moving the traffic from cloud 1 to cloud 2. The second layer controller, seeing such a huge traffic, aggressively increases the number of running VMs until the availability is back to desired value. On 00:50 cloud 1 recovers from its failure so the controller starts to send some requests back to this one since it is the cheapest. The switching from cloud 2 to cloud 1 is done in order to reduce costs and happens much slowly than the first switch. If we look back to Figure 6.3 we can see that in this period availability is not affected.

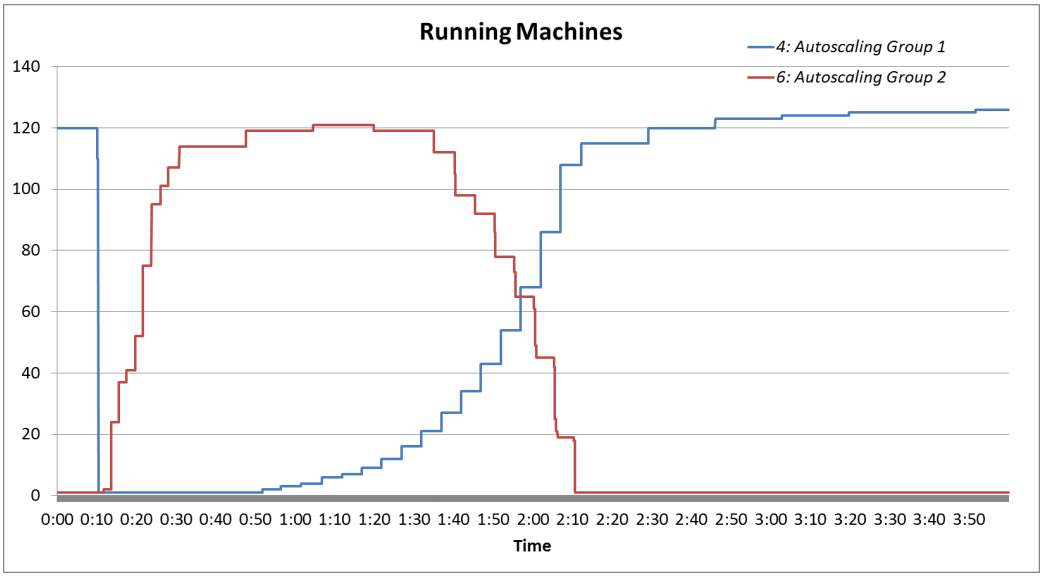


Figure 6.4: Number of active VMs of the system of Section6.1.1

	Cloud 1	**Cloud 2**	Controlled
C0	1	0	Controlled
Availability	81.57%	100%	95.90%
Cost	122.51$	251.51$	180.77$

Table 6.2: Controlled vs non controlled results

The total results of the availability and cost of the system is shown in Table 6.2. This table shows the total availability and cost of the system using only cloud 1, only cloud 2 or by using our control approach.

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

6.1.2 Scenario 2

The second scenario is quite similar to the one presented in Section 6.1.1. The length of the simulation is of four hours and the *arrival rate* is constant at $1e6$ requests per second. In this scenario both clouds' *availabilities* are kept constant at 100% but the maximum service rate of machines is changed as in Figure 6.5. This use case model a behavior that is quite usual in cloud providers and can be explained by the fact that in any reagion during

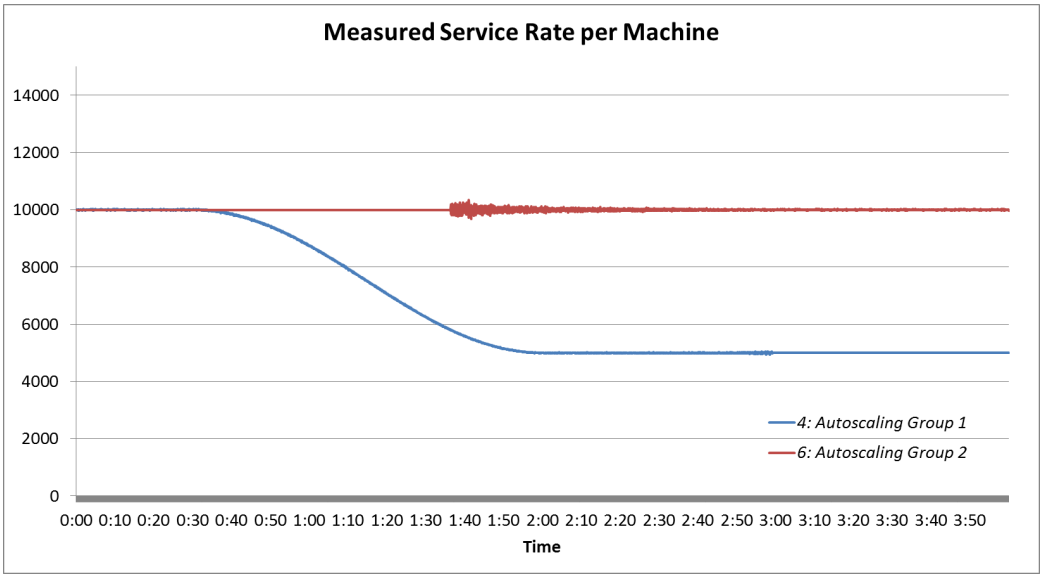


Figure 6.5: Maximum service rate of VMs

daytime hours the load on datacenters increases while it decreases at night. The increased workload causes the degradation of the performance of VMs that share resources with other users.

Figure 6.6 shows the values of the control variables that are computed by the controller. Figure 6.7 shows the utilization values of VMs. The red line is the average cpu utilization of machines of autoscaling group 1, the blue represent autoscaling group 2. The straight line at 70%, 80% and 90% represent respectively the minimum tolarated cpu usage, the desired cpu utilization and the maximum allowed cpu usage.

From these figures we can observe that when the service rate starts to decrease the load on the cpu of cloud 1 start to increase. When this value exceeds the maximum tolerated (around time 1:00 in Figure 6.7) the second layer controller scales up the number of machines in order to maintain the actual cpu load near the desired one. Since the maximum service rate continue to decrease the this behavior is repeated several times.

After a certain point the maximum service rate of cloud 1 falls behind a value that makes it inconvenient to use. This happens near 1:30 when the controller start to gradually move traffic from cloud 1 to cloud 2. The redirection of incoming requests causes the cpu usage of cloud 1 stop growing and, when enough percentage of the incoming traffic is redirected to the more convenient cloud 2, the CPU load on VMs on cloud 1 start to fall down. At the same time the new workload that enters cloud 2 makes the CPU of its machines to grow. The second layer controller of cloud 2 manages the

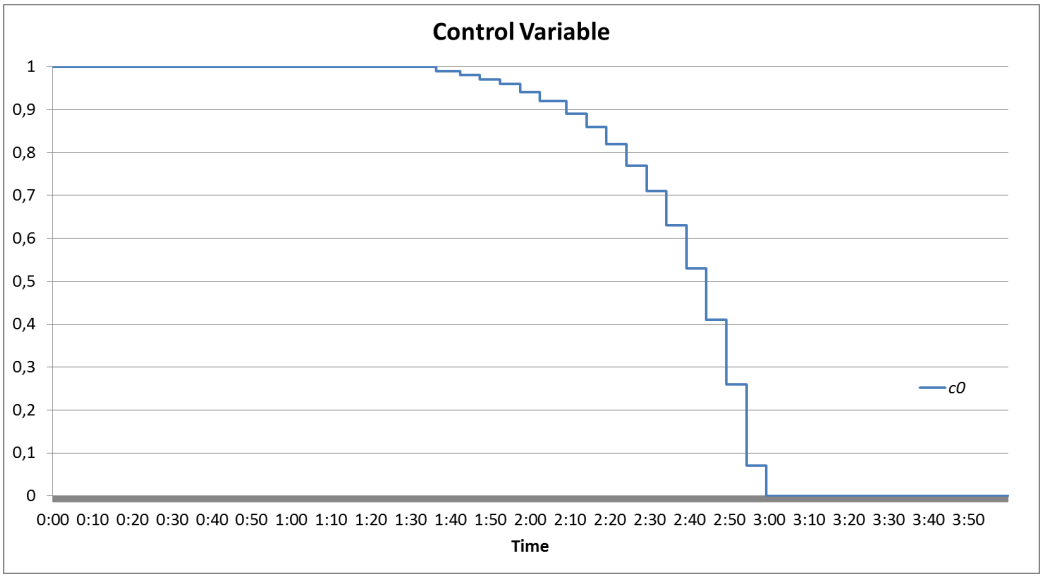


Figure 6.6: Control variable values

	Cloud 1	**Cloud 2**	Controlled
C0	1	0	Controlled
Availability	100%	100%	100%
Cost	240.99$	251.51$	225.47$

Table 6.3: Controlled vs non controlled results

growth of the incoming workload by scaling up the number of machines until the desired cpu load in reached (close to time 3:50).

From the results in Table 6.3 we can see that in all the presented cases the availability is 100% this is due to the fact that the second layer controller that manges autoscaling of nodes is always active and is capable of reacting to the gradual service degradation. On the other hand the cost of the controlled system is lower than the cost of both non controlled ones. This is due to the fact that the first layer controller redirects requests on the cloud that offers the same availability at the lowest price per request.

This scenario shows the fact that sometime in presence of gradual changes in the environment condition, the maximum service rate here, the second layer controller alone is capable to provide the desired availability. It also shows that the first layer controller can measure the effect of the degradation of machines performance and switch the application behavior in order to minimize costs.

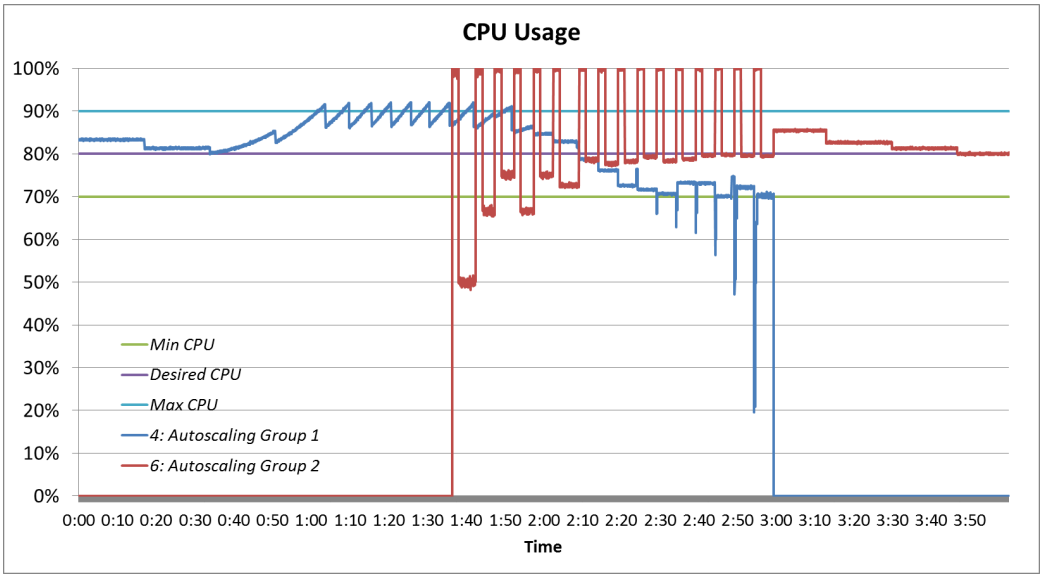


Figure 6.7: Cpu utilization values

6.1.3 Scenario 3

The last scenario for this use case is quite different and tests both the ability of the controller to react to changes in the availability of cloud providers and in the redefinition of the desired availability. Like in the previous scenarios the simulation time is of 4 hours and the arrival rate is constant. In this scenario the maximum service rate of VMs is kept constant to their nominal value shown in Table 6.1. We changed cloud 1 availability and the set point according to Figure 6.8 and Figure 6.10 (red line).

Figure 6.9 shows the availability of the system using only the first cloud provider. Figure 6.10 shows the availability of the system using only the second cloud provider. We can observe that cloud 1 alone is not capable of reaching the desired availability most of the time but from Table 6.1 it is the cheapest. Cloud 2 on the other hand is capable of satisfying the availability constraint all of the time but is more expensive. Figure 6.11 shows that the controller satisfies the availability constraint even if the set point is changed and react to these changes in a short period of time (10 minutes). By Figure 6.12 this is obtained by using a combination of both cloud 1 and cloud 2. The controller sends more requests to cloud 2 when the desired availability is raised and more to the cheaper cloud 1 when availability constraint is relaxed. We can see that from 3:00 on the desired system availability is 0.5 but the actual system availability is 0.8. This behavior is due to the fact that the system availability should be greater or equal to the set point. Figure

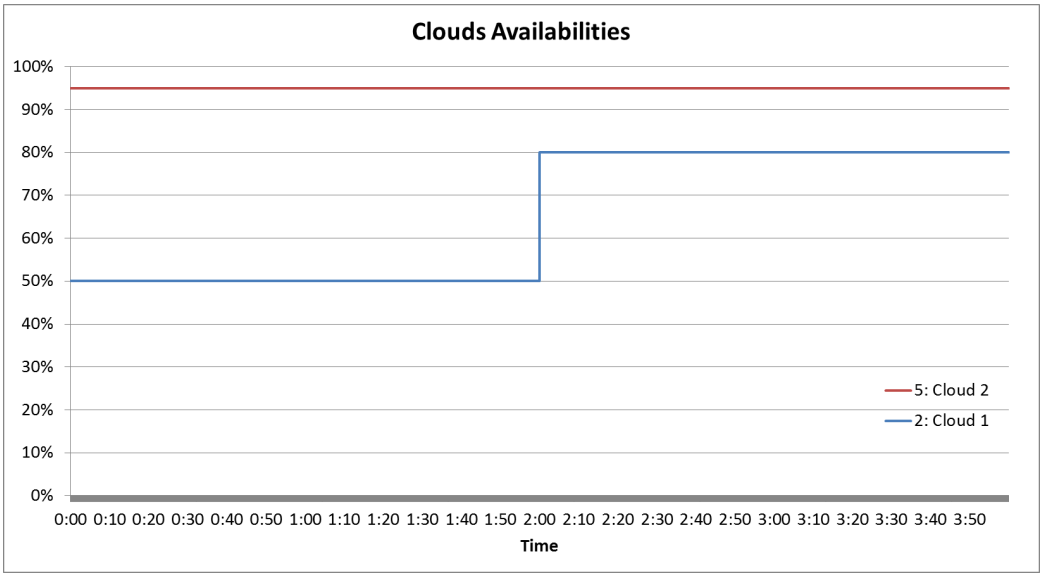


Figure 6.8: Cloud availabilities

	Cloud 1	**Cloud 2**	Controlled
C0	1	0	Controlled
Availability	64.89%	95%	82.12%
Cost	98.96$	241.20$	173.44$

Table 6.4: Controlled and non controlled results

6.12 shows that all requests are sent to cloud 1. This is due to the fact that cloud 1 is capable of providing the required availability at a lower cost.

The fact that the controller splits traffic among clouds in order to minimize costs while getting the required availability is also captured by Figures 6.13 and 6.14.

The final results are shown in Table 6.4. The availability values have a limited impact because the set point varies with time so the average value is not an accurate measure. On the other hand we can observe that the cost is a good trade off between the first cloud provider that offers a very low availability and the second one that costs much more.

6.2 A Multi-Region Scenario

This use case models the system reported in Figure 6.15. It is composed by a first load balancer that splits requests among the two cloud providers. The second layer controller splits requests among different regions inside

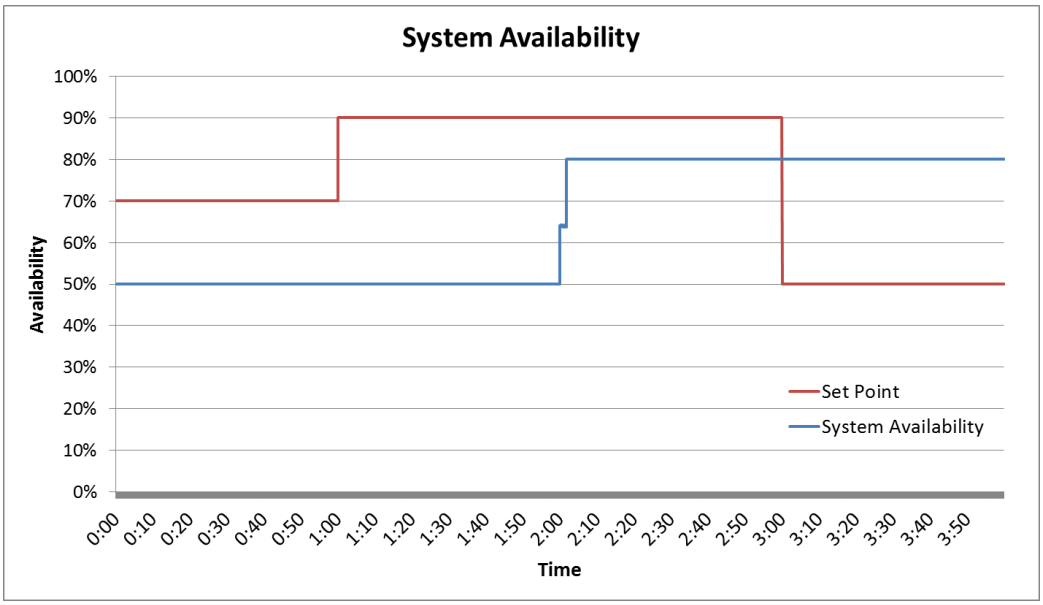


Figure 6.9: Cloud 1 system availability

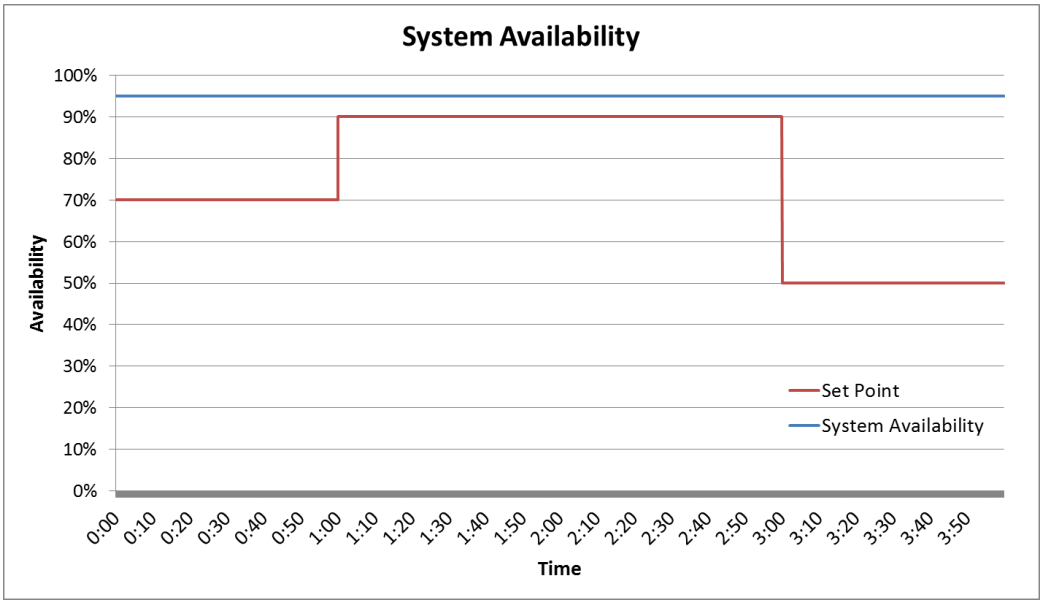


Figure 6.10: Cloud 2 system availability

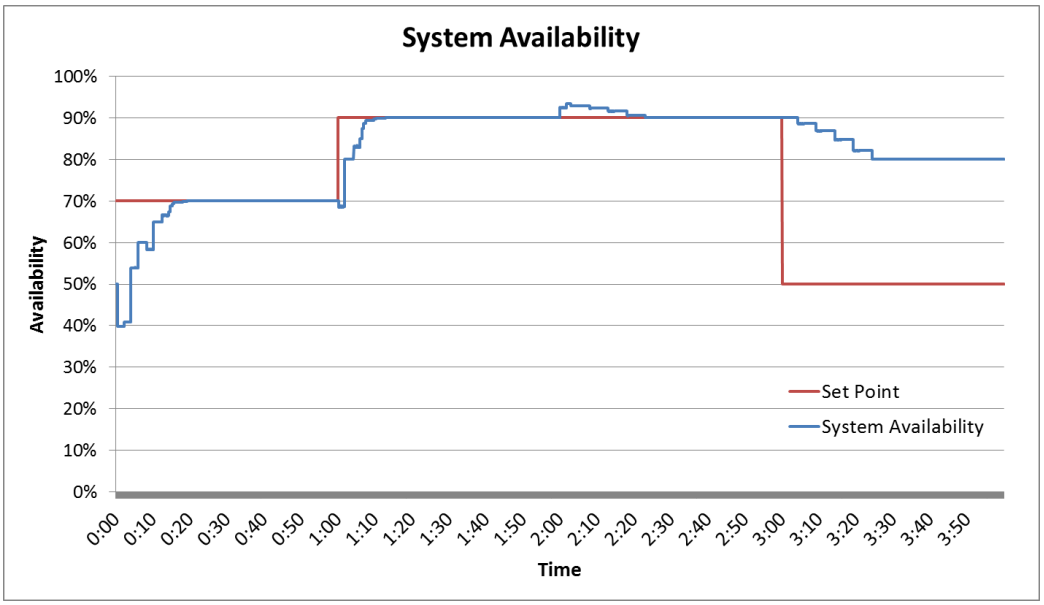


Figure 6.11: Controlled system availability

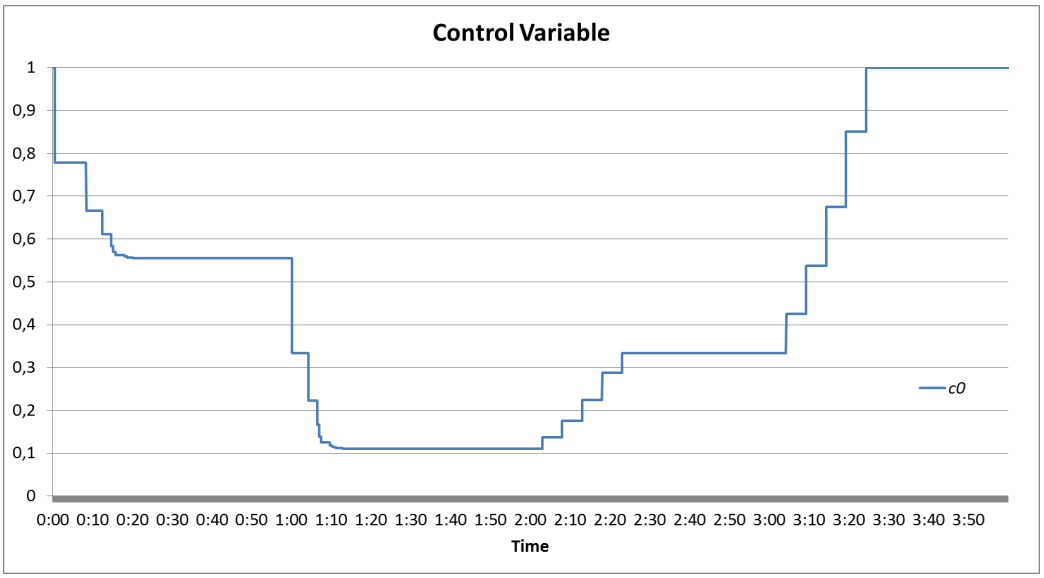


Figure 6.12: Control variable values

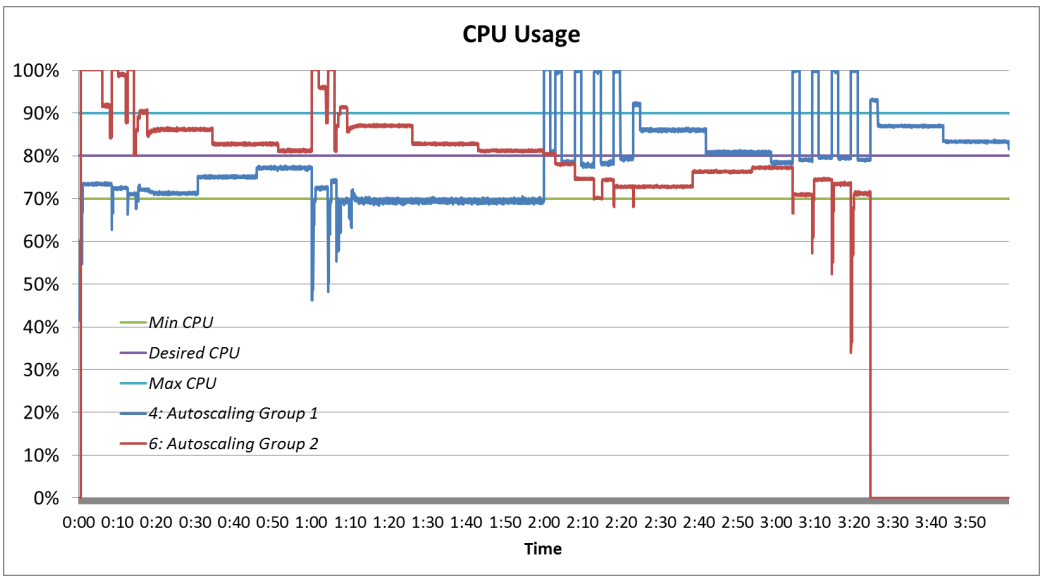


Figure 6.13: Average CPU utilization

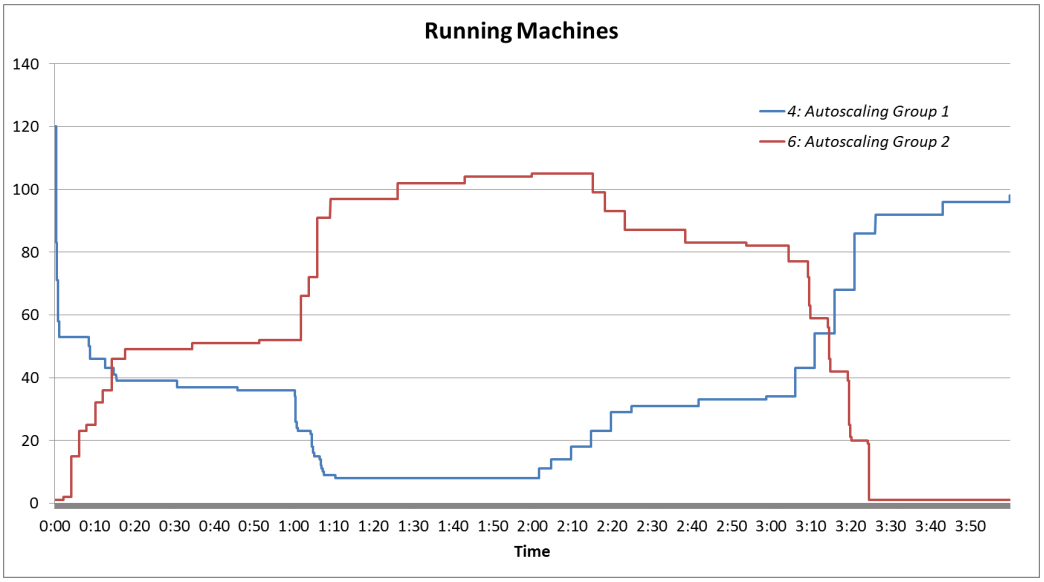


Figure 6.14: Number of VMs for the controlled system

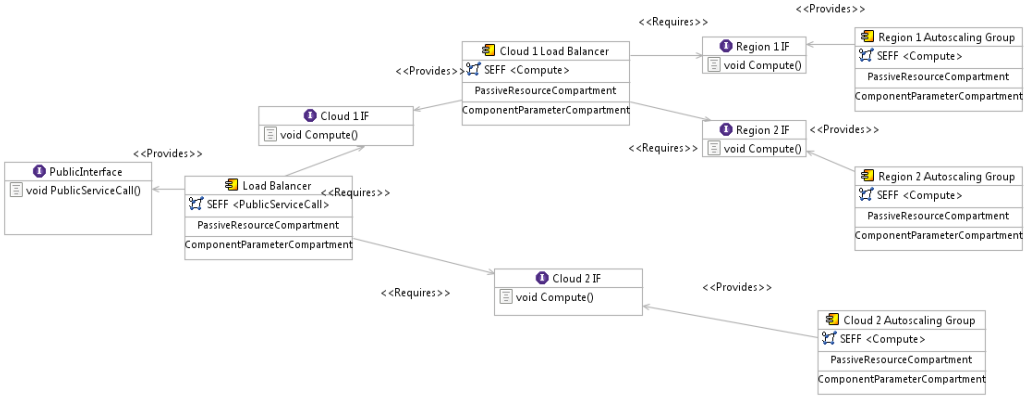


Figure 6.15: Palladio model for use case 2

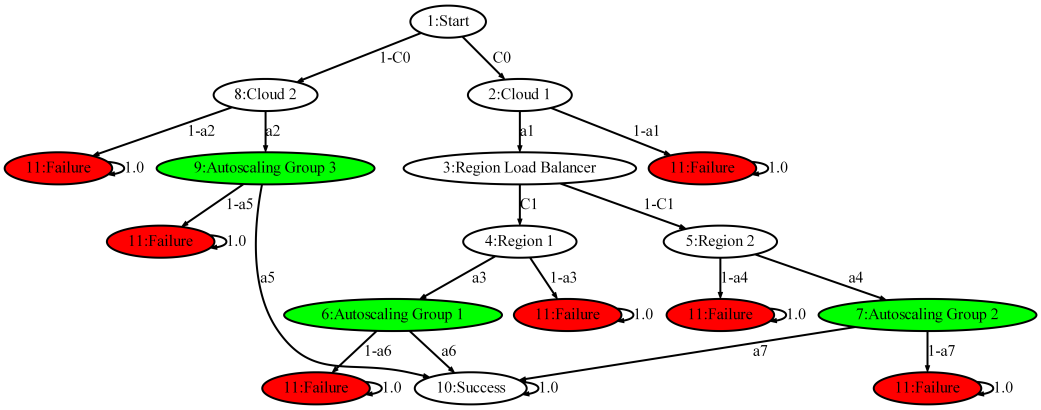


Figure 6.16: DTMC model for the second usecase

cloud 1. These two load balancers act according the probabilities given by the first layer controller presented in Section 4.3 as shown by Figure 6.16.

The value for the parameters of the autoscaling groups represented by green nodes in Figure 6.16 is shown in Table 6.5. The cost of the first cloud is varied during the simulation. The cost is set to 0.3$/hr at the beginning of the simulation and raised to 0.6$/hr at time 4:00. Costs are usually constant, but we want to be quite general in our approach, avoiding to bind to a specific cloud provider. A cloud provider like Amazon could change their prices in case spot instances are used, desribed in Section 2.1. Or else, a cloud provider may decide to change its pricing, after advising its customers, from a specific date.

The parameters we changed during the simulations are the set point that varies according to Figure 6.18 (red line). Also the availabilities of cloud providers are changed according to Figure 6.17. In this scenario the avail-

	Cloud 1 (R1)	Cloud 1 (R2)	Cloud 2
Cost per VM	0.30$/hr	0.30$/hr	0.45$/hr
VM startup time	100 s	100 s	100 s
VM nominal SR	$10,000 \frac{\text{reqs}}{\text{s}}$	$10,000 \frac{\text{reqs}}{\text{s}}$	$10,000 \frac{\text{reqs}}{\text{s}}$
CPU set point	80%	80%	80%
CPU tolerance	10%	10%	10%
Nominal cost per req	3.75E-5 $\frac{\$}{\text{req}}$	3.75E-5 $\frac{\$}{\text{req}}$	5.62E-5 $\frac{\$}{\text{req}}$

Table 6.5: Simulation parameters

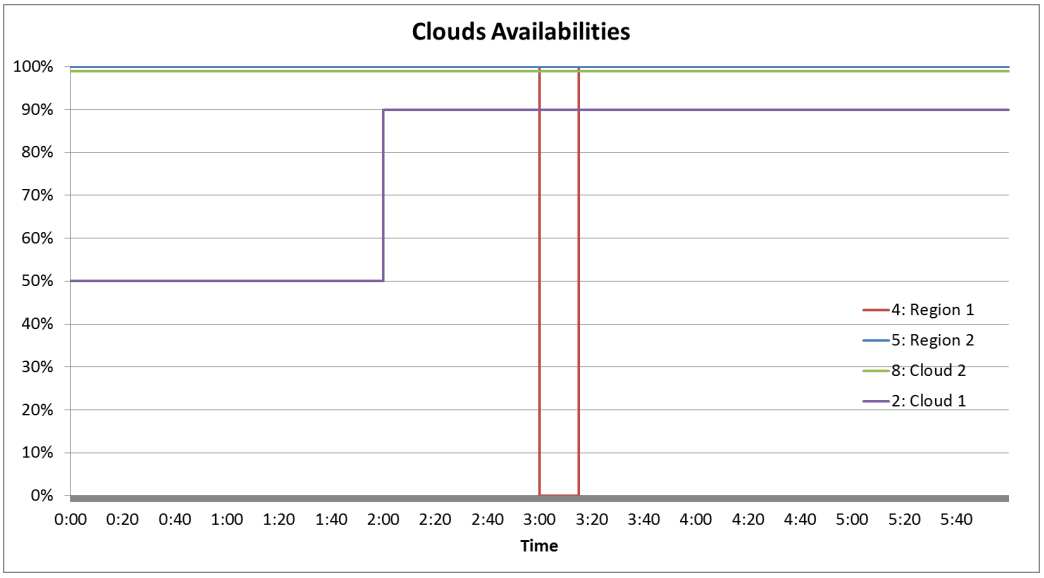


Figure 6.17: Availabilities of cloud providers

ability of cloud 2 is set to 100% and the availabilities of the two regions of cloud 1 change independently. The first region experience downtime between 3:00 and 3:20. Region 2 experiences a very low level of availability of 50% until 2:00, caused by a sudden traffic increase that caused the network to overload. From 2:00 on the network provider managed to partially solve the problem and availability grows to 90%. The duration of this simulation of 6 hours, the *arrival rate* is kept constant at $1e6$ requests per second.

The availabilities of the system simulated using only one region of cloud 1 or only cloud 2 is shown in Figures 6.18, 6.19 and 6.20. Region 1 of cloud 1 shows a low level of availability until time 2:00 and a drastic failure at time 3:00. It then recovers from the outage at time 3:20 when the system availability is brought back above the set point. Region 2 alone is not capable of satisfying the desired availability until time 2:00 but since it does not

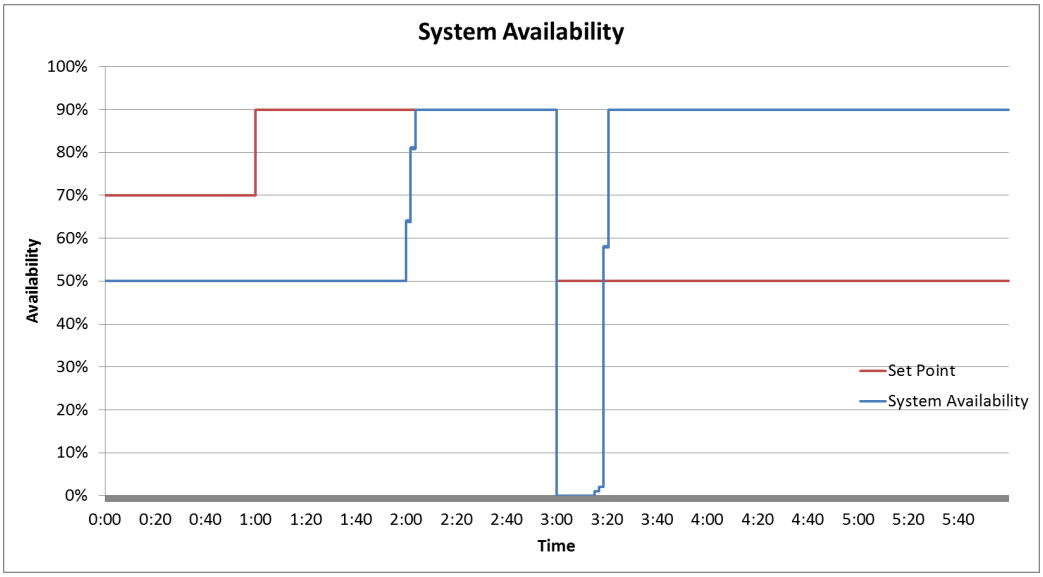


Figure 6.18: Availability of the system of using only region 1

suffer the outage of region 1 it offers a better availability value later in the simulation. Cloud 2 offers a very high system availability but it costs more than the other two clouds.

Figure 6.21 shows the availability of the controlled system. We can observe that the controller is capable of providing the required availability and to recover it after the failure of region 1.

Figure 6.22 shows that the controller keeps the desired availability by using both clouds 1 and 2 until time 1:00. When the set point is raised the system decides to send more resources to cloud 2 that offers higher availability until time 2:00 when the availability of cloud 1 is raised to 90% (Figure 6.17). The controller then slowly switches to use cloud 1 only that is cheaper and offers the desired availability. At time 3:00 the set point is lowered to 50% and simultanously region 1 experience an outage, due to bug in the cloud hypervisor software introduced after an update of the system. The controller reacts by moving some requests to region 2 and redirecting some other requests to cloud 2. When region 1 recovers from the outage at time 3:20 the controller switches back to use cloud 1 only. At time 4:00 the cost of using cloud 1 is raised so the controller starts switching slowly to cloud 2 wich is cheaper. During the switch the availability of the system increases because cloud 2 has an higher value of availability and a lower cost.

Figure 6.23 shows the number of running machines for the controlled system.

Table 6.6 shows the results of the simulations with the final costs. We

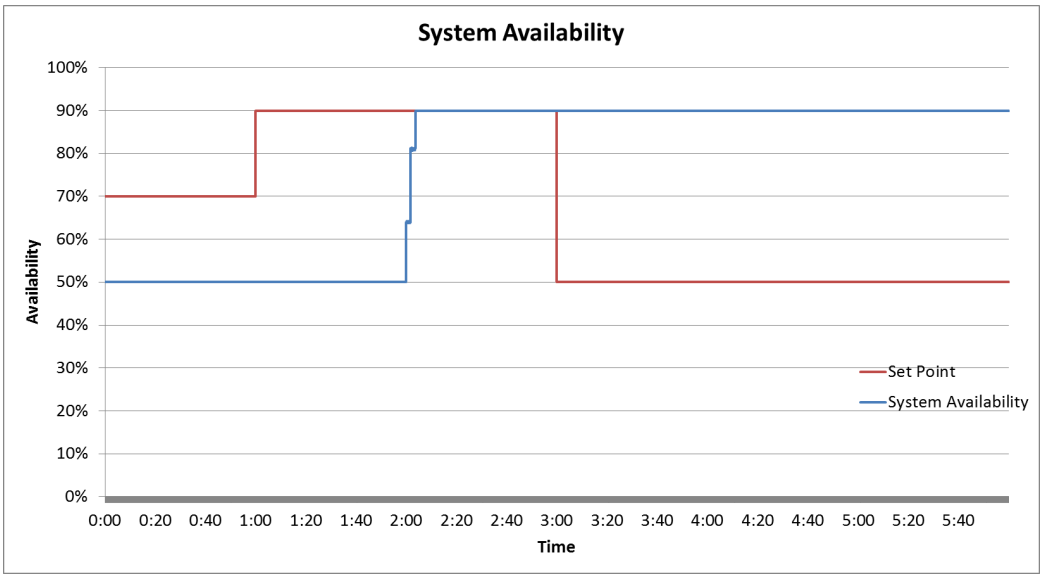


Figure 6.19: Availability of the system of using only region 2

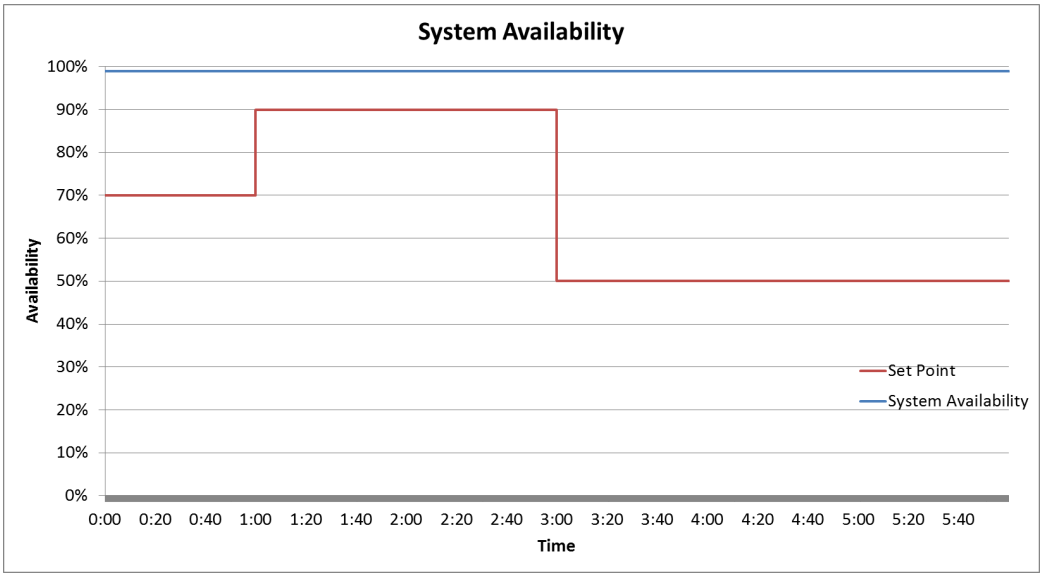


Figure 6.20: Availability of the system using only cloud 2

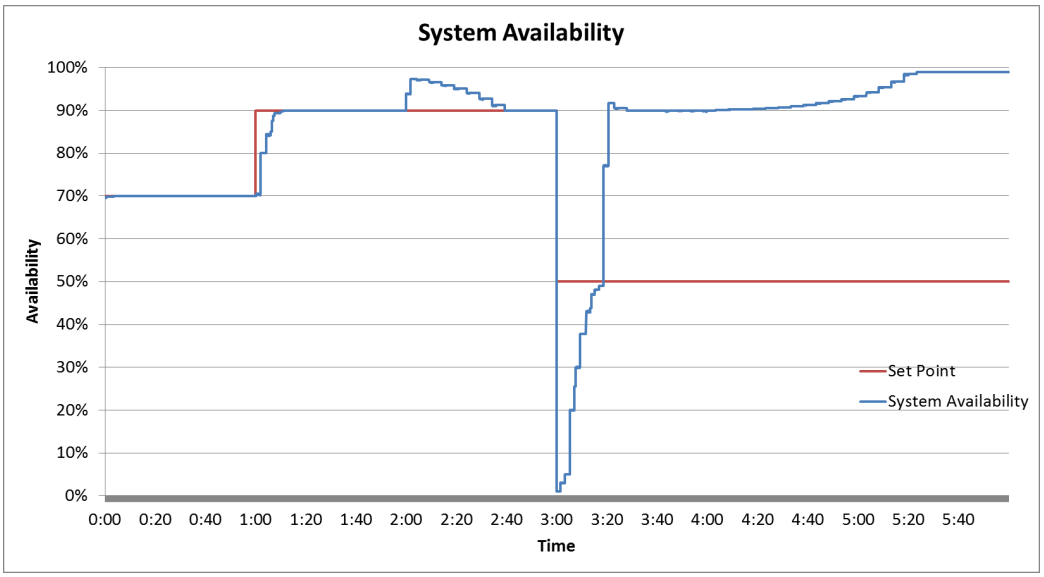


Figure 6.21: Availability of the controlled system

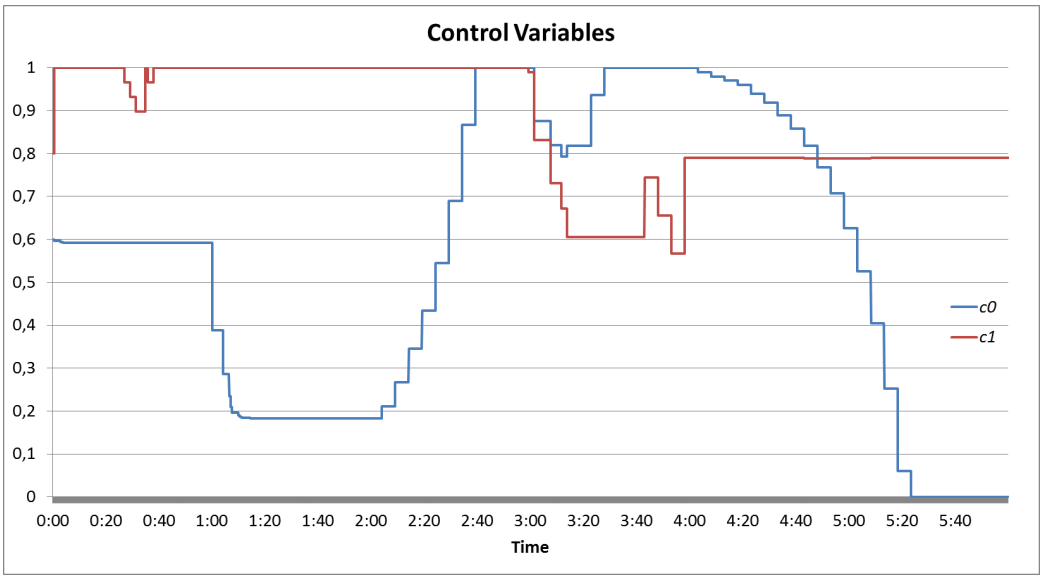


Figure 6.22: Control variables values

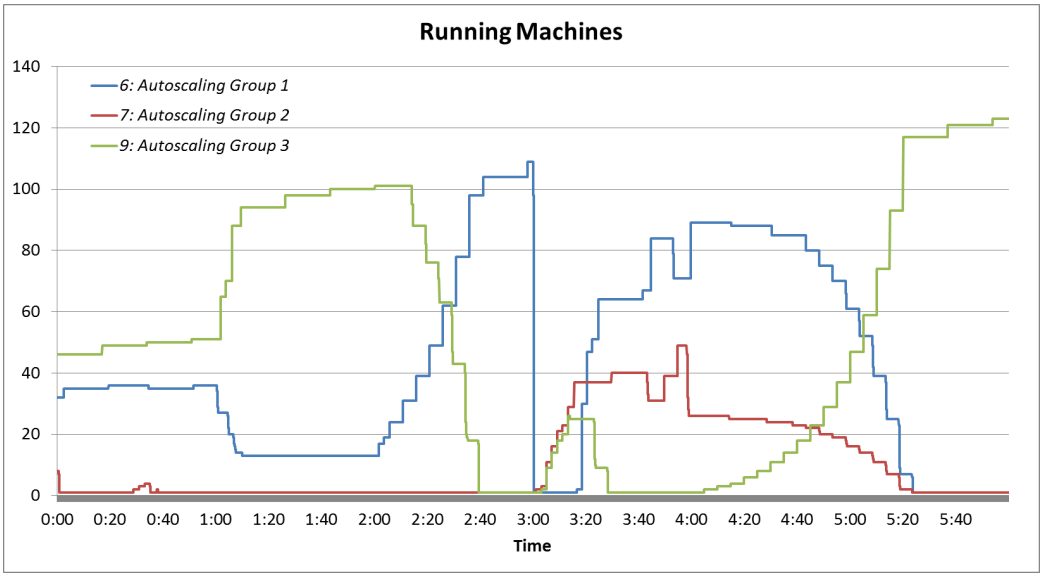


Figure 6.23: Number of running machines

	Cloud 1 (R1)	Cloud 1 (R2)	Cloud 2	Controlled
C0	1	1	0	Controlled
C1	1	0	–	Controlled
Availability	71.69%	76.50%	99%	85.22%
Cost	232.96$	245.75$	338.71$	276.41$

Table 6.6: Controlled vs non controlled results

can observe that, besides having satisfied the availability requirement for the entire simulation except for a maximum recovery time of 20 minutes after region 1 outage and sudden changes to the set point (Figure 6.21), we also kept costs low, avoiding to use only the expensive cloud 2 for most of the time, until if becomes more convenient for the raising of costs of cloud 1.

6.3 A Smart City Scenario

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

of the emergency including traffic light control to evacuate certain zones and clear path for rescuer squads.

Being a critical application the first and most important requirement is availability. In a deeply automated environment of a smart city the main response to emergency is given via its IT infrastructure, a failure in dealing with an emergency could result in severe damage to the city itself or even cause death.

Embedded sensors in buildings, on streets and on vehicles are already a reality. Once all these sensors are connected to the Internet the amount of data provided will be tremendous. Dealing with such a huge number of sensors involves processing of raw data on the order of TB/s that can vary over time of the day. This huge amount of data has to be cleaned from noise and aggregated. In order to process such a huge amount of data the infrastructure should be scalable. The last requirement of this application, quite obvious and popular this day, is to minimize costs of the IT infrastructure.

In order to fulfill these requirements the most reasonable choice these days is to exploit resources offered from cloud computing providers. A cloud platform like Amazon Web Services or Windows Azure can cope with the second requirement quite well and also help to reduce costs but can not guarantee the availability we wish to have.

The availability goal of our application is or five nines, which means that we wish that our application runs for 99.999% of the time. If we consider a general provider with an availability of 95%, as shown in [3], we should use at least four different providers of that kind, assuming that provider failures are independent of eachother. In fact the probability that n independent cloud providers with an average availability of 95% fail simoultaneously is 0.05^n. We can calculate the minimum number of n needed to fulfill the five nine requirement as shown:

$$0.05^n < 0.00001$$

$$n > \log_0 .05(0.00001)$$

$$n > 3.85$$

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

6.3.1 Application Model

The application is divided into three main layers. The first layer which take cares of collecting data from sensors, filtering, noise reduction and aggregation. The second layer receives aggregated data and update the process model which describes the dynamic state of the city. The third layer contains the reasoning module which is responsible of finding the best response

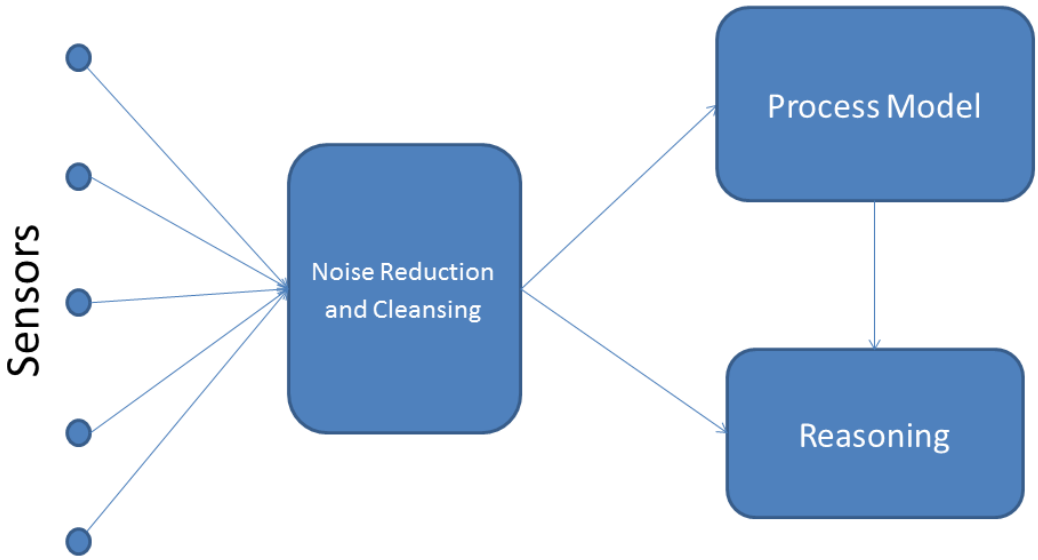


Figure 6.24: Structure of the application

to emergencies. In order to work properly this layer needs access to more information than the one deriving from aggregated data so it could instruct the first layer to reduce its aggregation policy or even to let some raw data pass directly to the reasoner. This structure is shown in Figure 6.24.

In order to adapt the behavior of the application to the environment conditions we thought of adding a middleware responsible for monitoring the healthiness of the application and taking decisions about design adaptation. We first focused on the filtering part since it's the most computationally intensive and, consequently, the one with greater impact on costs.

6.3.2 Filtering Part

Given the huge amount of data to be processed it's unreasonable to send it to all providers and make machines work redundantly to provide higher availability. The DTMC model of the system is shown in Figure 6.25. Nodes 4, 5, 6 and 7 represent the entry point of the four cloud providers. Green nodes attached to these nodes represent the auto scaling group of VMs that process requests in order to filter and clean data. Red nodes represent possible failures. Nodes with incoming arches labeled *1-a4*, *1-a5*, *1-a6* and *1-a7* represent the availability of each cloud provider. For example if we say that cloud provider 1 has an average availability of 95%, as in [3]. It means that the average proability of each request of going into state 8 is of 95%.

If we look again at cloud 1 we see two distinct failure states are repre-

sented. The first one receive requests that fail due reasons that affect the cloud provider infrastructure. The other one, with incoming arch *1-a8*, receive requests that fail because of limited performance of the autoscaling group. All failure states of Figure 6.25 have the same number and label because they are actually mapped to a single state in the DTMC which has been replicated here for clarity.

Nodes 1, 2 and 3 represent load balancing nodes whose probabilities are defined by the first layer controller of Section 4.3.

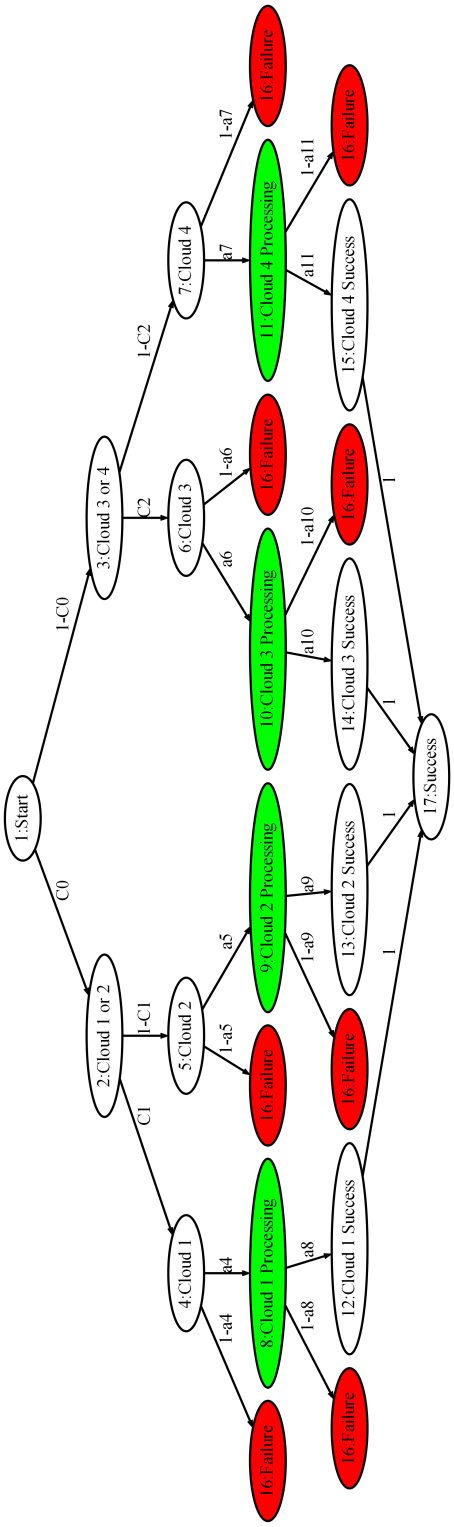


Figure 6.25: DTMC model of the filtering part of the smart city usecase

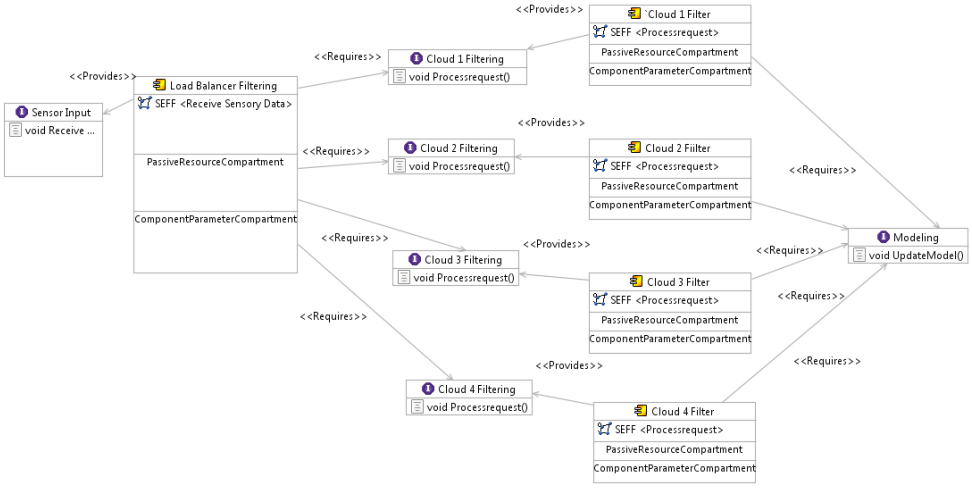
6.3.3 Process Model

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

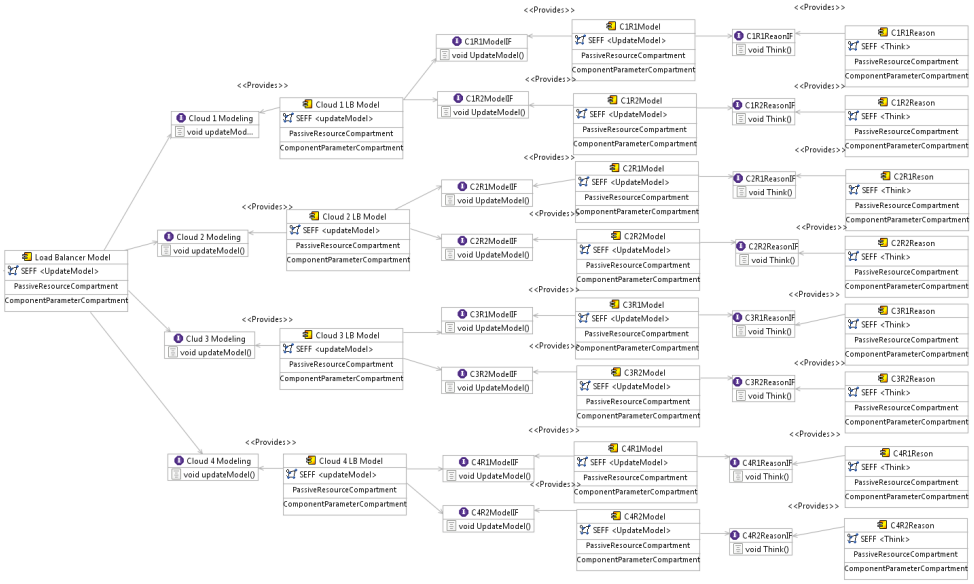
Reasoning

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

Figure 6.26 shows the entire model of the application. Since we are interested in controlling only the first layer the DTMC of Figure 6.25 is derived from the part of the model in Figure 6.26(a).



(a)



(b)

Figure 6.26: Palladio model of the smart city emergency system

	Cloud 1	**Cloud 2**	**Cloud 3**	**Cloud 4**
Cost per VM	0.35 $\frac{\$}{\text{hr}}$	0.40 $\frac{\$}{\text{hr}}$	0.60 $\frac{\$}{\text{hr}}$	0.55 $\frac{\$}{\text{hr}}$
VM limit	None	None	< 200	< 150
VM startup time	50 s	50 s	100 s	100 s
VM nominal SR	$10,000$ $\frac{\text{reqs}}{\text{s}}$	$10,000$ $\frac{\text{reqs}}{\text{s}}$	$20,000$ $\frac{\text{reqs}}{\text{s}}$	$20,000$ $\frac{\text{reqs}}{\text{s}}$
CPU set point	80%	80%	80%	80%
CPU tolerance	10%	10%	10%	10%
Nominal CpR	4.37E-5 $\frac{\$}{\text{req}}$	5.00E-5 $\frac{\$}{\text{req}}$	3.75E-5 $\frac{\$}{\text{req}}$	3.44E-5 $\frac{\$}{\text{req}}$

Table 6.7: Simulation parameters

The parameters used for simulating the scenario are reported in Table 6.7. We can observe that VMs offered by cloud providers 1 and 2 have the same performances and similar costs. Cloud providers 3 and 4 offers more performant VMs at higher costs. Though, as shown by the nominal cost per request (CpR), Cloud 1 and 2 offer more convenient machines.

The scenario simulates the usage of the system in a typical 24 hour period, the arrival rate is composed by a bimodal distribution shown in Figure 6.27 with two peaks at time 10:00 and 19:00. Maximum service rates for cloud providers 1, 2 and 3 are kept constat to their nominal values while cloud 4 experiences a degradation of its service rate between time 13:00 and 17:00 as shown in Figure 6.28. Also the availability of cloud providers are changed in order to simulate different failure scenarios. In particular the availability of cloud 2, the most expensive one, is constant at 100%. Cloud 4 experience a total downtime between time 8:00 and 13:00 that could be caused by the lacking of connectivity the cloud provider. Cloud 1 starts with availability of 95% which is not enough to satisfy the 5 nines availability constraint but from time 10:00 on its availability increases to 100% as shown in Figure 6.29 This scenario could happen if the workload of other users of the cloud decrease and its overall architecture has a lighter load so the availability increases. The availability of cloud 3 is shown in Figure 6.30. It starts from 100% and decreases to 95% between time 10:00 and 15:00.

Figure 6.31 shows the utilization of the cpu in the eventuality of using only cloud 2 to support the system without the controller. Figure 6.32 shows the number of running machines running on cloud 2 in the non controlled system. We can observe that the initial CPU load is above the desired value so the autoscaling group controller increases the number of running machines until the cpu is near 80%. When the peak of requests is reached, the cpu utilization increases and oscillates near the maximum allowed cpu because of the scaling effect of the controller. The same behavior can be observed

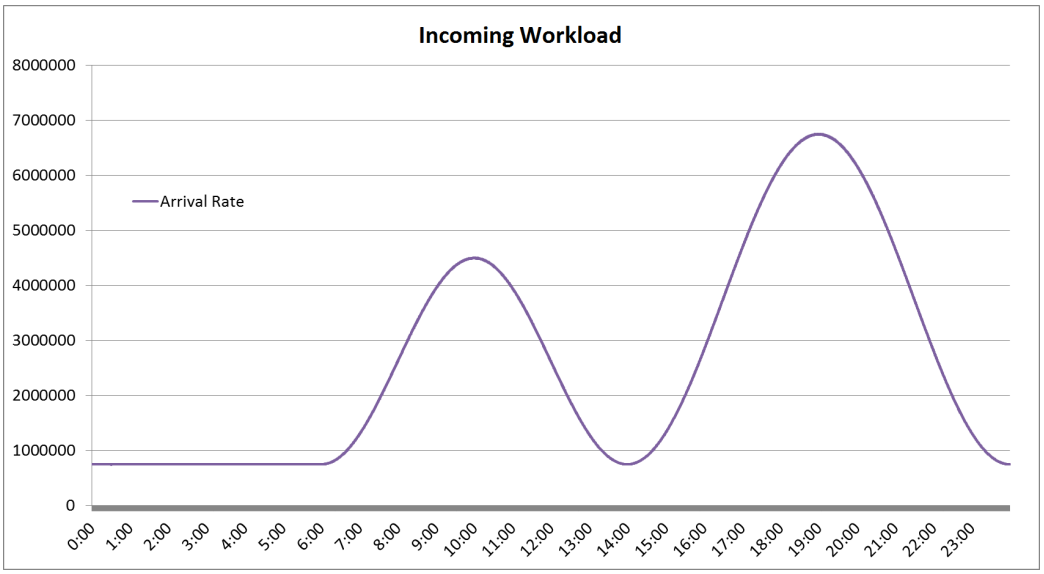


Figure 6.27: Bimodal requests arrival rate

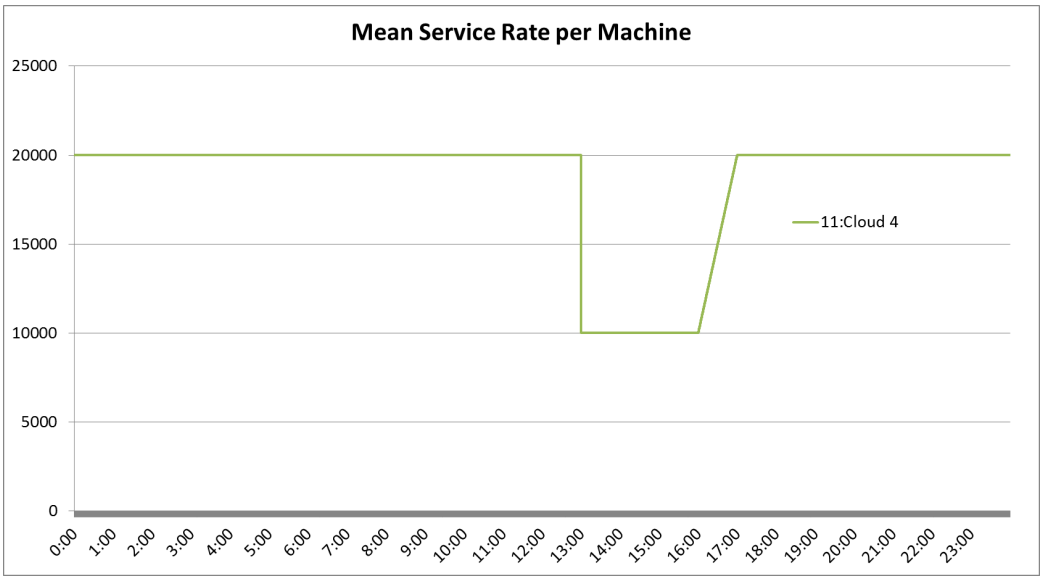


Figure 6.28: Cloud 4 service rate

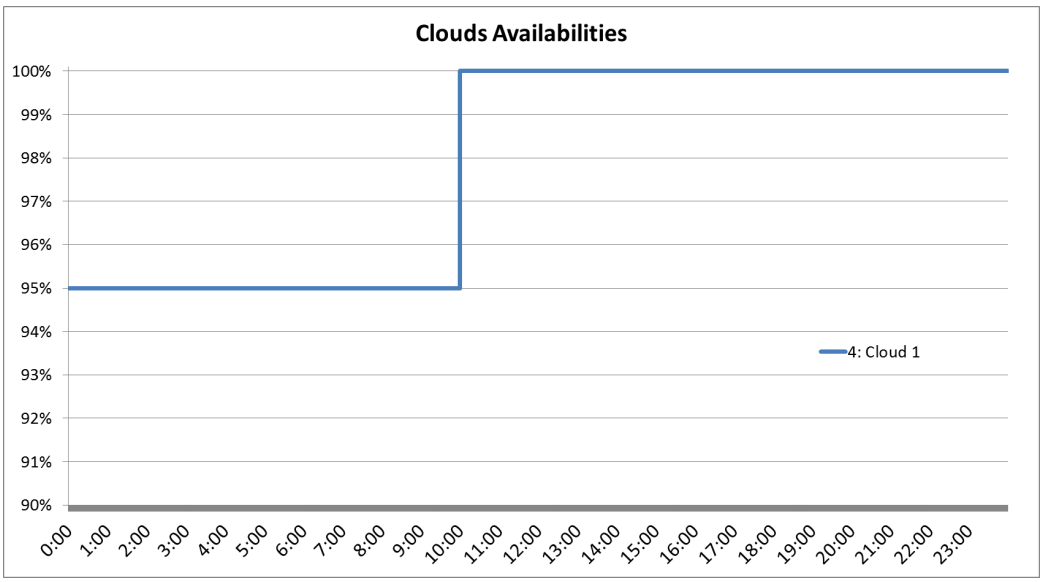


Figure 6.29: Cloud 1 availability

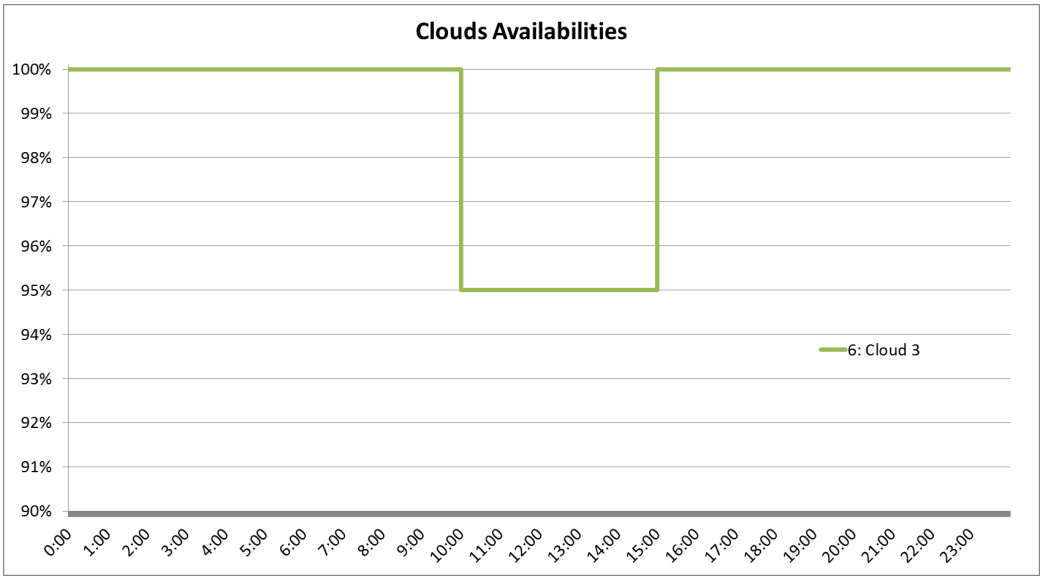


Figure 6.30: Cloud 3 availability

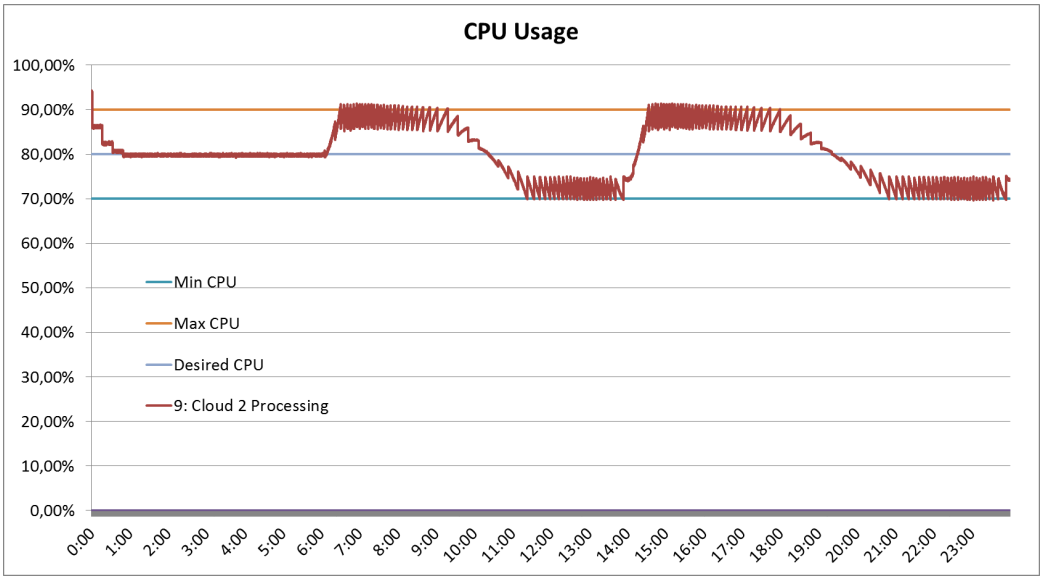


Figure 6.31: Cpu utilizaion of cloud 2

when the number of requests decreases and the cpu usage oscillates near the minimum allowed value because of the scale down actions of the controller.

If one decides to deploy his application only on top of cloud 3 he would observe the variation of the availability shown in Figure 6.33. Cloud 3 offers the cheapest price per request but, as shown in Figure 6.30, its availability decreases to 95% for some time. Cloud provider 3 also has an upper limit to the number of machine it offers. The loss of availbility can be detected by the loss of availaility of the entire system. At time 10:00 the system availability drops under 90% because even if the controller scales up the number of machines, when it reaches the machines upper limit, its system is overloaded and start rejecting requests. When the number of requests decreases, the system availability increases but it reaches only the value of 95% because of the limited availability of the cloud provider. At 15:00 the availability of the cloud provider is increased back to its initial value of 100% and the system availablity also follows this behavior. When the second, higher, peak of requests enter the system the maximum number of available machines is again the limitng factor of the availability of the system. These behaviors can are reported also in Figures 6.34 that shows the cpu usage and Figure 6.35 that shows the number of running machines.

If the system administrator decides to deploy its application only on cloud 4 he would see the availability shown in Figure 6.36. The large availability degradation that occurs between 8:00 and 13:00 is due to the loss of availability of the cloud in that time interval that could be caused, for example by

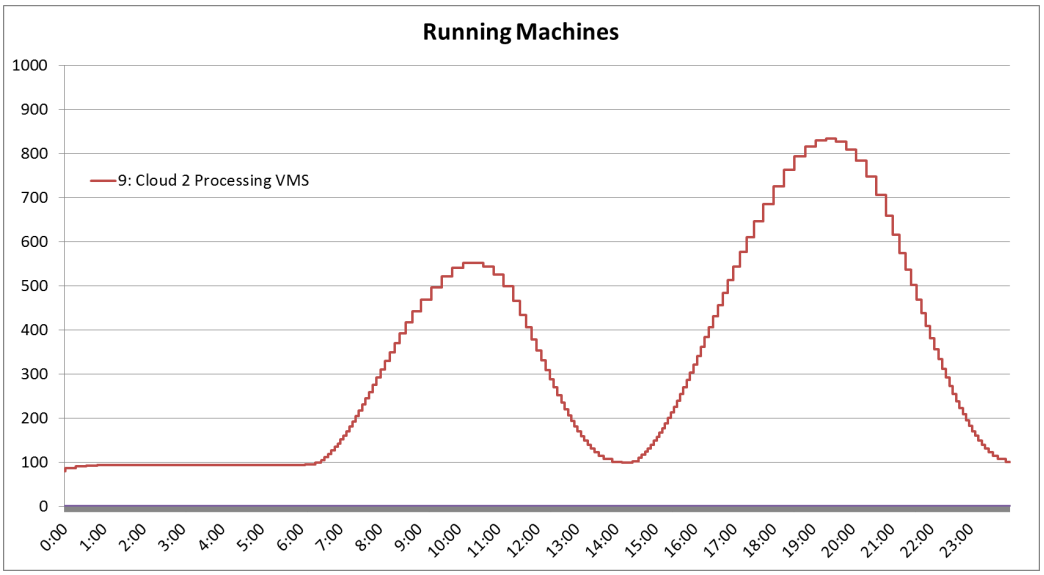


Figure 6.32: Number of running VM in cloud 2

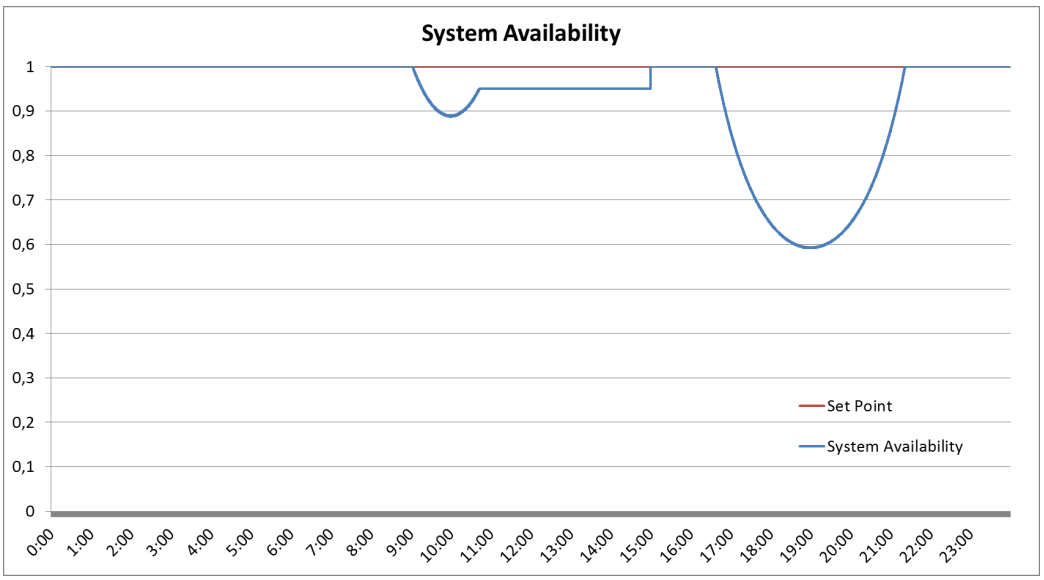


Figure 6.33: System avalability using only cloud 3

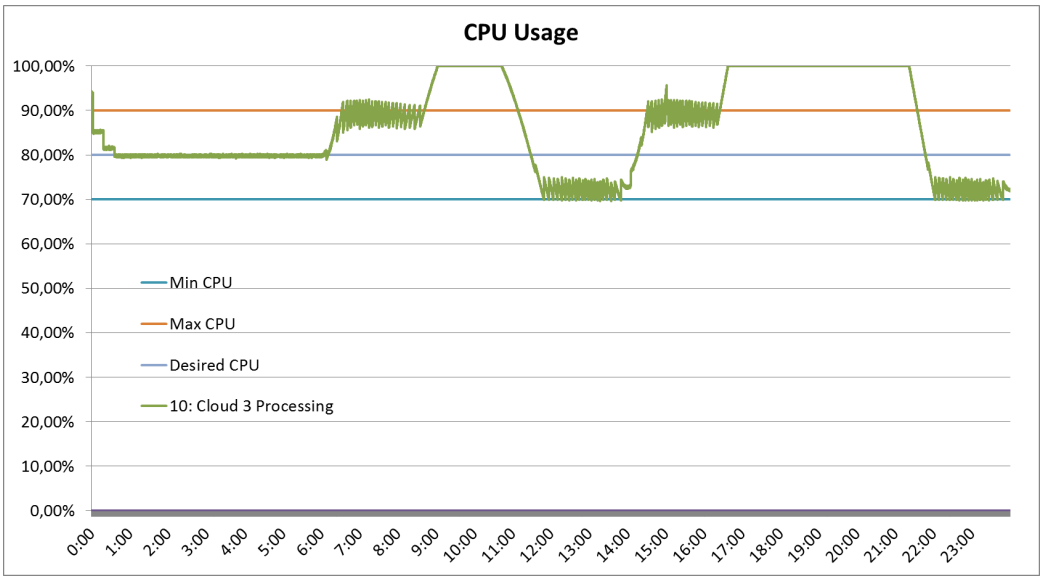


Figure 6.34: Cpu usage of machine usng only cloud 3

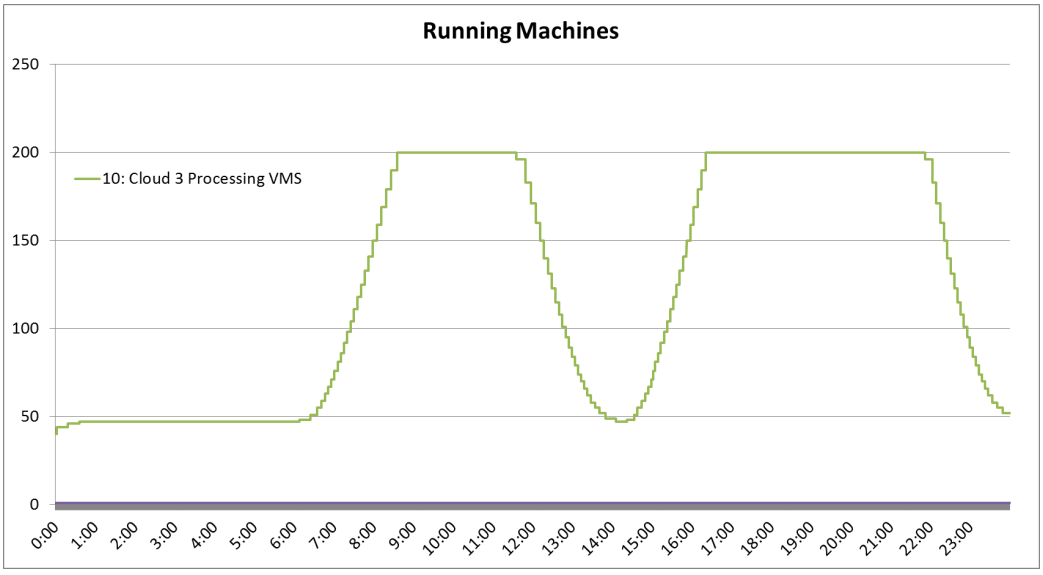


Figure 6.35: Number of VMs using only cloud 3

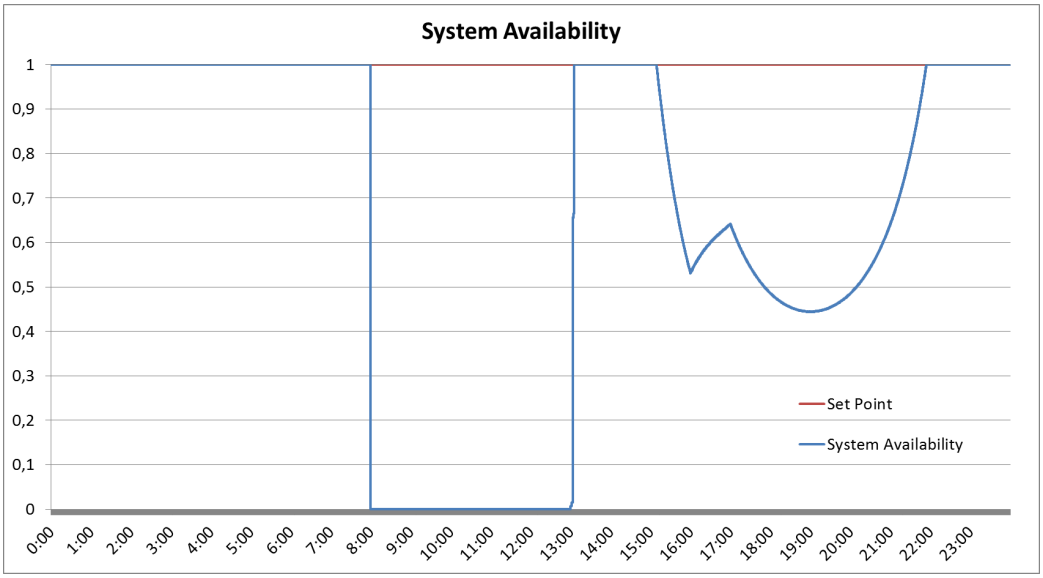


Figure 6.36: Availability of the system using only cloud 4

the impossibility of reaching that cloud provider due to networking reasons. The second loss of availability is due to the fact that, as shown in Figure 6.28, that the service rate of machines of cloud 4 degrades in that time interval and that the maximum number of machines that the cloud offer is not enough to serve all requests.

We now present the result of the simulations in which the controller is active and decides how to forward requests to clouds. The scenarios we present here have been simulated using all cloud behaviors introduced in previous examples of this section.

For this first scenario the set point has been set to 5 nines as required by the application. The availability of the controlled system is shown in Figure 6.37. We can observe that the system reaches the desired availability most of the time excepts for very short time intervals in which sudden cloud failures makes it decrease until the controller reroute some requests to other clouds. The utilization of the fourth cloud providers can be observed in Figure 6.39 that shows the value of control variables and Figure 6.38 that shows the number of running machines for each cloud provider.

We can observe that the controller uses only cloud 4, which is the cheapest one, until it fails at time 8:00. Then, the controller uses cloud 3 until the maximum number of available machines is reached. Since the workload keeps growing, the controller decides to use also cloud 2 (time 10:00). At the same time the availability of cloud 3 degrades to a point that the controller decides to switch some of its requests to cloud 1 and shut down cloud 3. Since from

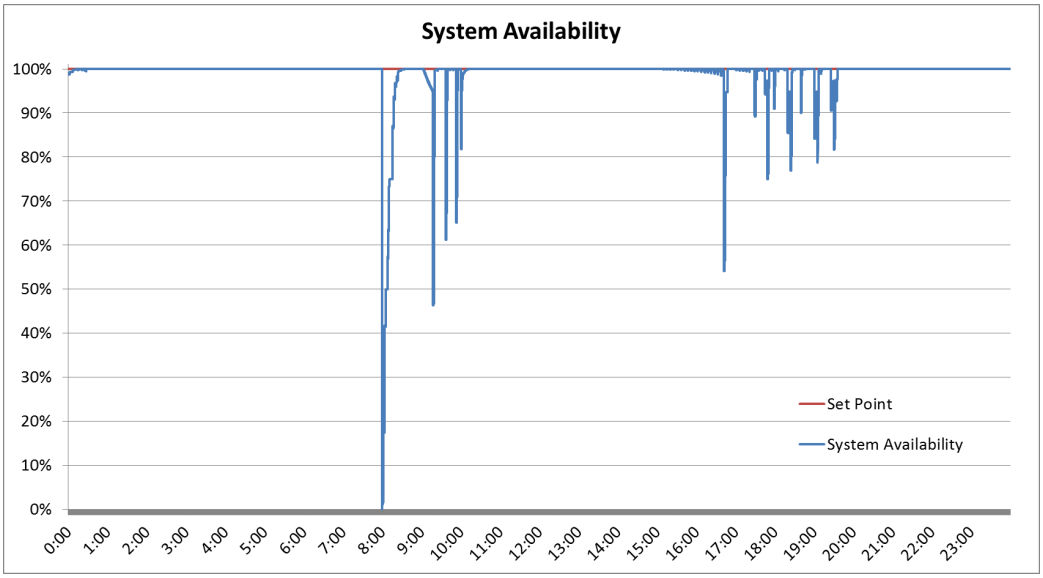


Figure 6.37: Availability of the controlled system with set point to 5-nines

10:00 the availability of cloud 1 is enough to fulfill the availability requirement and it is cheaper than cloud 2, the controller shuts down cloud 2 and redirects all requests to cloud 1. When the availability of cloud 3 and cloud 4 are set back to 100% the controller uses both clouds instantiating their maximum number of available machines and using clouds 1 and 2 only if the workload is too high for both clouds 3 and 4 to serve.

Figure 6.39 shows that the values of control variables oscillates in some situations. This is due to the fact that the controller tries to overload cheapest clouds that in this case have a fixed number of machines. The controller takes into account the fact that some cloud providers have a limited number of machines by positive term in the cost function. This effect could be avoided by adding a constraint to the controller to avoid it overloading limited clouds. Since the 5-nines constraint is very restrictive it is hard for the controller to minimize the cost function and the controlled variables values oscillate.

We have shown that the controlled solution gives better results with respect to most of the single cloud solutions in term of availability since it is capable of switching between clouds when the availability value decreases. The only cloud that performed better is cloud 2 since it did not occurred in degradation of any of its parameters during the simulation, the good behavior of cloud 2 is compensated by its very high cost.

The overall results of the simulation are shown in Table 6.8. This Table shows that the controlled system does not only overcome non controlled solutions in terms of availability but also in term of costs.

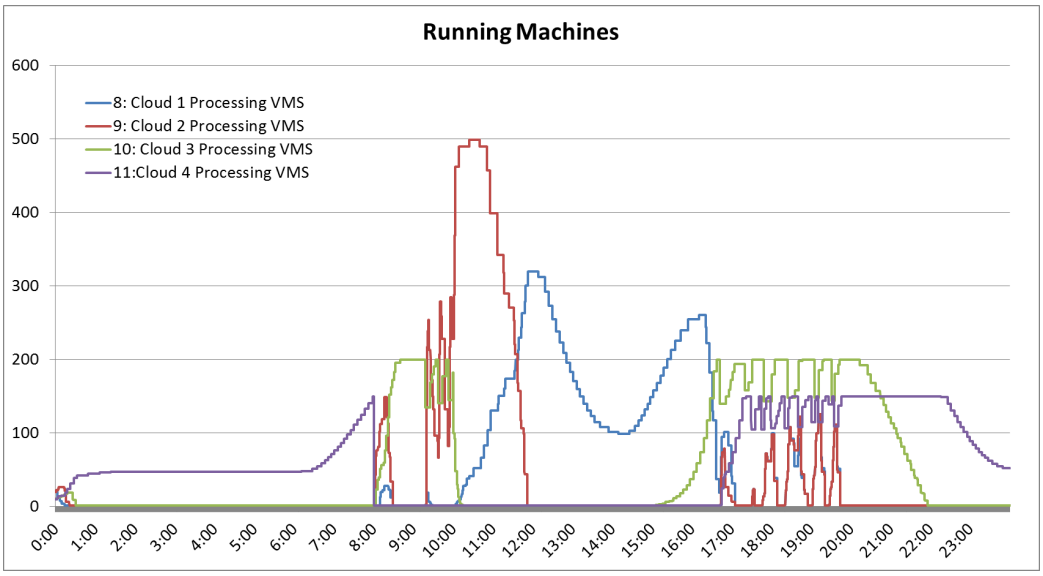


Figure 6.38: Number of running VMs for the controleld system with set point at 5-nines

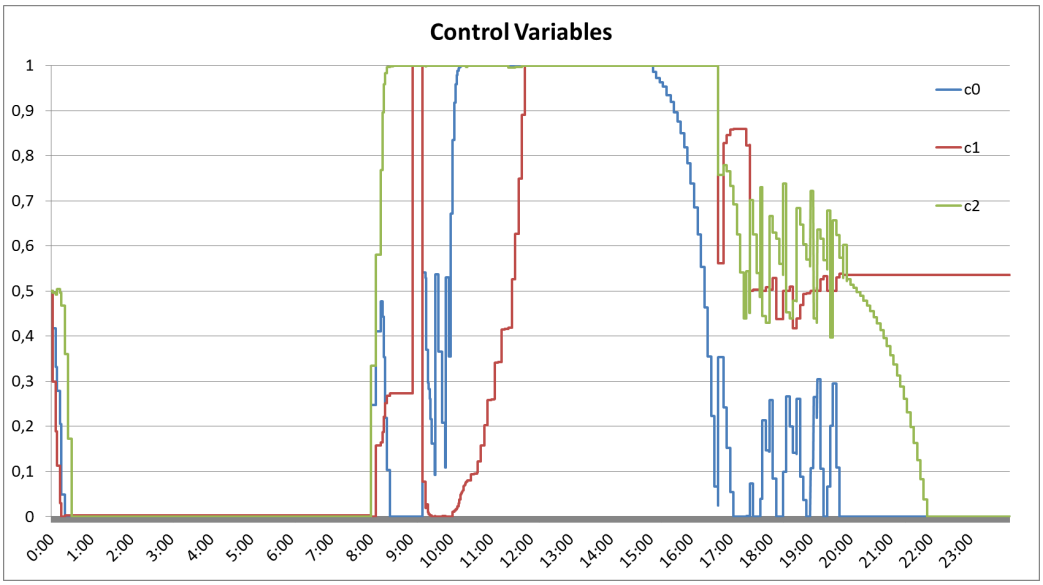


Figure 6.39: Control variables values for the controlled system with set point at 5-nines

	Cloud 1	Cloud 2	Cloud 3	Cloud 4	Controlled sp = 5 nines	Controlled sp = 99%	Controlled sp = 95%
C0	1	1	0	0	Controlled	Controlled	Controlled
C1	1	0	*	*	Controlled	Controlled	Controlled
C2	*	*	1	0	Controlled	Controlled	Controlled
Availability	98.81%	100%	85.16%	48.96%	98.24%	98.74%	97.70%
Cost	2,740.57$	3,164.28$	1,814.60$	1,088.18$	2,317.99$	2,189.66$	2,065.21$

Table 6.8: Smart city scenario results

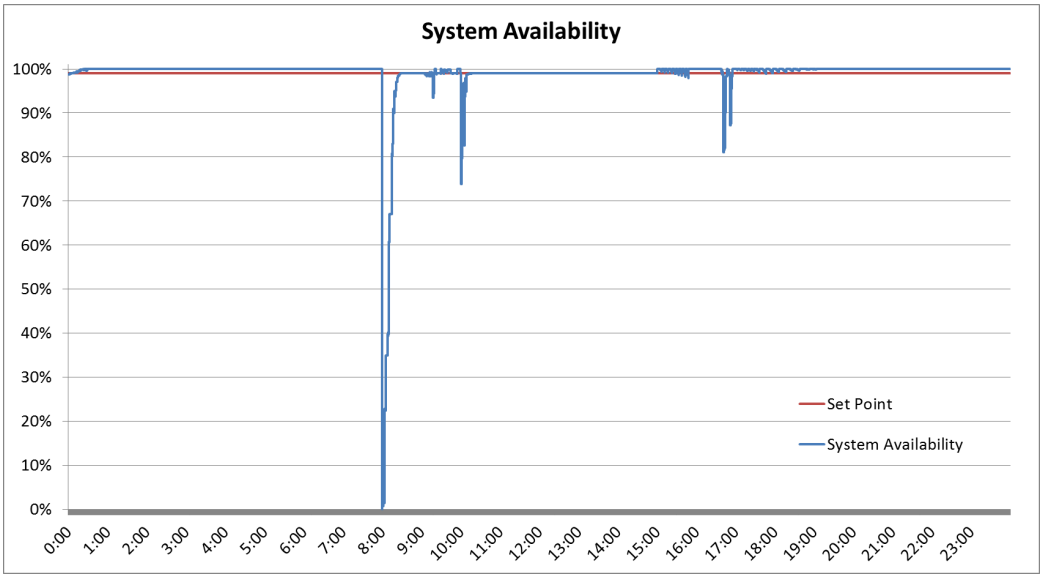


Figure 6.40: Availability of the controlled system with set point at 99%

In the second scenario we relaxed the availability requirement of the application by moving the set point of the desired availability to 99%. Figure 6.40 shows the availability of the system that is quite similar to the one of the previous scenario but less subject to failures. Figure 6.41 and Figure 6.42 shows the behavior of the controller. It is quite similar to one of the prevoius example but it differs from the fact that at time 10:00 the controller chooses to use cloud 1 instead of cloud 2. This Figure also shows the fact that the controller is less subject to oscillations. The coiche of cloud 2 in the previous example was due to the fact that at time 10:00 it was being overloaded by requests and scaling up.

In the last scenario we relaxed even further the availability constraint by setting it to 95%. Figure 6.43 shows that the system availability drops under the set point only for a very short period of time and most of the time stays over the desired value. Figure 6.44 show the number of running machines in the system and Figre 6.45 shows the value of control variables.

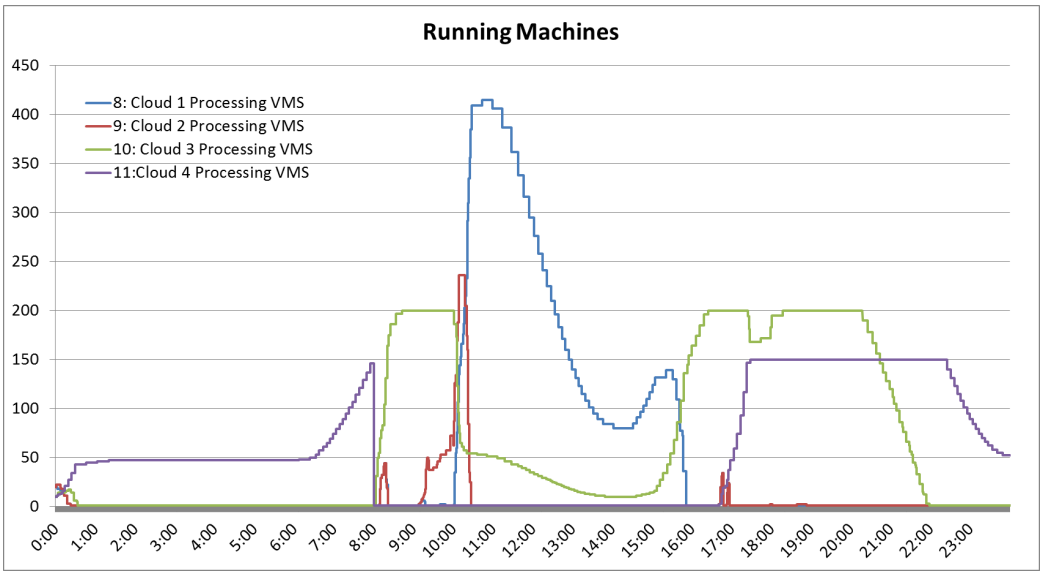


Figure 6.41: Number of running machines for the controlled system with set point at 99%

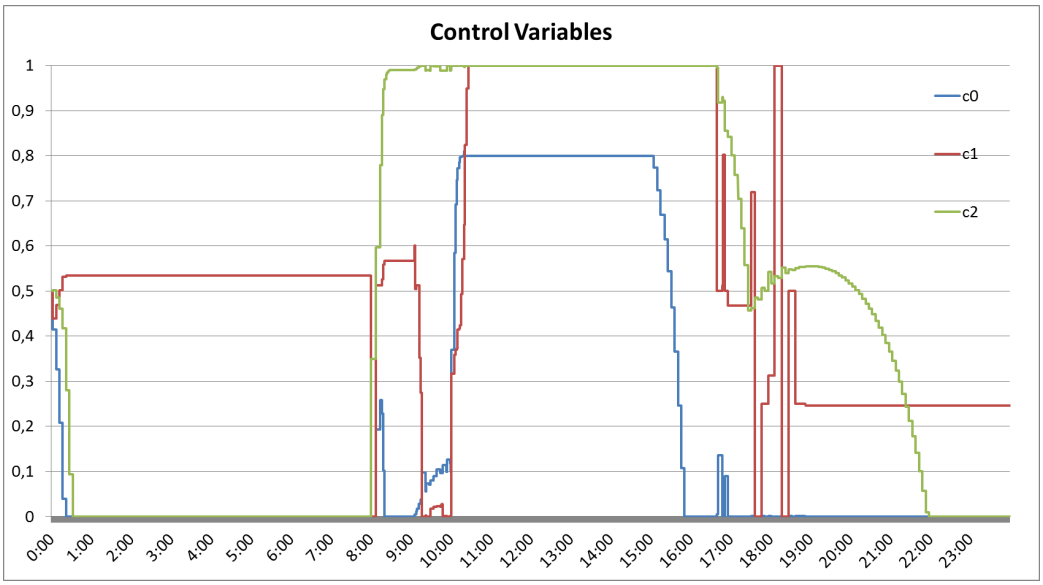


Figure 6.42: Control variables vaules for the controlled system with set point at 99%

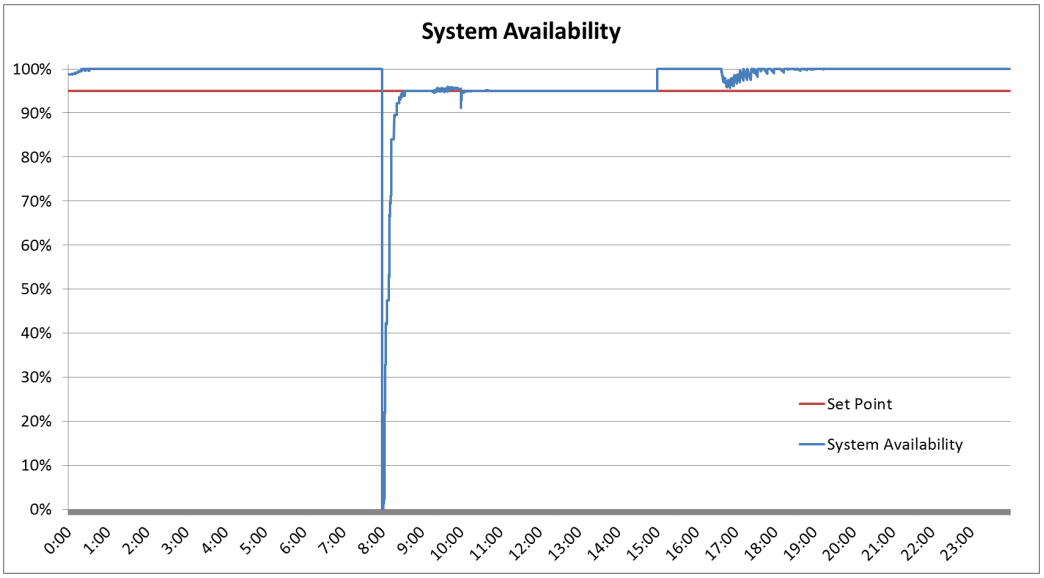


Figure 6.43: Availability of the controlled system with set point at 95%

The main difference between this behavior and the previous one is that the 95% constraint can be fulfilled even when the availability of cloud provider 3 experience degradation. This fact implies that cloud 1 and 2, most expensive ones, are used only to react fast to the loss of service at time 10 and when the capacity of cloud 3 and 4 is not enough to process the workload.

6.4 Results analysis

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

- The controller changes gradually the distribution of requests among cloud provider for *economical reasons*
- The controller reacts to *failure* or *degradation* of cloud performances and modify the behavior of the application to restore availability

The first kind of change can be observed in Figure 6.4 where the controller shuts down cloud 2 and redirects all traffic to cloud 1. Looking at Figure 6.3

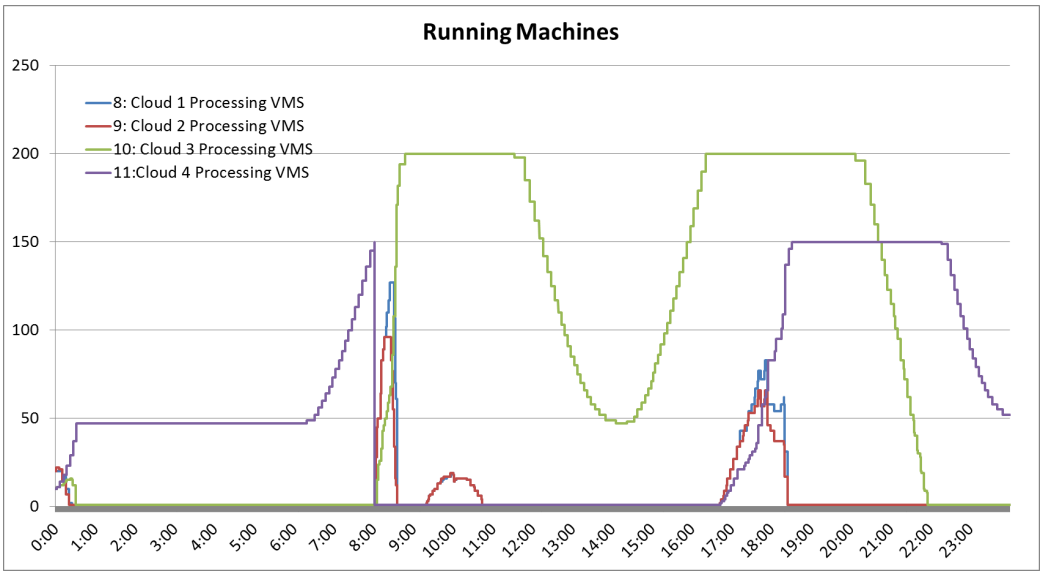


Figure 6.44: Number of running VMs of the controlled system with set point at 95%

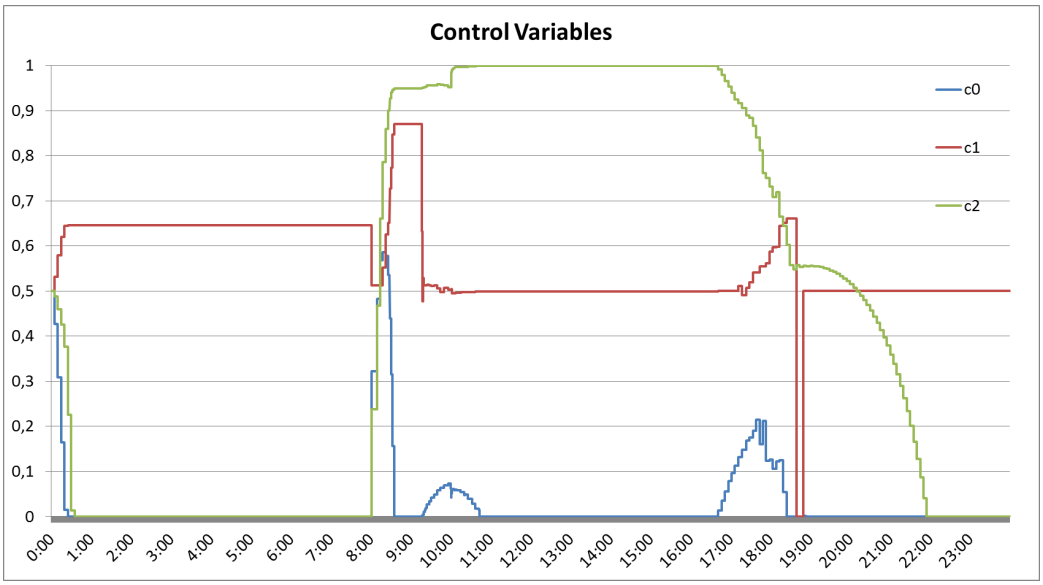


Figure 6.45: Control variables values for the controlled system with set point at 95%

we see that the availability of the system is not affected by this action of the controller.

The second kind of changes occurs in many simulated scenarios. The most challenging situation of this kind is the one shown in Figure 6.17 where a cloud provider experience a sudden complete outage. If, like in Section 6.2, the system was using that cloud to process requests it experiences a sudden degradation of its availability. In this case the controller redirects all traffic going to that cloud to both the other two cloud providers in order to make them scale and restore the computing capacity, as can be observe in Figure 6.23 between time 3:00 and 3:20. This sudden failure on a cloud provider that the system is using is the most difficult scenario in which to satisfy the availability requirement. The maximum time needed to bring back the availability of the system to the set point in this scenario can be found by applying the equation used for error convergence in [1]. Assuming that the system has converged when $e(k) \leq \epsilon$, then this happes when

$$k \geq \log_{\beta} \frac{\epsilon}{e(0))} \tag{6.3}$$

where $e(0)$ is the initial error. By setting the working conditions at time 3:00, that are, initial error $e(0) = 0.5$, $\beta = 0.3$ and setting $\epsilon = 0.01$ as the convergence tolerance, we obtain:

$$k \geq \log_{0.3} \frac{0.01}{0.5} \tag{6.4}$$

$$k \geq 5.64 \tag{6.5}$$

So we need 6 control steps of the load balancer controller. After each step of the load balancer it is inhibited until all autoscaling groups have stabilized their cpu usage within the predefined boundaries, that is, in this case, 70% - 90%. According to Section 4.3 the maximum number of steps needed to reach this value is 3. In our example machines take 100 seconds to boot up so the total time needed to restore the availability is given by: $3 \times 100s \times 6$ so 1800 seconds that are 30 minutes. This value overestimates the time needed for the controller to converge, since the function used in the reduction of error at each control step says that the error of the next step should be less then or equal to $\beta \cdot \textit{current error}$ so in some steps the error could be reduced by more than β. Also, as stated in Section 4.3, convergence depends on the initial number of active machines, the initial availability, and the accuracy of the estimated parameters of the model.

Other scenarios in which changes occurs more gradually, like the one in Section 6.1.2 in which the machine service rate degrades smoothly, are

handled by the controller in such a way that the system does not suffer loss of availability.

Chapter 7

Conclusions

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

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 its service rate inside the model.

Then, we defined a two layer controller to manage both the autoscaling policy of single nodes and the load balancing at run-time, reasoning on the defined models, which is kept alive and continuously updated at run-time. The layer dealing with the autoscaling is responsible of ensuring that the number of machines are enough to cope with the incoming workload, maintaining a user defined cpu usage desired level. The second layer is instead responsible of distributing the incoming workload so to keep the availability of the entire system over a user defined threshold, minimizing costs. This models start form the work in [1] that has been deeply modified in order to fit in the particular environment of cloud computing. We also expanded the controlling approach by adding costs and other kind of constraints specific to the cloud domain to the model, as suggested by authors in the future work section of the article.

Finally, we extended the already existing modeling Palladio Bench to allow developers to model their Multi-Cloud application and simulate different scenarios to test the availability and cost requirements, both while being monitored and controlled by our control system and without using it. We implemented the possibility to simulate different workload conditions, service

rate drop, network and cloud failures.

Results during experimental evaluation, reveal that our approach can indeed be valuable since even when dealing with clouds with average low availabilities, the controller is able to take decisions at run-time and distribute incoming workload to clouds so to cope with the user defined availability requirement and so to minimize costs. It turned out to be a valuable approach even in the case where clouds offer high availability but different (possibly varying) costs, since the controller is able to move the workload to the cheapest one.

Future research will first go through different improvements on the resolution of constraint optimization problems, so to cope with challenges cases like the Smart City Scenario presented in Section 6.3, where the adopted technique had some issues in finding the optimum solution.

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

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

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

Bibliography

[1] Alberto Leva Martina Maggio Antonio Filieri, Carlo Ghezzi. Self-adaptive software meets control theory: A preliminary approach supporting reliability requirements. 2011.

[2] Jie Li Ming Mao and Marty Humphrey. Cloud auto-scaling with deadline and budget constraints.

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

[4] Marty Humphrey Ming Mao. A performance study on the vm startup time in the cloud.

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

[6] http://aws.amazon.com/ec2/.

[7] http://www.rackspace.com/cloud/.

[8] http://www.terremark.com/services/infrastructure-cloud-services/enterprise-cloud.aspx.

[9] http://www.force.com/.

[10] https://developers.google.com/appengine/.

[11] http://www.google.it/intl/it/enterprise/apps/business/.

[12] http://www.netsuite.com/portal/home.shtml.

[13] http://www.freshbooks.com/.

[14] http://it.msn.com/.

[15] http://aws.amazon.com/s3/.

[16] Laprie J. C. Randell B. Landwehr C. Avizienis, A. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. on Dependable and Secure Computing*, 2004.

[17] Christel Baier and Joost-Pieter Katoen. *Principle of Model Checking*. April 2008.

[18] Dana Petcu. Portability and interoperability between clouds: Challenges and case study.

[19] NIST CCSRWG. Cloud computing standards roadmap.

[20] http://www.rackspace.com/cloud/public/servers/compare/.

[21] http://www.akamai.com/.

[22] Ralf Reussner a Steffen Becker, Heiko Koziolek. The palladio component model for model-driven performance prediction.

[23] Luciano Baresi and Carlo Ghezzi. The disappearing boundary between development-time and run-time. *FSE/SDP workshop on Future of software engineering research*, pages 17–22, 2010.

[24] Jussara M. Almeida, Virgilio A. F. Almeida, Danilo Ardagna, Italo S. Cunha, Chiara Francalanci, and Marco Trubian. Joint admission control and resource allocation in virtualized servers. *J. Parallel Distrib. Comput.*, 2010.