FEDERAL UNIVERSITY OF PERNAMBUCO
CENTER OF INFORMATICS

BACHELOR IN COMPUTER ENGINEERING

FIDEP - A Fault Injection Framework for Dependability Analysis on Cloud Computing Systems

Vandi Alves de Lira Neto

Bachelor's Dissertation

Recife
December 2016

FEDERAL UNIVERSITY OF PERNAMBUCO
CENTER OF INFORMATICS

Vandi Alves de Lira Neto

FIDEP - A Fault Injection Framework for Dependability Analysis on Cloud Computing Systems

Dissertation presented to the Program of Graduation in Computer Engineering, Center of Informatics, Federal University of Pernambuco, as a partial requirement for the achievement of Bachelor's degree in Computer Engineering.

Advisor: *Prof. Dr. Paulo Romero Martins Maciel*

Recife
December 2016

I devote this work to my family. My father Vandi, my mother Joselia, my sister Nayane. And especially to Marcele for all the support and comprehension that she showed me, during all this time.

Acknowledgements

I thank God for his grace. Also thanks to my colleagues in UFPE. In particular my advisor Paulo, thank you so much for all the guidance and patience.

It is not knowledge, but the act of learning, not possession but the act of getting there, which grants the greatest enjoyment

—CARL FRIEDRICH GAUSS

Abstract

In the last years, cloud-based infrastructures represent noticeable choices to host diverse applications ranging from social networks to scientific computing. A prominent cloud service corresponds to IaaS (Infrastructure as a Service), which delivers computing resources through virtual machines (VMs). Important IaaS providers apply service level agreements (SLA) to define the quality of service levels and penalties related to service outages. In this context, IaaS providers need to guarantee the contracted availability as defined in the SLA clauses. This work proposes a fault injection framework for cloud computing systems in order to support dependability studies combined with service monitoring; The framework is specifically designed for developers who are implementing fault injection tools for Linux-based cloud computing systems. It is also presented an experimentation of the proposed framework for the target system Eucalyptus Cloud System. It is proposed a Reliability Block Diagram (RBD) model to the adopted infrastructure, the model is generic enough to be extended for other arrangements. Results show that the availability simulated with the RBD model is within the confidence interval of the experimental availability calculated with the support of the fault injection tool. It is also presented another case study that applies a Stochastic Petri Net (SPN) model to represent the infrastructure and to estimate the availability of the service provided by Eucalyptus, the results also presented coherence, due to the availability estimated by the SPN model falls into the experimental confidence interval of availability.

Keywords: Fault Injection Framework, Cloud Computing, Infrastructure as a Service, Dependability Evaluation, Eucalyptus Cloud

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

This chapter provides a brief explanation of cloud computing systems and also of the dependability concept. It highlights the importance of the dependability evaluation in order to develop more reliable systems. Then it presents the motivation of this work, as well as the objectives and expected contributions. Finally, the related works are disclosed and then it is exposed the remaining structure of the presented dissertation.

1.1 Context and Motivation

The evolution of computer systems is the result of advances in research that aims to provide to the users new easy to use systems, diversified systems, reliable systems, low-cost systems and high-performance systems. For a system to meet such characteristics its developers have to be careful and attentive since the earliest stages of the development process until the system's implementation.

Dependability is a concept that includes attributes such as availability, reliability, operational safety, security, maintainability, testability and performability [54]. The use of some of these attributes in studies contributes to building systems that are justifiably more reliable.

A priority for the current systems is the assurance that the services provided are extremely reliable; Such assurance has become a key requirement for the new systems being developed. Thus, dependability assessment studies can and should be applied to obtain estimates of the behavior of the systems in production.

One way to conduct a system's dependability study is through the workload action. The use of a workload contributes to reveal any previously not noticed flaws that may exist in the system's components. Another way to conduct a dependability evaluation is through fault injection; The fault is injected and then the system's behavior is observed.

Fault injection tools can be applied in order to know the dependability levels of services provided under different scenarios; But a software designed to test a particular system, may not necessarily be used to act on a different system. For the designing of such tools it is necessary to implement a set of methods for the purpose of establishing means for communicating with the target system, for generating and controlling events, or even a way that allows random requests to the system; Random although based on a known statistical distribution. Accomplishing such tasks requires time since it is necessary to encode all the necessary methods, test, correct any errors, and test again.

We are surrounded by services provided on the cloud, thus cloud computing is an ascending paradigm; We use cloud computing more and more every day [5]. Companies known as cloud

providers are companies that provide cloud services to other companies or to final users [39].

Therefore, in such context of cloud computing, the motivation and the need for dependability studies is presented, independently in academic ways, or inside big corporations, because such companies need to provide reliable services.

1.2 Objective and Contributions

This work has as a general objective the implementation of a fault injection, repair and monitoring framework - the framework is called *Fidep*, which stands for Fault Injection framework for Dependability analysis; Such framework is capable of providing a high level interface to support the development of fault injection tools for Linux[57] based cloud computing systems, such as Eucalyptus [12], OpenNebula [55], OpenStack [56], Apache CloudStack [23] and other Linux based cloud computing systems. Besides the framework development, this work also implements a fault injection tool for the Eucalyptus platform [12], and the developed tool uses the proposed framework, *Fidep*, in order to show that the framework indeed makes it easier to implement fault injection tools; Eucalyptus is a cloud computing system that provides the level of Infrastructure as a Service (IaaS). The developed fault injection tool is then used to perform an availability evaluation on the Eucalyptus environment, such study is disclosed in the chapter 4 of this dissertation.

1.3 Related Works

Fault injection tools have been largely adopted for the evaluation of dependable systems. This section presents other works involving fault injection techniques to evaluate the dependability of systems. The injection of failures in systems contributes to evaluating at least two of the dependability properties: availability and reliability. It is also presented works involving availability evaluation on cloud computing systems, such like Eucalyptus Cloud.

[70] describe fault injection techniques; Such work presents five main techniques: (1) Hardware-based fault injection; (2) Software-based fault injection; (3) Simulation-based fault injection; (4) Emulation-based fault injection; (5) Hybrid fault injection. The before-mentioned paper also made some comparisons about the best technique to be used given a particular problem; besides presenting some tools in the literature that fall under each presented technique, description and method of operation, in addition to the strengths and weaknesses of each technique.

A Software-based tool is presented by [41] called WS-FIT (Web Service - Fault Injection), this tool injects faults in the network layer to test systems based on the SOAP protocol via SOAP messages changes when signing and encryption are being used.

[21] demonstrates the use of a hybrid mode of fault injection techniques by using the cooperation between simulation and emulation. This paper presents the FITSEC tool (Fault Injection Tool Based on Simulation and Emulation Cooperation), based on Verilog and VHDL; This tool was developed in order to support the entire system design process.

[68] presents a fault injection tool for real-time embedded systems, called DDSFIS (Debug-

based Dynamic Software Fault Injection System). The authors validated the effectiveness and performance of DDSFIS adopting real world experiments.

In [13] the authors present a benchmark for performance tests in databases for cloud-based systems. The authors defined a set of benchmarks and report results for four widely used databases (e.g., MySQL). Although the proposed benchmark also intends to be a tool for evaluating availability in databases for cloud systems, only performance and elasticity aspects are addressed in the work.

[58] initiate the study of fault injection into Eucalyptus cloud [12] with Eucabomber, which is a fault injection tool specific for the Eucalyptus Cloud System; Despite the important contribution, this version does not consider dependencies between components and also user-oriented metrics are not taken into account.

Other representative works adopt dependability models for evaluating cloud computing environments. In [27] a performability analysis for cloud systems is presented, in which the authors quantify the effects of workload variation, failure rate and system's capacity on service quality. In [4] the authors investigate software aging effects on Eucalyptus framework, and they also propose a strategy to mitigate such issues during the system's execution. [9] describes a system design approach for supporting transparent migration of virtual machines (VMs) by adopting local storage for their persistent state; The approach is transparent to migrated VMs, and it does not interrupt open network connections during VM migration. [64] presents an experiment that quantifies the effect of VM live migrations in the performance of an Internet application. Such study helps data center designers to plan cloud infrastructures satisfying service level agreements.

[15] presents a study on warm-standby mechanisms in Eucalyptus-based private clouds; Results demonstrate that replacing machines by more reliable counterparts would not produce significant improvements in system availability, whereas some fault-tolerance techniques can indeed increase dependability. [34] adopts a modeling approach for virtualized systems, which utilizes fault trees and CTMC. Fault trees are adopted on an upper level and CTMC in a lower level; The work's main contribution involves cloud system evaluation considering VM live migration. In [45] the authors present an evaluation technique to determine the parameters that cause the greatest impact on the availability of mobile cloud systems. The work adopts a combined evaluation of results from three sensitivity analysis techniques which complement each other; Results show that availability can be effectively improved by changing a reduced set of parameters.

The framework presented in this work considers fault injection and repair also in the VM life-cycle operations and dependency between cloud components e.g. the dependency between the Cloud Controller and the Node Controller (in the Eucalyptus environment), and also another dependency example is the dependency between the hardware and the operating system of the infrastructure that hosts the cloud system. The availability results contemplate user perspective, in which the service is available as long as the contracted VMs are operational. Additionally, a Reliability Block Diagram (RBD) model [59] is proposed in the Case Study (chapter 4) to evaluate availability in IaaS clouds. An experiment is conducted by adopting the fault injection tool which was implemented using the proposed framework; And the experiment results (real-world environment) were compared with the results obtained from the RDB model.

1.4 Structure

For better organization and understanding, this work is organized as follows: Chapter 2 contains the foundations and all the background information needed for a better understanding of this work; Chapter 3 covers the conceptualization, implementation, and all the technical details about the proposed and developed framework; Chapter 4 covers case studies where the proposed framework is indeed used to implement a fault injection tool for the Eucalyptus system, the same chapter also discusses the dependability study of the Eucalyptus platform and discloses the results; Chapter 5 concludes this dissertation and proposes some future directions of this work.

CHAPTER 2

Foundations

This chapter presents the fundamental concepts over which this Bachelor's dissertation was produced. First, the Object Oriented Frameworks concept is presented, then the Performance Evaluation concept is disclosed; After that, the Dependability concept is introduced, as well as the basic metric equations needed for a better understanding of this work. Afterward, Continuous Time Markov Chain definitions are disclosed, following it is revealed the definition of Stochastic Petri Nets as well as Reliability Block Diagram.

Another important theoretical basis is Cloud Computing Systems, and also one application of such concept, regarding the IaaS approach, which is the Eucalyptus platform, because it will be used during the experimental phase of the proposed framework in Chapter 4 case study. Afterward, it is presented the Fault Injection definition and its main categories: Hardware Implemented fault injection techniques, and Software Implemented fault injection techniques. That completes the necessary knowledge to understand the proposed framework in Chapter 3.

2.1 Object Oriented Frameworks

There are several definitions of frameworks, in the context of object orientation. According to [67] framework can be defined as a set of concrete and abstract classes with their relationships and constraints, designed for a specific application domain. While for [47] framework can be defined as a set of classes that incorporates an abstract project of solutions to a family of related problems.

Briefly, an object-oriented framework can be understood as a set of concrete and abstract classes that provide a partial implementation of a system or part of it for a given problem domain. A framework consists of classes that have relationships whose objects interact with one another, unlike a library that contains a set of classes that can be used separately . Another difference between libraries and frameworks resides in the way they are implemented, whereas libraries are implemented primarily focusing on reuse, frameworks do not just reuse code, but also significant parts of a project [47].

Object-oriented frameworks, or simply frameworks, can be recognized as generators since they are intended to be used as a foundation for the development of a series of applications belonging to the same application domain. Embedded into the source code, they determine the application architecture and redefine the project parameters, allowing the developer to devote himself to the specific details of the software being developed [67].

The emergence of object-oriented programming brought with it some concepts such as inheritance, polymorphism, interface, among others that made it possible to create frameworks.

These elements are basic for building the avowed hot-spots of a framework. Hot-spots are flexibility points in the framework that require complementation, where developers add their code by specifying the functionality of the software in production [46].

A framework should be simple enough for the developer to understand and should offer flexible methods so that some of their characteristics can be changed by redefining methods. Frameworks are quite attractive for providing agile systems development through the reuse process, often called the instantiation process [46].

According to [46], the main objectives of a framework are related to maintaining the organization's knowledge about the application domain within the organization; Minimizing the amount of code needed to deploy applications that belong to the same problem domain; And possible adaptations to specific application needs, such as through a subclass.

Object-oriented frameworks have some advantages and disadvantages. Among the advantages, it is presented the following:

- Alleviation of developed lines of code if the functionality required by the application and the functionality offered by the framework belong to the same domain;
- The framework code is already written and debugged;
- Frameworks offer not just code reuse, but also design reuse [46].

Among the disadvantages of object-oriented frameworks are:

- Challenges in the development of frameworks, since experience in the field of application, is necessary;
- Complex detection of hot-spots;
- Implementation and means to validate a framework are not mild.

2.1.1 Frameworks Characterization

An object-oriented framework can be characterized by different dimensions. The most important dimensions, according to [46], are the problem domain that the framework covers, its internal structure and how the framework can be used.

Three are the realms of a framework: application framework, domain framework, and support frameworks. The designated application frameworks encapsulate functionalities that can be applied to different domains. An example of this type of framework are the frameworks for building graphical interfaces. Domain frameworks capture knowledge and experience in a particular problem domain. An example of such type of framework are the frameworks for multimedia; And finally, support frameworks provide services for a low-level system such as device drivers.

The internal structure of a framework is related to the concepts of software architectures. [46] describes the framework architectures as:

- *Layered architectural framework:* helps structure applications that can be decomposed into groups of subtasks with different levels of abstraction;
- *Architectural Framework Pipes and filters:* can be used to structure applications that can be divided into several fully independent subtasks; Those tasks must be executed either in a sequential or in a parallel order;
- *MVC (Model-View-Controller) architectural framework:* defines a framework for interactive applications that separate the user interface from its functional core;
- *Architectural Framework PAC (Presentation-Abstraction-Controller):* suitable for structuring software systems that are highly interactive with human users;
- *Reflective architectural framework:* used for applications that need to consider a future adaptation to changes in the environment, technology, or requirements; Although without an explicit modification of its structure and implementation;
- *Architectural micro-kernel framework:* suitable for systems that offer different points of view about their functionalities and, which often have to be adapted to the new requirements of systems;
- *Architectural framework blackboard:* helps structure complex applications that involve several specialized subsystems for different domains;
- *Architectural framework broker:* Distributed structure software systems in which components interact decoupled through remote operations calls on a client and/or server.

An object-oriented framework can be used in two ways. Either the user derives new classes from it or instantiates and combines existing classes. The first approach is called directed architecture or focused inheritance. The main approach is to develop applications based on the inheritance mechanism. Users of the framework implement adaptations through class derivation and operations override. The second approach, instantiation and combining, is referred to as data-driven or composition-focused frameworks. The adaptation of the framework to the needs of the application implies reusing the composition of an object. The way objects can be combined is part of the framework description, but what the framework does depends on which object the user passes into the framework. A data-driven framework is generally easy to use, however, it is limited.

2.1.2 Frameworks Development

The development of an object-oriented framework requires considerable effort, but the benefits during the software development process often justify the initial effort, since a complete application or a significant part of it can be built from a framework. Constructed to reuse both the code and the most significant parts of a project, frameworks can be implemented in two ways. The first form is based on the deep domain analysis, while the second form is the result of the

knowledge acquired during the development of several applications that belong to the same domain.

The first way is very similar to software development, starting with extensive domain research, among other things. In this form of development examples of instances can be obtained through books or people with extensive knowledge of the domain studied. These instances are collected and analyzed in order to collect common parts and parts that may change.

The second form of development of a framework is relative to the knowledge acquired by the developers during the process of implementation of several software belonging to the same family. Based on the knowledge acquired, a generic design can be developed separating the common and specific parts.

Both of the above-mentioned forms of development present drawbacks. In the first form, it is possible to mention the cost of development since the first application instantiated is usually not commercial but developed as a form of validation of the framework. While in the second form, design defects of previously developed applications will be passed on to the next applications instantiated from the framework.

2.2 Performance Evaluation

Performance Evaluation is taken as an art [31]. And like all artwork, it can not be produced mechanically. Each evaluation requires a great knowledge of the system to be modeled, as well as a careful selection of the methodology, the workload, and the tools to be applied. Performance evaluation can be used, for example, whenever you want to compare two or more systems and find the best for a given set of applications, decide which is the best alternative of design among several options, amidst others [31] [40]. Even if there are no alternatives, evaluating the performance of systems can help determine the quality of their activities, and if improvements are indicated. Experiments involving performance evaluation of systems can be employed at each stage of the life cycle of a computational system [31] , [52]. However, the cycle in which the system currently is influences the choice of the technique to be used to carry out the studies. Performance evaluation has three major sets of techniques: techniques based on numerical/analytical modeling, techniques based on simulation modeling, and techniques based on measurement. Techniques based on numerical/analytical modeling describe the system in a mathematical form [40]. These techniques are generally used when the system does not yet exist or is not available at the moment of the study. The time period required to obtain the results turns this technique very attractive, because it occurs quickly. A simple analytical/numerical model can provide an agile overview of the behavior of the system or one of its components. Although analytical models are approximated, they are accepted because the models themselves can be used to explore design alternatives. This is sufficient to estimate approximately the expected behavior and performance. The time to build a model is relatively brief, as is the cost of its development.

Techniques based on simulation modeling are established on abstract models of the system, so they can be used at any stage of the system's life cycle. Such techniques require a considerable investment of time in the model derivation, design, and coding of the simulator. The simulator can be easily modified to study the impact of changes made to any component of the

simulated system. The complexity and level of abstraction employed in a simulator may vary according to the system. With simulations, it may be possible to search the space of the parameter values for optimal combination. Simulation modeling is more flexible, more accurate, and has greater credibility when compared to analytical/numerical modeling [52].

Unlike techniques based on numerical/analytical modeling and simulation, the measurement-based techniques can only be used when something similar to the proposed system already exists, even if it is a prototype [31]. Among the techniques presented, this is undoubtedly the one with the highest cost; Since it requires real equipment and time. The results obtained through this technique can be quite variable since the defined parameters, as well as the workload, can influence the results of the experiments. The credibility of the results obtained through this technique is regarded as the most reliable. In addition, the results are probably the biggest justification when considering the costs involved in their use.

Sometimes it is quite useful to use two or more techniques at the same time. For example, one can use analytical/numerical modeling techniques and simulation together in the same evaluation to validate the results obtained. [31] has three validation rules:

- Results obtained by simulation are not reliable until they have been validated by analytical/numerical modeling or by measurement;
- Results obtained through analytical/numerical modeling are not reliable until they have been validated by simulation or measurement;
- Results obtained through measurement are not reliable until they have been validated by analytical/numerical modeling or by simulation.

The third mentioned rule is commonly ignored. Measurements are susceptible to errors, as are the other techniques. The only requirement for validation is that the results should not be counter-intuitive. This method of validation is called expert intuition and is commonly used in simulation models.

Two or more techniques may be used sequentially. For example, a simple analytical model can be used to find the appropriate range for system parameters and the simulation subsequently used to study performance in that range. This reduces the number of simulation runs considerably and results in a more productive use of resources.

2.2.1 Performance Measurement

The measurement of computational systems essentially involves the monitoring of the system being studied while under the influence of workload. To achieve expressive results, the workload must be carefully selected. This load can be real or synthetic [22]. The actual workload is observed in a real system in production. This type of load is usually not the most suitable for carrying out performance studies due to not being repeatable [31], the size of the load may not be considerable, the data may have undergone many disturbances, or, for reasons of accessibility. Instead, a synthetic charge, the characteristics of which are similar to the actual charge, can be applied repeatedly and in a controlled manner.

The main reason for using synthetic workloads is that it is a representation or model of the actual workload. Some other reasons for using a synthetic workload are ease of modification; The simplicity of being converted to different systems due to its small size; And the load-generating application may also have built-in some measuring capability.

In the measurement technique, different types of metrics are generally required, depending on the nature of the system being tested [52]. The different measurement strategies have their basis the concept of the event, where this is a change in the state of the system. The precise event definition depends on the metric being measured [40]. For example, an event can be defined to be a reference to the memory, a disk access, a network communication, a change in the internal state of the processor, or the combination of other sub-events. From the event type point of view, these metrics can be organized into one of the following categories [40] [52]:

- *Event Count Metrics:* This category includes metrics that are simply counting the number of times that a given event occurs. For example, number of read/write requests on a disk storage;
- *Auxiliary event metrics:* Records the values of secondary system parameters, whenever a particular event occurs. For example, to determine the average number of messages in the queue that are sent to the buffer of a communication port, one must register the number of messages in the queue each time a message has been sent or removed from it. Thus, the events to be monitored will be the number of messages in and out of the queue;
- *Profile:* is an aggregate metric used to characterize the overall behavior of a program or an entire system. Typically, it is used to identify where the program or system is spending the most run-time.

The strategy used to measure the performance of the metric of interest can be decided based on the classification of the type of event discussed above, where the main strategies are [52]:

- *Event-driven:* Registers the information needed to calculate the performance metric whenever the events of interest occur. This strategy has the advantage that the generated overhead occurs only in the data record. On the other hand, if monitored events occur quite frequently, then such characteristic becomes a disadvantage;
- *Tracing:* This strategy is based on recording more data than just a single event. This results in the need for more storage space when compared to the event-driven strategy;
- *Indirect:* This strategy is used when the metric of interest can not be measured directly. In this case, one must find another metric that can be measured directly, from which the desired metric can be deduced;
- *Sample:* This strategy is based on recording the state of the system in time intervals the metric of interest. The sampling frequency determines the measurement overhead.

2.3 Dependability

The dependability of a system can be understood as the ability to provide a range of services with a certain level of confidence [59]. In fact, dependability is also related to concepts such as fault tolerance and reliability. Without loss of generality, the reliability is the probability that the system will provide a set of services for a certain time period [37]. In fault tolerant systems, reliability gives the probability of operation even when there are defective components. Availability is also another important concept, which quantifies the mixed effect of the fault and repair processes in a system. In general, availability and reliability are related concepts they differ in the fact that the availability may consider repairing components that have failed [19].

Metrics can be calculated using combinatorial models, such as Reliability Block Diagram (RBD), or models based on states, for example, SPN (RBD is the model adopted in this study). RBDs allow to represent networks of components and also to establish closed form equations. However, RBD designs have drawbacks for handling faults and repairs of the dependencies; Such cases are often found in the representation of maintenance policies and redundant mechanisms, especially those based on dynamic redundancy methods. It should also be noted that the RBD could be evaluated through simulation; Simulation is adopted whenever distributions more complex than the exponential distribution are used in the model.

2.3.1 Dependability Equations

For instance, if a designer is interested in calculating the availability *(A)* of a given device or system. It is needed the time to failure *(TTF)* and the time to repair *(TTR)*. Considering that the uptime and downtime are not available to the designer, then the most affordable way is the average value. If the designer needs only the average value, so the metrics commonly adopted are the mean time to failure *(MTTF)* and the mean time to repair *(MTTR)* (other central tendency measures can also be adopted). However, if the designer is also interested in the variability of the availability, then the standard deviation of the time to failure *(sd(TTF))*, and its repair time standard deviation *(sd(TTR))* give an estimate of the variation of the availability.

Availability *(A)* is obtained by analysis or simulation of the steady state. It is given by the following equation:

$$A = \frac{MTTF}{MTTF + MTTR} \tag{2.1}$$

Another useful concept is the instantaneous availability, which indicates the availability for a given time instant *(t)*, defined by:

$$A(t) = \frac{\mu}{\mu+\lambda} + \frac{\lambda}{\mu+\lambda} e^{-(\mu+\lambda)t} \tag{2.2}$$

Where μ indicates the repair rate *(1/MTTR)*, and λ represents the failure rate *(1/MTTF)*. It is considered that the failure time and the repair time obey exponential distributions.

Assume *f(t)* is a probability density function. Thus, through the analysis or transient simulation, the reliability *(R)* is obtained. So the *MTTF* can be calculated as the standard deviation

of time to failure *(TTF)*:

$$MTTF = \int_0^\infty R(t)dt = \int_0^\infty tf(t)dt \tag{2.3}$$

$$sd(TTF) = \sqrt{\int_0^\infty t^2 f(t)dt - (MTTF)^2} \tag{2.4}$$

It should be borne in mind that, for the computation of the reliability of a particular service or system, the respective service repair activity should not be represented. In addition, considering the unavailability calculation which is given by $UA = 1 - A$, and Equation 2.2, the following equation is derived:

$$MTTR = MTTF \times \frac{UA}{A} \tag{2.5}$$

So, the standard deviation of the repair time *(TTR)* can be calculated as follows:

$$sd(TTR) = sd(TTF) \times \frac{UA}{A} \tag{2.6}$$

In this work it is considered the availability of the system. The experiment disclosed in chapter 4 estimates the availability from the user point of view, which is the availability of the virtual machine services.

2.3.2 Confidence Interval of Availability

It is presented the approach to calculate the confidence interval of availability[65].

The failure rate is λ and the repair rate is specified by μ. Consider ρ as being the ratio λ/μ. If it is assumed that n failure and repair events were observed during the experiment, then the total failure time is S_n and the total repair time is Y_n. Therefore, the maximum-likelihood estimator for λ is defined by the Equation 2.7.

$$\hat{\Lambda} = \frac{n}{S_n} \tag{2.7}$$

A 100x(1 - α) confidence interval for λ is given by Equation 2.8.

$$\left(\frac{C^2_{2n;1-\frac{\alpha}{2}}}{2S_n}, \frac{C^2_{2n;\frac{\alpha}{2}}}{2S_n}\right) \tag{2.8}$$

An analogous process is performed estimate μ Equation 2.9.

$$\hat{M} = \frac{n}{Y_n} \tag{2.9}$$

Thus, the 100x(1 - α) confidence interval for μ is given by Equation 2.10.

$$\left(\frac{C^2_{2n;1-\frac{\alpha}{2}}}{2Y_n}, \frac{C^2_{2n;\frac{\alpha}{2}}}{2Y_n}\right) \tag{2.10}$$

Consequently, the maximum-likelihood estimator for the ratio λ/μ is $\hat{\rho}$ and it is defined by Equation 2.11.

$$\hat{\rho} = \frac{\hat{\Lambda}}{\hat{M}} = \frac{\frac{n}{S_n}}{\frac{n}{Y_n}} = \frac{Y_n}{S_n} \tag{2.11}$$

The 100x(1 - α) confidence interval for ρ is given by (ρ_l , ρ_u) through F-distribution probability function as specified in Equation 2.12.

$$\rho_l = \frac{\hat{\rho}}{f_{2n;2n;\frac{\alpha}{2}}} \quad \text{and} \quad \rho_u = \frac{\hat{\rho}}{f_{2n;2n;1-\frac{\alpha}{2}}} \tag{2.12}$$

Finally, the maximum-likehood estimator to availability is $\hat{A} = 1/(1+\hat{\rho})$. Since the availability A is a monotonically decreasing function of ρ, the 100x(1 - α) confidence interval for A is:

$$\left(\frac{1}{1-\rho_u}, \frac{1}{1-\rho_l}\right) \tag{2.13}$$

2.3.3 Continuous Time Markov Chain

Markov chain is a mathematical model proposed in 1906 by Andrey Markov [43] which corresponds to a type of stochastic process that has some special properties. This theory opened possibilities for the study of stochastic processes in various fields of applications, such as economics, meteorology, physics, chemistry and telecommunications. Particularly in computer science this formalism is quite convenient for describing and analyzing dynamic properties of computer systems [8]. Markov chain can be viewed as a fundamental technique for performance analysis and dependability analysis; Markov chain is the basis over other techniques are constructed [49]. In queuing theory, a queue may be reduced to a Markov chain and analyzed and, in addition, some major theorems of this theory are proved by Markov models [35].

To define the concept of continuous time Markov chain, initially will be presented the definition of stochastic process:

A stochastic process is defined by a family of random variables $X_t : t \in T$, where each random variable X_t is indexed by a parameter t and the set of all possible values of X_t represent the state space of the stochastic process.

Generally it is defined the t parameter as the time parameter, but just if it satisfies the condition: $T \subseteq \mathbb{R}^+ = [0,\infty)$. If the set T is discreet, the process is classified as discrete time, otherwise, it is continuous time. Likewise, the state space S can be discrete or continuous, dividing stochastic processes into two groups: discrete state and continuous state. A discrete space stochastic process can be called a chain.

A discrete state stochastic process is called a *Markov Process* if it satisfies the following property:

A stochastic process $X_t : t \in T$ is a Markov process if for all $0 = t_0 < t_1 < ... < t_n < t_{n+1}$ and for all $S_i \in S$, the probability $P(X_{t_{n+1}} \leq S_{n+1})$ depends only on the last value of X_{t_n}, but do not depend on the previous values $X_{t_0}, X_{t_1}, ..., X_{t_{n-1}}$. In other words, this property states that

once a system is in a particular state, then the transition to the next state depends only on the current state where the system is. The history of the previous states does not matter and is not remembered, so this property is also called the memory-less property.

Just as the stochastic processes in general, we divide the Markov chains into two classes according to the time parameter: DTMC (Discrete Time Markov Chain), and CTMC (Continous Time Markov Chain). The main difference between these two classes is that in the CTMCs the transitions can occur at any moment in time; While in DTMCs the transitions can only be performed at discrete points in time.

A Markov chain can be represented as a state diagram where the vertexes represent the states, and the arcs of the diagram represent possible transitions between the states. Transitions' weights have a meaning which varies if the chain is discrete or continuous time. In a DTMC, the weight of an arc from a state S_i to S_j represents the probability that once the system is in the state S_i, a transition occurs to the state S_j in the next time interval. Consequently, this value should be less than 1, and the sum of all weights of arcs leaving a S_i state also must not exceed 1.

In CTMCs, the weight of the arc represents the rate of migration from a state S_i to a state S_j. The inverse of this value corresponds to the average time spent in the state S_i. The memory-less property of Markov chains implies that this *time* should be driven by a distribution with this same property [8].

The only continuous time distribution to present such property is the exponential distribution. For DMTCs, the discrete distribution with such property is the geometric distribution.

2.3.4 Stochastic Petri Nets

The Petri Nets concept was conceived by Carl Adam Petri in his doctoral thesis entitled Kommunikation mit Automaten (Communication with Automatons) and presented at the University of Bonn in 1962 [53]. Since then, this formalism has been used in different areas, such as computer science, electrical engineering, physics, among others.

The graphical representation of the Petri Nets is composed by: Places (Figure 2.1 (a)), Transitions (Figure 2.1 (b)), Arcs (Figure 2.1 (c)) and Tokens (Figure 2.1 (d)). The state variables are represented by the places and the actions performed by the system are represented by the transitions [42] . Directed arcs interconnect these two components, places, and transitions. The distribution of tokens, or marks, in the Petri Net places, determine the current state of the system or the amount of a given resource available.

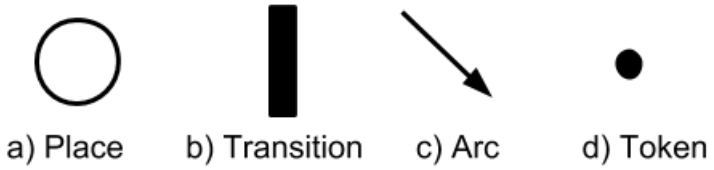

Figure 2.1 Elements of a Petri Net

Next, in Figure 2.2, it is presented a model of a Petri Net in which the operation of a web service is modeled. The places represent the states of the service, either the service is operational or the service is down, while the transitions represent the actions, which may be a random failure in the service or a service repair (could be a restart). In the initial state the service is operational, that is, functioning normally; This is represented by the token in the corresponding place, as shown in Figure 2.2 (a). At this point, the only possible transition will be the failure transition; Once the failure transition is triggered, the token goes to the corresponding place of the service down. At this point, the only available transition is the repair, Figure 2.2 (b), which once triggered takes the token back to its initial position, ie, the service is operational again.

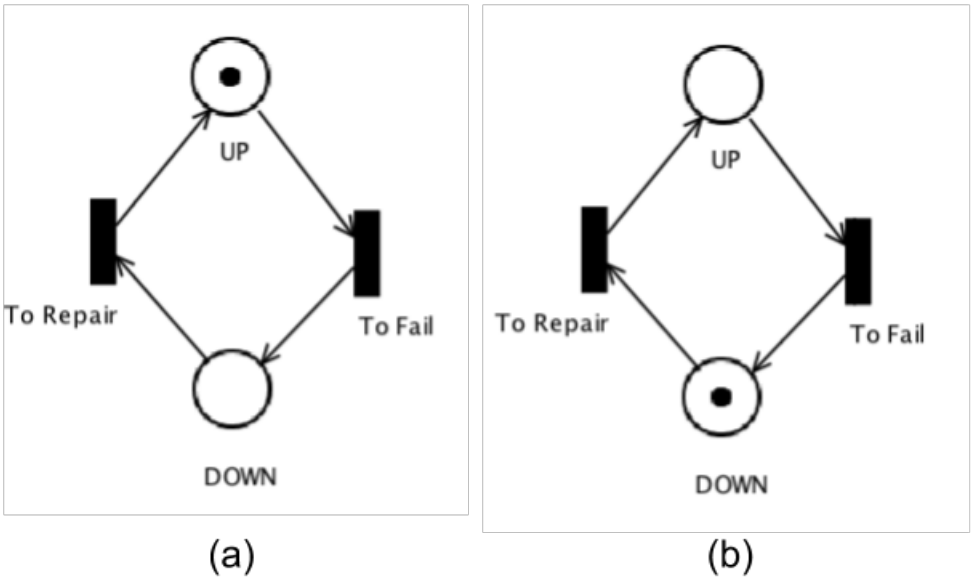

Figure 2.2 Example of a Petri Net

Initially, the Petri Nets proposed by Adam Petri did not have the time as notation due to the difficulties that specific parameter of the time would bring to the analysis of the models. Early work including time in Petri Nets was Merlin and D.J Faber [50] and J.D Noe and G.J Nutt [51].

There are different ways of embedding time into Petri Nets: by associating it with the location and type of time associated with it. As for location, time can be associated with places, transitions, and tokens. However, in most models, only transitions are associated with time. This is due to the fact that transitions represent activities, which in turn are actions that take some time to complete. But it is important to mention that choosing the location to associate time is typically a personal choice and also depends on the type of system being modeled [62]. Regardless of the type of location in which the notion of time will be used, several types of time can be modeled in a timed Petri Net: it can be deterministic, interval, or stochastic. Older models used deterministic times which in turn are simple to analyze. However, they limit their applicability. Thus, more recent authors use non-deterministic, ie, stochastic models [62] [42].

By allowing the addition of time in Petri Nets, other concepts such as the degree of habilitation emerge. This concept determines the number of times a particular transition can be triggered on a given markup before becoming disabled. Timing semantics indicate how many shots can be done per unit of time in a transition, as seen below:

- *Single-server (SS):* Only one token is hurled at a time, that is, the capacity of one place per transition is 1.
- *Multiple-server:* It is possible to perform *n* shots at a time, that is, the capacity of one place per transition is an integer textitn.
- *Infinite-server (IS):* You can do endless shots at once.

SPNs add time to Petri Nets formalism. This time is stochastic and exponentially distributed for timed transitions. In addition, there are immediate transitions, in which there is no associated time. The two types of transitions may have different levels of priority between them. However, immediate transitions have priority over tripping over timed transitions. Once the input arcs reach enough tokens and it becomes enabled, it must be triggered even if a timed transition is also enabled.

Formally, SPNs can be defined in several ways. In this work we adopt the definition according to [26] presented below:

An SPN is defined by the 9-tuple *SPN = (P, T, I, O, H,* Π*, G,* M_0*, Atts)*, where:

- *P = {p1,p2,...,pn}* is the set of places. *n* is the amount of places;
- *T = {t1,t2,...,tm}* is the set of timed and immediate transitions, $P \cap T = \emptyset$. *m* is the amount of transitions;
- $I \in (\mathbb{N}^n \rightarrow \mathbb{N})^{n \times m}$ is the matrix representing the input arcs, which may depend on markings;
- $O \in (\mathbb{N}^n \rightarrow \mathbb{N})^{n \times m}$ is the matrix representing the output arcs, which may depend on markings;
- $H \in (\mathbb{N}^n \rightarrow \mathbb{N})^{n \times m}$ is the matrix representing the inhibitory arcs, which may depend on markings;
- $\Pi \in \mathbb{N}^m$ is the vector that associates the priority level with each transition;
- $G \in (\mathbb{N}^n \rightarrow \{true, false\})^n$ is the vector that associates a guard condition related to the marking of the place at each transition;
- $M_0 \in (\mathbb{N}^n)$ is the vector that associates an initial marking of each place (initial state);
- *Atts = (Dist,Policy,Concurrency,W)*^{m} comprises the set of attributes associated with the transitions:

$Dist \in \mathbb{N}^n \rightarrow F$ is a probability distribution function associated with the time of each transition, but $F \leq \infty$. This distribution may be marking dependent;

$Policy \in \{prd, prs\}$ defines the memory policy adopted by the transition, where *prd* stands for preemptive repeat different, default value, which is identical to enabling memory policy; whereas *prs* stands for preemptive resume, which corresponds to the age memory policy;

$Concurrency \in \{ss, is\}$ is the concurrency degree of the transitions, where *ss* represents the single server semantics and *is* represents the infinite server semantics;

$W \in \mathbb{R}^+$ is the weight function, which associates a weight (w_t) with the immediate transitions and a rate λ_t to the timed transitions.

The SPN models describe the activities of systems by means of reachability graphs. These can be converted into Markovian models, which are used for quantitative evaluation of the analyzed system. Performance measurements are obtained through simulations and steady-state and transient analyses based on the Markov chain embedded in the SPN model [8].

The labeled SPNs and having a finite number of places and transitions in their structure are isomorphic to the Markov chains [44]. The isomorphism of a SPN model with a Markov chain is obtained from the reduced reachability graph, which is given by eliminating the volatile states, the label of the arcs with the rates of the timed transitions, and the weights of the immediate transitions.

2.3.5 Reliability Block Diagram

Reliability Block Diagram (RBD) is a combinatorial model; It was initially proposed as a technique for calculating the reliability of large and complex systems using intuitive block diagrams [59]. In general, block diagrams provide a graphic depiction of the system's components and connectors; Which may be adopted to determine the overall state of the system. The RBD's structure establishes the logic interaction between the components defining which combinations of active elements are able to sustain the operation of the system, in other words, it defines the combinations of the active components that are capable of maintaining the system running. This technique has also been extended to calculate the dependability metrics such as availability and reliability.

The blocks are generally organized using the following compositions: series, parallel, bridge, blocks k-out-of-n, or a combination of those approaches.

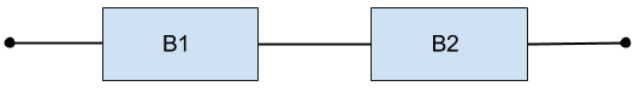

Figure 2.3 RBD example: Series

Figure 2.3 shows an example in which the blocks are arranged in series and Figure 2.4 shows an example in which the blocks are arranged in parallel.

In the series arrangement, if a single component fails, the whole system does not work anymore. Assuming a system with *n* elements, reliability *(R)* is given by:

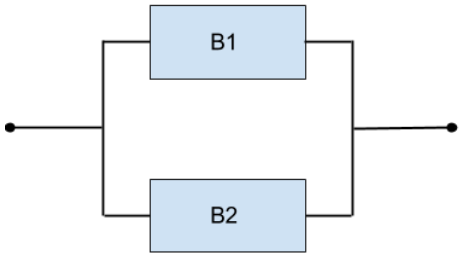

Figure 2.4 RBD example: Parallel

$$R_s(t) = \prod_{i=1}^{n} R_i(t) \tag{2.14}$$

where $R_i(t)$ is the reliability of the b_i block.

For a parallel arrangement, see Figure 2.4, at least one component must be operational for the entire system to be running. Taking into account n components, the system's reliability is given by:

$$R_p(t) = 1 - \prod_{i=1}^{n} 1 - R_i(t) \tag{2.15}$$

where $R_i(t)$ is the reliability of the b_i block. For other examples and more closed form equations, the reader should refer to [37].

2.4 Cloud Computing Systems

Cloud computing is a computing model in which resources such as processing power, networking, storage, and software are offered over the Internet and can be accessed remotely [5]. This model allows users to get the resources elastically, and at a low cost; All the services are delivered in a similar way to traditional services such as water, gas, electricity and telephone [18]. Among the many existing cloud computing definitions in the literature, it is presented the definition proposed by [63]:

> Clouds are a great set of easily usable and accessible virtualized resources such as hardware, deployment platforms and/or services. Such services can be dynamically reconfigured to a variable workload, allowing optimum utilization of resources. This feature set is typically exploited by a pay per use model, or pay as you go model, in which guarantees are offered by the infrastructure provider via customized Service Level Agreements (SLA).

Cloud computing is a new computing paradigm, but it was built based on existing technologies such as virtualization, grid computing, and utility computing. The computational resources of a grid are provided to customers as virtual machines (VMs) through virtualization of a data

center infrastructure, in such a manner that each customer only pays for the amount of resources consumed, instead of paying a flat fee (utility computing).

This computer model has several features that turn out to be attractive to many corporations. For instance, if a company would like to launch a service on the Internet, it can simply rent a set of VMs from a cloud service provider and then host its service on the VMs, without having to maintain their own server infrastructure [66].

The above features of cloud computing model help to reduce the time to market of their products/services and thus making it more competitive. Moreover, the elastically provisioning of cloud resources allow the system to adapt to abrupt changes in the workload without the need for sizing the used infrastructure. As an example, we can consider a portal for academic services (same as the Sig@ UFPE[1]); During registration and vestibular times, the volume of hits increases dramatically compared to the rest of the year; An administrator of such a system could allocate a greater number of more powerful VMs during this time of the year in order to meet such demand, however during the rest of the time of the year, when the number of hits is lower, it could be used modest VMs to save power and money.

The big advantage is that the service provider does not need to oversize a data-center infrastructure to support the peaks in workload, which as a consequence will be underutilized during most of the time. To illustrate this fact, Amazon - which these days is a major corporation in the cloud computing segment - used around 10% of the capacity of its data center, which had been designed to withstand peak times that in fact were not very common [29].

We can classify the computational clouds into two aspects. The first aspect relates to the cloud user's perspective and who maintains the infrastructure, by such point of view we can classify as a cloud [66]:

- *Private:* Cloud infrastructures designed to be used by a single organization. A private cloud infrastructure can be provided by an external agent, or it can be managed by the own company. This kind of cloud is often chosen by organizations that have very sensitive data, e.g. banks, government agencies, military agencies; That happens because usually it is not allowed that such confidential data be entrusted to third parties, in this case the third parties would be public cloud providers. The disadvantage of this approach in comparison with public clouds is the requirement of the initial investment in buying a proprietary server farm [66], just like the traditional model without the use of the cloud. However, we also count on the benefits of consolidation and resource management by the company, which allows a more efficient use of resources [69];

- *Public:* Public clouds are maintained by cloud service providers which offer their computing resources to other organizations and/or users. This type of cloud reflects more the unique benefits offered by cloud computing than the private clouds. Unlike a private cloud, the initial investment in infrastructure is zero, it is not necessary to acquire servers nor any other equipment itself. Public clouds are usually offered by large data-centers that have a computing power much higher than a private cloud. Thus, if the volume of accesses to a service increases dramatically, then it is possible to allocate more virtualized resources in order to adapt to the new workload. Another advantage is to delegate to

[1] https://siga.ufpe.br/ufpe/index.jsp

the cloud provider the responsibility for infrastructure management and for maintaining the combined SLA [11]. The disadvantage of this type of cloud is the lack of control over data, network and security settings; Which prevents some corporations from using this type of cloud, especially those that deal with highly sensitive data, already cited above;

- *Hybrid:* It is a combination of the public and the private cloud models with the goal to overcome the limitations of each approach. The advantage of this model is to provide the best of both models. The organization's critical data can be kept on the private part of the cloud, while it is possible to scale the service through the immense capacity of a public cloud. The disadvantage is a greater complexity in the cloud management and also the challenge of determining the optimal partitioning between the public and private components [66].

The other aspect on which the cloud computing can be classified is as its business model:

- *Infrastructure as a Service (IaaS)*: Through the use of virtualization, this business model partitions the resources of a data-center among users on the form of virtualized resources. In this business model, usually the user pays for the service according to the capacity allocated to the VM and according to the amount of time that the VM remains running. The user is responsible for maintaining the software stack in which the services will run;

- *Platform as a Service (PaaS):* In this business model, the service provider abstracts the VM and the operating system, and offers to the user a high-level software platform. This model has the advantage of a transparent scaling of hardware resources for the performance of the user services [63];

- *Software as a Service (SaaS):* This business model is geared to ordinary Internet users, rather than developers and system administrators. In this model, software and data generated by users are stored in the cloud and can be accessed from any computer connected to the Internet through a Web interface.

In addition to these main models described above, there is also the Data as a Service (DaaS) model [60], in which the cloud offers its customers a secondary storage service, ensuring high availability and data integrity. A DaaS can be seen as IaaS or SaaS, depending on the context in which it is used. Consider as an example the storage and backup service in the cloud *Dropbox*. The service that Dropbox [2] provides for its members can be seen as SaaS or DaaS. However, *Dropbox* uses the storage service from *Amazon S3* [1] to store the data of its customers. From the standpoint of *Dropbox*, *Amazon* storage service may be considered as IaaS.

2.4.1 Eucalyptus Cloud System

Eucalyptus stands for (Elastic Utility Computing Architecture Linking Your Programs To Useful Systems). It is a software that implements scalable styles of IaaS for private and hybrid clouds [12], it also provides an interface compatible with EC2 and S3 services provided by Amazon [6]. Such compatibility allows the user to run applications both on Amazon and Eucalyptus without modification.

The Eucalyptus platform uses the virtualization [12](hypervisor) from the underlying computer system to allow flexible allocation of resources, divorced from specific hardware. The Eucalyptus architecture is comprised of five high-level components, each with their own web interface, they are Cloud Controller (CLC), Node Controller (NC), Cluster Controller (CC) Storage Controller (SC), and Walrus. The following is a brief description of these components.

- *Cloud Controller (CLC):* It is the access point to the cloud infrastructure. This component uses its web interface to receive requests and interact with the rest of the components. The CLC is responsible for monitoring the availability of resources in various components of the infrastructure, resource arbitration and also the monitoring of the running instances (VMs)[15].

- *Node Controller (NC):* Runs on each node i.e. physical machine and it manages the VM lifecycle operation on the node. This component performs queries to find out if physical resources are available, for example, the number of CPU cores (Central Processing Unit), the size of main memory, and also to probe the instances' state within that node [32].

- *Cluster Controller (CC):* Manages one or more NCs. This component gathers information about a set of VMs and VM execution times in specific NCs. The CC performs four basic functions: requests instance (VM) execution to the NCs; controls the overlay virtual network comprises a set of VMs; gather information about a set of nodes, and report its status to the CLC [32].

- *Storage Controller (SC):* This component provides permanent storage to be used by the instances (VMs). This component implements access to networked storage block, similar to that provided by Amazon Elastic Block Storage (EBS) [6]. The main functions of the SC are persistent creation of EBS; allow the creation of snapshot volumes; and provide block storage protocols to AoE (ATA over Ethernet) or iSCSI (Internet Small Computer System Interface) for the instances.

- *Walrus:* It is a storage service based on a filesystem compatible with the Amazon S3 [1]. Can be used by VM images in addition to serving as a repository for VM file systems, ramdisks and Linux kernel images that are used to instantiate i.e. start VMs in the cloud physical nodes by the NC [32].

Figure 2.5 shows the layers with the respective Eucalyptus components as an overview of the Eucalyptus platform architecture [12].

2.5 Fault Injection Techniques

Computer systems need to maintain proper operation even in the presence of faults. However, a system does not always performs correctly the function for which it was designed. The causes and consequences of deviations from the expected operation by the system are called dependability factors, comprising [70]:

[2]http://www8.hp.com/us/en/cloud/helion-eucalyptus-overview.html

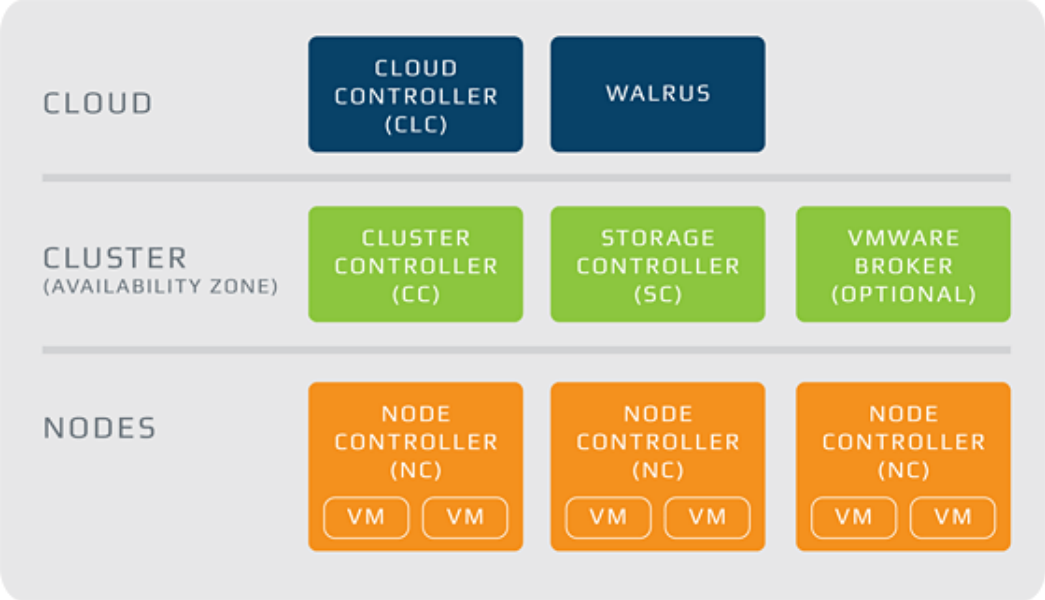

Figure 2.5 Eucalyptus Components. *Source: Eucalyptus Official Documentation*[2]

- *Fault:* is a physical defect, imperfection or flaw that occurs within some hardware or software component [70];
- *Error:* it is a deviation from the system accuracy, being a manifestation of a failure. A fault during processing can lead to a modification in the system state, which is considered an error [70];
- *Failure:* is the non-execution of some action that was expected by the system [70].

When a fault causes an incorrect change in the system state, an error occurs. Although the fault is located in the source code or located in the circuit, multiple errors can be generated and propagate in the system. When fault tolerance mechanisms detect an error, there are several actions that can be initiated in order to deal with the error and contain it. Otherwise, the system possibly has a malfunction.

There are two types of faults: hardware and software.

- *Hardware Fault:* They may arise during system operation and can be classified according to the length, they can be permanent, transient or intermittent. Permanent faults are irreversible and usually are caused by a damage to the physical component; transient faults are caused by specific environmental conditions, such as electromagnetic interference or radiation; intermittent faults are caused by hardware instability. This last type of failure can be repaired, for instance by replacement of components.
- *Software fault:* They are consequences of incorrect design, incorrect specification or incorrect design code time. However, many of such faults are latent in the source code of the software and only occur during system operation, especially under a high or unusual

workload. Since being a result of poor design, it can be assumed that all software faults are permanent. Interestingly, practice shows that, despite their permanent nature, their behavior is transient, i.e. when a bad system behavior occurs, the fault may not be seen again, even when great care is taken to repeat the situation in which it occurred.

Some of the hardware and software faults can be overcome when they are discovered in early stages of the development process, or in testing systems. The faults that are not removed from the system can affect the dependability of the system.

The dependability evaluation is usually done using fault injection. Fault injection corresponds to artificially insert faults into a system and then evaluate their behavior on the inserted faults [61]. Hardware faults are easily injected by hardware devices designed for such task or even applications built with such purpose [70]. In both cases, disturbances may be caused such as changing bits or for example bit misses. In this case, the choice of method for fault injection can determine the type of hardware failure, according to the classification described above. Software failures are usually the main cause for system operation interruption. This type of failure can be inserted into the system by applications called fault injectors. Such applications can be built to achieve both operating systems or specific programs/services.

For injecting faults in computer systems, some techniques may be employed. The fault injection techniques are classified into five categories [70]: hardware-implemented fault injection, software-implemented fault injection, simulation-based fault injection, emulation-based fault injection and hybrid fault injection. However, this work will only discuss the hardware-implemented fault injection and software-implemented fault injection. These two techniques are the subjects of the following subsections.

2.5.1 Hardware Implemented

Hardware implemented fault injection requires an additional hardware to insert faults on the target system. This type of fault injection allows disturbances to be entered into the system at the physical level of the machine. Depending on the fault and the location where it should be injected, the hardware fault injection techniques can be divided into two categories [70]:

- *Contact:* the hardware injector has physical contact with the target system by introducing electric current variation in the chip of the system under test;
- *Contact-less:* the injector does not have direct physical contact with the system under test. In this category, faults are introduced by means of external sources that produce some physical phenomena such as, electromagnetic interference and ion radiation.

Regardless of the used category, there are some benefits of using the hardware implemented fault injection technique, they have access to sites that are not accessible by other techniques, experiments are considered fast, and tests can be performed close to real time. The use of this technique also entails some drawbacks such as high risk of damage to the system under test; low portability and low control over faults; another drawback is reduced system behavior observation during testing.

2.5.2 Software Implemented

Software implemented fault injection is a more flexible approach than the hardware implemented approach, as it allows to simulate faults both in hardware, and applications such like operating systems [70]. This technique involves modifying the operating status of the software that is running on the target system.

Such technique is divided regarding the injected faults, into two categories [70]: Compilation time or during execution time. To inject faults while compiling the software instructions should be modified before the program image is loaded and executed. In this category, faults are injected through changes in the source code or assembly code of the target application. Thus, when the modified program runs then the faults are also activated. Note that the change in application source code can derail the software permanently.

In order to insert a fault during executing time it is required an additional mechanism to introduce the faults on the target system. This category can be divided according to the form how faults are inserted, they are:

- *Time-out:* the simplest of the methods. This method uses a timer (hardware or software) for controlling the injection of faults. When the predetermined period of time ends then the fault is injected;
- *Exception/trap:* in this case, hardware or software exceptions transfer the control to the fault injector, where faults are inserted whenever a particular event or condition occurs. The call to the fault injector must be inserted into the source code of the target application;
- *Code Insertion:* instructions are added to the target software to allow the fault to be triggered before certain commands. Unlike the compile time fault injection, this method just adds instructions to the system without modifying the existing source code.

Some benefits in choosing the software implemented fault injection are: experiments can be performed close to real time, does not require special hardware, and it is of low complexity and low cost of implementation. In contrast, some difficulties may be encountered in the use of this technique such as: require source code modifications, it has limited observation and limited controllability, and it is not possible to inject faults in locations that are not accessible via software.

This work proposes a framework for software implemented fault injection, in execution time, that injects faults by using a time-out method.

CHAPTER 3

The Framework - FIDEP

This chapter presents the proposed framework Fidep. Fidep is a framework designed to be supportive on dependability studies; **Fidep** stands for **F**ault **I**njection repair and monitoring framework for **DEP**endability studies for cloud computing systems. First the conceptualization is discussed, after the framework features; Followed by the implementation details, Java EE integration, as well as the developed framework architecture and its modules.

3.1 Conceptualization

Fidep is a framework for building fault injection tools for studies in the areas of performance evaluation and system dependability. Developed in Java [16], the framework has methods that can aid in the development of fault injection programs, the generation of synthetic events, such as creation and event control, such like faults and repairs. Thus, it is up to the development team to work out which failure and/or repair event will be produced, and choose how the application should interact with the target system. Flexibility is favored through the choice of one of the options that Fidep provides for creating and controlling fault / repair events.

Some object-oriented software programming terms will be constantly found throughout this work, such as class, abstract class, constructor, inheritance, instance, repeat loop, method, abstract method, object, package, subclass, superclass, thread, among others. The definitions of those terms will not be discussed in this work because this nomenclature is considered standard in projects involving object oriented programming. The meaning of each term can be found in rich detail in [16].

Once the problem was defined and the functionalities that the framework should contain were chosen, the Fidep design process was started. The sequence of steps followed to achieve the purpose of this work is illustrated by the flowchart present in Figure 3.1.

The first activity of the flowchart addresses the choice of the programming language used in the development of the framework. Considering the breadth, flexibility, and reusability of the framework, it was decided to use the Java language [14]. Another aspect that influenced the choice of this language was that it allows the construction of several types of applications, being mobile, corporate, among others. One of the most important features of Java is its portability: an application can run on different platforms and operating systems, as long as it supports Java virtual machine, aka JVM.

The second activity of the flowchart refers to the choice of the programming environment used to encode the framework. During this step, the Eclipse Integrated Development Environment [20] was selected.

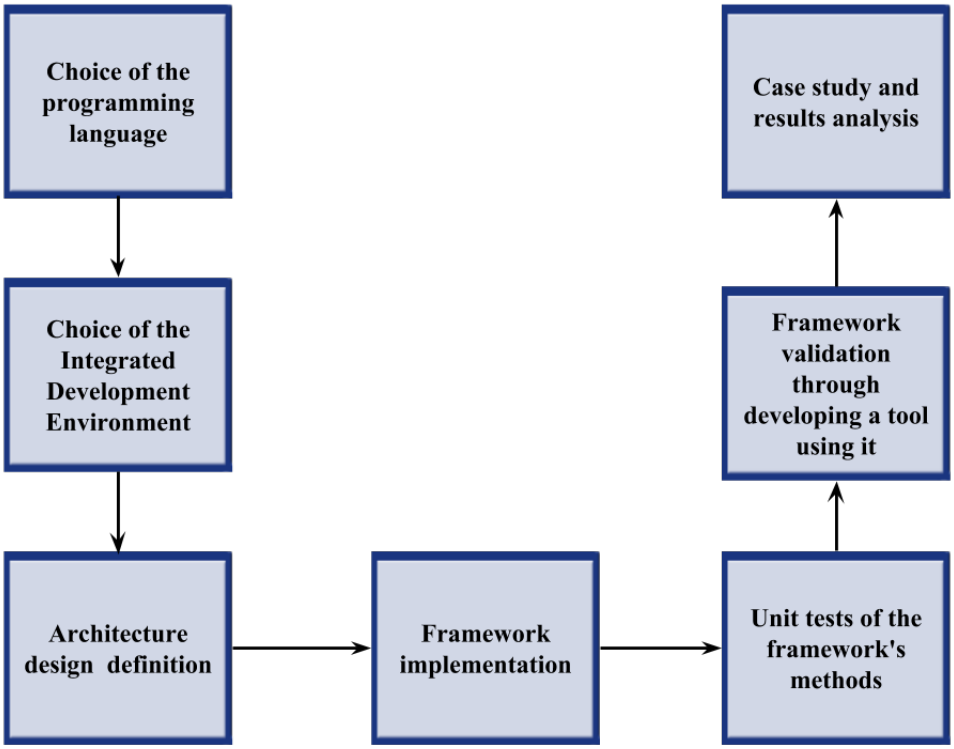

Figure 3.1 Flowchart of the activities adopted during the implementation of the framework.

The third activity in the flowchart comprises the definition of the architecture. In this stage, the Fidep architecture was defined and documented with the aid of class and sequence diagrams. This step is extremely important because it guided the step by step to be adopted during the implementation process of the framework. In a later activity, the documentation was expanded with the addition of Javadoc [36].

The fourth activity of the flowchart addresses the implementation of the framework. In this step, all the coding part related to the Fidep occurred, through the Eclipse programming environment, all classes and methods selected to integrate the framework were created.

The fifth activity corresponds to the test of the implemented methods. In this step all the methods that compose the framework were tested aiming to verify if they performed the activities for which they were expected to.

The sixth activity is related to validation of the framework. This activity is essential because it aims to analyze the effectiveness of the framework through the reduction in the effort of coding tools. In this activity, a tool was developed with the intention of injecting faults into the target system, which for this case study was chosen to be the Eucalyptus Cloud; Chapter 4 presents two case studies, which were performed using the developed tool.

The seventh and last activity presented in the flowchart is aimed at analyzing the efficiency of the tool built from Fidep. Through the results obtained in case studies, where these involve evaluating the availability of the system against the occurrence of failures. Chapter 4 describes in more detail how this step was performed.

Fidep is a framework that assists in the construction of fault-generating tools, with the option of repair, that emulates the absence of operations in the cloud system managed by cloud computing platforms. This framework was designed to support reliability and availability studies. The faults caused by the tools derived from the Fidep framework must always be transient faults, that is, the fault simulates a possible state of interruption of the execution of the system so that the fault can be repaired. The disturbances caused by the insertion of the faults in the cloud system are aimed at affecting the execution of the high-level components (of the Eucalyptus for example), as well as the physical infrastructure that houses the cloud computing system. The repair action aims to recover the system from any faults caused by the injector itself. The repair is only performed after the insertion of the faults, which implies that the tool does not repair a fault that was not injected by itself.

Fault tracing and repair, upon request, occurs after a period of time at random. Further details on inserting faults and repairs are given below.

Fidep supports the execution of events in the following modes of operation:

- *Hardware Faults:* This operating mode emulates hardware faults through the device hibernation mechanism, for example, server, desktop;
- *Hardware Faults and Repairs:* This operating mode includes both faults and hardware-related repairs. After the fault is injected the developer must inject the repair event which is a timer that waits for the time set at the time of hibernation. This operating mode requires the target machine to support and allow the Linux *rtcwake* standard, rtcwake is used to bring the system to a suspendable mode that can be configured by parameters and set a time for the system to return to its normal state of operation, rtcwake is amply discussed by [3].
- *Software Faults:* Faults in high-level components of the target system are included in this operating mode. The tool must act directly, suspending or terminating the execution of a target system process selected by the user/developer. It is noteworthy that in the same machine one can inject more than one type of software fault.
- *Software Faults and Repairs:* The difference between this operating mode and the previous one is that it not only acts causing faults but also performing the repair. The repair is performed by restoring the target system processes that had previously been terminated by the fault injection tool.

The framework enables the combination of more than one mode of operation previously described in a single experiment. This combination characterizes the possible scenarios of an experiment. In order for 100% of the functionality to be used, Fidep is currently restricted to environments that use the Linux operating system [57], but since Fidep was developed in Java [14], it is portable and can be adapted to run in other environments. However, the framework can be adapted to allow more elaborate fault events. Faults related to the suspend operation of the hardware can also be modified. In this case, the developer may add other hardware faults through software, such as loss of cached data.

The core of the framework consists of two fundamental parts: communication module and random number generation module. The connection module is responsible for providing communication between the fault injector and the target machine. The connection is established using the SSH2 [7] protocol that allows commands to be sent directly to the Linux operating system shell on the servers that compose the cloud (target system). The random number generation module is responsible for generating pseudo-random numbers that follow some of the available probability distributions. The tool uses these values as the time interval for the occurrence of events. The probability distributions will be detailed later.

The Fidep development process was marked by some design choices. The first choice was related to the protocol to be used to establish the communication between the framework and the system being tested. The SSH2 [7] protocol was chosen because of the possibility of sending commands that would cause software flaws in addition to software repairs. Subsequently, a solution was found that could inject faults and repairs on the target machines. The solution chosen to perform this task was the Linux *rtcwake* [3] standard. The main reason for this choice is because rtwake is standard on the Linux kernel.

3.1.1 Fidep's features

The high level functional software requirements of the framework are: to be compatible with the major linux distributions; to be capable of injecting faults on the most used cloud computing systems, IaaS providers under Linux which are: Eucalyptus[12], OpenNebula[55], OpenStack[56], CloudStack[23]. The framework is designed to be used by developers and/or scientists who need a flexible and easy way to conduct a fault injection experiment in one of the above cited cloud computing systems. The fault injector focuses on high-level components; the physical infrastructure and virtual machines. It is important to state that the framework provides the basic functionalities that are common to fault injectors, although the specific commands that simulate a fault will vary among the different target systems; as an example of use of the framework, it is implemented a fault injection tool for the Eucalyptus system[12].

As a suggestion of use the developer could create a component class to represent a high-level entity in the real system that can either fail or repair and be implemented as *HardwareComponent* or *SoftwareComponent*. Each component is associated with a *DistributedFunction* which controls event generation times. *HardwareComponent* represents hardware components and is independent of other components, since hardware itself will fail independently of other components, although the user has the flexibility to implement a fault injector just for the *SoftwareComponent*.

Fidep features are: To inject and repair (the injected faults) ; Embedded custimizable scripts for monitoring the target system, using the SysStat library[28]. The monitoring scripts should run on each machine to be monitored and the developer is responsible for the correct interpretation of its results.

3.2 Implementation

The Fidep framework is designed to be flexible; Fidep provides the main functionalities that are needed to implement fault injection tools, which are: event generation, network communication, and makes easier the random number generation.

The fault injection logic is based on components. Fidep was implemented by taking advantage of the compiled functions which will be discussed in the next subsections.

3.2.1 Java EE - Integration

Nowadays Web applications already have fairly complicated business rules. To codify these many rules represents a great job yet. In addition to these rules, also known as functional requirements of the application; There are other requirements that must be achieved through the infrastructure, which could be for instance: persistence in the database, transaction management, remote access, web services, thread management, HTTP connection management, object caching, the web session management, load balancing, among others; These are called non-functional requirements.

If we are also responsible for writing code that deals with all those other non-functional requirements, then we would have much more work to do; In accordance with that, a number of specifications that when implemented can be used by developers to take advantage and reuse all this ready-made infrastructure; Such ready-made infrastructure is the implementation of the Java EE [14].

The Java EE (Java Enterprise Edition) consists of a series of very detailed specifications, giving a recipe and how to implement software that makes each of these infrastructure services [14].

According to [14] Java Platform, Enterprise Edition (Java EE) is the standard in community-driven enterprise software; Java EE is developed using the Java Community Process, with contributions from industry experts, commercial and open source organizations, Java User Groups, and countless individuals. Each release integrates new features that align with industry needs, improves application portability, and increases developer productivity.

Fidep is integrated into a Java EE Web project. Fidep source code is well documented, by the use of JavaDoc[36], which makes it easier for the developer/user to understand and to extend the framework. The framework also provides a template web interface, which was implemented using Java Server Faces (JSF)[30], such web interface intends to provide a more friendly interaction with the user, but it is important to notice that it is not mandatory to use the web interface, the user can use all the functionalities direct into a new class and print out the results on the java console. Such Independence is possible thanks to the Model-View-Controller design pattern[38].

3.2.2 Spring framework

It was also used the Spring Framework[33]. The Spring Framework is an open source application framework and inversion of control container for the Java platform. The framework's core features can be used by any Java application, but there are extensions for building web applica-

tions on top of the Java EE platform. Fidep's development takes advantage of 3 main modules of the Spring framework: The first one is the inversion of control container, which is responsible for the dependency injection; the second is the Data access framework, which handles all the database operations and by using the Spring Data with hibernate is it possible to use any database manager without having to modify the source code of the application, because it uses a higher level entity, the only change needed is in the java connector configuration; Finally, the third module used was the model-view-controller framework[33].

3.2.3 Apache Maven

Maven simplifies the build process from the source code. It is a one time configuration tool provided by Apache [48]. The Maven tool is used to build and manage the Fidep Java project, it makes the build process easy, because it provides a uniform build system; it also provides guidelines for best practices development; it also allows transparent migration to new features. Maven is responsible for managing and downloading all the dependencies of the Fidep project, which makes it easy[48] for a developer to use the Fidep framework. Dependency management in Maven encourages the use of a central repository. Maven comes with a mechanism that the project's clients can use to download any dependency required for building the project from a central repository. This allows users of Maven to reuse dependencies across projects and encourages communication between projects to ensure that backward compatibility issues are dealt with.

The Figure 3.2 shows the principal files/classes of Fidep Framework, it is possible to understand the model view controller design pattern being applied; The packets are divided into controller, converter, enums, exception, model, model.repository, support, and view.

3.2.4 Fidep Architecture

Fidep provides random generation of events that emulate faults in high-level components. The purpose of these events is to simulate an action or a set of steps which leads to an unavailable state in the target component. Fidep is also capable of executing repair events in the target environment, however, the framework only supports repairing the system if the fault event was already generated by Fidep. To simulate real system behavior, the events of fault and repair follow randomly distributed times supported by the FlexLoadGenerator package [24].

Fidep's architecture design supports faults/repair generation in the following categories:

Infrastructure, the events generated in this category simulate hardware failures. These failures are implemented by hibernation commands to the Operating System (OS), which actually stop the activity of the machine for a limited time period. When the hibernation time is exceeded, all previous states of the system return to normal function, thereby performing a repair.

Considering the Eucalyptus Components, such mode is focused on interacting with the high-level Eucalyptus components previously mentioned. Fault events of this type are enacted by Eucalyptus component processes employing the service commands for Linux distributions. These commands act by stopping or starting processes of the selected component and may exhibit dependencies with associated hardware. Although it is possible to not associate a hardware component to an execution, it is highly recommended to do so if running parallel studies

on the same machine. This association is recommended to ensure a realistic scenario and a fault-free execution of the injection tool.

Considering the Virtual Machines interactions with the client service, which is represented by VMs, present dependencies with data center status. In other words, there is a connection with the eucalyptus Cloud Manager (the Cloud Controller) and the AWS Java API is employed to connect with Eucalyptus [12]. Therefore, in order to run failure scenarios on virtual machines, user credentials and access to the controller endpoint are required, as well as information regarding virtual machine service (i.e. image-id, virtual machine type). The injection of fault events is simulated with cloud environment terminate routines, and repair events are likewise simulated with creating new virtual machine routines.

Each component functions at the same time, so it is possible to specify a wide range of scenarios and generate multiple failure/repair tracks to the same component, machine or virtual machine. Dependencies are represented by directly linking each entity with its associated dependencies and it is the responsibility of the developer/tester to define these scenarios programmatically.

3.2.4.1 Fault Monitoring Module

This module is composed by two main sub-components: the event logger and the methods that check if a specific component of the target system is alive, in other words, if the component is actually responding and performing correctly. The developer should be able to configure all the parameters used by the framework, Fidep is flexible and customizable so that the user is able to define different scenarios to be tested.

3.2.4.2 Random Number Generation Module

The random number generation library is responsible for assisting in management between the occurrence of event triggers. The library generates samples with random values based on some of the main existing probability distributions, both continuous and discrete, through the application of random variable generation techniques [17]. The most widely used probability distributions in the areas of performance evaluation and dependability of computational systems are implemented in this library: Erlang, Exponential, Lognormal, Normal, Pareto, Triangular, Weibull and Uniform (continuous), And Geometric and Poisson (discrete). In addition to the mentioned distributions, this library also has the Empirical distribution implemented.

This library comes from the kernel present in WGCap - Workload Generator for Capacity Advisor, a tool designed by [25], without undergoing any changes to its coding. To that end, only the packages, and their respective classes have been preserved, referring to instructions for the operation of each probability distribution, control, and I/O information. The package responsible for the GUI, Graphical User Interface, was discarded because it was not necessary for use in Fidep. The generation of random numbers through the probability distributions functions promoted by the WGCap kernel has been extensively tested and its effectiveness is proven, as discussed in [24].

3.2.4.3 Communication Module

Fidep implements a generic communication module which can be used in order to send commands and receive outputs from the machines and components of the tested. Such communication module implements a Secure Shell (SSH); SSH is is a cryptographic network protocol for operating network services securely over an unsecured network[7]. The communication module is used to send the commands that inject the faults, as well as to send the repairing commands. This module has to be customized with the specific commands to the specific Linux distribution that the developer is using inside the testbed.

3.2.4.4 Finite State Machine

Each component has its own state machine in order to manage the four states that are possible during the life cycle of fault injection experiments: 1- Component Up/Running, 2- Framework waiting to inject the fault, 3- Component Down/Failed, 4- Framework waiting to repair.

- *Running:* In this state the component is up and running; The fault injection tool is able to generate a pseudo random time according to the specified probability distribution, and set up a timer in order to inject the fault. After the random time is generated the state machine goes to the next state.
- *Timer-Inject-Failure:* In this state the component is still running, and the injection tool is waiting for the timer event, when the time is over then the fault is injected; And then the state machine goes to the next state.
- *Failed:* In this state the component is down, which means that the fault was already injected and the injector tool may now generate another random time, but now the timer is set to perform the respective repair command. After the random time is generated the state machine goes to the next state.
- *Timer-Repair:* In this state the component is still failed, and the injection tool is waiting for the timer to end in order to inject the repair command to the component, and then the component will be up and running again, which brings the state machine to the very first state again.

It is responsibility of the developer to handle the exceptions that may arise according to the needs of the experiment to be performed.

The heart of a state machine is the transition table, which takes a state and a symbol, to a new state. That is just a two-index array of states. For sanity and type safety, the states and symbols are declared in Fidep as enumerations.

The Figure 3.3 shows an example of the implemented state machine for the Eucalyptus Node Controller Component; It depicts the actions performed through the four possible states and it allows any customizations needed by the developer/user.

Figure 3.4 shows an example of a test performed in order to assess the behaviour of the hardware controller component, the test intended to stress the implemented state machine and analyze the behavior. After the test the result is that the controller is stable and it did not show any type of instability.

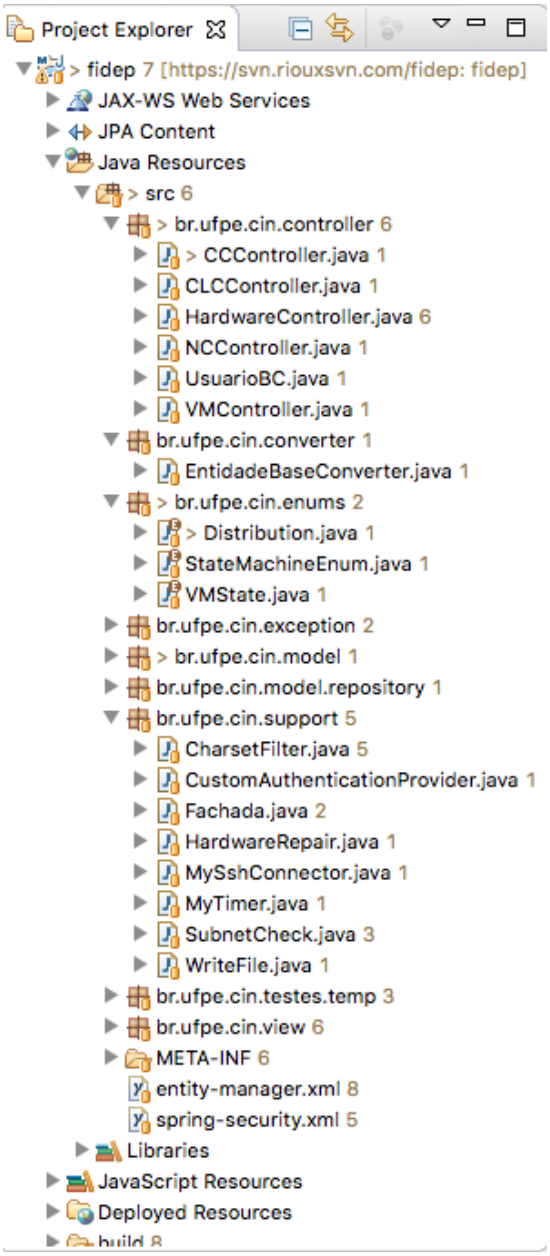

Figure 3.2 Overview of the core Java classes

```
public void runNCStateMachine() throws InterruptedException {
    switch (this.getState()) {
    case RUNNING:
        if (this.isAlive()) {
            int waitingTime = this.generateRandomFailureTime();
            this.setTimer(new MyTimer(waitingTime));
            this.setState(StateMachineEnum.TIMER_INJECT_FAILURE);
            WriteFile.logger("Generated Failure Time: " + waitingTime, "NCController_log.txt");
            WriteFile.logger(new Date().toString(), "NCController_log.txt");
        } else { } // Sleep again until the NC starts
        break;

    case TIMER_INJECT_FAILURE:
        if (this.getTimer().isExpired()) {
            this.stopNC();
            WriteFile.logger("Failed. NC: " + this.getSshConnection().getHost(), "NCController_log.txt");
            WriteFile.logger(new Date().toString(), "NCController_log.txt");
            Thread.sleep(30000); // 30 seconds sleep for the NC to actually stops
            this.setState(StateMachineEnum.FAILED);
        } else { } // Sleep again until the timer expires
        break;

    case FAILED:
        if (!this.isAlive()) {
            int waitingTime = this.generateRandomRepairTime();
            this.setTimer(new MyTimer(waitingTime));
            this.setState(StateMachineEnum.TIMER_REPAIR);
            WriteFile.logger("Generated Repair Time: " + waitingTime, "NCController_log.txt");
            WriteFile.logger(new Date().toString(), "NCController_log.txt");
        } else { } // Sleep again until the NC stops
        break;

    case TIMER_REPAIR:
        if (this.getTimer().isExpired()) {
            this.startNC();
            WriteFile.logger("Started. NC: " + this.getSshConnection().getHost(), "NCController_log.txt");
            WriteFile.logger(new Date().toString(), "NCController_log.txt");
            Thread.sleep(30000); // 30 seconds sleep for the NC to actually starts
            this.setState(StateMachineEnum.RUNNING);
        } else { } // Wait for the timer to expire
        break;
    }
}
```

Figure 3.3 Overview of a sample state machine method for the Node Controller Component

```
public class TestHardwareController {
    public static void main(String[] args) throws InterruptedException {
        System.out.println("Hardware Controller State Machine Testing");

        //SSH credentials to the Hardware
        MySshConnector con = new MySshConnector("root", "cloud1", "192.168.5.3");

        /*Here the user can set any distribution of the enumerator
       randF - Random Failure Time in milliseconds, the first parameter is the minimum value
       the second parameter is the maximum value, and the last one is the parameter of the distribution,
       in this case is the parameter of the exponential distribution, here for testing
       purposes, the values chosen are set between 2 and 10 minutes, mean value is 5 minutes. */
        RandomVariateGenerator randF = new ExponentialRandomVariateGenerator(120000, 600000, 300000);

        // randR - Random Repair Time between 1 and 10 minutes, mean value is 5 minutes
        RandomVariateGenerator randR = new ExponentialRandomVariateGenerator(60, 600, 300);

        HardwareController hw = new HardwareController(con, randF, randR);

        while (true) { //here we want to test the State Machine behavior, for that
                       //purpose we can set up an infinite loop
            hw.runHardwareStateMachine();
            Thread.sleep(50);
        }
    }

}
```

Figure 3.4 Hardware Controller Component Test

CHAPTER 4

Case Studies

In this chapter tow case studies are presented. The proposed framework *Fidep* is used in order to develop a fault injection repair and monitoring tool which is designed and compatible with Eucalyptus version 3.x; Afterwards, an availability evaluation is performed with the support of the fault injection tool implemented, for this case study the performed experiments measures the availability of Eucalyptus' capability to provide virtual machines (VMs), because that is the main service provided by an IaaS such as Eucalyptus; Then the test bed of the experiment is described, also the availability RBD model and finally the results of the experiment are disclosed. The second case study is performed varying the scenarios, also instead of a RBD model, it is proposed a SPN model to represent the target system, and the availability is estimated from the user's point of view, which means that a web service running inside the virtual machine is taken into account.

4.1 Eucalyptus Availability Study

The framework instantiation or experimentation is a fault injection tool for the Eucalyptus Platform [12]; Eucalyptus was chosen because it is widely adopted and it is a renowned academic cloud platform.

By adopting the *Fidep* framework, the implementation of the fault injection tool is very straightforward. The developer has to previously know and test the Eucalyptus platform's basic commands; Such basic commands for all the components basically start, stop and list the component. In this case the Eucalyptus platform provides the Euca-tools and the command *euca-run-instances* to start a new VM, *euca-terminate-instances* to shutdown a given VM, and *euca-describe-instances* to list all the VMs and their status.

The before-mentioned commands are just for the virtual machines components; But there are also other components, such like the *Cloud Controller*, *Cluster Controller*, *Storage Controller*, and so on; Thereby the developer must know and test all the *start/stop/list* commands, so that it is possible to communicate with the correct component in a proper way. Those commands will be used by the communication module of Fidep in order to inject and repair the faults. For the high-level components it can be used the Linux service command.

The main goal of this study is to measure the availability of the service; The service from the user's point of view is the system's capability of instantiating new virtual machines, in other words, the availability of Eucalyptus is determined if the user can still start a new virtual machine after terminating any other already running virtual machines. This experiment shows how reliable is the Eucalyptus capability of providing new VMs. In order to perform this study

it will be used the fault injection tool that was implemented. A fault injection and monitoring based strategy encompasses a workload generator, which is a random fault and repairing activities generator; a fault injector, a system monitor and the target system. The role of the fault injector is to inject the faults into the target system. The workload generator generates commands, faults and repairing actions previously configured, and drives the fault injector, whilst the system monitor observes the target system status and behavior and collects data. In this case, the target system is Eucalyptus.

The failure and repair commands are injected into the target system at run-time. To compute the time between failures and repair, the monitoring module has a clock that records downtime between event activations. As previously explained, system behavior is observed considering two possible states: UP when it is working properly and DOWN, when it is not. Thus, the state of the system oscillates between UP and DOWN according the respective probability distribution of failure and repair.

4.1.1 Testbed Description

The test environment consists of three machines Figure 4.1. An Eucalyptus front-end (Computer 1) which consists in four components: Cloud Controller (CLC), Cluster Controller (CC), Storage Controller (SC), Walrus; A fault injector, repair and monitor (Computer 2), which runs the fault injection tool and keeps track of the experiment by storing all the logs; Computer 3 run the Node Controller (NC) which controls the physical resources and provides the virtual machines (VMs) with the KVM hypervisor [12].

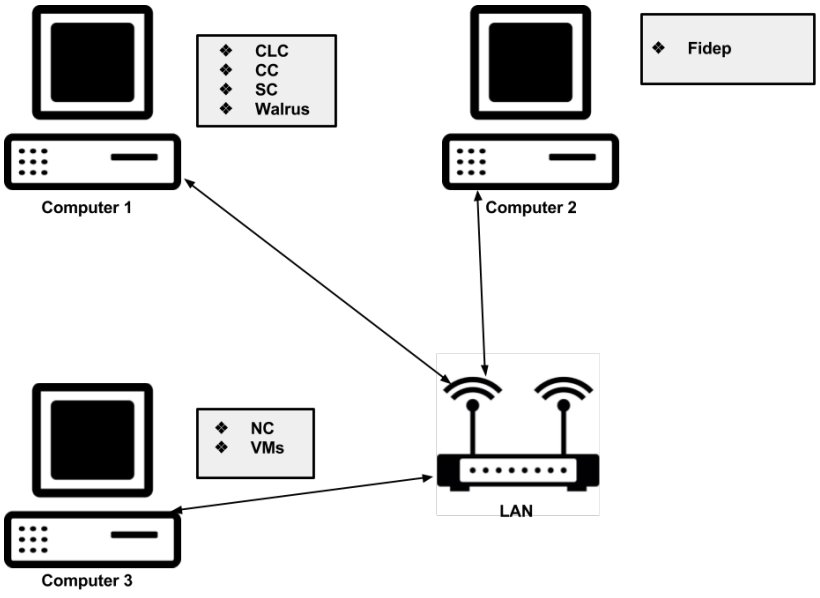

Figure 4.1 Components of the test bed - Case Study I

The Eucalyptus version used in this experiment is Eucalyptus 3.3, running over a CentOS 6. All the physical machines have the same configurations, with 8 GB RAM, Intel i3 Processor

and 500 GB hard disk. The environment was isolated by using a single Gigabit Switch to connect all the machines.

At the end of the experiment there will be a log containing an assortment of UP and DOWN strings. Each of these entries is a measure of server status at a given time. Therefore, it is possible to approximate this discontinuous data to continuous data by normalizing the values between points, and finally estimate the confidence interval of availability. By using Equation 2.11, it is possible to calculate the ρ estimator from the number of UP (uptime or S_n) and DOWN (downtime or Y_n) entries in the log file. The number of errors (n) is easily calculated by comparing the number of UP and DOWN states. Adopting the ρ estimator, the n number of faults, and the confidence level (α), the availability can be computed with the Equations 2.12 and 2.13.

4.1.2 Availability RBD Model

In this subsection it is presented the availability model that represents the target system within the proposed testbed environment; For the purposes of this experiment, the service is the capability to provide a virtual machine, thus from the Eucalyptus' user point of view if Eucalyptus system fails to deliver a new virtual machine, then the monitor module considers that the service is down, otherwise if the virtual machine is instantiated without any issues, then the service is up and running. The maximum number of running virtual machines is taken into account during the whole experiment.

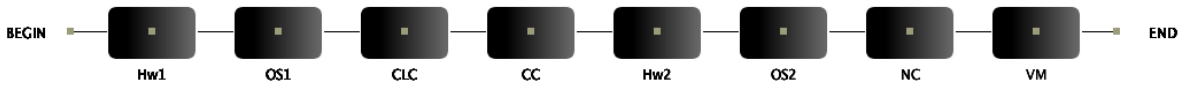

Figure 4.2 RBD model of the target system

Figure 4.2 shows the RBD model. Each block represents one hardware component or software component; from the begin label the Hw1 represents the physical hardware of the computer 1; the OS1 represents the operating system from the computer 1; the CLC represents the Cloud Controller component from Eucalyptus; the CC represents the Cluster Controller from Eucalyptus; the Hw2 represents the hardware from the computer 3; the OS2 represents the operating system from the computer 3; the NC represents the Node Controller component from Eucalyptus; and the VM represents the virtual machine instantiated within the KVM hypervisor, managed by the Node Controller. The computer 2 with the Fidep and the fault injection tool is not part of the cloud system, thus it is not represented in the model.

4.1.3 Results

Here it is discussed the results from the RBD model as well as is disclosed a comparison between the model results and the experiment's results. The case study purpose is to verify the impact of distinct failures and repairs in this environment. The scenario considers one virtual

machine to assume the service operational. The experiment did not include a backup instance to user service. The experiment employed exponential distributions to represent mean time to fail (MTTF) and mean time to repair (MTTR). Finally, the results were compared with the respective results obtained from RBD model. Faults were injected in all testbed components, including the physical machine hardware. This case study aim to evaluate the service availability (running VMs) considering the cloud environment behavior. The scenario employs real estimated parameters, accelerated by 1000, for Eucalyptus high-level components and hardware [15] (see Table 4.1).

Component	Experimental MTTF	Experimental MTTR	Real MTTF	Real MTTR
Hardware	31536 s	6 s	8760 h	100 min
Cloud Controller	2838 s	1 s	788 h	1 h
Cluster Controller	2838 s	1 s	788 h	1 h
Node Controller	2838 s	1 s	788 h	1 h
Virtual Machine	10414 s	1 s	2893 h	15 min

Table 4.1 Parameters of the Experiment Scenario

The experiment was executed during a 24 hours period, and it was repeated twice. The log files generated by the monitoring module shows a total up-time of the system of 23,93 hours, that means that the Eucalyptus was responding correctly and it was able to run a virtual machine for the majority of the time of the experiment; thus the remaining 4,2 minutes were downtime, which means that Eucalyptus was not able to run a virtual machine for such period. During the 24 hours of experiment a total of 14 faults were generated, injected and repaired.

The RBD model estimated availability for the system was 0.9935857; After straightening the data from the log file, by calculating the availability and confidence interval, the result is the following confidence interval of availability (0.993420433 , 0.998626244). As the estimated availability, from the RBD model, is contained in the confidence interval of availability from Fidep's experimentation, then there is no evidence to reject the hypothesis that the results are equivalent for this case study.

4.2 Web Service Availability Study

This section presents the adopted Eucalyptus infrastructure and the proposed availability model. Although the depicted model is presented for a specific Eucalyptus configuration, it is generic enough to be extended for other arrangements. The SPN model presented in this subsection was proposed by [10]. An experiment is performed in order to to assess Eucalyptus availability, but the virtual machines life cycle is also taken into account, which means that there will be different scenarios and each scenario will determine the number of virtual machines that must be running/available in order for us to consider that the service is operational.

4.2.1 Testbed Description

The test environment consists of four machines Figure 4.3. A fault injector and monitor (Machine 4). Front-end (Machine 1) that contains the Eucalyptus Cloud Controller (CLC) and Cluster Controller (CC). Machines 2 and 3 execute Eucalyptus Node Controller (NC and KVM hypervisor) and host virtual machines.

The Eucalyptus version used in this experiment is Eucalyptus 3.3, running over a CentOS 6. All the physical machines have the same configurations, with 8 GB RAM, Intel i3 Processor and 500 GB hard disk. The environment was isolated by using a single Gigabit Switch to connect all the machines.

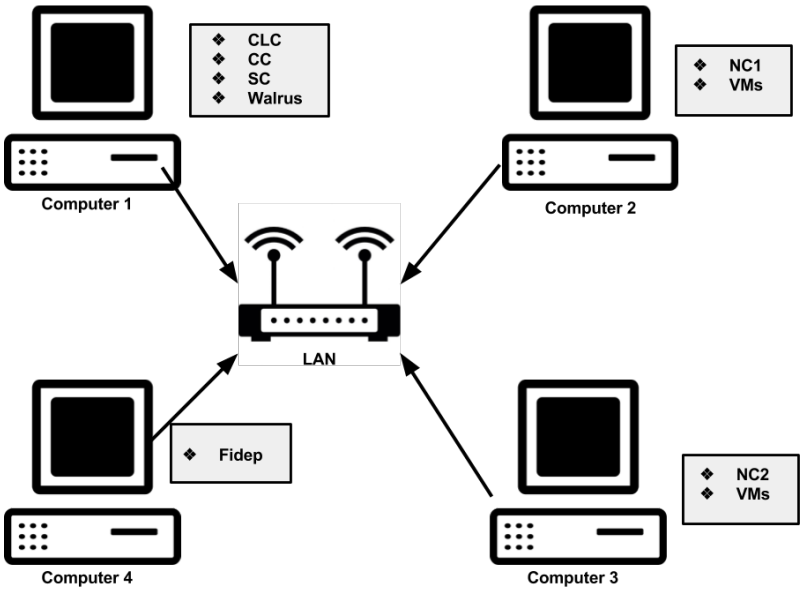

Figure 4.3 Components of the test bed - Case Study II

All servers are on the same private network and the fault injection tool can access other components/machines using SSH connection. The computer 4 also controls the monitoring scripts. With this testbed environment, faults can be injected in the main Eucalyptus resources (e.g., physical and virtual machines), as well as service status can be traced by means of monitoring scripts.

Once again, it is important to understand that at the end of the experiment there will be a log file; Such log file was generated by the monitoring scripts. The log file contains an group of UP and DOWN lines. Each of these entries is a measure of the service status at a given time. Hence, it is possible to approximate this discontinuous data to continuous data by normalizing the values between points, and finally estimate the confidence interval of availability of the service. By using Equation 2.11, it is possible to calculate the ρ estimator from the number of UP (uptime or S_n) and DOWN (downtime or Y_n) entries in the log file. The number of errors (n) is easily calculated by comparing the number of UP and DOWN states. Adopting the ρ estimator, the n number of faults, and the confidence level (α), the availability can be computed with the Equations 2.12 and 2.13.

4.2.2 Availability SPN Model

This section presents the SPN availability model for representing the proposed testbed environment. This model is divided into five main parts: Eucalyptus Front-End Services for modeling CC, CLC and Frontend hardware (Machine 1) Figure 4.6; NC1 SERVER Figure 4.4 and NC2 SERVER Figure 4.5 for representing the hardware and software of Machines 2 and 3; VMS NC1 and VMS NC2 that correspond to running VMs on Machines 2 and 3 are all represented by the model depicted in Figure 4.7. As the components of Machine 4 (Fault injection tools and monitoring scripts) are not part of the cloud system, so these entities are not represented in the model. The source of the Figures 4.6, 4.4, 4.5, 4.7 is [10].

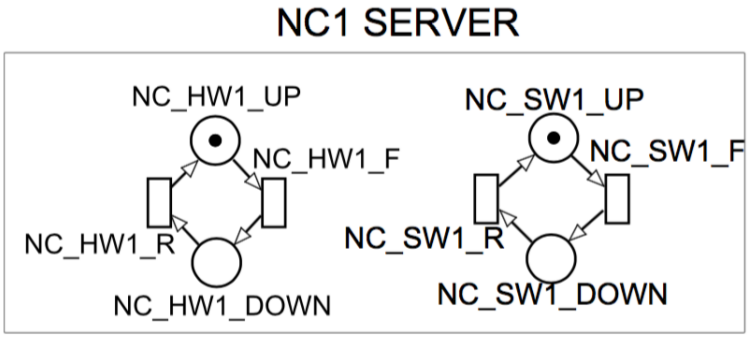

Figure 4.4 SPN model of the target system Node Controller 1. [1]

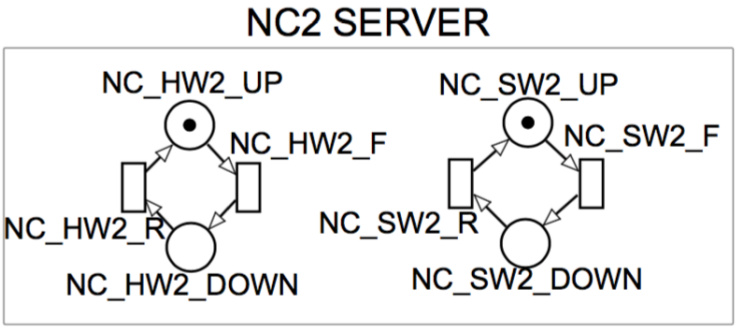

Figure 4.5 SPN model of the target system Node Controller 2 [1]

Components with no redundancy mechanisms or dependency relations, e.g. NC hardware, are represented as submodels Figure 4.8, which are composed of two states and two transitions. Assuming CC as the represented component in the Figure 4.8, this submodel might be in two states, it can be operational (CC_UP) or it can be failed (CC_DOWN). Transitions *CC_F* and *CC_R* denote respectively the component's failure action and the component's repair action.

This submodel has two parameters, yet not shown in Figure 4.8, namely *X_MTTF* and *X_MTTR*, which represent delays associated to transitions *X_F* and *X_R*, respectively. The Table 4.2 depicts the attributes related to these transitions. In this context, a component is working if there is no tokens in place *X_DOWN*. Therefore, to represent a failed component,

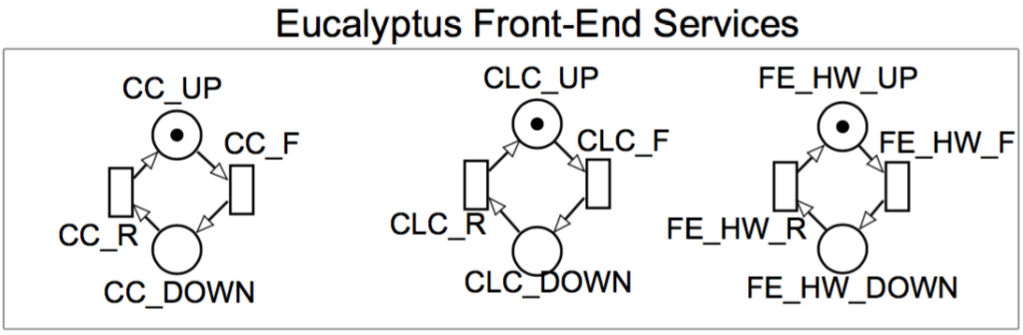

Figure 4.6 SPN model of the Eucalyptus front services [1]

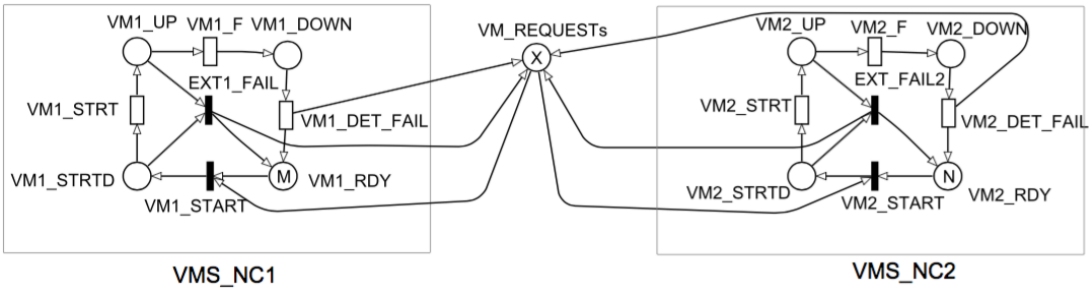

Figure 4.7 SPN model of the target system - VMs dependency [1]

the number of tokens in *X_UP* must be zero. Table 4.3 shows the submodels adopted in Figures 4.6, 4.4, 4.5, and, 4.7 for representing components with no dependency relations.

Transition	Delay	Description
X_F	MTTF	Component failure event
X_R	MTTR	Component repair event

Table 4.2 Transition attributes associated with a component

Submodel's name	Description
NC_HW1	NC1's hardware
NC_SW1	NC1's software
NC_HW2	NC2's hardware
NC_SW2	NC2's software
FE_HW	Front end's hardware
CC	Cluster controller hardware
CLC	Cloud controller hardware

Table 4.3 Submodels for representing no redundancy components

In Figure 4.7, VMS_NC1 and VMS_NC2 submodels represent the set of VMs that run on NC1 and NC2 servers, and whenever a dependent device, e.g. underlying hardware, fails, then

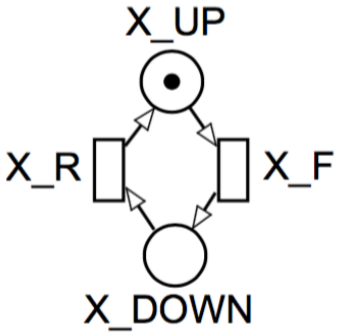

Figure 4.8 Basic component - SPN submodel [1]

the respective VMs fail too. VMS_NC1 and VMS_NC2 are two analogous behavior, then just VMS_NC1's structure will be detailed bellow. VMS_NC1 is composed of places VM1_UP, VM1_DOWN, VM1_STRTD and VM1_RDY. These places denote, respectively, the amount of VMs in states operational, failed, starting, and waiting for request. The Place VM_REQUESTs represent a number of requested VMs, which can be executed on NC1 or NC2 server.

Exponential transitions VM1_DET_FAIL, VM1_F and VM1_STRT model the fault detection time, failure and starting activities related to NC1 virtual machines, see Table 4.4. The association with the underlying infrastructure is carried out by immediate transitions EXT1_FAIL and VM1_START, and the respective guard conditions of such transitions are shown in Table 4.5. It is important to state that the virtual machine stops working whenever the respective physical machine fails. To start a VM, the respective physical machine, NC, CC, CLC and front-end machine must be operational. Therefore, EXT1_FAIL fires if the respective physical machine fails. Transition VM1_START denotes the opposite idea, in the sense that virtual machines start only if the required infrastructure is operational.

Transition	Delay	Description
VM1_F	VM1_MTTF	VM fault event
VM1_DET_FAIL	Detection_Time	VM fault detection event
VM1_STRT	VM_Start_Time	VM Start event

Table 4.4 Transitions of the VM life-cycle model

The variables X, M and N represent respectively the number of requested VMs, the maximum number of running VMs on NC1 and NC2 servers. Considering the presented environment, the values of M and N are the same and equal to four. The availability is calculated based on the total amount of virtual machines running in both node controllers. Consequently, the availability is estimated as $P\{(\#VM1\_UP + \#VM2\_UP) \geq X\}$.

4.2.3 Results

The case study purpose is to verify the impact of distinct failures and repairs in this environment, and show that it is possible to employ a fault injection tool to evaluate diverse scenarios

(in this case A and B). Each scenario presents two testing rounds. The first round considered only one VM to assumes the service operational (A1 and B1) and the second scenario assumes two VMs (A2 and B2), which means that in the second scenario if there is just one of the virtual machines running then the service is not considered to be operational. No experiment included a backup instance to user service and both employed exponential distributions to represent mean time to fail and mean time to repair. Finally, the results were compared with the respective results obtained from the previously mentioned SPN models.

Transition	Condition
VM1_START	(#*NC*_HW1_UP > 0) AND (#*NC*_SW1_UP > 0) AND (#*CC*_UP > 0) AND (#*CLC*_UP > 0) AND (#*FE_HW_UP* > 0) AND (#*VM*1_RDY > 0) AND (#*VM*_REQUESTs > 0)
EXT1_FAIL	(#*NC*_HW1_UP = 0) AND (#*VM*1_UP + #*VM*1_STRTD) > 0

Table 4.5 Condition to enable the immediate transitions

Faults were injected in all testbed components, including the physical machine hardware. This case study aims to evaluate the service availability, which means the running VMs, considering the cloud environment behavior. The experimental study consists of two distinct scenarios. Each scenario was submitted for two testing rounds. The first scenario (A) employs real estimated parameters (accelerated by 1000) for Eucalyptus high-level components and hardware [15] see Table 4.1.

Scenario B, as detailed in Table 4.6, is a hypothetical case where the MTTF is closer to the MTTR value. Therefore, a lower value for availability would be expected. In a real situation, it would be expected that a higher availability value would be demonstrated in the situation where there is less reliance on virtual machines.

Component	Experimental MTTF	Experimental MTTR	Real MTTF	Real MTTR
Hardware	5600 s	1200 s	1555 h	333 h
Cloud Controller	7200 s	1200 s	2000 h	333 h
Cluster Controller	5600 s	600 s	1555 h	166 h
Node Controller 1	3600 s	600 s	1000 h	166 h
Node Controller 2	2700 s	300 s	750 h	83 h
Virtual Machine	2700 s	900 s	750 h	250 h

Table 4.6 Parameters of Scenario B

Table 4.7 shows uptime and downtime results of the experiments executed over a 24 hour period. As expected, the scenarios running more virtual machines to serve the client's application lost more time with faults. This behavior is easily understood since each VM is another component that the user's application depends on and therefore faults here would further impact the total downtime.

It is important to highlight that the measured values occur over a small time frame and so if scaled up even the difference of 2 minutes between A1 and A2 would have a major impact on the user's service. In Scenario B, with smaller differences between repair and failure times, this behavior is even more apparent. In this case, a comparison of B1 (dependent on one VM) with B2 (dependent on two VMs) exhibits an increase in downtime of over 266 percent.

Scenario	Number of VMs	Uptime	Downtime
A1	1	23,93 h	4,2 min
A2	2	23,89 h	6,6 min
B1	1	21,98 h	121,2 min
B2	2	18,66 h	320,4 min

Table 4.7 Uptimes and Downtimes from the experiment

Scenario	Number of Faults	Confidence Interval of Availability	Estimated Availability
A1	14	(0.993420433 , 0.998626244)	0.9935857
A2	20	(0.987658509 , 0.993537325)	0.9879326
B1	9	(0.807462045 , 0.965819035)	0.8279075
B2	20	(0.650996494 , 0.867691751)	0.7921259

Table 4.8 Availability Evaluated from the Experiments

In Table 4.8, the data has been straightened by calculating the availability and confidence interval. As predicted, Scenario B produced lower availability values and greater confidence intervals than scenario A. Due to the low rate of failure in Scenario B1, the impact of one more dependence was greater than it was in scenario A. The last column provides the availability metric estimated with the respective SPN model. As the estimated results by the SPN models are contained in the results confidence interval, then there is no evidence to reject the hypothesis that the results are also equivalent for this case study.

CHAPTER 5

Conclusions

Although the creation of event generation and fault injection tools is not the main activity during the design process of a new software product, it must be taken into account that the need to implement such functionalities can lead to delays in the delivery of the software system. One of the ways to minimize this disorder is through the use of frameworks.

This work presented Fidep which is a framework for linux based Cloud computing systems; By using this framework a developer is able to implement a fault injection repair and monitoring tool for linux distributions; Also by adopting Fidep the developer is able to implement a fault injection tool that intends to provide support for dependability evaluations.

For the development of the framework, initially, the literature was sought the necessary knowledge of how to develop a framework, its positives, and negatives, main differences between frameworks and libraries, means of implementation, flexibilization through hotspots and validation forms.

It was shown that the Fidep framework can be of important assistance to the cloud provider administrator in the support of calculation of dependability metrics by supporting availability prediction and providing more information concerning SLAs for future users. This work also demonstrated how to adopt RBD as well as SPN models for representing IaaS cloud environments. Therefore, cloud designers can decide which technique, modelling or fault injection, should be adopted to evaluate dependability in their IaaS cloud environments.

The case studies results show how Fidep can be used to test the Eucalyptus cloud environment and be an important tool for supporting maintenance operation plans and architectural changes beneath the infrastructure. The results show that either model-based and fault injection techniques can be adopted to support dependability evaluation in IaaS systems.

5.1 Future Works

Considering the limitations of this work, there are several improvements to be done. The intention is to look at other component types within the Cloud System's infrastructure that have not been explored at this time; for example, the hypervisor process within the OS; another example that can be affected by faults is the networking equipment. Another point to be taken into account in the future work is the new concept of hybrid clouds, as well as distributed data centers, a further intention is to expand the study of virtual machines life-cycles to incorporate migration activities, because fault injection techniques can be utilized to perform analysis in virtual machine migrations.

Bibliography

[1] Amazon S3 (2013). Amazon simple storage service (amazon s3). `http://aws.amazon.com/s3/`.

[2] Dropbox (2013). Dropbox. `http://www.dropbox.com`.

[3] Marc D Alexander and Peter A Woytovech. Using rtc wake-up to enable recovery from power failures, June 18 2002. US Patent 6,408,397.

[4] Jean Araujo, Rubens de S Matos, Paulo Romero Martins Maciel, and Rivalino Matias. Software aging issues on the eucalyptus cloud computing infrastructure. In *SMC*, pages 1411–1416, 2011.

[5] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[6] Amazon AWS. Amazon web services (amazon aws). `http://aws.amazon.com/`.

[7] Daniel J Barrett and Richard E Silverman. *SSH, the Secure Shell: the definitive guide*. " O'Reilly Media, Inc.", 2001.

[8] Gunter Bolch, Stefan Greiner, Hermann de Meer, and Kishor S Trivedi. *Queueing networks and Markov chains: modeling and performance evaluation with computer science applications*. Wiley-Interscience, New York, 1998.

[9] Robert Bradford, Evangelos Kotsovinos, Anja Feldmann, and Harald Schiöberg. Live wide-area migration of virtual machines including local persistent state. In *Proceedings of the 3rd international conference on Virtual execution environments*, pages 169–179. ACM, 2007.

[10] Jonathan Brilhante, Bruno Silva, Paulo Maciel, and Armin Zimmermann. Dependability models for eucalyptus infrastructure clouds considering vm life-cycle. In *2014 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 1336–1341. IEEE, 2014.

[11] J Carolan and S Gaede. Introduction to cloud computing architecture, sun microsystems inc. white paper. *Sun Microsystems Inc., Santa Clara*, 2009.

[12] Eucalyptus Cloud. Official documentation for eucalyptus cloud. `http://docs.hpcloud.com/eucalyptus`.

[13] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[14] Oracle Corporation. Official documentation java ee - overview. `http://www.oracle.com/technetwork/java/javaee/overview/index.html`.

[15] Jamilson Dantas, Rubens Matos, Jean Araujo, and Paulo Maciel. An availability model for eucalyptus platform: An analysis of warm-standy replication mechanism. In *Systems, Man, and Cybernetics (SMC), 2012 IEEE International Conference on*, pages 1664–1669. IEEE, 2012.

[16] Paul Deitel and Harvey M Deitel. *Java™ for Programmers*. Prentice Hall Professional, 2011.

[17] Luc Devroye. Sample-based non-uniform random variate generation. In *Proceedings of the 18th conference on Winter simulation*, pages 260–265. ACM, 1986.

[18] Hoang T Dinh, Chonho Lee, Dusit Niyato, and Ping Wang. A survey of mobile cloud computing: architecture, applications, and approaches. *Wireless communications and mobile computing*, 13(18):1587–1611, 2013.

[19] Charles E. Ebeling. *An Introduction to Reliability and Maintainability Engineering*. Waveland Press, Inc., 1997.

[20] IDE Eclipse. Eclipse foundation, 2007.

[21] Alireza Ejlali, Seyed Ghassem Miremadi, Hamidreza Zarandi, Ghazanfar Asadi, and Siavash Bayat Sarmadi. A hybrid fault injection approach based on simulation and emulation co-operation. page 479. IEEE, 2003.

[22] Paul Fortier and Howard Michel. *Computer systems performance evaluation and prediction*. Digital Press, 2003.

[23] The Apache Software Foundation. Apache cloudstack: Open source cloud computing. `https://cloudstack.apache.org/`.

[24] Hugo ES Galindo, Erica AC Guedes, Paulo RM Maciel, Bruno Silva, and Sérgio ML Galdino. Wgcap: a synthetic trace generation tool for capacity planning of virtual server environments. In *Systems Man and Cybernetics (SMC), 2010 IEEE International Conference on*, pages 2094–2101. IEEE, 2010.

[25] Hugo ES Galindo, Wagner M Santos, Paulo RM Maciel, Bruno Silva, Sérgio ML Galdino, and José Paulo Pires. Synthetic workload generation for capacity planning of virtual server environments. In *Systems, Man and Cybernetics, 2009. SMC 2009. IEEE International Conference on*, pages 2837–2842. IEEE, 2009.

[26] Reinhard German. *Performance analysis of communication systems with non-Markovian stochastic Petri nets*. John Wiley & Sons, Inc., 2000.

[27] Rahul Ghosh, Kishor S Trivedi, Vijay K Naik, and Dong Seong Kim. End-to-end performability analysis for infrastructure-as-a-service cloud: An interacting stochastic models approach. In *Dependable Computing (PRDC), 2010 IEEE 16th Pacific Rim International Symposium on*, pages 125–132. IEEE, 2010.

[28] Sebastien Godard. Sysstat utilities home page, 2010.

[29] Robert D. Hof. Jeff bezos' risky bet. `http://www.bloomberg.com/bw/stories/2006-11-12/jeff-bezos-risky-bet`. Accessed: 2015-12-22.

[30] James Holmes and Chris Schalk. *JavaServer faces: the complete reference*. McGraw-Hill, Inc., 2006.

[31] Raj Jain. *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling*. John Wiley & Sons, 1990.

[32] D Johnson, K Murari, M Raju, RB Suseendran, and Y Girikumar. Eucalyptus beginner's guide–uec edition, css corp. 2010, 2010.

[33] Rod Johnson, Juergen Hoeller, Keith Donald, Colin Sampaleanu, Rob Harrop, Thomas Risberg, Alef Arendsen, Darren Davison, Dmitriy Kopylenko, Mark Pollack, et al. The spring framework–reference documentation. *Interface*, 21, 2004.

[34] Dong Seong Kim, Fumio Machida, and Kishor S Trivedi. Availability modeling and analysis of a virtualized system. In *Dependable Computing, 2009. PRDC'09. 15th IEEE Pacific Rim International Symposium on*, pages 365–371. IEEE, 2009.

[35] Leonard Kleinrock. Queueing systems, volume i: theory. 1975.

[36] Douglas Kramer. Api documentation from source code comments: a case study of javadoc. In *Proceedings of the 17th annual international conference on Computer documentation*, pages 147–153. ACM, 1999.

[37] Way Kuo and Ming J Zuo. *Optimal reliability modeling: principles and applications*. John Wiley & Sons, 2003.

[38] Avraham Leff and James T Rayfield. Web-application development using the model/view/controller design pattern. In *Enterprise Distributed Object Computing Conference, 2001. EDOC'01. Proceedings. Fifth IEEE International*, pages 118–127. IEEE, 2001.

[39] Ang Li, Xiaowei Yang, Srikanth Kandula, and Ming Zhang. Cloudcmp: Comparing public cloud providers. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10, pages 1–14, New York, NY, USA, 2010. ACM.

[40] David J Lilja. *Measuring computer performance: a practitioner's guide*. Cambridge university press, 2005.

[41] Nik Looker, Malcolm Munro, and Jie Xu. Ws-fit: A tool for dependability analysis of web services. pages 120–123. IEEE, 2004.

[42] Paulo RM Maciel, Rafael D Lins, and Paulo RF Cunha. *Introdução às redes de Petri e aplicações*. UNICAMP-Instituto de Computacao, 1996.

[43] Andrey Andreyevich Markov. Extension of the law of large numbers to dependent quantities. *Izv. Fiz.-Matem. Obsch. Kazan Univ.(2nd Ser)*, 15:135–156, 1906.

[44] Marco Ajmone Marsan, Gianfranco Balbo, Gianni Conte, Susanna Donatelli, and Giuliana Franceschinis. *Modelling with generalized stochastic Petri nets*. John Wiley & Sons, Inc., 1994.

[45] Rubens Matos, Jean Araujo, Danilo Oliveira, Paulo Maciel, and Kishor Trivedi. Sensitivity analysis of a hierarchical model of mobile cloud computing. *Simulation Modelling Practice and Theory*, 50:151–164, 2015.

[46] M Mattsson. Object-oriented frameworks-a survey of methodological issues", licentiate thesis, department of computer science, lund university, coden: Lutedx/(tecs-3066)/1-130/(1996). Technical report, also as Technical Report, LU-CS-TR: 96-167, Department of Computer Science, Lund University, 1996.

[47] Michael Mattsson and Jan Bosch. Framework composition: Problems, causes and solutions. In *Technology of Object-Oriented Languages and Systems, 1997. TOOLS 23. Proceedings*, pages 203–214. IEEE, 1997.

[48] In Maven. Apache maven project, 2011.

[49] Daniel A Menasce, Virgilio AF Almeida, Lawrence W Dowdy, and Larry Dowdy. *Performance by design: computer capacity planning by example*. Prentice Hall Professional, 2004.

[50] Philip Merlin and David Farber. Recoverability of communication protocols-implications of a theoretical study. *IEEE transactions on Communications*, 24(9):1036–1043, 1976.

[51] Jerre D Noe and Gary J Nutt. Macro e-nets for representation of parallel systems. *IEEE Transactions on Computers*, 100(8):718–727, 1973.

[52] Mohammed S Obaidat and Noureddine A Boudriga. *Fundamentals of performance evaluation of computer and telecommunications systems*. John Wiley & Sons, 2010.

[53] Carl Adam Petri. Kommunikation mitautomaten. *Bonn: Institut für Instrumentelle Mathematik, Schriften des NM*, (3), 1962.

[54] Dhiraj K Pradhan. *Fault-tolerant computer system design*. Prentice-Hall, 1996.

[55] OpenNebula Project. Opennebula - flexible enterprise cloud made simple. `http://opennebula.org/`.

[56] OpenStack Project. Openstack - open source cloud computing software. `https://www.openstack.org/`.

[57] Mark Shacklette. Linux operating system. *Handbook of Computer Networks: LANs, MANs, WANs, the Internet, and Global, Cellular, and Wireless Networks, Volume 2*, pages 78–90, 1995.

[58] Débora Souza, Rubens Matos, Jean Araujo, Vandi Alves, and Paulo Maciel. A tool for automatic dependability test in eucalyptus cloud computing infrastructures. *Computer and Information Science*, 6(3):57, 2013.

[59] Kishor S Trivedi, Steve Hunter, Sachin Garg, and Ricardo Fricks. Reliability analysis techniques explored through a communication network example. 1996.

[60] Hong-Linh Truong and Schahram Dustdar. On analyzing and specifying concerns for data as a service. In *Services Computing Conference, 2009. APSCC 2009. IEEE Asia-Pacific*, pages 87–94. IEEE, 2009.

[61] Juliano C Vacaro and Taisy S Weber. Injeção de falhas na fase de teste de aplicações distribuídas. *Anais do Simp. Bras. de Eng. de Software, SBES*, pages 161–176, 2006.

[62] Wil MP Van Der Aalst, Kees M Van Hee, and Hajo A Reijers. Analysis of discrete-time stochastic petri nets. *Statistica Neerlandica*, 54(2):237–255, 2000.

[63] Luis M Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A break in the clouds: towards a cloud definition. *ACM SIGCOMM Computer Communication Review*, 39(1):50–55, 2008.

[64] William Voorsluys, James Broberg, Srikumar Venugopal, and Rajkumar Buyya. Cost of virtual machine live migration in clouds: A performance evaluation. In *Cloud Computing*, pages 254–265. Springer, 2009.

[65] Wendai Wang and Dimitri B Kececioglu. Confidence limits on the inherent availability of equipment. In *Reliability and Maintainability Symposium, 2000. Proceedings. Annual*, pages 162–168. IEEE, 2000.

[66] Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of internet services and applications*, 1(1):7–18, 2010.

[67] Tiange Zhang, Xiaochun Xiao, and Leqiu Qian. Modeling object-oriented framework with z. In *Computer Science and Computational Technology, 2008. ISCSCT'08. International Symposium on*, volume 2, pages 165–170. IEEE, 2008.

[68] Yunjia Zhang, Bin Liu, and Qing Zhou. A dynamic software binary fault injection system for real-time embedded software. In *Reliability, Maintainability and Safety (ICRMS), 2011 9th International Conference on*, pages 676–680. IEEE, 2011.

[69] Ming Zhao and Renato J Figueiredo. Experimental study of virtual machine migration in support of reservation of cluster resources. In *Proceedings of the 2nd international workshop on Virtualization technology in distributed computing*, page 5. ACM, 2007.

[70] Haissam Ziade, Rafic A Ayoubi, Raoul Velazco, et al. A survey on fault injection techniques. *Int. Arab J. Inf. Technol.*, 1(2):171–186, 2004.