Computer and Information Science; Vol. 6, No. 3; 2013
ISSN 1913-8989 E-ISSN 1913-8997
Published by Canadian Center of Science and Education

A Tool for Automatic Dependability Test in Eucalyptus Cloud Computing Infrastructures

Débora Souza[1], Rubens Matos[1], Jean Araujo[1,2], Vandi Alves[1] & Paulo Maciel[1]

[1] Center of Informatics, Federal University of Pernambuco, Recife, Brazil

[2] Academic Unit of Garanhuns, Federal Rural University of Pernambuco, Garanhuns, Brazil

Correspondence: Paulo Maciel, Center of Informatics, Federal University of Pernambuco, Recife, Brazil. Tel: 55-81-2126-8430 ext. 4009. E-mail: prmm@cin.ufpe.br

Received: April 2, 2013 Accepted: May 3, 2013 Online Published: May 22, 2013

doi:10.5539/cis.v6n3p57 URL: http://dx.doi.org/10.5539/cis.v6n3p57

Abstract

Cloud Computing is a paradigm that dynamically provides resources as services through the Internet. The constant concern about the trust placed in cloud computing systems inspires dependability studies. A possible way of performing dependability studies, especially regarding reliability and availability, is through fault injection tools, which enable to observe the system's behavior during the occurrence of fault events. This paper presents a fault injection tool, called EucaBomber, for reliability and availability studies in the Eucalyptus cloud computing platform. The tool supports fault injections in Eucalyptus hardware and software components at runtime, and also upholds reparation of both types of injected faults. The efficiency of EucaBomber is tested through a case study involving two different scenarios where faults and repairs of hardware and software are injected in the Eucalyptus platform simulating the system's events. Such a tool assists the system administrator and planners to evaluate the system's availability and maintenance policies.

Keywords: cloud computing, dependability evaluation, fault injection

1. Introduction

Cloud Computing is a computational paradigm which is in continuous development and was born from the combination of different technologies such as grid computing, cluster computing, utility computing and virtualization (Sousa et al., 2012). This technology allows the user to dispose of dynamically scalable resources in the form of services (Furht & Escalante, 2010), through standard interfaces and web-based protocols (Eucalyptus, 2013). This feature allows users to focus on their daily activities without worrying about spending efforts and funds with processes such as maintenance or upgrading equipment.

Cloud service providers must offer assurances that the services supplied meet requirements such as availability and security at a reasonable cost. In this case, one of the difficulties is to ensure continuity of service in the face of possible system failures. The impossibility of eliminating all negative effects related to failures of a cloud inspires dependability studies. Those studies help to evaluate the quality and the reliability of the system as well as to elaborate service level agreements (SLAs).

A possible way of testing the reliability and availability of cloud services is through fault injection tools. We can use such tools to create and observe situations of failures in laboratory, thus avoiding interrupting or disturbing the tasks of an operational cloud system. The observation of the system behavior during occurrence of faults can significantly contribute to the planning and improvements of cloud infrastructures and its maintainability.

The Eucalyptus framework (Eucalyptus, 2013) is an open-source cloud environment that implements private and hybrid clouds as Infrastructure-as-a-Service (IaaS). It is interface-compatible with the EC2 and S3 Amazon services (Amazon, 2013), enabling to test applications in private infrastructures before deploying on Amazon, or easily deploy in the private cloud the applications that already run on the public infrastructure. The Eucalyptus architecture is based on five high-level components, which together constitute a cloud system that provides the environment's control and management. However, the occurrence of failures in one of the components can disturb the correct operation of the system with different degrees of impact, depending on the specific failed device and the deployed architecture.

In order to enable support for dependability studies, focusing in reliability and availability aspects in the

Eucalyptus platform, we developed a tool for fault injection. We proposed a tool that is able to generate fault events at runtime, disrupting the processes of high-level components of the Eucalyptus platform, and is capable to interfere in the functioning of the hardware that hosts the environment, and also enables the repair of such failures after simulating the repair time period. The current version of the tool permits the injection of failures and repairs into the Eucalyptus platform only. Although, the proposed application can be easily adapted to deal with other cloud systems that can be accessed through the SSH2 protocol. The focus in this research is to guide the deployment and maintenance of cloud infrastructures, enabling a system administrator to plan the acquisition of hardware and software with higher reliability, as well as improvements in the maintenance procedures in order to increase the availability of the cloud environment.

This paper is organized as follows. Section 2 provides an overview about dependability, fault injection techniques and cloud computing. The EucaBomber is presented in detail in Section 3, as well as the methodology of failure generation and the operating mode. Case studies are presented in Section 4, demonstrating results obtained using the EucaBomber on a Eucalyptus infrastructure testbed. Finally Section 5 concludes the paper and discusses future works.

2. Background

This section presents a synthesis of concepts related to dependability, fault injection and the Eucalyptus framework, which are fundamental to understand this work. Along with the concepts, this section also mentions some related works regarding fault injection and its applicability.

2.1 Dependability Concepts

Many of the wished features of a cloud system are related to the concept of dependability. There is no unique definition of dependability (Nurmi et al., 2009; Avizienis et al., 2004). By one definition, it is the ability of a system to deliver the required specific services that can justifiably be trusted (Nurmi et al., 2009).

Dependability basically owns six metrics: reliability, availability, safety, confidentiality, integrity and maintainability.

Reliability is the probability that the system does not fail up to instant t, assuming that it operational at $t=0$. The simplest definition of Availability is expressed as the ratio of the expected system uptime to the expected system up and downtimes. Safety may be defined as absence of catastrophic consequences on the user(s) and the environment. Confidentiality is the absence of unauthorized disclosure of information. Integrity is absence of improper system state alterations; and Maintainability may be thought as the ability to undergo repairs and modifications (Avizienis et al., 2004).

Cloud computing is a large-scale distributed computing paradigm and its applications are accessible anywhere, anytime, and in anyway, dependability in cloud system becomes more important and more difficult to achieve (Sun et al., 2010).

2.2 Fault Injection Techniques

Fault injection can be used to study systems that need high levels of reliability (Zhang et al., 2011; Ejlali et al., 2003). Ziade et al. (2004) presents a survey on fault injection techniques and explains some supporting tools. Zhang et al. (2011) presents an injector called DDSFIS (Debug-based Dynamic Software Fault Injection System) wich aims at real time embedded systems. Friesenbichler et al. (2010) presents an approach in QDL (Quasi Delay-Insensitive) circuits where its operation uses saboteurs placed in the circuits.

According to Ziade et al. (2004), fault injection techniques can be divided into five main areas:

• Hardware-based fault injection: requires additional hardware for fault injection at a physical level. The interference provided by this technique may change the electrical characteristics of the circuit, e.g., modifying the voltage attributed to pins. Generally this technique is classified in two subcategories, with or without physical contact. The injector with contact has at least one direct connection with the system to be tested. An injector without contact takes advantage of other means to produce the desired events, such as electromagnetic induction.

• Software-based fault injection: it is a more flexible approach than the previous one since it can be targeted to software applications, operating systems and hardware. However, this approach is limited up to where software commands can reach. This technique allows the injection of faults that can occur in two different moments: (a) *compile-time*, the application is modified before it is loaded and executed, errors are injected in the source code of the target application emulating effects of faults; (b) *run-time*, it requires a specific software to trigger the faults. One of the mechanisms commonly used to trigger fault events is known as *time-out*. This method uses a

timer (hardware or software) to control the injection of the failure. When the predetermined time ends, the fault is injected.

• Simulation-based fault injection: it involves the creation of models for fault injection, since only the model of the system is ready or available to perform the tests. Baraza et al. (2000) present an automatic and model-independent tool based in VHDL (VHSIC Hardware Description Language). This tool is used in IBM-PC or compatible systems that are aimed to inject faults into VHDL models.

• Emulation-based fault injection: it deals with some limitations imposed by the simulation like the time spent during a simulation-based fault injection. This technique uses FPGAs (Field Programmable Gate Arrays) to accelerate the failure simulation. Weinkopf et al. (2006) propose a model for non-classical faults emulation besides a generic environment that supports different models in this scope. This environment is called PARSIFAL (Platform for Analysis and Reduction of Safety-critical Implementations) and can be adapted to perform different emulation systems.

• Hybrid fault injection: mix between two or more fault injection techniques previously mentioned. Ejlali et al. (2003) demonstrate the use of this technique through cooperation between simulator and emulator fault injection. The authors present the tool FITSEC (Fault Injection Tool based on Simulation and Emulation Cooperation) based on both Verilog and VHDL. FITSEC aims at supporting the entire process of system design.

2.3 Eucalyptus Framework: An Overview

Eucalyptus (2013) is a software that implements scalable IaaS-style private and hybrid clouds (Eucalyptus, 2009) and is interface-compatible with the commercial services Amazon EC2 and S3 (Eucalyptus, 2013; Amazon, 2013). This API compatibility enables one to run an application on Amazon and on Eucalyptus without modification. In general, the Eucalyptus cloud-computing platform uses the virtualization capabilities (hypervisor) of the underlying computer system to enable flexible allocation of resources decoupled from specific hardware. Eucalyptus architecture is composed of five high-level components, each one with its own web service interface: Cloud Controller, Node Controller, Cluster Controller, Storage Controller, and Walrus (Eucalyptus, 2009).

Figure 1 shows an example of Eucalyptus-based cloud computing environment, considering two clusters (A and B). Each cluster has one Cluster Controller, one Storage Controller, and various Node Controllers. The components in each cluster communicate to the Cloud Controller and Walrus to serve the user requests.

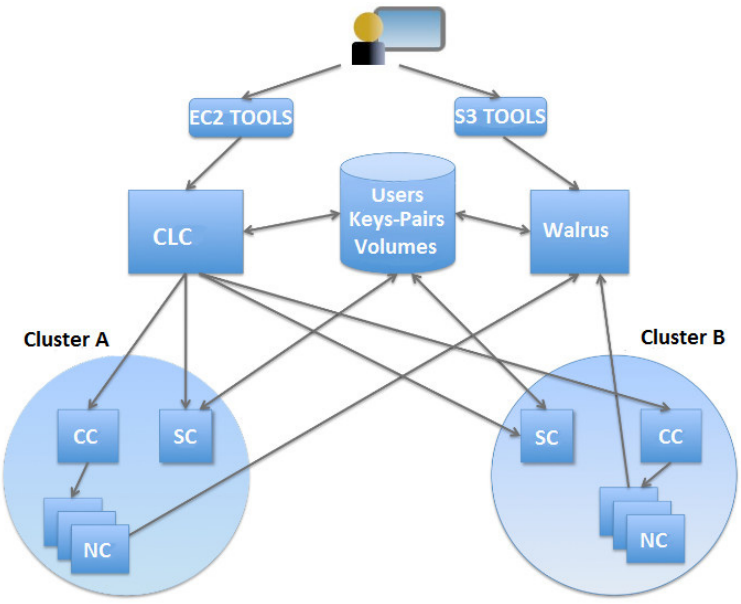


Figure 1. Example of Eucalyptus-based environment

A brief description of these components follows:

• Cloud Controller (CLC): is the front-end to the entire cloud infrastructure, and it is responsible for identifying and managing the underlying virtualized resources (servers, network, and storage) via Amazon EC2 API (Amazon, 2013).

• Node Controller (NC): runs on each compute node and controls the life cycle of VM instances running on the node, and it makes queries to discover the node's physical resources (e.g., number of CPU cores, size of main memory) as well as to probe the state of VM instances on that node (Eucalyptus, 2009; D., 2010).

• Cluster Controller (CC): gathers information on a set of VMs and schedules the VMs execution on specific NCs. The CC has three primary functions: scheduling requests for execution of VM instances; controlling the virtual network overlay composed by a set of VMs, gathering information about the set of NCs, and reporting their status to the CLC (Eucalyptus, 2009).

• Storage Controller (SC): provides persistent storage to be used by virtual machine instances. It implements block-access network storage, that is similar to that provided by Amazon Elastic Block Storage – EBS (Amazon, 2011).

• Walrus: is a file-based storage service compatible with Amazon's Simple Storage Service (S3) (Eucalyptus, 2009). It provides a storage service for VM images. Root filesystem as well as the Linux kernel and ramdisk images used to instantiate VMs on NCs can be uploaded to Walrus and accessed from the nodes.

3. EucaBomber: An Eucalyptus Failure Generator Tool

EucaBomber is a tool for generation of events, injection of failures and repairs that offers support for studies of dependability, more specifically reliability and availability, in cloud computing systems under the Eucalyptus framework. The failures and the repairs injected by the tool aim at affecting the behavior of the high-level components of Eucalyptus and the physical infrastructure that makes up the cloud.

EucaBomber has kernel of functions that supports the basic resources needed for the construction of the injector of failures and repairs.

3.1 Kernel

The kernel was implemented considering a framework called FlexLoadGenerator, that aims at assisting the creation of tools to execute performance and dependability tests. Starting from this framework we can implement workload generators and fault injectors, for example.

Hereinafter, we will not discuss about the FlexLoadGenerator framework, but just about the components that were used to compose the kernel of EucaBomber.

The kernel consists of two fundamental parts, Connection Mode module and Random Variate Generation module. The Connection Mode module is responsible for providing the communication between the host where the fault injector runs and the host that will be affected by that fault. The connection is established using the SSH2 protocol, which enables commands to be issued directly in the Linux shell of the Eucalyptus hosts. Random variate generation module is responsible for generating pseudo-random numbers that follow a given probability distribution, which are used by the tool as the timeout to execute both fault and repair actions. This feature is based on a software library of the WGCap (Workload Generator Capacity Advisor) tool (Galindo, 2009). In our research, we observed that the library that serves as kernel for WGCap can be adapted to the purposes of EucaBomber. We have the following probability distributions available: Erlang, Exponential, Geometric, Lognormal, Normal, Pareto, Poisson, Triangular, Uniform and Weibull.

3.2 EucaBomber

EucaBomber tool was designed to generate and inject failures that emulate the absence of operation. Failures caused by EucaBomber are transient, i.e., simulate a possible turned-off state of the system so that it can be repaired.

The repair action aims to recover the system from eventual failures caused by the fault injector. The tool does not generate the repair of a failure that was not injected by itself. The time between the occurrence of the fault and its repair is simulated by the random variate generation functions supported in the kernel. This strategy aims to simulate the time needed to recover the system and to emulate failures.

The EucaBomber envisages to perform failures and repairs in the following operation modes:

• Hardware failures: this operational mode emulates hardware failures by shutting down the hardware device (server, desktop, etc).

• Hardware failures and repairs: this operational mode includes both failures and repairs related to the hardware. After shutting the machine down, EucaBomber boots the host up again according to the time interval generated . This scenario requires that the target machine supports the WOL (Wake on LAN) (Popa & Slavici, 2009) feature. This is essential because the repair action is performed by sending a "magic packet" through the network, which turns on the machine.

• Sofware failures: failures of Eucalyptus high-level components previously mentioned are included in this operational mode. The tool operates directly suspending the execution of a Eucalyptus process selected by the user. It is worth highlighting that the same machine can be induced to undergo more than one type of software failure.

• Software failures and repairs: this mode enables to restore Eucalyptus processes that had been previously terminated by EucaBomber.

The tool may also combine the previously mentioned operational modes, with the option of performing hardware and software failures in the same machine, with or without the corresponding repair events. It is worth emphasizing that if, by chance, an only-failure scenario is initiated, the test team will be responsible to implement the necessary repair without the aid of EucaBomber.

The EucaBomber can be adapted to test other cloud systems. In order to adapt the tool, the developer has to change the fault injection commands in the source code to the corresponding commands for the new target system. The names of processes and Linux services relative to the Eucalyptus components may be replaced by others, or even changed to a set of instructions that perform a more elaborate fault event. Failures related to the shutdown action may also be modified and the developer can add other hardware faults via software, such as cache data losing.

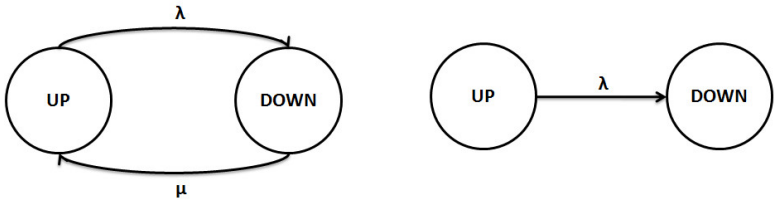
3.3 Methodology for Generation of Failures and Repairs

In this work we decided to perform fault injection at runtime on the target system. For activation of both the fault injections and repairs we used the *time-out* mechanism, considering that it would be appropriate since the tester only needs to inform the events' probability distributions and its respective parameters (ex.: exponentially distribution and failure and repairing rates). Therefore, the time instants at which the events will be carried out are generated using the kernel support. The time unit of the time-out is determined by the users, that is they can choose whether events occur in months, days, hours, minutes, seconds or milliseconds. The syntax adopted to inform the time-out is: *M-x;D-x;H-x;m-x;s-x*, where *M* corresponds to months, *D* is for days, *H* is for hours, *m* for minutes and *s* for seconds, while *x* represents the value for each of the mentioned time units.

Some parts of this syntax can be suppressed when not needed, for example an user wants to define events following an exponential distribution, with average of 2 hours between events. In this case the time-out may be defined with the syntax *H-2*. In other case, if the event occurs every 5 months, 20 hours and 40 minutes, the time-out can be expressed as *M-5;H-20;m-40*. It is important to stress that distinct time units must be separated by a semicolon (";") without spaces between them.

The only exception to the use of syntax occurs when the user wants to specify the timeout in milliseconds. In this case, the user only needs to inform the numeric value, without any other symbol. As an example, an user can inform a value 1000 as the rate parameter of the exponential distribution, and EucaBomber reads this value as 1000 ms. Parameters of specific distributions which have no time unit may also be expressed as single vaues, without using the specified syntax. An example of such case is when an user defines an event expressed by the Erlang distribution, with rate of 5 hours and 2 minutes and shape 2. The rate is expressed as *H-5;m-2* whereas the shape as simply *2*.

Figure 2 presents two models of the system behavior: when the repair option is selected (Figure 2a) and when the repair option is not selected (Figure 2b). Note that both behaviors can be emulated by the EucaBomber either applied to whole system or individually to each Eucalyptus high-level component or even the hardware. The UP state symbolizes that the hardware and/or software is in functional, while the DOWN state represents its failure in any of them. The transition between UP and DOWN states of Figure 2a is determined by λ and μ. The λ to the transition arc represents the failure rate. The failure is generated at the end of the respective time delay, causing the component or system to leave the UP state and go to the DOWN state. The μ annotated to transition arc represents repair rate.



(a) With repair (b) Without repair

Figure 2. Models of system's behavior

The repair begins at the end of the respective time duration, causing the component or system to leave the DOWN state and go to the UP state. The same process for the generation of faults expressed in Figure 2a is repeated in Figure 2b except that in this case only fault events are generated.

3.4 Operation

The EucaBomber tool works with inputs provided through its graphical interface. The steps required for fault injection in the Eucalyptus platform and underlying hardware are described below:

1) Inform the IP (Internet Protocol) address, MAC (Media Access Control) address, user name and password of an user with administrative rights in the host.

2) Select type of failure (hardware, software or both). If software is selected one must choose which of the Eucalyptus processes the tool must operate. One of the following must be selected: Cloud Controller, Cluster Controller, Storage Controller, Walrus or Node Controller.

3) Select if whether there will be or not the repair for the generated failure.

4) Choose one of the available probability distributions. One must also set the values for each parameter of the distribution, informing the time unit to the occurrence of the events. The data informed in this step generate random variables that are used as delay between the current state of the system and trigger the next event. Note that in this step different probability distributions can be chosen for failure and repair in the same machine as well as different values for the parameters of the distributions.

5) Choosing whether to incorporate other machines or software components to the test. If the option to add new faults is selected, one will need to repeat the steps from 1 to 4 informing the new data. For each new computer inserted, one can choose different fault injection scenario (i.e., only-hardware, only-software, etc.).

6) Set the run time of the tool which can be set from months to milliseconds.

7) Run the scenario.

At the end of the execution, the tool generates a CSV (Comma-separated values) file reporting the experiment. The file contains the IP of the machine, the type of the fault and repair actions (if it is the case), and the respective time instants in which the event occurred.

4. Experimental Study

This section presents a case study in which the tool has been applied. The system evaluated was a private cloud testbed composed of six Core 2 Quad machines (2.66 GHz processors, 4 GB RAM). Five of them have installed the Ubuntu Server Linux 11.04 (kernel 2.6.38-8 x86-64) and the Eucalyptus System version 2.0.2. One machine, used as a client for the cloud, has the Ubuntu Desktop Linux 11.04 (kernel 2.6.38-8 x86-64). The cloud environment under test is fully based on the Eucalyptus framework and the KVM hypervisor.

This study may focus on three aspects: 1) check whether the generated time values for failure and repair actions match the selected probability distributions; 2) use the tool to verify the impact of distinct parameter (MTTF or MTTR) values on the system availability; 3) validate the availability measured during the experiments and the expected availability, computed with analytical models using the same parameters of the experimental testbed. In this particular case study, however, we concentrate only on aspects 1 and 2.

4.1 Testbed Environment

Figure 3 shows the components of our testbed environment. The Cloud Controller, Cluster Controller, Storage Controller and Walrus were installed in the same machine (the host 2 in our environment). Other four hosts run the Node Controller, so they are able to execute VMs.

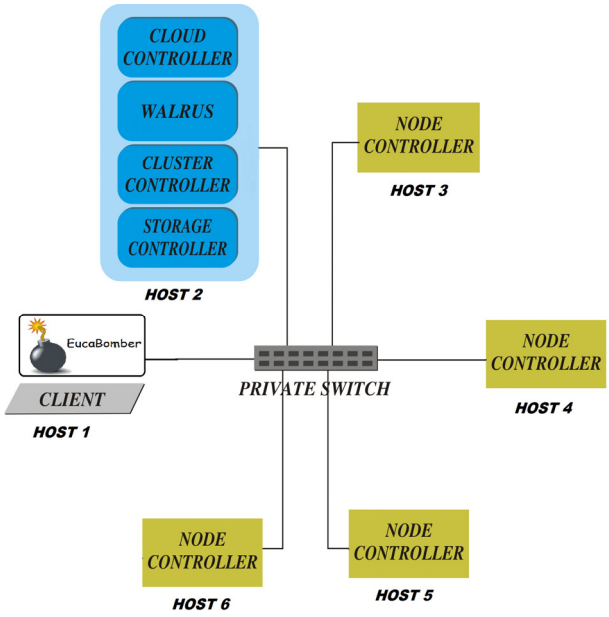


Figure 3. Components of testbed environment

A specific shell script was implemented to check periodically how many resources are available in the cloud. The script assumes that the cloud is available only if all four nodes are available to run VMs. Therefore, if a single software process stops, in any of the machines, the cloud is considered unavailable. The same is considered for the physical machine failures (host shutdown). Other failure modes may also be easily defined and evaluated.

The monitoring script verifies the system condition each 5 seconds. The lack of response from the Cloud Controller in this time interval is also considered as system unavailability.

4.2 Scenarios

The experimental evaluation is composed of four scenarios. Scenarios 1a and 1b consider the exponential probability distribution. Scenarios 2a and 2b adopt the Erlang probability distribution. Scenarios 1a and 2a consider the MTTF and MTTR values obtained in (Dantas et al., 2012; Kim et al., 2009; Hu et al., 2010) reduced by a factor of 1000. Scenarios 1b and 2b use MTTR values which are 20% higher than those values adopted in Scenarios 1a and 2a. Such configurations help to determine what would be the impact to system availability if the repair activity is slowed down due to lack of resources (e.g., missing spare equipments, or absence of trained personnel). Tables 1 and 2 show the MTTF and MTTR values used for each scenario. The shape parameter, k, for the Erlang distribution in Scenarios 2a and 2b was defined as $k=2$.

Table 1. Parameters of Scenarios 1a and 2a

Type of component	Experiment		Real	
	MTTF	MTTR	MTTF	MTTR
Software	2836.8 s	14.4 s	788 h	4 h
Hardware	31536 s	28.8 s	8760 h	9 h

Table 2. Parameters of Scenarios 1b and 2b

Type of component	Experiment		Real	
	MTTF	MTTR	MTTF	MTTR
Software	2836.8 s	17.3 s	788 h	4.8 h
Hardware	31536 s	34.5 s	8760 h	9.6 h

4.3 Results

Each scenario was executed for 48 hours (*totaltime*), and the availability results are presented in Table 3. The availability was computed as:

$$A = uptime/totaltime \quad (1)$$

Where the *uptime* is the sum of all time intervals where the system is available, i.e., all hardware and software components are working. Similarly, the unavailability was computed as:

$$UA = downtime/totaltime \quad (2)$$

Where *downtime* is the summation of all time intervals where the system is unavailable.

Table 3. Availability results of all scenarios

Metric	Scenario			
	1a	1b	2a	2b
Availability (%)	94.4213	93.1626	95.4167	94.8438
Unavailability (hardware) (%)	1.1892	1.3194	1.1140	1.1574
Unavailability (software) (%)	4.3895	5.5179	3.4693	3.9988
Unavailability (total) (%)	5.5787	6.8374	4.5833	5.1563

The results in Table 3 show that, as expected, the availability decreases from Scenario 1a to 1b, due to the increase in the MTTR. The experiments also show that the increase in MTTR has a bigger effect on software unavailability than it has on hardware unavailability. This behavior occurs in both, Exponential-based and Erlang-based, experiments because, in this environment, the software failures are more frequent than the hardware ones. Therefore, software repair actions are also more often than hardware repairs. This conclusion is stressed by Table 4, which shows the results using the time perspective (uptime and downtime). Notice that the difference between scenarios 1a and 1b is about 11%, when considering hardware downtime (34.2 minutes in 1a versus 38 minutes in 1b). On the other hand, when considering software downtime, the difference is around 25% (126.4 minutes in 1a versus 158.9 minutes in 1b). Similar conclusions are drawn when observing the results of scenarios 2a and 2b in Table 4. The hardware downtime increases only 3.7% from 2a to 2b, whereas the software downtime increases 15.2%.

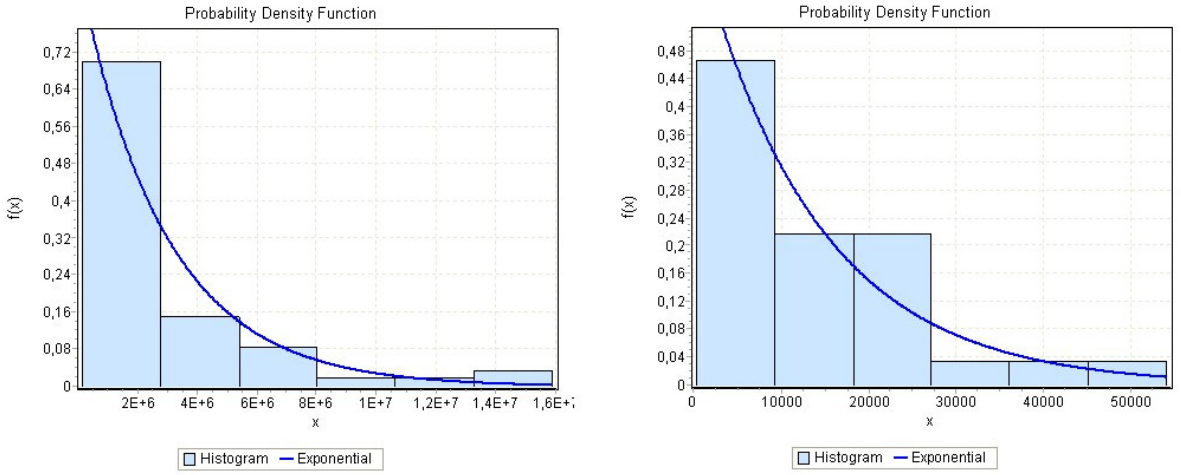
Table 4. Uptime/downtime of all scenarios

	Scenario			
Metric	1a	1b	2a	2b
Uptime (hours)	45.3	44.7	45.8	45.5
Downtime (hardware) (min.)	34.2	38	32.1	33.3
Downtime (software) (min.)	126.4	158.9	99.9	115.1
Downtime (total) (min.)	160.6	196.9	132	148.5

It is also important to highlight the difference observed between Scenario 1a and 1b, where total annual downtime increases more than 30 minutes.

The presented results demonstrate how EucaBomber is useful to system administrators who need to predict the reliability or availability behavior of their private cloud infrastructures under distinct conditions, and plan changes in corrective, preventive or predictive maintenance policies. We also verified the accuracy of time intervals generated for the failure and repair events, according to the selected probability distributions. Figures 4a and 4b show the exponential curve fitting of time values for failure and repair events, respectively, considering the Scenario 1a. The probability density function (pdf) for both, failure and repair events, has a fair degree of approximation to the exponential curve. The goodness of fit is confirmed by the Kolmogorov-Smirnov (K-S) test, whose values are 0.0944 for the failure events, and 0.0882 for the repair events. For both cases, the

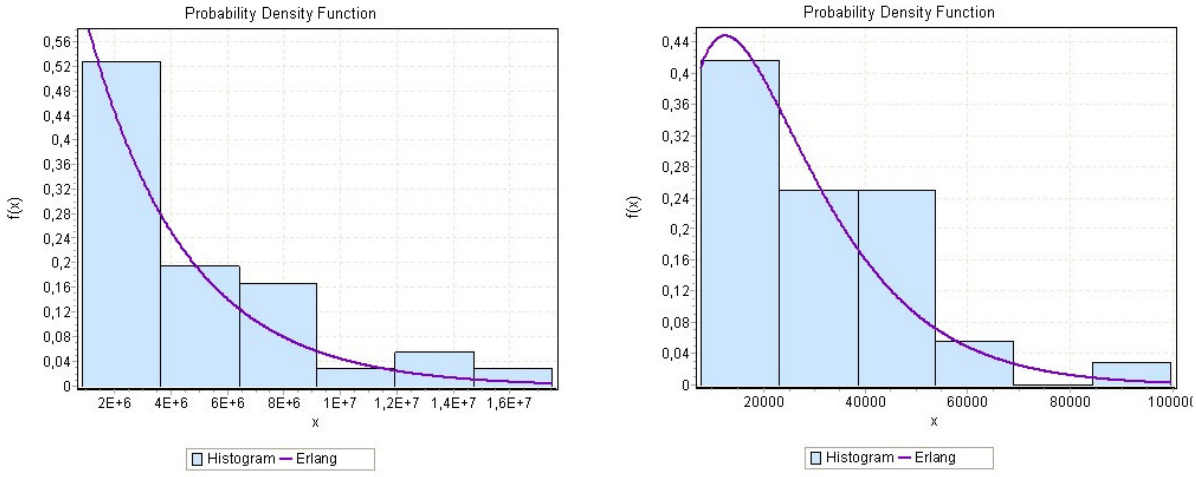
index is close to zero, therefore the experimental data have a good fit to the exponential distribution.



(a) PDF of failure events (b) PDF of repair events

Figure 4. Distribution of events in Scenario 1a

Figures 5a and 5b show the fitting for the Scenario 2a, considering the Erlang distribution. Failure and repair events have a good fit, as seen in the plots. The K-S test indicates an index of 0.2528 for the failure events, and 0.1597 for the repair ones, so demonstrating the accuracy of EucaBomber in generating the events according to the Erlang probability distribution.



(a) PDF of failure events (b) PDF of repair events

Figure 5. Distribution of events in Scenario 2a

The goodness of fit for Scenarios 1b and 2b is not shown here because they are based on the same probability distributions of Scenarios 1a and 2a, respectively.

5. Final Remarks

EucaBomber is a tool tailored to support dependability studies in the Eucalyptus platform. It generates hardware and software fault events at runtime and also enables the repair of the respective failures considering a set of impotant probability distributions. The results of case studies show that EucaBomber helps to test cloud infrastructures, regarding reliability and availability issues of hardware and software components of a Eucalyptus cloud environemnt, being an important tool for supporting maintenance operation plan and architectural changes. It can also be used to support system availability prediction, hence aiding the definition and accomplishment of

service level agreements for future users of the cloud infrastructure. In future works, we intend to study other types of faults, such as crashes on the virtual machines or in the applications running on them. Also we could extend the tool in order to apply to other cloud systems, such as OpenNebula or OpenStack.

Acknowledgements

We would like to thank the Coordination of Improvement of Higher Education Personnel – CAPES, National Council for Scientific and Technological Development – CNPq, Foundation for Support to Science and Technology of Pernambuco – FACEPE and MoDCS Research Group for their support.

References

Amazon. (2011). *Amazon Elastic Block Store (EBS)*. Retrieved from http://aws.amazon.com/ebs

Amazon. (2013). *Amazon Elastic Compute Cloud - EC2*. Retrieved from http://aws.amazon.com/ec2/

Avizienis, A., Laprie, J., Randell, B., & Landwehr, C. (2004). Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing, 1*, 11-33. http://dx.doi.org/10.1109/TDSC.2004.2

Baraza, J. C., Gracia, J., Gil, D., & Gil, P. (2000). A prototype of a VHDL-based fault injection tool. Defect and Fault Tolerance in VLSI Systems. *Proceedings. IEEE International Symposium on* (pp. 396-404). http://dx.doi.org/10.1109/DFTVS.2000.887180

Dantas, J. R., Matos, R., Araujo, J., & Maciel, P. (2012). An availability model for eucalyptus platform: An analysis of warm-standy replication mechanism. In *The 2012 IEEE Int. Conf. on Systems, Man, and Cybernetics (IEEE SMC 2012), Seoul*.

Ejlali, A., Miremadi, S. G., Zarandi, H., Asadi, G., & Sarmadi, S. (2003). A hybrid fault injection approach based on simulation and emulation co-operation. In *Dependable Systems and Networks, 2003. Proceedings. 2003 International Conference on* (pp. 479-488). http://dx.doi.org/10.1109/DSN.2003.1209958

Eucalyptus Systems. (2009). *Eucalyptus Open-Source Cloud Computing Infrastructure - An Overview*.

Eucalyptus Systems. (2013). *Eucalyptus - The Open Source Cloud Platform*. Retrieved from http://open.eucalyptus.com/

Friesenbichler, W., Panhofer, T., & Steininger, A. (2010). A deterministic approach for hardware fault injection in asynchronous QDI logic. In *Design and Diagnostics of Electronic Circuits and Systems (DDECS), 2010 IEEE 13th Int. Symp. on* (pp. 317-322). http://dx.doi.org/10.1109/DDECS.2010.5491758

Furht, B., & Escalante A. (2010). *Handbook of Cloud Computing*. Springer. http://dx.doi.org/10.1007/978-1-4419-6524-0

Galindo, H., Santos, W., Maciel, P., Silva, B., Galdino, S., & Pires, J. (2009). Synthetic workload generation for capacity planning of virtual server environments. In *Systems, Man and Cybernetics, 2009. SMC 2009. IEEE Int. Conf. on* (pp. 2837-2842). http://dx.doi.org/10.1109/ICSMC.2009.5346600

Hu, T., Guo, M., Guo, S., Ozaki, H., Zheng, L., Ota, K., & Dong, M. (2010). Mttf of composite web services. In *Parallel and Distributed Processing with Applications (ISPA), 2010 Int. Symp. on* (pp. 130-137). http://dx.doi.org/10.1109/ISPA.2010.91

Johnson, D., Kiran, M., Murthy, R., Suseendran, R. B., & Yogesh, G. (2010). *Eucalyptus Beginner's Guide* (UEC ed.).

Kim, D. S., Machida, F., & Trivedi, K. (2009). Availability modeling and analysis of a virtualized system. *Dependable Computing, 2009. PRDC '09. 15th IEEE Pacific Rim Int. Symp. on* (pp. 365-371). http://dx.doi.org/10.1109/PRDC.2009.64

Nurmi, D., Wolski, R., Grzegorczyk, C., Obertelli, G., Soman, S., Youse, L., & Zagorodnov, D. (2009). The eucalyptus open-source cloud-computing system. In *Proc. the 9th IEEE/ACM Int. Symp. on Cluster Computing and the Grid (CCGrid)* (pp. 124-131). Washington, DC, USA. IEEE Computer Society. http://dx.doi.org/10.1109/CCGRID.2009.93

Popa, M., & Slavici, T. (2009, May). Embedded server with wake on lan function. In *EUROCON 2009, EUROCON'09. IEEE* (pp. 365-370). http://dx.doi.org/10.1109/EURCON.2009.5167657

Sousa, E., Maciel, P., Medeiros, E., Souza, D., Lins, F., & Tavares, E. (2012). *Evaluating eucalyptus virtual machine instance types: A study considering distinct workload demand* (pp. 130-135). The Third Int. Conf. on Cloud Computing, GRIDs, and Virtualization.

Sun, D., Chang, G., Guo, Q., Wang, C., & Wang, X. (2010). A dependability model to enhance security of cloud environment using systemlevel virtualization techniques. In *Proc. First Int. Conf. on Pervasive Computing, Signal Processing and Applications (PCSPA 2010), Harbin*. http://dx.doi.org/10.1109/PCSPA.2010.81

Weinkopf, J., Harbich, K., & Barke, E. (2006). Parsifal: A generic and configurable fault emulation environment with non-classical fault models. In *Field Programmable Logic and Applications, 2006. FPL '06. International Conference on* (pp. 1-6). http://dx.doi.org/10.1109/FPL.2006.311220

Zhang, Y., Liu, B., & Zhou, Q. (2011). A dynamic software binary fault injection system for real-time embedded software. In *Reliability, Maintainability and Safety (ICRMS), 2011 9th Int. Conf. on* (pp. 676-680). http://dx.doi.org/10.1109/ICRMS.2011.5979375

Ziade, H., Ayoubi, R. A., & Velazco, R. (2004). A survey on fault injection techniques. *Int. Arab J. Inf. Technol., 1*, 171-186.