Pós-Graduação em Ciência da Computação

"Software Aging Monitoring Strategies and Rejuvenation Policies for Eucalyptus Cloud Computing Platform"

By

Jean Carlos Teixeira de Araujo

M.Sc. Dissertation

Federal University of Pernambuco
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/~posgraduacao

RECIFE-PE, MARCH/2012

Federal University of Pernambuco
Informatics Center
Post-Graduation in Computer Science

Jean Carlos Teixeira de Araujo

"Software Aging Monitoring Strategies and Rejuvenation Policies for Eucalyptus Cloud Computing Platform"

A M.Sc. Dissertation presented to the Informatics Center of the Federal University of Pernambuco in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

Advisor: *Paulo Romero Martins Maciel*

RECIFE-PE, MARCH/2012

Catalogação na fonte
Bibliotecária Monick Raquel Silvestre da Silva, CRB4-1217

Araujo, Jean Carlos Teixeira de
Software aging monitoring strategies and rejuvenation policies for Eucalyptus cloud computing platform / Jean Carlos Teixeira de Araujo. - Recife: O Autor, 2013.
xxiv, 96 p.: il., fig., tab.

Orientador: Paulo Romero Martins Maciel.
Dissertação (mestrado) - Universidade Federal de Pernambuco. CIn, Ciências da Computação, 2013.

Inclui bibliografia e apêndice.

1. Avaliação de desempenho. 2. Redes de computadores e sistemas distribuídos. 3. Computação nas nuvens. I. Maciel, Paulo Romero Martins (orientador). II. Título.

004.029 CDD (23. ed.) MEI2013 – 016

Dissertação de Mestrado apresentada por **Jean Carlos Teixeira de Araujo** à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título "**Software Aging Monitoring Strategies and Rejuvenation Policies for Eucalyptus Cloud Computing Platform**", orientada pelo **Prof. Paulo Romero Martins Maciel** e aprovada pela Banca Examinadora formada pelos professores:

_____

Prof. Nelson Souto Rosa
Centro de Informática / UFPE

_____

Prof. Rivalino Matias Júnior
Faculdade de Computação / UFU

_____

Prof. Paulo Romero Martins Maciel
Centro de Informática / UFPE

Visto e permitida a impressão.
Recife, 2 de março de 2012.

_____

Prof. Nelson Souto Rosa
Coordenador da Pós-Graduação em Ciência da Computação do
Centro de Informática da Universidade Federal de Pernambuco.

I dedicate this dissertation to my grandfather José Domingos and my grandmother Maria Pereira, essentials in my growing and presents in all moments of my life. You are my personal examples of dedication and love to family.

Acknowledgements

First of all, I would like to thank God for the gift live and all moments that you are with me. Also, I am very grateful my family, friends and professors who gave me all necessary support to get here. In special to my advisor Paulo Maciel that believed me and my work, there are few words that can express my esteem and appreciation.

Thanks to my mother Josinete, present in every moment of my life and for teaching me to fight and never give up of my personal objectives. A thanks to my girlfriend Danise Vivian, for your patience, comprehension, love and support in my decisions.

I would like to thank to Rivalino Matias and Ibrahim Beicker to help me on some experiments. I am very grateful to magistrates Ícaro Matos, Glautemberg Bastos and Cláudio Pantoja and my boss and friend Daniel Malta for their support.

Thanks to my dear friend and young researcher Rubens Matos, for their help in anytime. It is a great honor to work with you; I can't forget Álvaro Ribas, my best friend and the brother that I never had; Also, I am very grateful to Cristiano and Nira, for opening the doors of your home. You are my family too.

I would like to thank the National Council for Scientific and Technological Development – CNPq and for everyone of MoDCS Research Group for their support.

Do not rush
Tomorrow anything can happen
including anything
Do not rush
The caterpillar crawls up the day
That creates wings
Do not rush
Than the little donkey of happiness
Never late
Do not rush
Tomorrow she stops at the door
Your home

Do not rush
Every journey begins
In the first step
Nature does not hurry
Follow your compass
Inexorably get there
Do not rush
Look who's going up the hill
Be Princess or valet
To go higher will have to sweat

—ACCIOLY NETO (Adapted from: The Nature Of Things)

Resumo

A necessidade de confiabilidade e disponibilidade tem aumentado em aplicações modernas a fim de lidar com a crescente demanda da utilização de serviços de TI, proporcionando um serviço ininterrupto. Sistemas de computação em nuvem fundamentalmente fornecem acesso a grandes conjuntos de recursos computacionais através de uma variedade de interfaces, de forma semelhante a gestão de recursos grid e HPC, que também permitem a implantação flexível de aplicativos por meio de máquinas virtuais. Este trabalho investiga o vazamento de memória, a fragmentaćão da memória e outros efeitos do envelhecimento sobre a infraestrutura de computação em nuvem Eucalyptus, considerando as cargas de trabalho composta por intensas solicitações para a implantação de máquinas virtuais, mostrando a existência destes "sintomas" no ambiente estudado. Estratégias de rejuvenescimento são propostas e seus benefícios são destacados através de um experimento que ativa o mecanismo quando o processo de rejuvenescimento do Eucaliptus atinge níveis críticos de utilização de memória virtual. Esta dissertação apresenta também uma abordagem que utiliza séries temporais para estimar o tempo de rejuvenescimento, de modo a reduzir o tempo de inatividade prevendo o momento apropriado para realizar o rejuvenescimento. Nós mostramos os resultados da nossa abordagem através de experimentos usando o framework de computação nas nuvens Eucalyptus.

Palavras-chave: Envelhecimento e rejuvenescimento de software; computação nas nuvens; platforma Eucalyptus; análise de dependabilidade; vazamento e fragmentação de memória; séries temporais

Abstract

The need for reliability and availability has increased in modern applications, in order to handle rapidly growing demands the use of IT services while providing uninterrupted service. Cloud computing systems fundamentally provide access to large sets of data and computational resources through a variety of interfaces, similarly to existing grid and HPC, which also enable the flexible deployment of applications by means of virtual machines. This work investigates the memory leak, memory fragmentation and others aging effects on the Eucalyptus cloud computing infrastructure considering workloads composed of intensive requests for deployment of virtual machines, showing the existence of these "symptoms" in the studied environment. Rejuvenation strategies are proposed and its benefits are highlighted through an experiment that activates the rejuvenation mechanism when the eucalyptus process reaches critical levels of virtual memory utilization. This dissertation presents too an approach that uses time series to estimate the time to rejuvenation, so as to reduce the downtime by predicting the proper moment to perform the rejuvenation. We show the results of our approach through experiments using the Eucalyptus cloud computing framework.

Keywords: Software aging and rejuvenation; cloud computing; Eucalyptus platform; dependability analysis; memory leak and fragmentation; time series

Contents

List of Figures

List of Tables

List of Acronyms

API	Application Programming Interface
AWS	Amazon Web Services
CC	Cluster Controller
CLC	Cloud Controller
CPU	Central Processing Unit
EBS	Elastic Block Store
EC2	Amazon Elastic Compute Cloud
FTP	File Transfer Protocol
GB	Gigabyte
GCM	Growth Curve Model
HPC	High-Performance Computing
KVM	Kernel-based Virtual Machine
LTM	Linear Trend Model
MAD	Mean Absolute Deviation
MAPE	Mean Absolute Percentage Error
MB	Megabyte
MSD	Mean Squared Deviation
NC	Node Controller
NIST	US National Institute of Standards and Technology
OS	Operating System
PID	Process Identifier
QTM	Quadratic Trend Model
SC	Storage Controller

SCTM S-Curve Trend Model

S3 Simple Storage Service

UID User Identifier

VM Virtual Machine

1 Introduction

A mente que se abre a uma nova ideia jamais voltará ao seu tamanho original.
The mind that opens to a new idea never returns to its original size.
—ALBERT EINSTEIN (Phrase)

Information systems are present in all activities of our daily lives, from communication, education, health, finance, security and entertainment. Hence, unavailability, low reliability and poor performance of such systems have been taken attention of service providers, infrastructure managers, application designers and users, thus demanding solutions from the scientific community.

Our society is becoming increasingly dependent on computer systems and the interconnection of their networks. Computer systems are complex and implemented by hardware and software layers. In specific circumstances, external or natural events can cause the expected behavior is not executed. In these circumstances, the system may not provide the services for which it was specified.

The development of software applications, from client-server architectures to multi-level architectures, and recently web services, led to the creation of increasingly distributed applications. Such applications usually require large data center infrastructures, with a large amount of available resources such as CPU, memory, storage, and network bandwidth. All this processing power requires intensive use of refrigeration, resulting in high-energy consumption and other related costs. Hence, service providers need to use flexible computing infrastructures in order to reduce costs and easily adapt the service to the different levels of workload demand. Cloud computing enables the use of computing resources by means of

providers that may be in any geographic location, accessible through de Internet, i.e., in the clouds.

Deployment of cloud-based architectures has grown over recent years, mainly because they constitute a scalable, cost-effective and robust service platform. Such features are made possible due to the integration of various software components that enable reservation and access to computational resources mainly based on web services, by means of standard interfaces and protocols. Virtualization is an essential requirement to build a typical cloud-computing infrastructure (Armbrust *et al.*, 2009).

Eucalyptus (Eucalyptus, 2009) is a software framework used to implement private clouds and hybrid-style Infrastructure as a Service - IaaS. Its architecture is quite modular and its internal components use web services, easing their replacement and expansion. It implements the API AWS (Amazon Web Services), allowing interoperability with other AWS-based services.

In the distributed system area, the study of software reliability and availability is required, since the consequences of software failures very often cause enormous economic and reputational losses (Thein and Park, 2009). To Avizienis *et al.* (2004) a system failure is an event that occurs when the delivered service deviates from correct service. A fault is thus a transition from correct service to incorrect service, i.e., to not implementing the system function. The delivery of incorrect service is a system outage. A transition from incorrect service to correct service is service restoration.

Cloud-oriented data centers allow the success of massive user-centric applications, such as social networks, even for start-up companies that would not be able to provide, by themselves, the performance and availability guarantee needed for those systems. Although availability and reliability are major requirements for cloud-oriented infrastructures, an aspect usually neglected by many service providers is the effect of software aging phenomenon (Grottke *et al.*, 2008), which has been verified (e.g.,(Grottke *et al.*, 2008; Matias and Filho, 2006; Bao *et al.*, 2005)) to play an important role to the reliability and performance degradation of many software systems.

While flexible and essential to the concept of “elastic computing”, the usage of virtual machines and remote storage volumes requires memory and disk intensive operations, mainly during virtual machines initialization, reconfiguration or destruction. Such operations may speed up the exhaustion of hardware and operating system resources in the presence of software aging due to software faults or poor system design (Grottke *et al.*, 2008).

An important threat for availability is the phenomenon of software aging, an inevitable process where the application processes suffer from performance degradation

throughout their use. Some proactive actions may be taken to minimize the aging effects, but these solutions are very specific to each environment under analysis. The aging phenomenon in cloud systems deserves special attention, since such environment should provide characteristics such as high availability, high stability, high fault tolerance and dynamical extensibility.

In order to counteract this problem, there is the technique of software rejuvenation, which involves occasionally stopping the software application, removing the accrued error conditions and then restarting the application in a clean environment. The procedures remove the accumulated errors and frees up or defragments operating system resources, thus preventing, in a proactive manner, unplanned and potentially expensive future system outages (Huang *et al.*, 1995).

This dissertation presents a study on the software aging effects in Eucalyptus cloud computing infrastructure. We detect and demonstrate the aging effects in this system, by monitoring the usage of resources such as CPU, memory and disk space, as well as other indirect consequences of software aging that could cause performance degradation or service failures as well. The case studies is carried out by using repeated virtual machines instantiations as our workload.

We aim to evaluate the aging effects in a cloud-computing platform, focusing on memory-related aging effects such as memory fragmentation and leaking, which are proven to cause serious undesired consequences on system performance and availability (Matias *et al.*, 2010). We also propose rejuvenation strategies to Eucalyptus cloud computing infrastructure.

The remainder of this chapter describes the focus of this dissertation and starts by presenting its motivation in Section 1.1 and a clear definition of the objectives in Section 1.2. A justification of this research is presented in Section 1.3. Section 1.4 presents the main contributions. Section 1.5 describes some related works and, finally, Section 1.6 describes how this dissertation is organized.

1.1 Motivation

Deployment of cloud-based architectures has grown over recent years. Features such as scalability, cost-effectiveness and robustness has been widely requested by the market of Information Technology. Usage of virtual machines and remote storage volumes requires memory and disk intensive operations. Such operations may speed up the exhaustion of hardware and operating system resources in the presence of software aging: Software faults, Poor system design.

Researchers worldwide have investigated the factors of software aging and in some cases proposed action of rejuvenation in various environments, such as web servers (Grottke *et al.*, 2006; Matias and Filho, 2006), OS kernel (Matias *et al.*, 2010), clustered systems (Vaidyanathan *et al.*, 2001), telecommunication billeting applications and safety-critical military equipment (Marshal, 1992; Office, 1992), and strategies to improve rejuvenation, such as the use of Regenerate Markov Stochastic Petri Nets. However, this is the first work that studies software aging and rejuvenation in a cloud computing platform.

To identify the occurrence of software aging is not so simple, especially when it comes to something that was never published by other researchers. If it was not detected in time to a proactive action to be activated, the software aging effects can be catastrophic, causing inestimable damage and losses. So, what we would like to do is to identify the software aging and rejuvenation to bring an action in a cloud computing environment.

To propose a rejuvenation strategy it requires a good knowledge of the system, and specially of their resources aging, because this technique looks for reducing the aging effects during the software runtime until the aging causes are fixedin a new version of the software.

Therefore, research and combat to an important factor in occurrence of unavailability by important services as used by hundreds of thousands of users, is the case of cloud computing, and to be the first research in the study of software aging and rejuvenation in the cloud computing is an important motivation for conducting this dissertation.

1.2 Objectives

This section describes the general and specific objectives of this research. The overall objective is to investigate software aging effects and strategies to rejuvenate the Eucalyptus cloud computing environment.

The specific objectives are:

- To assemble a test environment to investigate the software aging effects;
- Definition of procedures for monitoring and analyse of computacional resources;
- To evaluate the aging effects in a cloud-computing platform;
- Identification of resources that age more quickly;
- To propose rejuvenation actions to avoid the system outages due to aging;

- To use strategies of time prediction for activating the rejuvenation action;
- To propose strategies to reduce system downtime during the execution of the rejuvenation action.

1.3 Justification

The study of software aging is a very important factor to be considered, especially in systems that receive a lot of user requests, which is the case of a cloud computing platform, because a system failure can affect thousands of customers.

The choice of Eucalyptus software has been done because of its advanced stage of development, with bug fixes and regular updates and mainly because its open source version of the software has been used by Amazon Elastic Compute Cloud (EC2). The fact that Eucalyptus is an open source software, makes it easy to access and to control the platform and hence the development of monitoring scripts and workloads.

Proposing a rejuvenation action is not trivial, given that each software has specific characteristics, therefore the action of rejuvenation should consider these differences. This action will directly influence the functioning of the system, correcting flaws, fixing errors or taking any action to avoid disrupting the system. Thus, the thorough knowledge of the environment is essential.

1.4 Aimed contributions

In this research, we obtained results that can confirm the existence of the software aging phenomenon in the Eucalyptus cloud computing infrastructure, a factor that degrades the system and causes, among other things, loss of performance, failure to perform activities and can result in system downtime.

Identifying such occurrence in a software system, we then propose a proactive action that allows the software rejuvenation, making it returns to its previous state, where it performs its functions normally without unwanted stopping.

The bringing of a rejuvenation action may not be enough, because the system may be unavailable for a few moments during the execution of this action, in addition to the time of its activation that would be related to the administrator empirical knowledge. Time series is used to estimate the growth in the resources usage, allowing the system administrator to identify the most suitable moment for activation of the rejuvenation action.

To identify the occurrence of aging in Eucalyptus software allow users and developers to take knowledge of this degrading factor, enabling them to take preventive measures and/or proactive to avoid downtime. The proposal of the rejuvenation action can be used in similar architecture environments, in order to be a specific action. But the use of time series can be applied in any environment in order to be a general methodology, where the history of any monitored software can be used not only to predict the rejuvenation activation but also to reduce the system downtime during the execution of the rejuvenation action.

1.5 Related works

The characteristics, architectures and applications of several popular cloud computing platforms are analyzed and discussed in (Peng *et al.*, 2009), that aims to clarify the differences among the considered platforms. The authors conclude that though each cloud computing platform has its own strength, one thing should be noticed is that there is lots of unsolved issues in all platforms. Such issues include the continuously high availability, dealt mechanisms of cluster failure in cloud environment, consistency guarantee, synchronization in different clusters, interoperation, standardization, and security.

In (Cordeiro *et al.*, 2010), a comparative description about three most popular cloud computing solutions - Xen Cloud Platform, Eucalyptus and OpenNebula - is presented. The work also describes examples of use for each platform, and it supposes that by understanding some of the main differences between them, one may decide where and when each solution may be more appropriate.

The enhancement of communication performance, robustness, and security of services using cloud concepts is addressed in (Mckinley *et al.*, 2006). It describes Service Clouds, a distributed infrastructure designed to facilitate rapid prototyping and deployment of services. The infrastructure combines adaptive middleware functionality with an overlay network substrate in order to support dynamic instantiation and reconfiguration of services. The Service Clouds architecture includes a collection of lowlevel facilities that can be either invoked directly by applications or used to compose more complex services. Two experimental case studies were conducted to improve throughput of bulk data transfer and to enhance the robustness of multimedia streaming. These case studies demonstrate the usefulness of the Service Clouds infrastructure in deploying new services, and the performance results indicate the benefits of the services themselves.

A great number of technologies have been developed to provide comprehensive high availability to virtualized systems, by means of fast fault detection and replication based on checkpointing techniques. The problem of application fault resilience in virtualized

environments is addressed by (Lan *et al.*, 2008), which presents an on-going project on the design and development of adaptive fault tolerance for HPC applications. It aims to enable parallel applications to avoid anticipated failures via preventive migration, and in the case of unforeseeable failures, to minimize their impact through selective checkpointing.

Mihailescu *et al.* (2011) proposes improvements for the resilience of cloud applications to infrastructure anomalies, by means of OX, a runtime system that uses application-level availability constraints and application topologies discovered on the fly. This system allows application owners to specify groups of highly available virtual machines. To discover application topologies, OX monitors network traffic among virtual machines transparently, and based on this information dynamically implements VM placement optimizations to enforce application availability constraints and reduce and/or alleviate application exposure to network communication anomalies, such as traffic bottlenecks.

An important technique for providing high availability in virtualized environments is presented in (Cully *et al.*, 2008), under the name of Remus, an extension to the Xen hypervisor that works by continually live-migrating a VM from the primary physical host to the backup. Such approach prevents outages due to hardware failures and unusual software bugs, but it cannot avoid or fix problems commonly caused by software aging. In fact, a continuous software replication mechanism may copy bad aspects of system state, such as memory fragmentation or garbage resulting from application's faulty resource (de-)allocation.

By considering software aging related to application domains other than cloud computing, (Matias *et al.*, 2010) presented a study where they explored OS Linux kernel using instrumentation techniques to measure software aging effects.

Matias and Filho (2006) adopted SIGUSR1 Linux signal for the rejuvenation purpose in web servers. When this signal is sent to the master *httpd*, it indicates that it must reinitialize each of its slaves. However this action only takes place when the slave finishes the request processing that it is currently in progress. The authors state that the downtime during rejuvenation is nonexistent, because the master *httpd* - by continuing execution - ensures the setting of new connections on port 80, even in those situations in which all slaves are being rejuvenated at the same time. The experiment was performed along the period of 9 hours. It was defined so that when the memory usage by the *httpd* processes reaches 395 megabytes, the rejuvenation mechanism would be activated.

Carrozza *et al.* (2010) proposes a practical approach to detect aging phenomena caused by memory leaks in distributed objects Off-The-Shelf middleware, which are commonly used to develop critical applications. The approach, which is validated on a real-world case study from the Air Traffic Control domain, defines algorithms and support tools to

perform data filtering and for trading off experimentation time and statistical accuracy of aging trend estimates.

The work in (Iosup *et al.*, 2011) analyses the performance of cloud computing services for scientific computing workloads, quantifying the presence in real scientific computing workloads of Many-Task Computing (MTC) users, that is, users who employ loosely coupled applications comprising many tasks to achieve their scientific goals. That study was followed by an empirical evaluation of the performance of four commercial cloud computing services. Last, trace-based simulation was used to compare the performance characteristics and cost models of clouds and other scientific computing platforms, for general and MTC-based scientific computing workloads. The results indicate that the current clouds need an order of magnitude in performance improvement to be useful to the scientific community, and show which improvements should be considered first to address this discrepancy between offer and demand.

1.6 Structure of the dissertation

The remaining parts of the dissertation are organized as follows:

In Chapter 2 we present fundamental concepts about software aging, software rejuvenation, cloud computing, dependability, measurement and time series. Chapter 3 presents the main aspects of the Eucalyptus framework. Chapter 4 presents Sofware Aging in Eucalytpus Cloud Computing Environment. Chapter 5 describes Rejuvenation Strategies to Eucalytptus platform. In Chapter 6 we presents the case studies and results obtained from the experiments. Chapter 7 draws some conclusions, shows aimed contributions and possible future works.

2
Background

Não é preciso ter olhos abertos para ver o sol, nem é preciso ter ouvidos afiados para ouvir o trovão. Para ser vitorioso você precisa ver o que não está visível
No need to open eyes to see the sun or you to have ears sharp to hear the thunder. To be successful you need to see what is not visible.
—SUN TZU (The Art of War)

This chapter presents fundamental concepts about Cloud Computing, Dependability, Software Aging and Rejuvenation, Performance Measurement and Time Series.

2.1 Cloud computing

Cloud computing is the access to computers and their functionalities via the Internet or a local area network. It is called "cloud computing" because the user can not actually see or specify the physical location and organization of the equipment hosting the resources they are ultimately allowed to use (Eucalyptus, 2012).

Although several researchers have tried to define cloud computing, no single, agreed-upon definition exists yet. The US National Institute of Standards and Technology - NIST (NIST, 2011), defines cloud computing as follows: "*Cloud computing is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction*".

Numerous advances in application architecture have helped to promote the adoption of cloud computing. These advances help to support the goal of efficient application

development while helping applications to be elastic and scale gracefully and automatically (Microsystems, 2009). Cloud computing is seen by some as an important forward-looking model for the distribution and access of computing resources because it offers these potential advantages:

- *Scalability*: Applications designed for cloud computing need to scale with workload demands so that performance and compliance with service levels remain on target (Eucalyptus, 2012; Microsystems, 2009).
- *Security*: Applications need to provide access only to authorized, authenticated users, and those users need to be able to trust that their data is secure (Microsystems, 2009).
- *Availability*: Regardless of the application being provided, users of Internet applications expect them to be available every minute of every day (Microsystems, 2009).
- *Reliability and fault-tolerance*: Reliability means that applications do not fail and most important they do not lose data (Microsystems, 2009) i.e. this is the ability to perform and maintain its functions in unexpected circumstances.
- *As-needed availability*: Aligns resource expenditure with actual resource usage thus allowing the organization to pay only for the resources required, when they are required (Eucalyptus, 2012).

2.1.1 Business model

Cloud computing employs a service-driven business model. In other words, hardware and platform-level resources are provided as service an on-demand basis. The most common services styles are referred to by the acronyms IaaS (Infrastructure as a Service), PaaS (Platform as a Service), and SaaS (Software as a Service):

IaaS (Infrastructure as a Service) style clouds provide access to collections of virtualized computer hardware resources, including machines, network, and storage. With IaaS, users assemble their own virtual cluster on which they are responsible for installing, maintaining, and executing their own software stack (Eucalyptus, 2012). IaaS is the delivery of computer infrastructure as a service. This layer differs from PaaS in that the virtual hardware is provided without a software stack. Instead, the consumer provides a VM image that is invoked on one or more virtualized servers (Jones, 2008). Examples of IaaS providers

include Amazon EC2 (Amazon, 2011b), GoGrid (GoGrid, 2011) and Flexiscale (FlexiScale, 2011).

PaaS (Platform as a Service) style clouds provide access to a programming or runtime environment with scalable computing resources and data structures embedded in it. With PaaS, users develop and execute their own applications within an environment offered by the service provider (Eucalyptus, 2012). PaaS can be described as an entire virtualized platform that includes one or more servers (virtualized over a pool of physical servers), operating systems, and specific applications (such as Apache and MySQL for Web-based applications) (Jones, 2008). Examples of PaaS providers include Google App Engine (Google, 2011), Microsoft Windows Azure (Microsoft, 2011) and Force.com (SalesForce, 2011a).

SaaS (Software as a Service) style clouds deliver access to collections of software application programs. SaaS providers offer users access to specific application programs controlled and executed on the provider's infrastructure. SaaS is often referred to as "Software on Demand" (Eucalyptus, 2012). SaaS is the ability to access software over the Internet as a service. Another perspective on SaaS is the use of software over the Internet that executes remotely. This software can be in the form of services used by a local application (defined as Web services) or a remote application observed through a Web browser (Jones, 2008). Examples of SaaS providers include Salesforce.com (SalesForce, 2011b) and Rackspace (RackSpace, 2011).

2.1.2 Cloud types

There are many issues to consider when moving an enterprise application to the cloud environment. For example, some service providers are mostly interested in lowering operation cost, while others may prefer high reliability and security. Accordingly, there are different types of clouds, each with its own benefits and drawbacks (Zhang *et al.*, 2010). Cloud types (including public, private, and hybrid - see Figure 2.1) refer to the nature of access and control with respect to use and provisioning of virtual and physical resources.

Public clouds refers to a cloud service delivery model in which a service provider makes massively scalable IT resources, such as CPU and storage capacities, or software applications, available to the general public over the Internet. Public cloud services are typically offered on a usage-based model (Furht and Escalante, 2010). The basic model for a public cloud is similar to that for a public power utility: a third-party vendor manages the infrastructure necessary to deliver computing capability to customers who pay usage fees

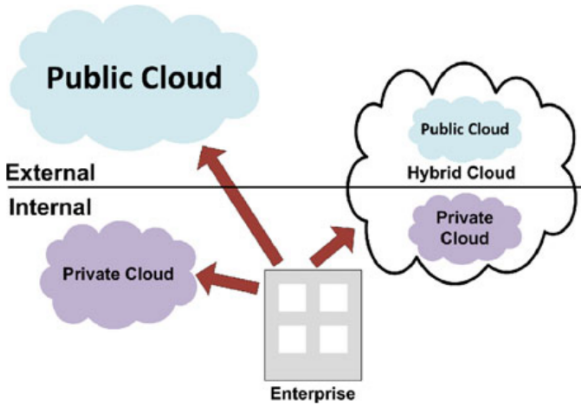

Figure 2.1 Cloud Types (Furht and Escalante, 2010)

(Eucalyptus, 2012). However, public clouds lack fine-grained control over data, network and security settings, which hampers their effectiveness in many business scenarios (Zhang *et al.*, 2010).

A *Private cloud* is a cloud that implements the "cloud computing" model in a "private" setting where only a single organization has access to the resources that are used to implement the cloud (Marks and Lozano, 2010; Eucalyptus, 2012). It means, this is a cloud that an organization implements using its own resources (machines, networks, storage, data centers, etc.). Recent advances in virtualization and data center consolidation have allowed corporate network and datacenter administrators to effectively become service providers that meet the needs of their customers within their organization (Furht and Escalante, 2010).

A *Hybrid cloud*, as the name implies, is a combination of the two types of clouds (Eucalyptus, 2012; Furht and Escalante, 2010). Combines computing resources (e.g., machines, network, storage, etc.) drawn from one or more public clouds and one or more private clouds at the behest of its users. Hybrid clouds offer more flexibility than both public and private clouds. Specifically, they provide tighter control and security over application data compared to public clouds, while still facilitating on-demand service expansion and contraction (Zhang *et al.*, 2010).

2.1.3 Related technologies

Cloud computing is often compared to the following technologies, which share certain aspects with cloud computing:

Grid computing: This is a distributed computing paradigm that coordinates networked resources to achieve a common computational objective. Cloud computing is similar

to Grid computing, since it also employs distributed resources to achieve application-level objectives. However, cloud computing takes one step further by leveraging virtualization technologies at multiple levels (hardware and application platform) to realize resource sharing and dynamic resource provisioning (Zhang *et al.*, 2010; Stanoevska-Slabeva *et al.*, 2009).

Utility computing: Utility computing represents the model of providing resources on-demand and charging customers based on usage rather than a flat rate. With on-demand resource provisioning and utility-based pricing, service providers can truly maximize resource utilization and minimize their operating costs. Cloud computing can be perceived as a realization of utility computing (Zhang *et al.*, 2010; Stanoevska-Slabeva *et al.*, 2009).

Virtualization: One of the most important ideas behind cloud computing is scalability, and the key technology that makes that possible is virtualization (Menken and Blokdijk, 2009). Virtualization is the emulation of hardware within a software platform. This allows a single computer to take on the role of multiple computers. A virtual machine behaves exactly like a physical computer and contains its own virtual CPU, RAM hard disk and network interface card (NIC). In practical terms, virtualization provides the ability to run applications, operating systems, or system services in a logically distinct system environment that is independent of a specific physical computer system (von Hagen, 2008). The virtualization technology has existed since the early days of computer science and attracts a heavy attention from the research community even today (Thein and Park, 2009).

2.1.4 Open source clouds

Exists some open source clouds that can be used to provide services in a private or a hybrid cloud. Three open platforms are listed: Eucalytptus, Nimbus and OpenNebula:

Eucalyptus is a software platform for the implementation of private cloud computing on computer clusters. There is an open-core enterprise edition and an open-source edition. Currently, it exports a user-facing interface that is compatible with the Amazon EC2 and S3 services but the platform is modularized so that it can support a set of different interfaces simultaneously (Eucalyptus, 2012). Eucalyptus can use a variety of virtualization technologies including VMware, Xen and KVM hypervisors to implement the cloud abstractions.

Nimbus Platform is an integrated set of open source tools that allow users to easily leverage IaaS cloud computing systems. This includes application instantiation, configuration, monitoring, and repair. Nimbus is an open source project focused on cloud

computing, it is built around three goals targeting three different communities: Enable resource owners to provide their resources as an infrastructure cloud; Enable cloud users to access infrastructure cloud resources more easily; Enable scientists and developers to extend and experiment with both sets of capabilities. The Nimbus project has been created by an international collaboration of open source contributors and institutions (Nimbus, 2012; Sempolinski and Thain, 2010).

OpenNebula is an open-source cloud computing toolkit for managing heterogeneous distributed data center infrastructures. The OpenNebula toolkit manages a data center's virtual infrastructure to build private, public and hybrid IaaS clouds. OpenNebula orchestrates storage, network, virtualization, monitoring, and security technologies to deploy multi-tier services as virtual machines on distributed infrastructures, combining both data center resources and remote cloud resources, according to allocation policies (OpenNebula, 2012). OpenNebula, by default, uses a shared file system, typically NFS, for all disk images files and all files for actually running the OpenNebula functions (Sempolinski and Thain, 2010).

2.2 Dependability: Basic concepts

Many of the wished features of a cloud system are related to the concept of dependability. There is no unique definition of dependability. By one definition, it is the ability of a system to deliver the required specific services that can justifiably be reliable (Nurmi *et al.*, 2009). It is also defined as the system property that prevents a system from failing in an unexpected or catastrophic way (Avizienis *et al.*, 2004).

Indeed, dependability is also related to disciplines, such as availability and reliability. Availability is the ability of the system to perform its slated function at a specific instant of time or over a stated period of time. It is generally expressed in the form of a ratio of the units of time when service was available and the agreed service period (Trivedi *et al.*, 2009).

In general, the basic reliability concept is defined as the probability that a system will perform its intended function during a period of running time without any failure (Musa, 1998). A failure causes the system performance to deviate from the specified performance (Xie *et al.*, 2004).

Dependability is a very important property for a cloud system as it should provide services with high availability, high stability, high fault tolerance and dynamical extensibility. Because 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).

Software aging effects in cloud systems may affect the performance of communication among their components, what in a critical level would also have an impact on the system dependability. Therefore, the presence of many software layers in cloud systems raise the need of monitoring aging effects and proposing proper rejuvenation mechanisms in order to assure the requered dependability aspects cited here.

2.2.1 Fault versus failures

Software engineers use the terms defects, faults, and bugs interchangeably and occasionally interject the term failures when speaking about undesirable system behavior. Faults are defects that are in the system at some point in time. Failures are faults that occur in operation. Defects metrics measure faults. Reliability metrics measure failures. Bugs are synonymous with defects and faults (Laird and Brennan, 2006).

If the code contains faults but the faults are never executed in operation, then the system never fails. The mean-time-between-failures (MTBF) for the system will approach infinity and software availability will be 100% (Laird and Brennan, 2006).

2.3 Software aging and rejuvenation

This section presents some aspects about Software Aging and Rejuvenation.

2.3.1 Software aging

Software aging is the deterioration in the availability of system or application resources, data corruption, or numerical error accumulation (Castelli *et al.*, 2001). Potential fault conditions gradually accumulate over time, leading to either performance degradation or transient failures, or both. Some of the frequent culprits of aging are memory leaks, unreleased file locks, hanging threads, data corruption, storage space fragmentation, and accumulation of round-off errors (Laird and Brennan, 2006).

Software aging can be defined as a growing degradation of software's internal state during its operational life (Grottke *et al.*, 2008). The causes of software aging have been verified as the accumulated effect of software faults activation (Avizienis *et al.*, 2004) during the system runtime (Huang *et al.*, 1995; Shereshevsky *et al.*, 2003). Aging in a software system, as in human beings, is an accumulative process (Huang *et al.*, 1995; Trivedi *et al.*,

2000). The accumulating effects of successive error occurrences directly influence the aging-related failure manifestation. Software aging effects are the practical consequences of errors caused by aging-related fault activations (Matias and Filho, 2006). These faults gradually lead the system towards an erroneous state (Trivedi *et al.*, 2000; Huang *et al.*, 1995).

This gradual shifting is consequence of aging effects accumulation, being the fundamental nature of the software aging phenomenon. It is important to highlight that a system fails due to the consequences of aging effects accumulated over the time. For example, considering a specific load demand, an application server system may fail due to unavailable physical memory, which may be caused by the accumulation of memory leaks related to the lack of some variables deallocation, or similiar software faults. In this case, the aging-related fault is a defect in the code (e.g., erroneous use of $malloc\&free$ functions) that causes memory leaks; the memory leak is the observed effect of an aging-related fault (Matias and Filho, 2006).

The aging factors (Grottke *et al.*, 2008) are those input patterns that exercise the code region where the aging-related faults may be activated. The occurrence of such faults may lead the system to erroneous states. Considering such memory leakage per aging-related error occurrence, the server system may fail due to the main memory unavailability. Nevertheless, the related effect may be observable only after long run of the system.

The time to aging-related failure (TTARF) is an important metric for reliability and availability studies of systems suffering from software aging (Grottke *et al.*, 2008). Previous studies on the aging-related failure phenomenon show that the TTARF probability distribution is deeply influenced by the intensity with which the system gets exposed to aging factors, such as system workload (Bao *et al.*, 2005).

Due to the cumulative property of the software aging phenomenon, it occurs more intensively in continuously running software systems that are executed over a long period of time (Castelli *et al.*, 2001), such as cloud-computing framework software components (Matias *et al.*, 2010). In a long-running execution, a system suffering from software aging increases its failure rate due to the aging effect accumulation caused by successive aging-related error occurrences, which monotonically degrades the system internal state integrity (Castelli *et al.*, 2001; Matias and Filho, 2006). Problems such as data inconsistency, numerical errors, and exhaustion of operating system resources are examples of software aging consequences (Grottke *et al.*, 2008; Garg *et al.*, 1998).

Since the notion of software aging was introduced (Huang *et al.*, 1995), many theoretical and experimental researches have been conducted in order to characterize and

understand this important phenomenon. Monitoring the aging effects is an essential part of any aging characterization study (Avritzer and Weyuker, 1997). Many past-published studies have implemented aging monitoring in different system levels, however to the best of our knowledge this is the first work discussing the aging effects in a cloud-computing environment.

2.3.2 Software rejuvenation

Software rejuvenation is a proactive fault management technique aimed at cleaning up the system's internal state to prevent the occurrence of more severe crashes or failures in the future (Huang *et al.*, 1995; Laird and Brennan, 2006; Kourai and Chiba, 2011).

Once the aging effects are detected, mitigation mechanisms might be applied in order to reduce the impact of the aging effects on the applications or the operating system. The search for aging mitigation approaches resulted in the so-called software rejuvenation techniques (Matias and Filho, 2006; Vaidyanathan and Trivedi, 2005).

Since the aging effects are typically caused by hard to track software faults, software rejuvenation techniques look for reducing the aging effects during the software runtime, until the aging causes (e.g., a software bug) are fixed definitively (Huang *et al.*, 1995; Grottke *et al.*, 2008). Examples of rejuvenation approaches may be software restart or system reboot. In the former, the aged application process is killed and then a new process is created to substitute it (Matias and Filho, 2006). Replacing an aged process by a new one, we remove the aging effects accumulated during the application runtime. The same applies to the operating system in a system reboot.

A common problem during rejuvenation is the downtime caused during restart or reboot, since the application or system is unavailable during the execution of the rejuvenation action. In (Matias and Filho, 2006) is presented a zero-downtime rejuvenation technique for the apache web server, which was adapted for this work.

The methods of software rejuvenation are (Laird and Brennan, 2006):

- System restarts;
- System cleanups (partial rejuvenation);
- Application/process restart (partial rejuvenation);
- Node/application failover (in a cluster system).

Although reboots are dreaded by most of us, below are shown three important properties that make them desirable (Candea and Fox, 2001) :

1. a restart will unequivocally return software to its start state, which is usually the best understood and best tested state of the system. For example, the Patriot missile defense system (Office, 1992), used during the Gulf War, had a bug in its control software that could be circumvented only by rebooting every 8 hours. It was due to an error in a mathematical computation, which would accumulate and eventually render the system ineffectively (Marshal, 1992);

2. reboots provide a high confidence way to reclaim resources that are stale or leaked. At a major Internet portal, the front end Apache web servers are routinely quiesced, killed and restarted, in order to control known memory leaks that quickly accumulate under heavy load.

3. rebooting is easy to understand and employ. Simple mechanisms, in general, benefit from being easy to implement, easy to debug, and easy to automate.

System cleanups are processes that execute in the background, which monitor the ongoing resource consumption and clean up the internal state. Releasing file locks, garbage collection, flushing operating system kernel tables, and killing zombie processes are all examples of system cleanups (Laird and Brennan, 2006).

Rejuvenation action can occur either on a regularly scheduled (such as once every ten days, for example) or as needed, triggered by the monitoring of resource exhaustion or performance degradation.

2.4 Performance measurement

Performance analysis as applied computer science and engineering should be thought of as a combination of measurement, interpretation, and communication of a computer system's 'speed' or 'size' (sometimes referred to as 'capacity') (Lilja, 2000). Performance measurement can be done only if the actual system or a prototype exists.

Measurements of real systems are not very flexible, however, they provide information about the specific system being measured. A common goal of performance analysis is to characterize how the performance of a system changes as certain parameters are varied (Lilja, 2000).

There are many metrics that are commonly used in performance analysis. For instance, the system *response time* is the amount of time that elapses from when a user submits a request until the result is returned from the system. System *throughput* is a measure of the job numbers or operations that are completed per unit time (Lilja, 2000). However, the standard and simplest measures of variability are *range, deviation, variance, standard deviation*, *and index of variation* (Laird and Brennan, 2006).

- *Range*: The range of values for the mapping of the data that is calculated by subtracting the smallest from the largest.
- *Deviation*: The distance away from the mean of the measurement.
- *Variance*: A measurement of spread, which is calculated differently if it is for a full population or a sample population. For a full population, where N is the number of data points,

$$Variance = \sum(Deviations^2)/N, \tag{2.1}$$

  For a sample population,

$$Variance = \sum(Deviations^2)/(N-1), \tag{2.2}$$

- *Standard Deviation (SD)*: The "typical" distance from the mean.

$$SD = \sqrt{Variance}, \tag{2.3}$$

- *Index of Variation (IV)*: An index that indicates the reliability of the measurement.

$$IV = SD/mean \tag{2.4}$$

Figure 2.2 illustrates lower and higher variance. With low variance, the values cluster around the mean. With higher variance, they spread farther out.

The time is a fundamental quantity that needs to be measured to determine almost any aspect of a computer system's performance.

Any measurement tool has three important characteristics that determine the overall quality of its measurements (Lilja, 2000):

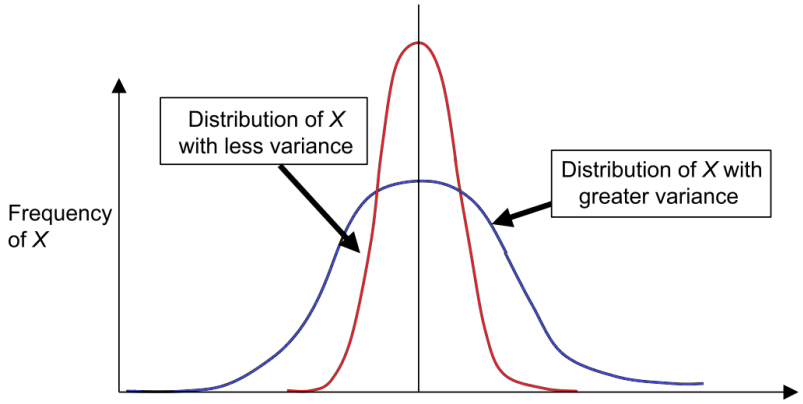

Figure 2.2 Examples of variance - adapted from (Laird and Brennan, 2006)

1. *Accuracy*. In the case of the timer, accuracy is an indication of the measurement timer closeness to a standard measurement of time defined by a recognized standards organization. More generally, accuracy is the absolute difference between a measured value and the corresponding reference value.

2. *Precision*. Relates to the repeatability of the measurements made with the tool. It is sometimes easier to think of precision in terms of its inverse. *Imprecision* is the amount of scatter in the measurements obtained by making multiple measurements of a particular characteristic of the system being tested.

3. *Resolution*, is the smallest incremental change that can be detected and displayed. The finite resolution of a measuring tool introduces a quantization effect into the values it is used to measure.

To obtain an intuitive view of the differences between accuracy and precision, Figure 2.3 shows an illustration plot using hypothetical measurements.

The accuracy of a measurement is the degree to which a measurement reflects reality. Precision of a measurement represents the size of differentiation possible with a measurement (Laird and Brennan, 2006).

There are some types of errors in experimental measurements. Presents two more important errors: systematic error and random error (Laird and Brennan, 2006).

- *systematic* error, meaning it is an error in the measurement system. It will show up every time. It affects the validity of the measurement. It is the “bias” with a measurement.

  Figure 2.4 is a graph of systematic error. Systematic errors change the mean but not the variance.

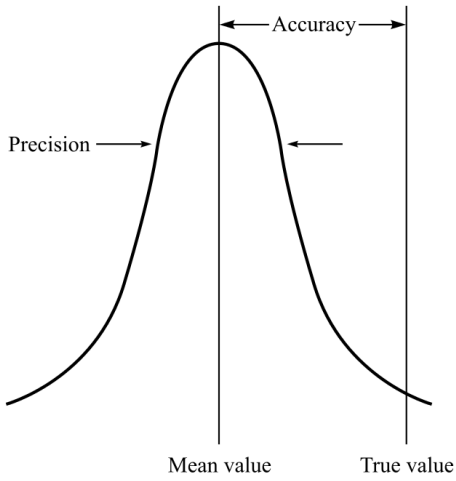

Figure 2.3 A illustration plot showing the differences between accuracy and precision (Lilja, 2000)

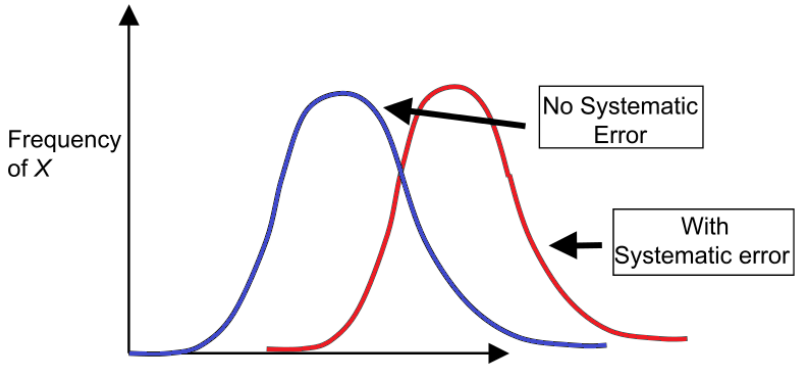

Figure 2.4 Distributions of X with and without systematic error - adapted from (Laird and Brennan, 2006)

- *random* error, meaning it is an error in the measurement itself, which will appear "at random". It affects the reliability of the measurement. It is the "noise" within a measurement.

  Figure 2.5 is a graph of random error. Random errors increase the variance but do not change the mean.

Figure 2.6 is a final look at measurement error. If reality is the vertical line on the left, then the distance from the actual mean to reality is the bias, or systematic error.

One important concern with any measurement strategy is how much it perturbs the system being measured. We list yet four measurements strategies (Lilja, 2000):

1. *Event-driven*. This measurement strategy records the information necessary to calculate the performance metric whenever the preselected event or events occur. One of the

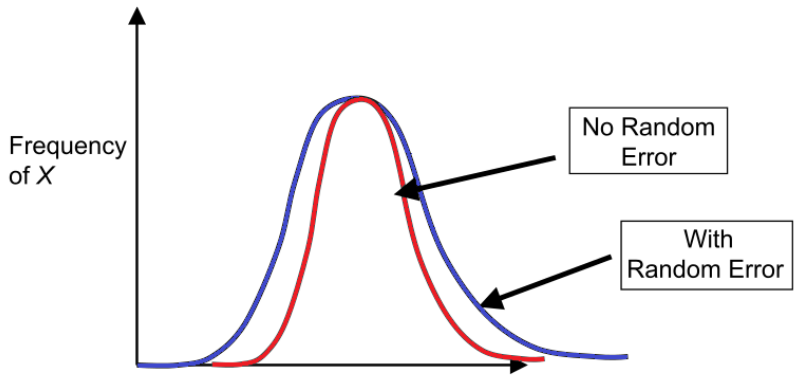

Figure 2.5 Distributions of X with and without random error - adapted from (Laird and Brennan, 2006)

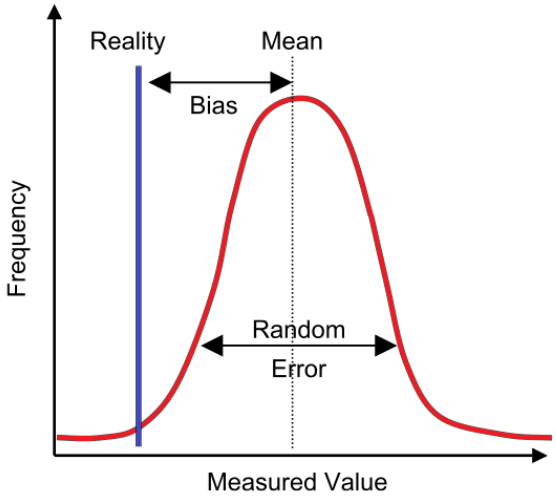

Figure 2.6 Measurement error - adapted from (Laird and Brennan, 2006)

advantages of an event-driven strategy is that the system overhead required to record the necessary information is incurred only when the event of interest actually occurs. If the event never occurs, or occurs only infrequently, the perturbation to the system will be relatively small. This characteristic can also be a disadvantage, however, when the events being monitored occur very frequently.

2. *Tracing*. A tracing strategy is similar to an event-driven strategy, except that, rather than simply recording that fact that the event has occurred, some portion of the system state is recorded to uniquely identify the event. For example, instead of simply counting the number of page faults, a tracing strategy may record the addresses that caused each of the page faults.

3. *Sampling*. In contrast to an event-driven measurement strategy, a sampling strategy

records at fixed time intervals the portion of the system state necessary to determine the metric of interest. As a result, the overhead due to this strategy is independent of the number of times a specific event occurs.

4. *Indirect*. An indirect measurement strategy must be used when the metric that is to be determined is not directly accessible. In this case, you must find another metric that can be measured directly, from which you then can deduce or derive the desired performance metric.

2.5 Time series

To Chatfield (1996) a time series is a collection of observations made sequentially in time. Examples occurs in a variety of fields, ranging from economics to engineering, and methods of analysing time series constitute an important area of statistics.

A time series can be represented by a set of observations of a random variable, arranged sequentially over time (Kedem and Fokianos, 2002). The value of the series at a given instant t can be described by a stochastic process, i.e. a random variable $X(t)$, for each $t \in T$, and T is an arbitrary set and its associated probability distribution. In most situations, t represents time, but can also represent another physical quantity, for example, space. The applications of time series analysis are mainly: description, explanation, process control and prediction.

- *Description*: series properties as, for example, the trend pattern, the existence of structural changes, etc.
- *Explanation*: to build models that explain the behavior observed during the series.
- *Process control*: for example, statistical quality control.
- *Prediction*: predict future values based on past values.

Time series enables one to build models that explain the behavior of the observed variable and the types of time series analyses may be divided into frequency-domain (Bloomfield, 2000; Kedem and Fokianos, 2002) and time-domain methods (Akaike, 1969; Box and Jenkins, 1970; Chatfield, 1996).

The modeler must decide how to use the chosen model according to her goals. Many forecasting models are based on the “least squares” method. The most common time series models are based on errors (or regression), autoregressive moving average (ARMA)

models, the autoregressive integrated moving average (ARIMA) models, the long memory ARIMA (also called ARFIMA models), the structural models and the nonlinear models (Kedem and Fokianos, 2002).

2.5.1 Trend analysis

To model a time series, the first thing to do is a graphical representation of the series, because through the graph we can identify the characteristics that may be relevant to the study of that series. The next step is an analysis of the observed data in order to identify possible trends.

This work adopts five models, namely: the linear model, the quadratic model, the exponential growth model, the model of the Pearl-Reed logistic, and the ARIMA model. These models are briefly described as follows, based on $Y_t = E[X(t)]$:

- *Linear Trend Model (LTM)*: this is the default model used in the analysis of trends. Its equation is given by $Y_t = \beta_0 + \beta_1 \cdot t + e_t$ where β_0 is known as the y-intercept, β_1 represents the average of the exchange of one time period to the next, and e_t is the error of fit between the model and the real series (Montgomery *et al.*, 2008).

- *Quadratic Trend Model (QTM)*: this model takes into account a smooth curvature in the data. Its representation is given by $Y_t = \beta_0 + \beta_1 \cdot t + \beta_2 \cdot t^2 + e_t$ where the coefficients have similar meanings as the previous item (Montgomery *et al.*, 2008).

- *Growth Curve Model (GCM)*: this is the model of trend growth or fallen in exponential form. Its representation is given by $Y_t = \beta_0 \cdot \beta_1^t \cdot e_t$.

- *S-Curve Trend Model (SCTM)*: this model fits the logistics of Pearl-Reed. It is usually used in time series that follow the shape of the curve S. Its representation is given by $Y_t = 10^a/(\beta_0 + \beta_1 \beta_2^t)$.

- *ARIMA:* in general, an ARIMA model is characterized by the notation ARIMA(p,d,q) where p,d and q denote orders of auto-regression, integration (differencing) and moving average, respectively. Such parameters may be estimated by means of autocorrelation analysis and verification of differencing steps needed to transform the series in a stationary series (Box and Jenkins, 1970).

Error measures (Schwarz, 1978) are adopted for choosing the model that best fits the observed data. MAPE, MAD and MSD were the error measures adopted in this work:

- *MAPE (Mean Absolute Percentage Error)* represents the precision of the estimated values of the time series expressed in percentage. This estimator is represented by:

$$MAPE = \frac{\sum_{t=1}^{n} |(Y_t - \widehat{Y}_t)/Y_t|}{n} \cdot 100, \tag{2.5}$$

  where Y_t is the actual value observed at time t ($Y_t \neq 0$), $\widehat{Y}_t$ is the estimated value and n is the number of observations.

- *MAD (Mean Absolute Deviation)* represents the accuracy of the estimated values of the time series. It is expressed in the same unit of data. MAD is an indicator of the error size and is represented by the statistics:

$$MAD = \frac{\sum_{t=1}^{n} |(Y_t - \widehat{Y}_t)|}{n}, \tag{2.6}$$

  where Y_t, t, $\widehat{Y}_t$ and n have the same meanings index MAPE.

- *MSD (Mean Squared Deviation)* is a more sensitive measure than the MAD index, especially in large forecasts. Its expression is given by

$$MSD = \frac{\sum_{t=1}^{n} |(Y_t - \widehat{Y}_t)|^2}{n}, \tag{2.7}$$

  where Y_t, t, $\widehat{Y}_t$ and n have the same meanings of the previous indexes.

3

Eucalyptus cloud computing infrastructure

O que fazemos em vida, ecoa na eternidade.
What we do in life echoes in eternity.
—MAXIMUS (Gladiator Movie)

EUCALYPTUS - Elastic Utility Computing Architecture Linking Your Programs To Useful Systems - is a software that implements scalable IaaS-style private and hybrid clouds (Eucalyptus, 2010a). It was created with the purpose of cloud computing research and it is interface-compatible with the commercial service Amazon EC2 (Jones, 2008; Eucalyptus, 2012). The API compatibility enables to run an application on Amazon and on Eucalyptus without modification. Eucalyptus was developed at the University of California, Santa Barbara, for the purpose of cloud computing research.

In general, the Eucalyptus cloud computing platform uses the virtualization capabilities (hypervisor) of the underlying computer system to enable flexible allocation of computing resources decoupled from specific hardware (Eucalyptus, 2010a). Eucalyptus (Eucalyptus, 2009) is compatible with and packaged for multiple distributions of Linux including Ubuntu, RHEL, OpenSuse, Debian, Fedora, and CentOS.

There are five high-level components in the Eucalyptus architecture (Figure 3.1), each with its own web service interface: *Cloud Controller, Cluster Controller, Node Controller, Storage Controller*, and *Walrus* (Eucalyptus, 2010a). A brief description of the components within the Eucalyptus system follows.

3.1 Cloud Controller (CLC)

The *Cloud Controller (CLC)* is the front-end to the entire cloud infrastructure. The CLC is responsible for exposing and managing the underlying virtualized resources (servers,

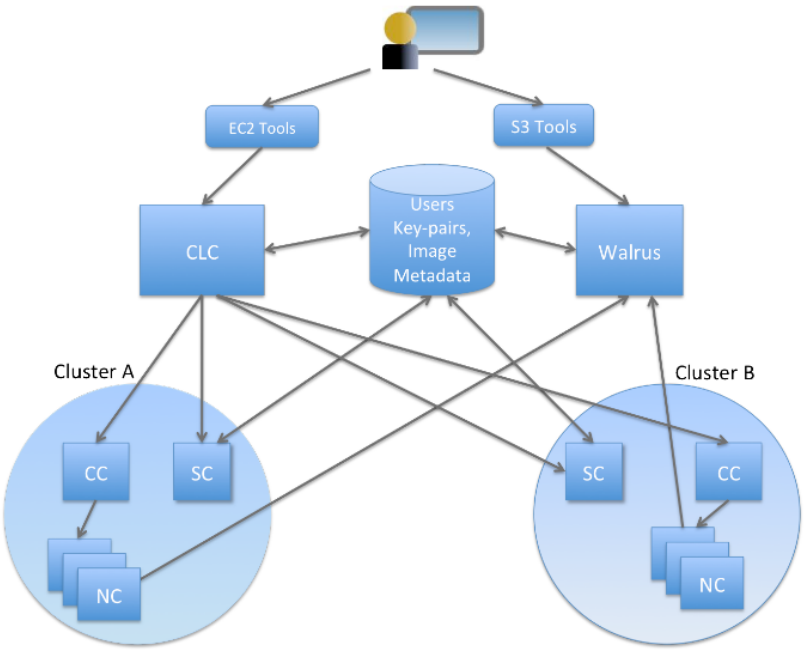

Figure 3.1 Five Eucalyptus high-level components (Eucalyptus, 2010b)

network, and storage) via Amazon EC2 API (Microsystems, 2009). This component uses web services interfaces to receive the requests of client tools on one side and to interact with the rest of Eucalyptus components on the other side.

3.2 Cluster Controller (CC)

The *Cluster Controller (CC)* usually executes on a cluster front-end machine (Eucalyptus, 2010a) (Eucalyptus, 2009), or on any machine that has network connectivity to both the nodes running *Node Controllers (NC)* and to the machine running the *Cloud Controller*. CCs gather information on a set of VMs and schedules VM execution on specific NCs. The *Cluster Controller* has three primary functions: schedule incoming instance run requests to specific NCs, control the instance virtual network overlay, and gather/report information about a set of NCs (Eucalyptus, 2009).

3.3 Node Controller (NC)

Node Controller (NC) runs on each node and controls the life cycle of instances running on the node. The NC interacts with the operating system and with the hypervisor running on the node. The actions of a NC are managed by the *Cluster Controller (CC)*.

NCs control the execution, inspection, and termination of VM instances on the host where it runs, fetches and cleans up local copies of instance images. It queries and controls the system software on its node in response to queries and control requests from the *Cluster Controller* (Eucalyptus, 2010a). A NC makes queries to discover the node's physical resources - number of CPU cores, size of memory, available disk space - as well as to learn about the state of VM instances on that node (Eucalyptus, 2009; Murari *et al.*, 2010).

3.4 Storage Controller (SC)

Storage controller (SC) provides persistent block storage for use by the virtual machine instances. It implements block-accessed network storage, similar to that provided by Amazon Elastic Block Storage - EBS (Amazon, 2011a), and it is capable of interfacing with various storage systems (NFS, iSCSI, etc.). An elastic block storage is a Linux block device that can be attached to a virtual machine but sends disk traffic across the locally attached network to a remote storage location. An EBS volume can not be shared across instances (Murari *et al.*, 2010).

3.5 Walrus

Walrus is a file-based data storage service, that is interface compatible with Amazon's Simple Storage Service (S3) (Eucalyptus, 2009). Walrus implements a REST interface (through HTTP), sometimes termed the "Query" interface, as well as SOAP interfaces that are compatible with S3 (Eucalyptus, 2009; Murari *et al.*, 2010). Users that have access to Eucalyptus can use Walrus to stream data into/out of the cloud as well as from instances that they have started on nodes. In addition, Walrus acts as a storage service for VM images. Root filesystem as well as kernel and ramdisk images used to instantiate VMs on nodes can be uploaded to Walrus and accessed from nodes.

3.6 Other additions

Here some additions to Eucalyptus:

Command Line Tools (Euca2ools) - Euca2ools from Eucalyptus provide a bunch of command line tools to manage the eucalyptus setup (Murari *et al.*, 2010; Eucalyptus, 2012). These commands help you manage images, instances, storage, networking etc. The tools were inspired by command-line tools distributed by Amazon (*api-tools and ami-tools*)

and largely accept the same options and environment variables. However, these tools were implemented from scratch in Python, relying on the Boto library and M2Crypto toolkit Eucalyptus (2012).

Elasticfox - This is an open source Mozilla Firefox extension for interacting with Amazon Compute Cloud (Amazon EC2): Launch new instances, mount Elastic Block Storage volumes, map Elastic IP addresses, and more. This was originally written for EC2, but, since version 1.7, Elasticfox added Eucalyptus support as well, because API of Eucalyptus is compatible with that of EC2. The source code also functions as an example of how to use the Amazon EC2 Query API from JavaScript.

Hybridfox was forked from Elasticfox to make it usable with Eucalyptus, when Elasticfox worked only with AWS. Hybridfox can be used as the single interface to AWS and Eucalyptus. The interface of Hybridfox is similar to Elasticfox. Even after the recent release of Elasticfox (version 1.7) with support for Eucalyptus, Hybridfox has the following additional features that make it attractive for users of Eucalyptus: Raising instances with Private IP; Some quirks related to hard coded mapping of instance types and architecture types addressed; Support for Eucalyptus 1.5.x as well as 1.6.x; Other usability enhancements Murari *et al.* (2010).

4

Sofware aging in Eucalytpus cloud computing environment

Se soubéssemos o que era que estávamos fazendo, não seria chamado pesquisa, seria?
If we knew what it was we were doing, it would not be called research, would it?

—ALBERT EINSTEIN (Phrase)

To investigate the software aging effects of Eucalyptus platform can be quite complicated in view of the environment that is very complex and have many resources to be monitored, where each one could show signs of aging. Therefore, the system administrator should be familiar with the environment. It was decided to look at all possible resources, such as CPU, disk and memory of the entire environment, beyond the use of specific resources of Eucalyptus processes such as *eucalyptus-cloud* and *eucalyptus-nc (Apache2)*, besides the existence of zombie processes.

It is necessary to use a strategy or mechanism to monitore automatically all infrastructure resources. As there is no only one specific software to obtain all data needed but a lot of software, the creation of scripts with specific functions is required. Some programming languages could be used for this task.

Languages like Python and Shell Script can be used (Mitchell *et al.*, 2001; Blum, 2008) to monitore system resources. Tests using Python and Shell script language shows that the both options are effectives in capturing and manipulating data, but for this moment Shell Script was chosen. The first one shows problems in converting string to integer data and others problems with data capture in rows and columns. Each script was built using

native Linux programs, such as: *date*, *mpstat*, *pidstsat*, *free*, *ps*, *du*, among others.

Some steps are required during a software aging investigation. Thess steps are show in Figure 4.1 and enumerated below:

1. The first step is the implementation of monitoring scripts, your function is to collect system information during the evaluation period.

2. The second step is to choose the workload that will be used during the experiment. The workload is essential in the software aging investigation, because will accelerate the software lifetime and, thus, to present the degradation effects that occurs in the system. After, the script with the workload choosed is implemented.

3. In the third step, a stressful test is executed. This test is performed in a short period, the objective is to stress all environment resources. If any aging effect is observed in this preliminary experiment, then passes it to the next step. Otherwise, the cycle ends.

4. Where aging effects were observed, a stressful experiment is performed, this time for an extended period.

5. This step is to analyze the experimental data obtained by monitoring. All data are analyzed to verify the aging occurrences.

6. This step gathers all results obtained with this experiment, ending the cycle.

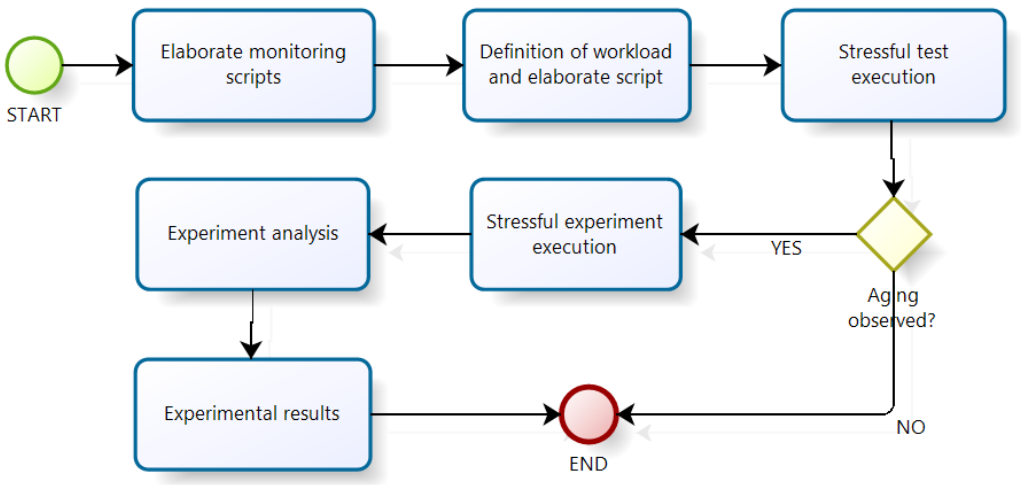

Figure 4.1 Approach overview software aging

4.1 General resources monitoring

The resources below were monitored using custom scripts to capture only useful data to research. They all contain the $date --rfc-3339 = seconds$ command to obtain the exactly moment that the resource information was captured. The historic was collected every 60 seconds and stored in a text document for further data analyse. This frequency for data collection was chosen to not miss any important events in system behavior, and also to avoid interference by monitoring mechanism in the system behavior. All general resources monitored and their major commands are shown below:

- CPU utilization is monitored by *mpstat* command. This command shows the total use of CPU applied by the User, System, I/O wait and idle. The complete script can be seen in Appendix B.1.
- Disk utilization is monitored by *df (disk free)* command. This command shows informations about all disk partitions, disk space used and available. The complete script can be seen in Appendix B.2.
- Memory - *free* command. This command returns information about the overall memory of the machine, such as free memory, used and buffer cache, in addition to free memory and swap used. The complete script can be seen in Appendix B.3.

4.2 Specific resources monitoring

The resources here named specifics are so called precisely because they do not look at the overall system, but only for specific processes of Eucalyptus. Knowing that these processes change the PID (Process Identifier) often, all processes below were located using the *ps* command plus basic information about the column position of a particular line caught with *awk* command, which allows to obtain the PID of each process. Even changing their PID, the process will be monitored in the same way because the new PID is detected automatically by the script.

Like the previous scripts, they are also followed by the $date --rfc-3339 = seconds$ command to obtain the date and time the process was monitored. The information from these resources are also obtained every 60 seconds and stored in a text document for later data analysis. All specific resources monitored and their major commands are shown below:

- *Eucalyptus-cloud* process utilization resources is monitored by *pidstat* command. After locating this process using *ps* command, the *pidstat* command is used to obtain specific informations from resources used by this process, such as CPU, virtual memory and resident memory. *Eucalyptus-cloud* process is only found in the cloud controller. The complete script can be seen in Appendix B.4.

- *Eucalyptus-nc (Apache2)* process utilization resources is monitored by *pidstat* command. After locating this process using *ps* command, the *pidstat* command is used to obtain specific informations from resources used by this process, such as CPU, virtual memory and resident memory. *Eucalyptus-nc (Apache2)* process only exists on node controllers. The complete script can be seen in Appendix B.5.

- *Zombie* processes occurrences are monitored by *ps* command. A zombie or defunct process is a process that has completed execution but still has an entry in the process table. Also found in a similar way to the previous cases, but in this case the ps command capture the letter "Z" in the "STAT" column on data field of this process. The complete script can be seen in Appendix B.6

Zombies that exist for more than a short period of time typically indicate a bug in the parent program, or just an uncommon decision to reap children Herber (1997). If the parent program is no longer running, zombie processes typically indicate a bug in the operating system. The presence of a few zombies is not worrisome in itself, but may indicate a problem that would grow serious under heavier loads. Since there is no memory allocated to zombie processes except for the process table entry itself, the primary concern with many zombies is not running out of memory, but rather running out of process ID numbers.

4.3 Log files

Another way to get information from the environment is to use the logs generated by the Eucalyptus. The activity log contains comprehensive information about the operations being performed. The log provides an audit trail of all tasks performed (with the latest activities related to last one) and can be useful for solving problems that may occur.

Usually when an issue arises in Eucalyptus, records can suggest the nature of the problem either in the eucalyptus log files or in the system log files. Assuming Eucalyptus is installed in the root (/) directory, the Eucalyptus log files by default are located on each machine hosting a component in the following directory: $var/log/eucalyptus/$. Here are the relevant logs for each component:

- Cloud Controller (CLC): cloud-debug.log, cloud-error.log, cloud-output.log, axis2c.log
- Node Controller (NC): nc.log, httpd-nc_error.log, euca_test_nc.log
- Cluster Controller (CC): cc.log, httpd-cc_error_log, registration.log
- Storage Controller (SC): sc-stats.log, registration.log
- Walrus: walrus-stats.log, registration.log

4.4 Memory leaking monitoring

Memory leaking in long running programs can cause system performance degradation, and even system crash (Xu and Zhang, 2008).

Many memory leaking problems are subtle and hard to be detected. They have few symptoms other than the slowing down system's performance and steady increase in memory consumption (Xu and Zhang, 2008). Memory leaks occur because a block of memory was not freed when it should, hence it is a primary cause for possible further system failure (Hastings and Joyce, 1992).

To Ni *et al.* (2008) memory leak has been exposed to be one of the most serious bugs which are hard to locate and fix in computer programs. Eventually, in the worst case, too much of the available memory may become allocated and all or part of the system or device stops working correctly, the application fails, or the system slows down unacceptably due to thrashing.

Considering the large number of resources in the system a priority list of resources to be monitored had to be defined. The top list processes chosen to be monitored were *eucalyptus-cloud* and *eucalyptus-cc (Apache2)* in the Cloud Controller, and the *eucalyptus-nc (Apache2)* process in both 32-bit and 64-bit machines. These processes have been chosen since they are the main Eucalyptus process running on each machine.

Python was chosen as the language to write the scripts due to its support for concatenating monitored information and its fast learning curve. The script allows finding the PID of the process through the *ps* command, after it carried out the monitoring directly from */proc (/proc/"+pid+"/status)*, where it is possible check a lot of information. In this case, the objective is memory consumption of each process.

A history of the memory usage is stored in a spreadsheet which allows its analysis and detection of possible anomalies. As already mentioned, the time interval between measures is 60s.

The memory leaking, in short, occurs when a process uses the memory required for their execution and continues after several requests by allocating more memory without deallocating the previously one, which may cause the machine resource depletion. The three possible memory states and their transitions are shown in Figure 4.2.

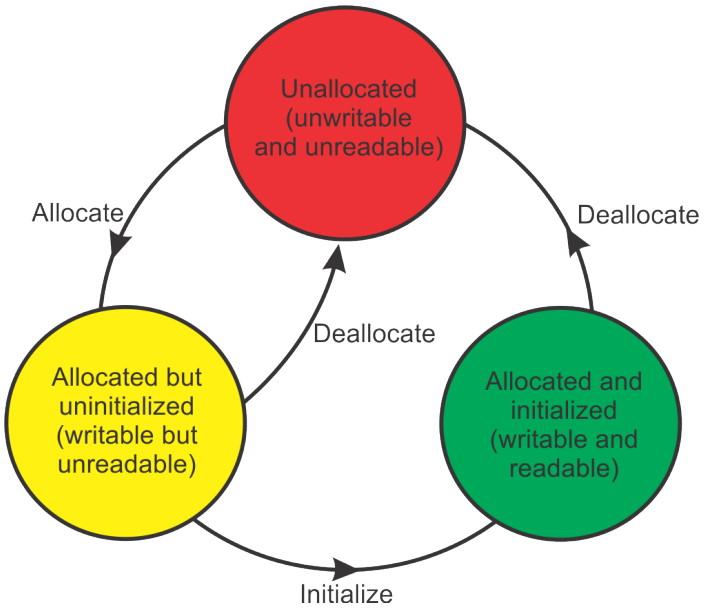

Figure 4.2 Memory state transition diagram

The allocation algorithm first searches for blocks of pages of the size requested. It follows the chain of free pages that is queued on the list element of the *free_area* data structure. If no blocks of pages of the requested size are free, blocks of the next size are looked for. This process continues until all of the *free_area* has been searched or until a block of pages has been found *matias2010-measur, Rusling:1999*. Allocating blocks of pages tends to fragment memory with larger blocks of free pages being broken down into smaller ones. The page deallocation code recombines pages into larger blocks of free pages whenever it can. Each time two blocks of pages are recombined into a bigger block of free pages the page deallocation code attempts to recombine that block into a yet larger one. In this way the blocks of free pages are as large as memory usage will allow.

A *write* to memory that contains any bytes that are currently in an *unwritable* state causes a diagnostic message be printed; a similar message printed if the program-under-test reads bytes marked *unreadable*. Writing *uninitialized* memory causes memory's state to become *initialized*. When *malloc* allocates memory, the memory's state is changed from *unallocated* to *allocated-but-uninitialized*. Calling *free* causes the affected memory to enter the *unallocated* state (Hastings and Joyce, 1992).

4.5 Memory fragmentation monitoring

Fragmentation is a phenomenon in which memory space is used inefficiently, so that the system is not able to reuse memory that is free (Baker, 1995). Memory fragmentation may reduce system capacity and performance and it is considered a software aging effect (Grottke *et al.*, 2008).

Memory fragmentation is one of the serious problems that occurs in software. The system reports a memory full message even if the total free space is more than the requested memory slot. This might be related to the lack of contiguous memory space for the respective allocation (Baishnab *et al.*, 2010). In this contiguous memory allocation, each process is contained in a single contiguous section of memory, for example memory addresses 1000-2000, as opposed to "fragmented allocation" where the memory comes as several smaller blocks in different places, for example memory addresses 1000-1050, 2050-2125. Memory fragmentation exists when there is enough total memory space to satisfy a request, but the available spaces are not contiguous. Figure 4.3 shows a memory map illustration of contiguous memory and fragmented memory.

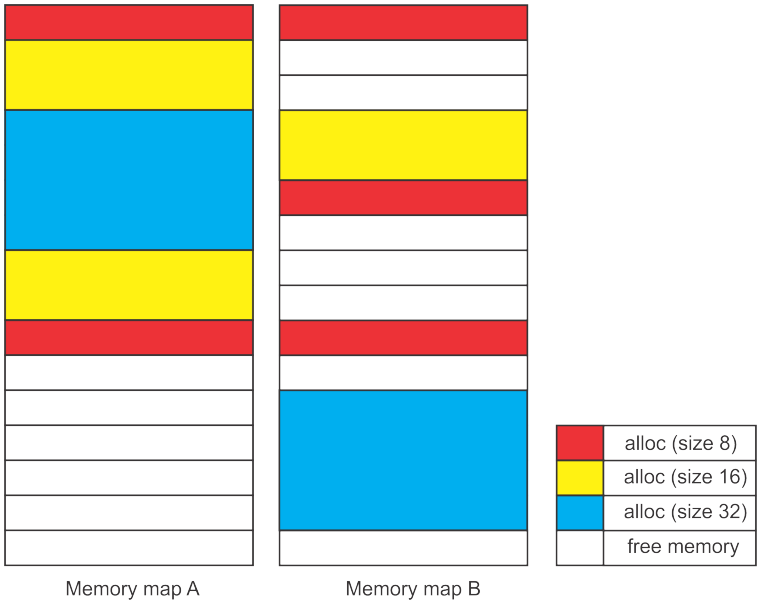

Figure 4.3 Memory map ilustration - A) Continguous memory and B) Fragmented memory

Internal Fragmentation may be quantified by the extra unused memory that a policy or system has allocated beyond actual memory size requested by the program (Skotiniotis and Chang, 2002).

By using *SystemTap* (SystemTap, 2012), memory fragmentation events were mea-

sured during the experiment, by means of a Linux kernel tracepoint named *mm_page_alloc_extfrag* (Matias *et al.*, 2010). Kernel tracepoints are a static kernel instrumentation support mechanism for Linux kernel source code, allowing special tools such as LTTng (LTTng, 2012) or SystemTap (SystemTap, 2012) to trace information exposed by these probe points.

A fragmentation monitoring script was developed using the specific language of SystemTap (Jacob *et al.*, 2009). Each time that occurs a fragmentation event in the OS kernel, the script identifies and records such occurrence in a spreadsheet for further data analysis. The collected data are the name of parent process, PID and UID, and the same information from child processes.

The use of this strategy is somewhat different from those previously used, because there is no telling at what time it will occur fragmentation on an event. This means that within a minute both may be several records of fragmentation, but none of them can occur during this period.

As during the testing phase various fragmentation events were found in a short period of time, the log events generated by the SystemTap was very extensive and, therefore, difficult to analyze. In this sense, to facilitate analysis of data, such information had to be filtered by a second script written just for this purpose, i.e, to filter only the data of the research interest. Another possible strategy is to use an array that stores the records and only from time to time (every ten minutes for example), write the information to the spreadsheet.

In the case of memory fragmentation, the mere fact of its existence does not immediately suggest the occurrence of software aging. However, a large amount of memory fragmentation events caused by the same process is a symptom that may explain an aging phenomenon.

4.5.1 SystemTap

SystemTap is a tool for the Linux Operating System that allows developers and system administrators to deeply investigate the behavior of the kernel and even user space applications in order to discover error conditions, performance issues, or just to understand how the system works (Jacob *et al.*, 2009; Eigler, 2010; Domingo and Cohen, 2011; SystemTap, 2012). Data may be extracted, filtered, and summarized quickly and safely, to enable diagnoses of complex performance or functional problems (Eigler, 2010).

SystemTap is a command line application that utilizes a plain text file (a script) as input and generates plain text output. The input file is a script created with the specific SystemTap language, which is similar to the C language (Jacob *et al.*, 2009). Figure 4.4 shows SystemTap processing steps.

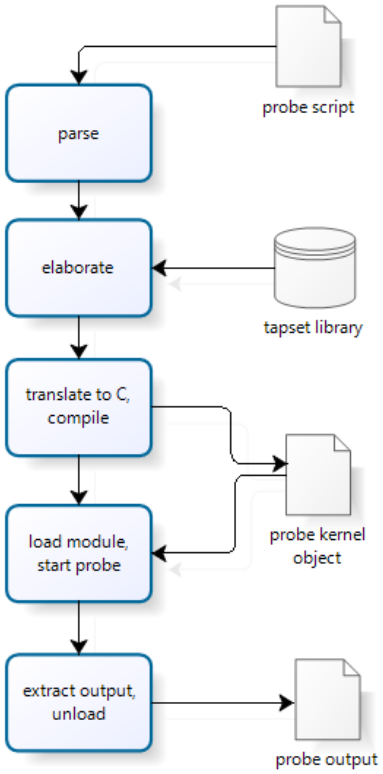

Figure 4.4 SystemTap processing steps - adapted from (Prasad *et al.*, 2005)

Elaboration is a processing phase that analyzes the input script and resolves references to the kernel or user symbols and tapsets. Tapsets are libraries of script or C code used to extend the capability of a basic script. Once a script has been elaborated, it is translated into C. A compiler converts the instrumentation script and tapset library into C code for a kernel module. After compilation and linking with the SystemTap runtime, the kernel module is loaded to start the data collection. Data is extracted from module into userspace via reliable and high performance transport. Data collection ends when the module is unloaded from the kernel (Prasad *et al.*, 2005).

A kernel event could be a function name, a specific line of code, an exception or interruption, meaning that when this event happens the handler will be executed (Mitchell *et al.*, 2001).

Two types of Kprobes are utilized by SystemTap, *Kprobes* and *return probes* Jacob *et al.* (2009):

1. A *Kprobe* is a general purpose hook that can be inserted in the kernel code to probe an instruction. The first byte of the instruction is replaced with the breakpoint instruction for the architecture being used (Dandamudi, 2005). When this breakpoint is hit, *Kprobe*

takes over execution, executes its handler code for the probe, and then continues execution at the next instruction (Jacob *et al.*, 2009).

2. A *return probe*, also called a *kretprobe*, attaches to the entry point of a function like a regular *Kprobe*. When the function is called, the *return probe* gets the return address and replaces it with a *trampoline address*. When the function exits, it returns to the *trampoline address* instead of where it was originally set to return, and the handler code for the probe is called. *Return probes* have access to the return values from functions (Jacob *et al.*, 2009).

Figure 4.5 provides a graphical representation of how a *kretprobe* works.

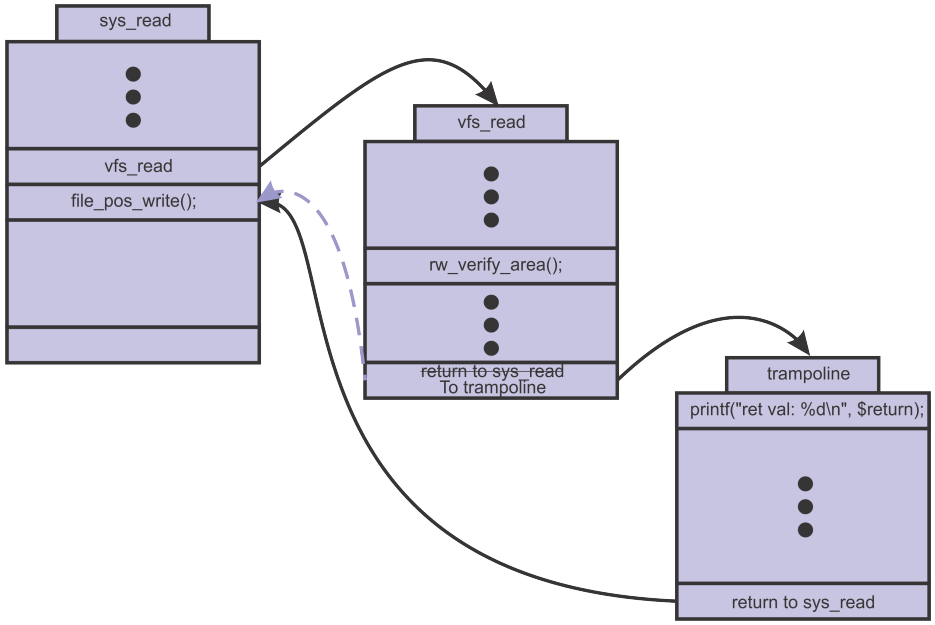

Figure 4.5 SystemTap operations overview - Adapted from (Jacob *et al.*, 2009)

4.6 Testbed environment

In order to analyze possible aging effects in the Eucalyptus cloud-computing framework, we use a test bed composed of six Core 2 Quad machines (2.66 GHz processors, 4 GB RAM) running the Ubuntu Server Linux 10.04 (kernel 2.6.35-24) and the Eucalyptus System version 1.6.1. The operating system running in the virtual machines is based on a custom image of the Ubuntu Linux 9.04 that runs a simple FTP server. The cloud environment under test is fully based on the Eucalyptus framework and the KVM hypervisor.

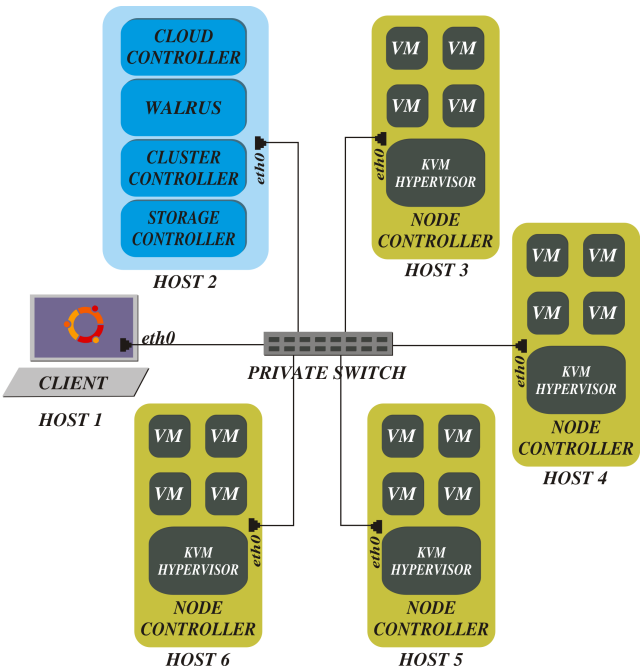

Figure 4.6 Components of the test bed environment - adapted from Murari *et al.* (2010)

The Figure 4.6 shows the components of our experimental environment.

The *Cloud Controller, Cluster Controller, Storage Controller* and *Walrus* have been installed on the same machine, and the VMs were instantiated on four physical machines, so that each of them run a *Node Controller*. Three of those machines have installed a 32-bit (i386) Linux OS, whereas one has the 64-bit (amd64) Linux OS, which allow us to capture possible different aging effects related to the system architecture. A single host is used to monitor the environment and also to perform requests to the *Cloud Controller*. That single host is a client for the cloud-computing infrastructure in our testbed. This infrastructure represents a small cloud with different system architectures.

In order to verify whether it occurs or not the phenomenon of software aging in Eucalyptus, we decide to monitor the environment in full operation during a certain time, so that it would be prone to constant failures. Thus, it is necessary to generate a workload able to stress the system and accelerates the failure process.

The impact caused by the monitoring mechanism is irrelevant to affect the system behavior, because the most affected processes are related to the Eucalyptus functioning .

4.6.1 Workload

Monitor the environment and "wait" that it presents the aging effects can take months or even years, maybe never occur. Thus, it was necessary to generate a workload able to stress the system and accelerates the failure process. According to the empirical knowledge acquired in the testing phase, it was found that activities such as instantiate, terminate and restart VMs increased the resources consumption and, therefore, could be used for generating the workload required to accelerate possible aging effects.

The workload chosen is the use of some of the Eucalyptus features to accelerate the life cycle of the VMs, which is composed by four states: *Pending, Running, Shutting Down* and *Terminated*, as shown in Fig. 4.7.

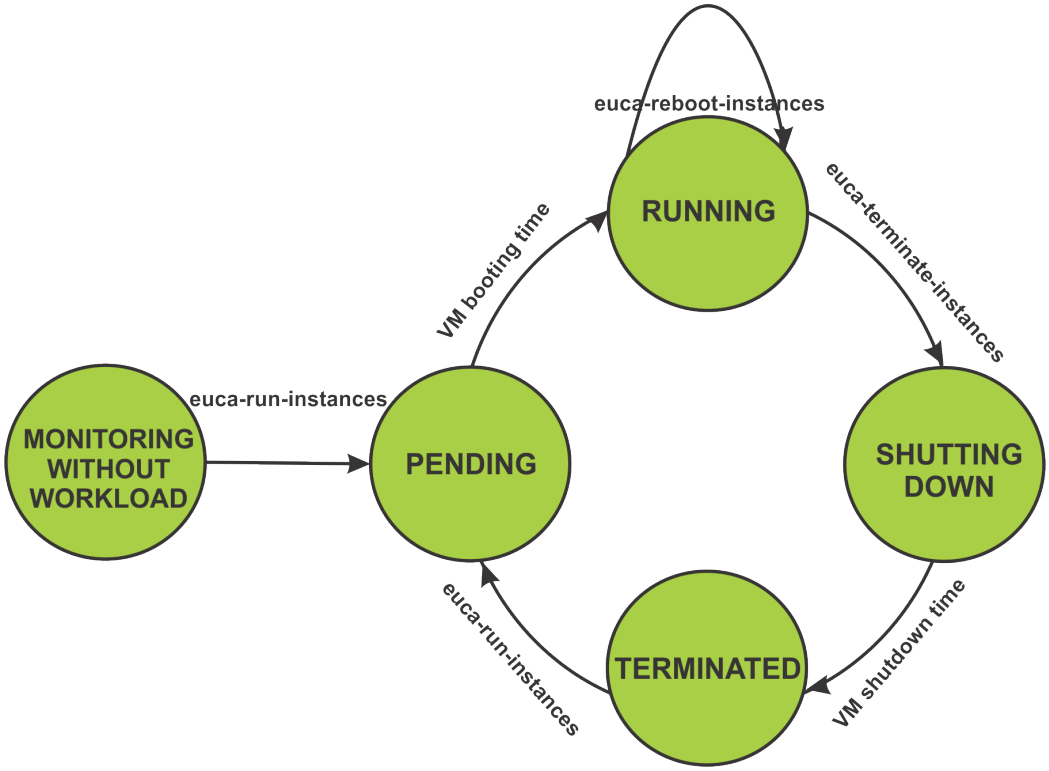

Figure 4.7 Workload Illustration

Scripts are used in order to start, reboot and terminate the VMs in a short time period. Such operations are central for scaling up and down the so-called elastic computing (Amazon, 2011b).

Cloud-based applications adapt themselves by instantiating new virtual machines, and save resources by terminating underused VMs when the load of requests is low. VM reboots are also essential to high availability mechanisms (VMware, 2007). The workload

cycle was implemented by means of shell script functions that perform the operations we have just mentioned, as it may be seen below:

- *Instantiate VMs Function*: This function uses *euca-run-instances* command to instantiate 8 VMs using type *m1.xlarge* in a cluster named *cluster-modcs*. Those VMs are instances of an Ubuntu Server Linux running a FTP server. This function is represented by a Shell script that is shown below:

```
1 InstancesVMs(){
2     i=0
3     while [ $i -lt 8 ]
4     do
5      euca-run-instances -t m1.xlarge -z cluster-modcs -k admin emi-3
           FDE12D4
6      i=`expr $i + 1`
7     done
8     sleep 10
9 }
```

- *Terminate VMs Function*: This function uses *euca-describe-instances* command to find out which instances are running in the cloud and terminate them all with *euca-describe-instances* command.

```
1 terminateVMs(){
2     instances=`euca-describe-instances | grep INSTANCE | awk '{print
          $2}'`
3     euca-terminate-instances $instances
4 }
```

- *Reboot VMs Function*: Much like the previous function, it also finds all existing instances (*euca-describe-instances* command), but instead of terminating, it requests their reboot (*euca-reboot-instances*).

```
1 rebootVMs(){
2     instances=`euca-describe-instances | grep INSTANCE | awk '{print
          $2}'`
3     euca-reboot-instances $instances
4 }
```

Every ten minutes the script checks whether more than five hours have passed from the beginning of the last initialization. If so, all VMs are killed, otherwise all VMs are

rebooted. Empirically, was decided to use ten minutes to reboot eucalyptus service and five hours to execute the terminate command and, after, this cycle is started again. This amount of executions generates a workload that can stress the Eucalyptus infrastructure. The times used here were chosen just to accelerate the aging effects and do not represent values from the "real life". The complete script can be seen in Appendix A.

It is important to highlight that the workload was set just to speed up the effects of aging, so that the time values that we have chosen do not attempt to illustrate real cases. For this experiment, monitoring scripts were implemented using Linux utilities, such as *date, mpstat, free, ps* and *du* (Blum, 2008). These scripts monitor the utilization of resources such as CPU, disk and memory usage. Those resources were monitored for the entire system as well as specific services like *eucalyptus-cloud* (in the *Cloud Controller*) and *eucalyptus-nc* (in the *Node Controllers*).

Once the experiment starts, information is gathered from the monitored resources every sixty seconds. Nevertheless, in the first two hours, no workload is provided to the system. After this first two hours, then, the scripts that instantiate the virtual machines are executed and the workload cycle (previously described) starts.

4.7 Performance data analysis

Data analysis is a process of inspecting, cleaning, transforming, and modeling data with the goal of highlighting useful information, suggesting conclusions, and supporting decision making.

Identifying the growth in the use of a particular resource alone may not characterize the occurrence of software aging, but it may indicate that this resource deserves attention, since patterns and/or abnormalities is a key issue for detecting aging phenomenon.

5

Rejuvenation strategies to Eucalytptus cloud computing environment

As oportunidades multiplicam-se à medida que são agarradas.
Opportunities multiply as they are seized.
—SUN TZU (The Art of War)

In few words, the software rejuvenation strategy aims to restore the state of the software when the beginning of its execution, acting proactively to avoid unwanted interruptions or failures. The software rejuvenation technique best known and perhaps generic for all applications is system restart, which theoretically restores the original state of the software by forcing it to start again from the beginning.

The big problem with this technique is the large downtime for the application. The downtime includes the time to restart the machine and the time to start again the same application. However, even in a pessimistic view, such an action is effective and it causes less damages than an unplanned service disruption. Usually, the restoration of the system is much easier in preventive actions than in corrective actions. A web server is an example where a large system downtime can affect a lot of user and cause catastrophic results. In the Case Studies Chapter is explained with more details about the downtime caused by the rejuvenation strategy adopted.

It is important to stress that the mere fact of finding an effective rejuvenation mechanism may not be enough. It is necessary to implement the action with minimum downtime. The software restart is always a valid technique, but each application has specific functions and characteristics, which may require a rejuvenation action that takes into account these characteristics. Other actions might bring the system to a previous (and better) state

with less service unavailability than the simple restart.

The proposal of a rejuvenation action to the Eucalyptus environment requires a good knowledge of the platform, and specially of the aging of their resources. Such an understading about the platform under analysis is essential to develop a suitable rejuvenation strategy. The knowledge acquired in the previous experiments enabled the identification of some characteristics of Eucalyptus to propose the most appropriate rejuvenation action.

Many resources were monitored and several indications about aging effects were found. Although, the memory utilization of the node controller was the resource that deserved more attention, since the NC process has crashed at certain times, in the 32-bit machines, while the 64-bit machine remained in full operation. At the end of the preliminary experiments, analyzing data from each machine, it was noted the high growth of the virtual memory usage by *eucalyptus-nc (Apache2)* process, reaching the maximum allowed by the 32-bit architecture in Linux (3 GB), which led to the inability of VMs instantiation on those machines. In the 64-bit machine, even with no problems in the instantiation of new VMs, the virtual memory usage by the same process continued to grow. The non-occurrence of the same problem that occurred in the 32-bit machines is explained by the fact that the 64-bit architecture enables up to 250 TB of virtual memory usage per process. However, this does not mean that this exaggerated growth may not reach critical levels. Specifically in this case the depletion of disk space is imminent, given this is also a limited resource.

By means of a preliminar analysis of the virtual memory usage of the *eucalyptus-nc* process, it was noted that after the restart command there was a decrease on its use. Although it returned to grow until the limit of 3 GB, requiring another restart of the process. This behavior remained until the end of the experiment. As the restart process times coincided with the times when the virtual memory usage fell, it was noticed that a restart was a possible rejuvenation action to be adopted. It is noteworthy that this command does not restart the machine, nor the operating system, but only one Eucalyptus process, which returns to normal operation within a few seconds, which is an important factor to choose such an action.

Figure 5.1 shows the steps to be followed during the software rejuvenation experiments. The first step is to define a resource limit to activate the rejuvenation action. The second step is to implement a script that going execute the rejuvenation action when the resource utilization reach the limit defined previously. The third step is to execute a rejuvenation test. This preliminar experiment is performed in a short period time, the objective is verify if the resource utilization decreased. If any reduction is observed in the resource consumption, then passes it to the next step. Otherwise, the cycle ends. However, when any reduction is observed, the rejuvenation experiment is performed, this time for an extended

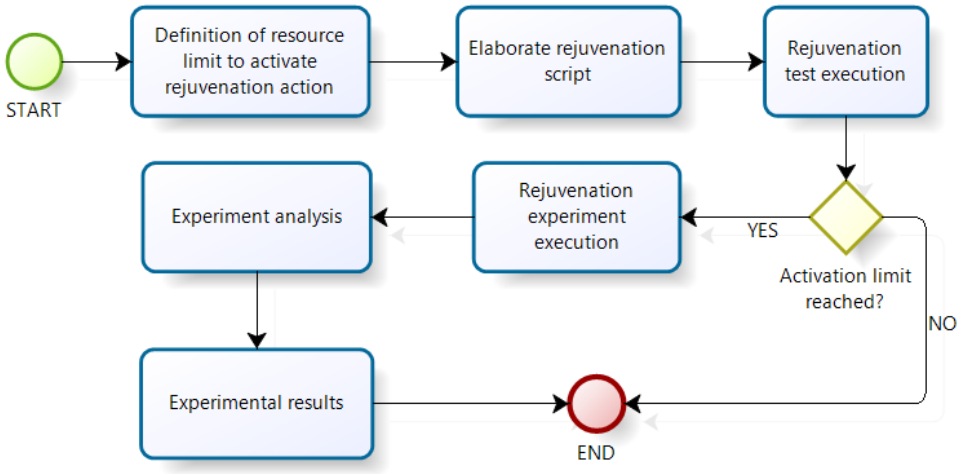

Figure 5.1 Approach overview software rejuvenation

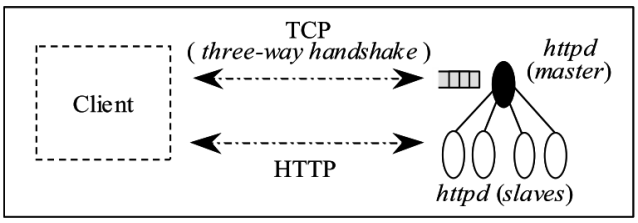

Figure 5.2 Apache process model (Matias and Filho, 2006)

time. The next step is to analyze the experimental data obtained by monitoring. All data are analyzed to verify the rejuvenation executions number and if the scripts was correctly executed. The end step gathers all results obtained with the experiment, ending the cycle.

The problem now was to minimize the system downtime system caused by this rejuvenation action or find another solution that does not cause the system down. Considering that the *eucalyptus-nc* process is an Apache process, it was decided test a rejuvenation technique to web server proposed by Matias and Filho (2006), whose rejuvenation downtime during the action is absent.

5.1 Rejuvenation strategy based on virtual memory utilization

The *eucalyptus-nc* process is primarily responsible for the high utilization of virtual memory in the node controllers. As this is an Apache process, the rejuvenation solution should be proposed taking into consideration the Apache process characteristics. Figure 5.2 shows the Apache process model.

The Apache implements the handling of software signals for general purposes. One of these Apache-manipulated signals is SIGUSR1. This signal, when sent to the master *httpd*, indicates that it must reinitialize each of its slaves, however this action only takes place when the slave finishes the request processing that is currently in progress (Laurie and Laurie, 2002). Apache uses SIGUSR1 to request a "graceful" process restart/shutdown.

Based on the software aging symptoms observed in the Eucalyptus environment, an automated method is proposed for triggering a rejuvenation mechanism in private cloud environments that are backed up by the Eucalyptus cloud computing framework. This method is based on the rejuvenation mechanism presented in (Matias and Filho, 2006), that sends a signal (SIGUSR1) to the apache master process, so that all slave idle processes are terminated and new ones are created. This action cleans up the accumulated memory and has a small impact on the service, since the master process waits for the established connections to be closed. Creating a new process to replace the old one causes a delay of around 5 seconds, due to the load of eucalyptus configurations during process startup, as observed in previous experiments.

With an effective rejuvenation strategy, another important step is to define the exact moment when the action will be executed. As the most critical resource observed is the virtual memory, so the rejuvenation mechanism should be activated by the size of virtual memory. This proposal is simple. When the virtual memory usually reaches a threshold considered ideal, before reaching the failure threshold (3 GB for example), the rejuvenation action is performed. The limit chosed should be not too close to the critical limit.

For this purpose, it was needed a constant monitoring of the resource. To not interfering in the system performance with constant requests, it was decided to collect information every 1 minute. To automate the sending of the SIGUSR1 command, a script was implemented to check every minute if the virtual memory usage has reached to the 2 GB (empirically chosen limit) . If so, an immediate rejuvenation action is performed. The rejuvenation mechanism can be activated every time is necessary.

A good point of this strategy is that the virtual memory utilization never will reach the critical limit of the 32-bit architeture (3 GB). However, one downside of this strategy is that there is no way to know which day and time the rejuvenation will occur.

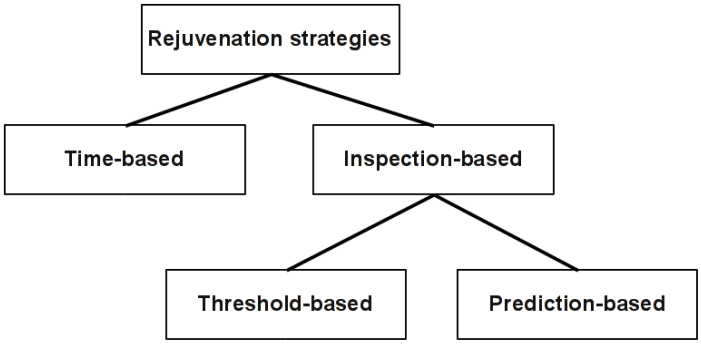

Figure 5.3 Classification of rejuvenation strategies

5.2 Threshold and predictions based rejuvenation strategies

In a production environment, the interval between process restarts should be as large as possible, in order to reduce the effects of summing up small downtimes during a large runtime period. One approach to achieve that maximum interval is based on a high-frequency monitoring of process memory usage. At the exact moment when the memory limit is reached, the rejuvenation is triggered. Since a small sampling interval may affect system's performance, so we consider that a 1-minute interval is the minimum time duration to avoid interference in the system.

Despite the ability to provide good results, a problem can be identified in such approach. It is possible that the node controller process reaches its memory limit between two monitoring points in time. An additional downtime is introduced in this way, described now as "monitor-caused downtime".

The proposed triggering mechanism aims to remove this additional downtime, as it tries to keep the interval between process restarts as large as possible. The prediction about when the critical memory utilization (CMU) will be reached is used for that purpose. Therefore, considering the classification of rejuvenation strategies presented in Figure 5.3, our approach has characteristics of two categories, since it is a threshold based rejuvenation but it is aided by predictions.

The rejuvenation activation is the prediction of when the critical or ideal threshold is reached. That way you can know which day and time the rejuvenation action is activated. Analyzing again the virtual memory uses graphics, we find that growth follows an almost linear pattern. Therefore, it is fully possible to predict the approximate time that the virtual memory usage will reach the limits of interest. For this purpose, it is necessary to use Time

Figure 5.4 Schematic representation of a Time Series

Series.

A time series is a time-oriented or chronological sequence of observations on a variable of interest (Montgomery *et al.*, 2008). Given an observed time series, one may want to predict teh future values of the series (Chatfield, 1996).

A Time Series uses information previously existent (database) and from these data it is possible estimate a future value. In this case, the data resources collected during the experiment will be used to find the rejuvenation moment that the mechanism will be activated. Using this strategy, the resource utilization can be used in full operation, activating the machanism only when it is needed. Figure 5.4 shows a schematic representation of a Time Series

This strategy is a bit more complex than before. Initially, we have to figure out which time series is best suited to perform the prediction (LTM, QTM, ACG, or SCTM). For this purpose, the data obtained a few hours monitored were used to calculate with time series mentioned above. The series to be chosen is the one that showed the best rates MAPE, MAD and MSD. After finding the time series with the best results, this was be used to predict the rejuvenation moment activation.

Time-series fitting enables us to perform a trend analysis and therefore state, with an acceptable error, the time remaining until the process reaches the CMU. This information makes it possible to schedule the rejuvenation to a given time (T_{rej}), which takes into account a safe limit (T_{safe}) to complete the rejuvenation process before the CMU is reached. The safe limit should encompass the time spent during the rejuvenation and the time relative to the time-series prediction error. Therefore, we can state $T_{rej} = T_{CMU} - T_{safe} = T_{CMU} - (T_{restart} + T_{PredError})$.

As seen in Figure 5.5, it is important to highlight that the trend analysis is started only after the monitoring script detects the node controller process has grown over a time series computation starting point, TSC_{SP}, that in our case is 80% of the critical memory utilization. This starting point was adopted to avoid unnecessary interference on the system due to the computation of time series fitting.

When a limit of 95% of CMU is reached, the last prediction generated by the trend analysis is used to schedule the rejuvenation action, i.e., the last computed T_{CMU} will be used

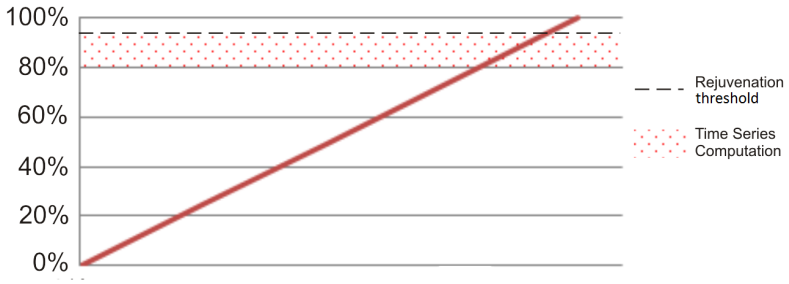

Figure 5.5 Chart projection

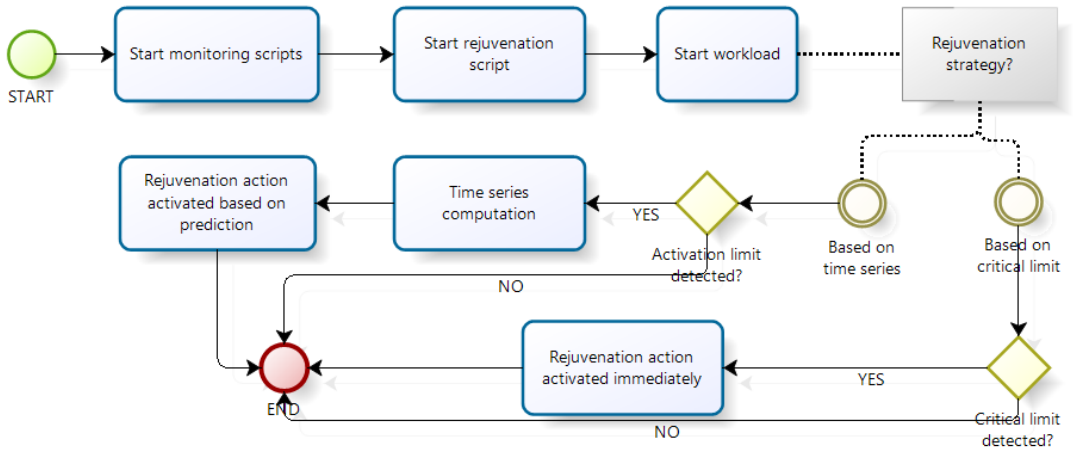

Figure 5.6 Approaches of rejuvenation strategies

to assess the T_{rej} and the system will be prepared so that the rejuvenation occurs gracefully in time T_{rej}. We see that by reaching this time series computation final point, TSC_{FP}, there is no benefit in continuing computing new estimates, and it would be a risk to postpone the scheduling of rejuvenation mechanism. It is important to stress that the values adopted for TSC_{FP} and TSC_{SP} are specific to our environemnt, and therefore may vary if this strategy is instantiated for other kinds of systems.

Figure 5.6 shows two approaches of rejuvenation strategies: Threshold-based (critical limit) and Prediction-based (time series). In both cases it is necessary to start the monitoring scripts to collect the data of the resources utilization. After, the rejuvenation script can be executed. The next step is start the workload to stress the environment. In this moment, the rejuvenation strategy should be chose. If based on critical limit, when the critical limit is reached, the rejuvenation action is activated immediatly. If based on time series, when activation limit is detected, there is a prediction using a time series computation. After, the rejuvenation mechanism is executed based on this prediction.

5.3 Considerations

The rejuvenation strategy used to reduce the growth of virtual memory can also be used in other resources such as resident memory. In this study, it was decided to adopt its use only in the virtual memory because this feature has been the most critical due to its fast growth, reaching the maximum allowed in 32-bit architecture. It is evident that the resident memory is also scarce and that its growth may also reach critical levels, especially in the case of machines with modest settings.

No rejuvenation experiment using the amount of *zombie* processes was performed. However, if it was necessary, it could bring an action in sending consistent rejuvenation manual (or via script) signal SIGCHLD, using the kill command. If the parent process still refuses to reap the *zombie*, the next step would be to remove the parent process. When a process loses its parent, *init*[1] becomes its new parent. The reason we did not find one even higher number of *zombie* processes is because *init* periodically executes the wait system call to reap any zombies with *init* as parent.

[1] *Init* is the parent of all processes on the system, it is executed by the kernel and is responsible for starting all other processes; it is the parent of all processes whose natural parents have died and it is responsible for reaping those when they die. Processes managed by init are known as jobs and are defined by files in the */etc/init* directory.

6

Case studies

Você não sabe o quanto eu caminhei pra chegar até aqui...
You do not know how I walked to get here...
—CIDADE NEGRA (A Estrada)

This chapter presents some case studies and the respective results. The requirements, characteristics, methods and strategies used in these experiments were explained in the previous two chapters.

We analyzed the utilization of hardware and software resources in a scenario where the adopted workload performs operations related to instantiation of virtual machines.

We have used the testbed environment described in the previous chapter to perform experiments aiming at finding out aging symptoms characterized by anomalous usage of the following resources: CPU, memory space, hard disk space, and process IDs. These experiments provided the basis for the rejuvenation method presented and other experiments were used to test the proposed approach.

6.1 Case study one

The main goal of the case sutdy one was to verify the existence of software aging symptoms in the Eucalyptus cloud computing infrastructure. Some indicators of software aging had already been observed, but there was a need for confirmation and characterization of this phenomenon. Therefore, the experiment was performed during a 30 days period. The duration of the experiment was based on empirical observation of the time elapsed until the appeareance of some possible aging symptoms, considering the workload that was adopted to stress the system.

The virtual and resident memory usage, for the Eucalyptus node controller process, are the most representative results found in this experimental study. Other important results found were the number of zombie processes, CPU utilization and swap memory in the cloud controller host. Disk usage metrics did not show any perceptible aging behavior, so they have not been included in this study.

6.1.1 General resources

Figure 6.1 shows the CPU utilization results of the cloud controller machine. In almost all experiment the CPU usage did not exceed 5%. However, some major growth spurts following a nearly linear pattern can be observed throughout the experiment. We realize that such peaks of resource usage will increase over time, which may be a sign of performance degradation.

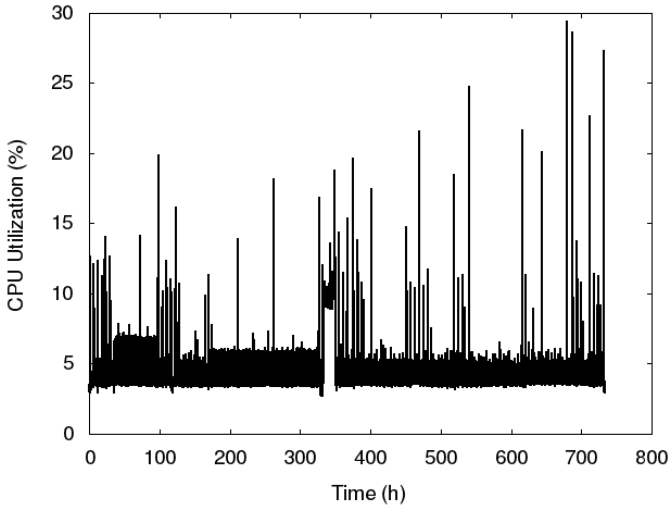

Figure 6.1 CPU utilization in the cloud controller machine

There was also a considerable growth in swap memory use on the cloud controller machine. In Figure 6.2 we can see that this growth has come close to 14 GB. The growth is constant, without drops, since the host continued responding to the request of VM instantiation throughout all the experiment. However, even without stopping the service, this behavior deserves attention because the swap space is a limited resource and in a longer period the growth of its usage may lead to resource depletion, then to the system crash.

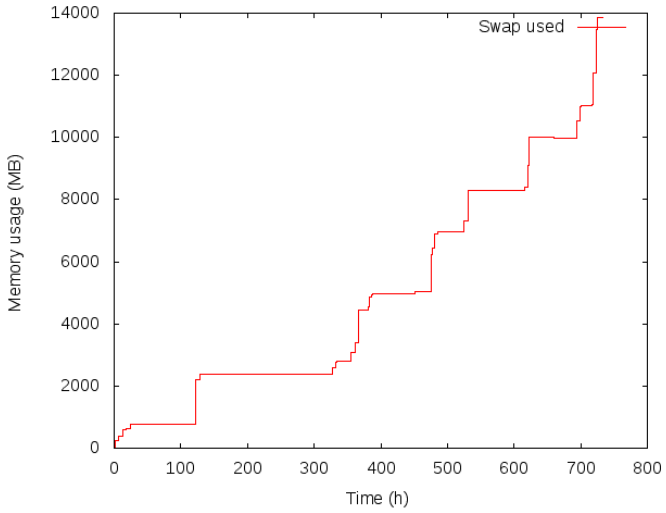

Figure 6.2 Swap memory used in the CLC machine

6.1.2 Specific resources

Figure 6.3 shows the virtual memory utilization of the node controller process at *host3*. It is worth emphasizing the significant memory utilization growth that was observed during the reboots, shutdowns and instantiations of VMs. Such behavior continues until the process reaches about 3055 MB of virtual memory, time at which the process stops growing. At that point, the node controller can no longer instantiate VMs, probably due to the virtual memory exhaustion related to the limitations of the 32-bits operating system. A manual restart of the process responsible for the Eucalyptus node controller makes the memory utilization to drop to about 110 MB.

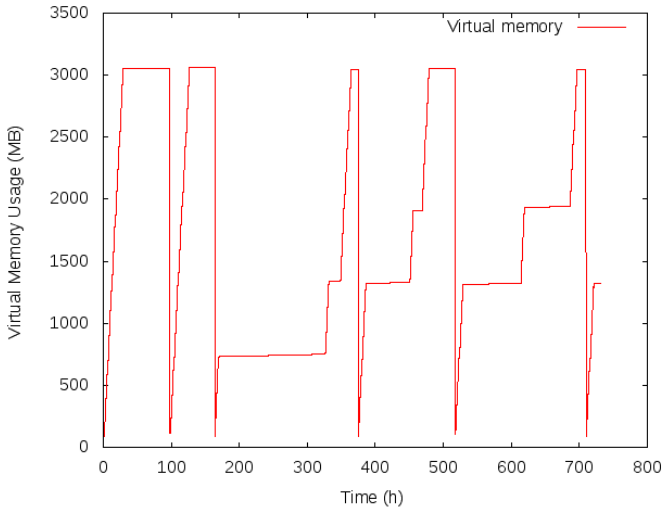

Figure 6.3 Virtual memory usage of the NC process at *Host3*

After restarting the process, the same behavior pattern is observed. The virtual memory grows until reaching about 3064 MB and again the node controller is not able to handle virtual machines instantiation requests. Nodes 5 and 6 (the other nodes with 32-bit OS) showed the same behavior as node 3.

In the 64-bit enabled host, a monotonic growth of virtual memory utilization was also observed, as shown in Figure 6.4. In this case, no drops were observed because the node controller process never failed, so the process was not restarted. The 64-bit operating system architecture allows up to 256 TB of virtual memory addressing, however in a long period, the virtual memory usage would exhaust the hard disk space available for this purpose.

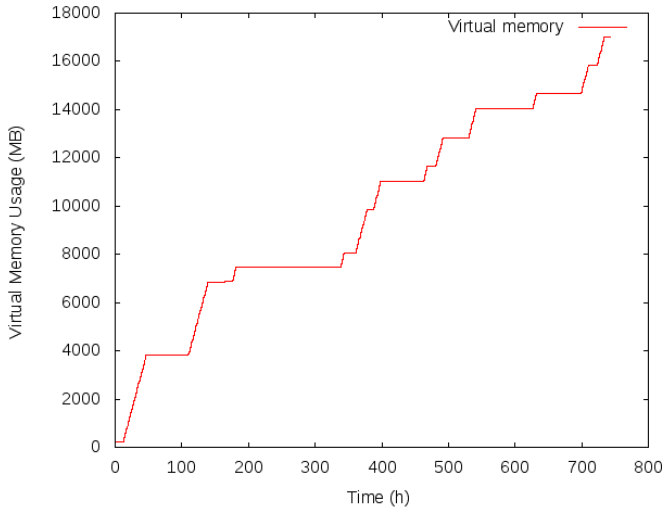

Figure 6.4 Virtual memory usage of the NC process in a 64-bits machine

Another monitored resource was the use of resident memory of the node controllers. Figure 6.5 shows an evident growth of resident memory. The drops are related to the moments when the node controller process is restarted. However, the observed growth does not take a larger proportion because the virtual memory reaches its limit, about 3GB, as previously mentioned.

The resident memory usage in the 64-bit OS, depicted in Figure 6.6, has an uninterrupted growth. The almost linear behavior in Figure 6.6 motivated the execution of a regression study for the resident memory usage of NC process in that 64-bit machine. We have found the following relation: $rm = 44,157 + 2.850 \cdot t$, where rm is the amount of resident memory (in kilobytes) used by the node controller process, and t is the time of execution for the process. The obtained equation enables one to estimate the amount of memory used by the process at a given time.

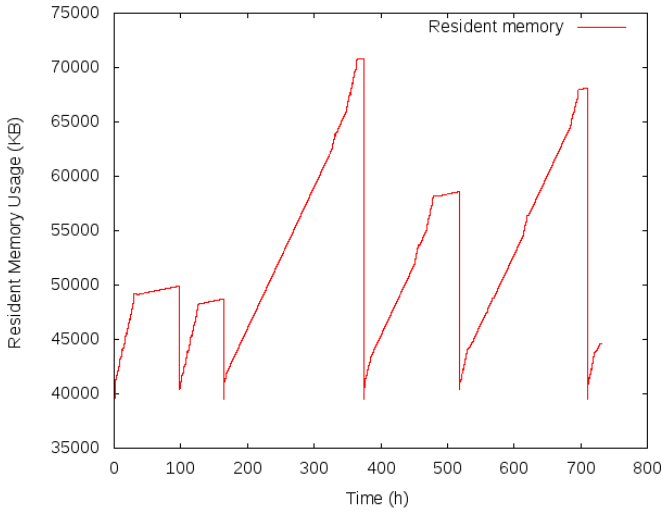

Figure 6.5 Resident memory usage of the NC process at *Host3*

Table 6.1 Regression-based estimates for resident memory usage in NC process

Time (months)	Estimative (MB)
2	283.591
4	524.060
6	764.528
8	1004.997
10	1245.466
12	1485.935

Table 6.1 shows the estimates for distinct time periods, expressed in months. According to the regression, after 8 months of uninterrupted execution, this single process would reach 1 GB of resident memory, approximately 1/4 of the available resource. This prediction is an example of the worry rised by this increasing memory utilization.

Figure 6.7 shows the usage of resident memory by the cloud controller process. We can see some peaks of growth, probably related to the fact that some node controller stopped working at a given instant, and the cloud controller was trying to instantiate new VMs, but it was not successful in this task. The amount of virtual memory used by the cloud controller process did not present a significant increase, so it is not shown here.

The number of zombie processes is another measure that highlights an aging effect in the machine that hosts the cloud controller. Figure 6.8 shows that the number of zombie processes in cloud controller machine grows in some periods. The major part of processes that fall to the zombie state are apache processes which are executed by the Eucalyptus services. The native Linux cleaning mechanism is executed periodically and removes zombies from the process table, so the amount of zombie processes detected does not cause

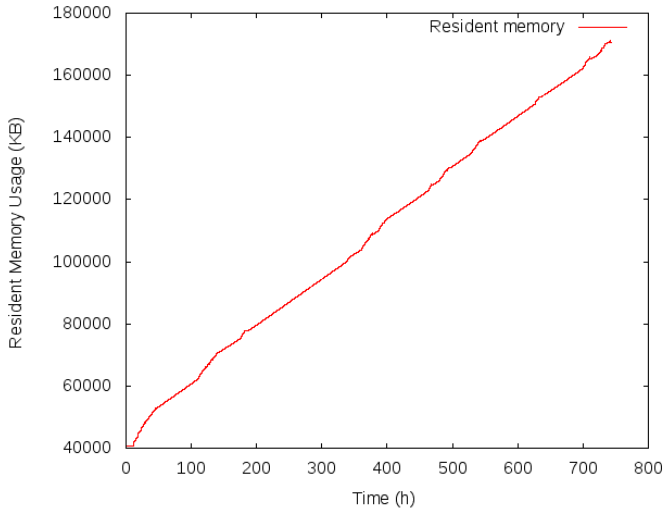

Figure 6.6 Resident memory usage of the NC process in a 64-bits machine

problems to system activities. Nevertheless, the number of zombie process may become a critical issue when the system is operating under a heavier workload.

6.2 Case study two

In order to confirm the previously described software aging effects, and identify new symptoms, another experiment was conducted. This case study uses a similar workload, but memory fragmentation is an additional focus in the measurements, besides the memory leak problems. Kernel instrumentation (Matias *et al.*, 2010), by means of the SystemTap infrastructure, was used to collect the occurence of fragmentation events, whereas the memory usage was collected directly from the /proc pseudo-filesystem.

6.2.1 Memory leaking monitoring

A Python script has been implemented for detecting memory leaking of Eucalyptus-related processes, collecting the data in the file /proc/*pid*/stat, where *pid* is the Linux id of the process being monitored. Figure 6.9 depicts the resident memory size used by the *eucalyptus-cloud* process in the machine *Host2* - that holds the *Cloud Controller*. The reader may observe the memory usage increase when the workload begins - two hours after the start of the experiment, but the process seems to reach a steady behavior as the time goes by. Therefore, we can not point out an aging phenomenon related to this specific process.

The subsequent experiment conducted was monitoring the *apache2* process, also

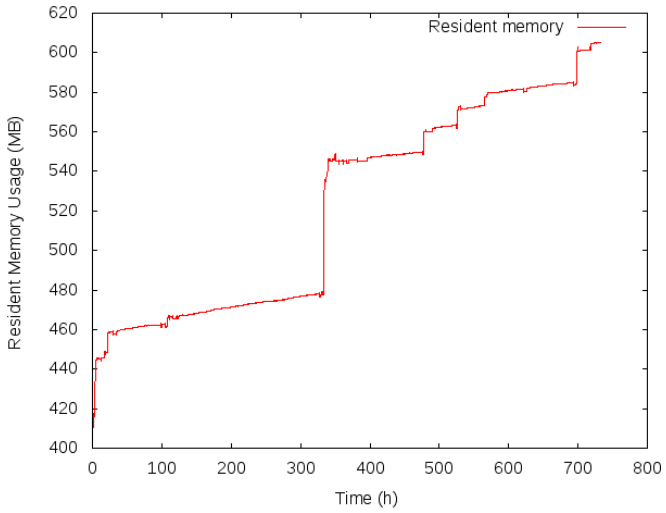

Figure 6.7 Resident memory usage of the cloud controller process

in machine *Host2*. Figure 6.10 shows sharp increases in memory usage, mainly due to problems in NCs, which can be understood in details by analyzing the memory leak results for the machines *Host3, Host4, Host5*, and *Host6*.

Since *Host3, Host5* and *Host6* are similar 32-bit OS, the results presented are those related to *Host3*.

Figure 6.11 shows the memory related to the node controller process at *Host3*, during the reboots, shutdowns and instantiations of VMs. The increase in memory usage is a typical symptom of software aging, since it seems that the process does not free memory space, even when the virtual machines running in that node are terminated.

During the experiment, the 32-bit node controllers (machines *Host3, Host5* and *Host6*) stopped responding to requests of VM instantiation. This phenomenon occured as soon as the virtual memory utilization of the corresponding process (*apache2*) reached about 3064 MB (limit of this architecture). In order to continue the experiment, the process responsible for the node controller was restarted every time this problem happened.

The drop in memory usage, seen in the middle of Figure 6.11, is due to the restart process. There are also time intervals when memory usage is almost constant. They correspond to the moments when that process reached the maximum virtual memory allowed in 32-bit operating systems, or when other node controllers reached their own limits, what temporarily paused the workload.

The machine *Host4* did not failed running VM instances in any time. This fact gives more strength to the hypothesis that the service problems in the other NCs were caused

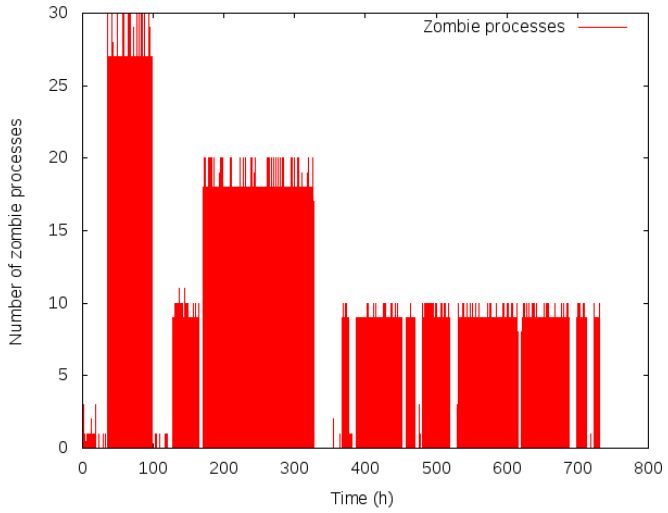

Figure 6.8 Number of zombie process in the cloud controller machine

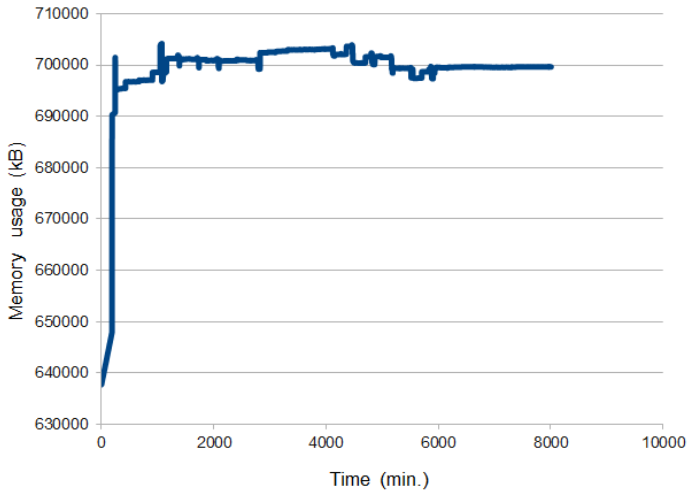

Figure 6.9 Memory usage in the Eucalyptus-cloud process at *Host2*

by the limitations of a 32-bit operating system. In the machine with the OS using 64-bit architecture (*Host4*), there was a monotonic growth of memory usage for the *eucalyptus-nc* process, as shown in Figure 6.12. While this behavior did not lead to failures during the observed time period, problems similar to those seen in 32-bit machines may happen in a longer time interval.

A regression study was conducted for the resident memory usage of NC process in the 64-bit machine. It produced the following equation: $rm = 41,695 + 4.084 \cdot t$, where rm is the amount of resident memory (in kilobytes) used by the node controller process, and t is the time of execution of that process. That equation enables one to estimate the amount of memory used by that process in a specific time instance, and the other way around.

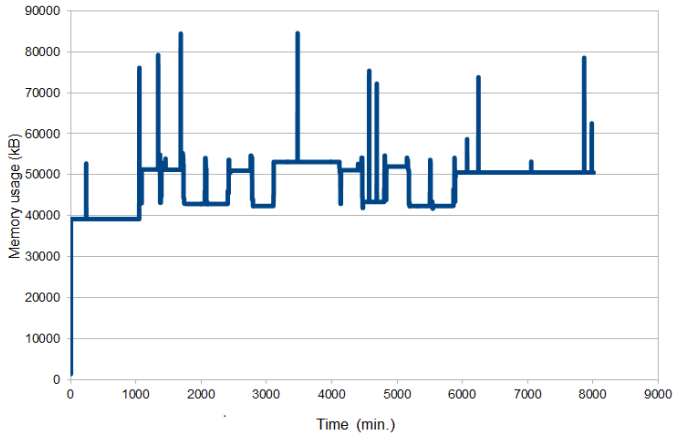

Figure 6.10 Memory usage in the Apache (eucalyptus-cc) process at *Host2*

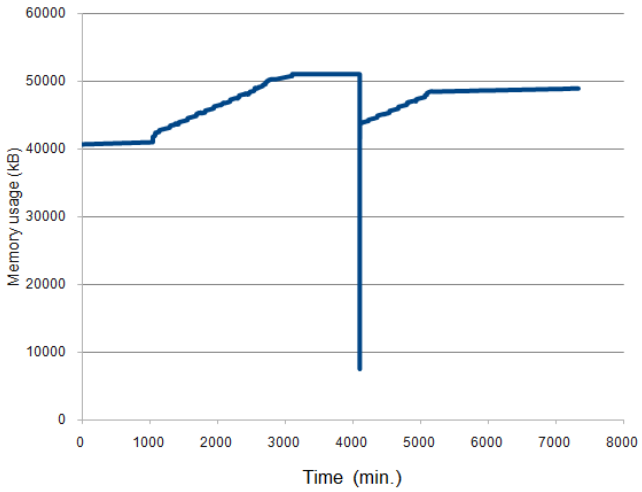

Figure 6.11 Memory usage of the Apache (eucalyptus-nc) process in a 32-bits OS(*Host3*)

Table Table 6.2 shows estimates for distinct time periods, expressed in months. Those estimates indicate that the memory usage deserves attention. The usage of resident memory of is supposed to reach about 1 GB after 6 months of uninterrupted execution and 2 GB after one year. Those values are high because they are related to just one process running in that machine. The node controller process is using too much resources, that should be available for the virtual machines executing on top of that host.

To validate this model, a comparison was made between experimental measurements and the estimates obtained from the regression equation. Table 6.3 presents some points of the comparison and its analysis suggests that the regression-based estimates are accurate.

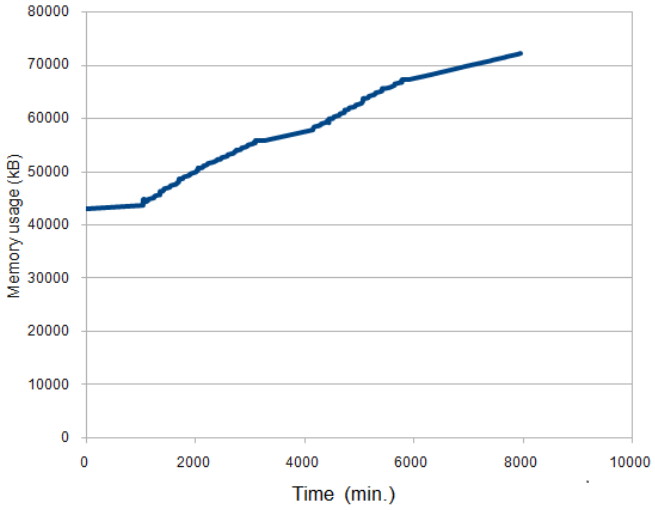

Figure 6.12 Memory usage of the *eucalyptus-nc process* in a 64-bits OS(*Host4*)

Table 6.2 Regression-based estimates of resident memory usage by the NC process - Case study two

Time (months)	Estimative (MB)
2	385,305
4	729,893
6	1074,480
8	1419,068
10	1763,655
12	2108,243

6.2.2 Memory fragmentation monitoring

The fragmentation occurrences produced by each running process in the testbed environment were recorded at 1-minute time interval between measurements.

Figure 6.13 shows the cumulative amount of fragmentation occurrences for two processes related to Eucalyptus running in *Host2* for a 24-hours period. The cloud controller runs in this host. The processes named *Eucalyptus-cloud* and *Apache2* have suffered fragmentation along all time of the experiment.

As it can be seen in Figure 6.14, the measures obtained from *Host3* show an even more intense fragmentation behavior than that for *Host2*. The *Apache2* process that implements the node controller service has presented about three times more fragmentation occurrences than the *Apache2* process corresponding to the cloud controller. *Ksmd* and *Libvirtd* are the other processes that were the most affected by memory fragmentation.

Libvirtd is tightly related to the management of virtual machines, since it is the daemon for the Linux standard virtualization API. *Ksmd* is a daemon in the kernel that

Table 6.3 Comparison between measurements and estimates for resident memory usage in NC process

Time (hours)	Measurements (MB)	Estimates (MB)
24	45.76	46.46
48	53.24	52.20
72	57.59	57.95
96	65.09	63.69
120	68.59	69.43

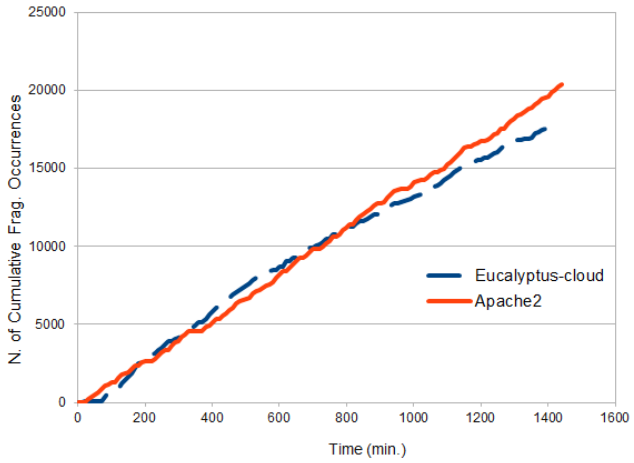

Figure 6.13 Fragmentation per processes in *Host2*

periodically performs page scans to identify duplicate pages and merges duplicates to free pages for other uses (Arcangeli *et al.*, 2009). *Ksmd* can perform the consolidation of identical memory pages from concurrent virtual machines, managed by KVM. *Ksmd* is specially stressed by the workload, because two identical VMs start in each cycle, therefore many memory pages may likely be shared by them.

Figure 6.15 depicts the fragmentation results for the Host4 - the machine that runs a 64-bit operating system. The process *Ksmd* was highly affected by the fragmentation, despite the amount of occurrences was smaller than in *Host3*. *Libvirtd* and *Apache2* processes also had their space memory fragmented, resulting in more than 5000 occurrences after the 24-hours period. This is a lot of occurrences in a very small time.

A regression study was carried out for the fragmentation occurrences in *Apache2* and *Eucalyptus-Cloud* processes that run at the *Cloud Controller (host2)*. The regression equations are $foa = 1359 + 14.3 \cdot t$ and $foec = 2150 + 12.4 \cdot t$ to *Apache2* and *Eucalyptus-Cloud* processes, respectively. Those equations demonstrate the similarity in the occurrence of fragmentation for both processes along the time, since the coefficients corresponding to their slopes (14.3 and 12.4) are close one to each other.

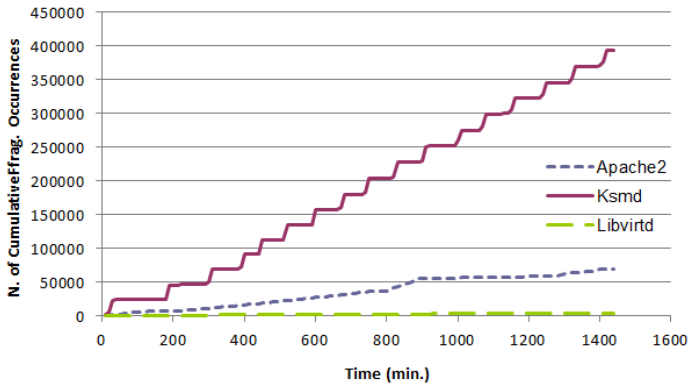

Figure 6.14 Fragmentation for processes in *Host3*

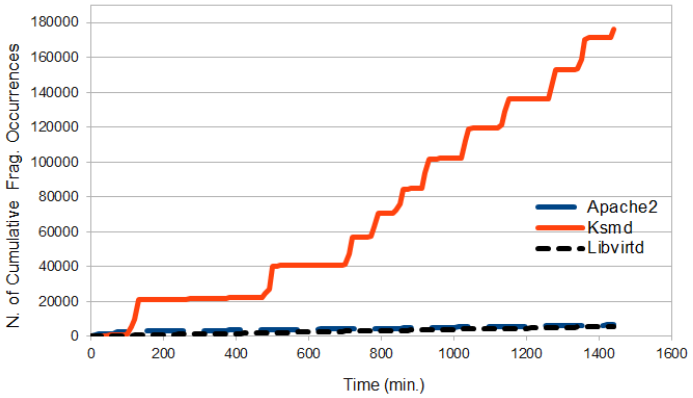

Figure 6.15 Fragmentation for processes in *Host4*

Table 6.4 shows the regression-based estimates for distinct time periods, expressed in months. Those estimates highlight again the need for attention to the increase of fragmentation occurrences. Some action should be taken in order to avoid that only two processes fragment the overall memory space, degrading system performance and making difficult the memory allocation for other processes.

The linear regression analysis was also carried out to find the growth pattern of fragmentation for the process *Ksmd*, at *host3* (32-bit) and *host4* (64-bit) machines (see Table 6.5). *Apache2* process was not included in this analysis because it fragmented in a lesser extent in comparison to *Ksmd*.

Similarly as it was previously performed for the *host2*, the first two hours (period without workload) were not considered in the computation of linear regression for *host3*. However, as the *host4* showed stronger growth in the second half of the experiment, the first half was discarded for the calculation of linear regression. The regression equations obtained for *Ksmd* are $foh3 = 8757 + 290 \cdot t$ and $foh4 = 56660 + 167 \cdot t$, where $foh3$ and

Table 6.4 Regression-based estimates for fragmentation occurrences in the *Host2*

Time (months)	Apache2	Eucalyptus-Cloud
2	1,236,879	1,073,510
4	2,472,399	2,144,870
6	3,707,919	3,216,230
8	4,943,439	4,287,590
10	6,178,959	5,358,950
12	7,414,479	6,430,310

Table 6.5 Regression-based estimates for fragmentation occurrences in Ksmd process

Time (months)	Fragmentation (Host3)	Fragmentation (Host4)
2	25,064,757	14,485,460
4	50,120,757	28,914,260
6	75,176,757	43,343,060
8	100,232,757	57,771,860
10	125,288,757	72,200,660
12	150,344,757	86,629,460

$foh4$ are the amounts of fragmentation ocurrences in the node controllers process (at *host3* and *host4*), and t is the time of execution for the process. The values in Table 6.5 and the slopes in the regression equations reveal that the fragmentation in more intense in the Ksmd process located in the *host3* than it is in the *host4*. The difference between 32-bit and 64-bit architectures may be the reason for this phenomenon, since the allocatable memory space is higher in the latter.

6.3 Case study three

The case study one showed that increasing usage of virtual memory stops the node controller process, that was not able to respond to VM instantiation commands. A manual restart of Eucalyptus node controller service, in 32-bit hosts, made the virtual memory usage falls to less than 110 MB.

The objective of this case study is to implement a rejuvenation strategy e demonstrate that a proactive action can be to avoid systems downtime.

After restarting the service, the same behavior pattern was resumed. As seen in Figure 6.16, the process's virtual memory has grown until about 3064 MB and again the node controller was not able to service the requests of virtual machines instantiation.

A proactive action is needed to prevent the failure of the cloud environment. Since

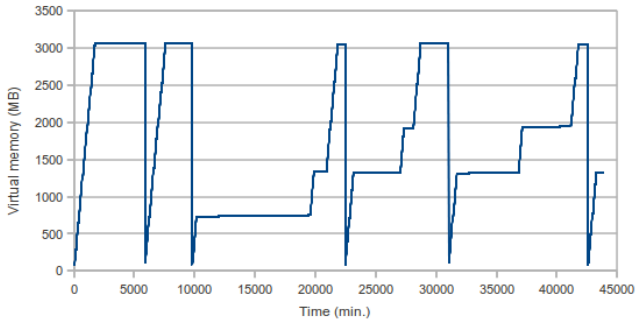

Figure 6.16 Virtual memory used in the NC process at *Host3* (previous experiment)

the node controller activities are performed by an Apache process, the rejuvenation strategy adopted in (Matias and Filho, 2006) was adapted to this environment. The adopted method sends the Linux signal SIGUSR1 to the master *httpd* process, when a memory usage threshold is reached. When this signal is sent, the *httpd* process restarts each of its slave processes. This strategy has less impact than restarting the Eucalyptus node controller service, because it only takes place when the slave finishes the request processing that is currently in progress. In our case, Apache processes are named *apache2*, and we send SIGUSR1 signal to the Apache master (also called parent) process.

In order to test this strategy, a new experiment was performed using the same environment previously described, but using a more intense workload in order to accelerate the virtual memory growth.

We implemented a script that every five seconds checks if the *apache2* process has reached 2 GB, which is our chosen rejuvenation threshold. If so, SIGUSR1 signal is sent to the *apache2* master, reducing the use of virtual memory without effects to service availability. Figure 6.17 shows the effect of rejuvenation mechanism, which limits the growth of memory usage by the node controller's *apache2* process. This time needs to be short to the resource consumption monitored not reach high levels.

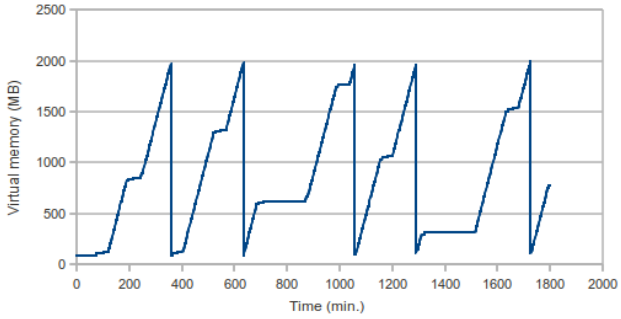

Figure 6.17 Virtual memory used in the NC process at *Host3*, during rejuvenation experiment

Table 6.6 Statistical summary of instantiation times

	Without rejuvenation	With rejuvenation
Mean	462 min.	73 min.
Minimum	24 min.	23 min.
Maximum	3847 min.	194 min.
Std. Dev.	1,269	64

Figure 6.18 shows the evolution of the time to instantiate all VMs in a sequence of 9 workload cycles, during the first experiment, without a rejuvenation process, as well as in the second experiment, with the execution of the rejuvenation method.

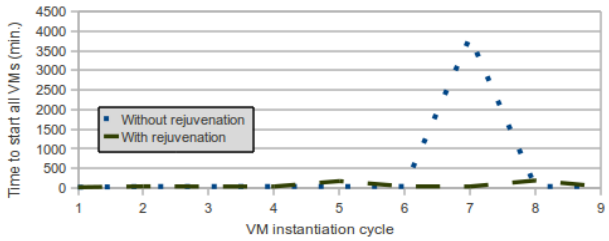

Figure 6.18 Time to instantiate 8 VMs in each workload cycle

The largest instantiation time occurs when Eucalyptus fails starting VMs, when the virtual memory usage reaches the maximum allowed by the 32-bit architecture. Such failure causes repeated instantiation attempts by the workload generator script, demanding human interaction to restart the Eucalyptus service. As seen in the same figure, the measured instantiation time in the rejuvenation experiment does not present big increases.

Table 6.6 presents the mean, minimum, maximum, and standard deviation values of instantiation times, considering both experiments, with and without rejuvenation. This statistical summary shows the benefits of the adopted rejuvenation mechanism. We confirm that the maximum instantiation time is more than 20 times smaller when the rejuvenation is performed in comparison to the other experiment. The minimum and mean times are also smaller when the proposed approach is used. In especial, we consider that the reduced standard deviation highlights that proactive action turns the system more stable and prevents problems caused by large times waiting for virtual machines instantiation.

6.4 Case study four

The knowledge acquired in the previous case studies enabled the proposal of an enhancement to the rejuvenation method presented in Section 6.3. The previous method only triggers

Table 6.7 Summary of the accuracy indices for each model (NC virtual memory)

Model	$\widehat{Y}_t$	MAPE	MAD	MSD
LTM	$44157.1 + 2.85t$	1%	900	1472447
QTM	$43354.8 + 2.95830t - 0.000002t^2$	1%	872	1343698
GCM	$53013.6(1.00003^t)$	6%	6014	52619294
SCTM	$(10^6)/(4.70218 + 16.8036(0.999942^t))$	2%	1259	3449812

the rejuvenation action after a critical level of resource utilization is reached (e.g. 3 GB of virtual memory for a given process). Due to this characteristic, the rejuvenation may take place after the system has already denied service to a client request. Therefore, there is room to improvements aiming to reduce the downtime. The approach proposed in this case study is based on the computation of time series to forecast the moment when the critical level of resource utilization will be reached.

The objective of this case study is use Time Series to estimate the critical utilization of resources and when the rejuvenation action needs to be activated. The rejuvenation of the Node Controller process is the focus in the experiments, since it has shown the major aging effects among all monitored components.

It is noteworthy that the use of time series to reduce the system downtime during the execution of a rejuvenation action may also be applied to other computing environments. The time series itself is not involved directly in the system, but indicates the appropriate time at which an action should be taken.

The data collected in preliminary experiments were used to find out which kind of time series has the better fitting for the growth of virtual memory usage in Node Controller processes.

The trend analysis included four models: LTM (Linear), QTM (Quadratic), GCM (Exponential growth), and SCTM (S-Curve). A summary of the results of the series and their error rates are shown in Table 6.7, where $\widehat{Y}_t$ is the predicted value of the memory consumption at time t.

It can be seen in Table 6.7 that the values of the indices MAPE, MAD and MSD are smaller for the LTM and QTM models. So the choice must be made in relation to these two models. It is also observed that despite the MAPE values of the indices are the same for these two models, the index values of the QTM model are smaller than for the LTM model. So the QTM model was chosen as the best fit for the trend analysis of virtual memory utilization in Eucalyptus node controllers, therefore it was adopted in the rejuvenation scheduling.

Table 6.8 Comparison of Experiments

	Experiment #1	Experiment #2
Availability	0.999584	0.999922
Number of nines	3.38	4.11
Downtime	108 seconds	20 seconds

The actual experimental study for verifying the rejuvenation method was performed in two parts. In the first, the cloud environment was stressed with the workload described in Section 4.6, and the rejuvenation mechanism was triggered only when the critical limit was reached. In the second experiment, the same workload was used, but the rejuvenation was scheduled based on the time-series predictions.

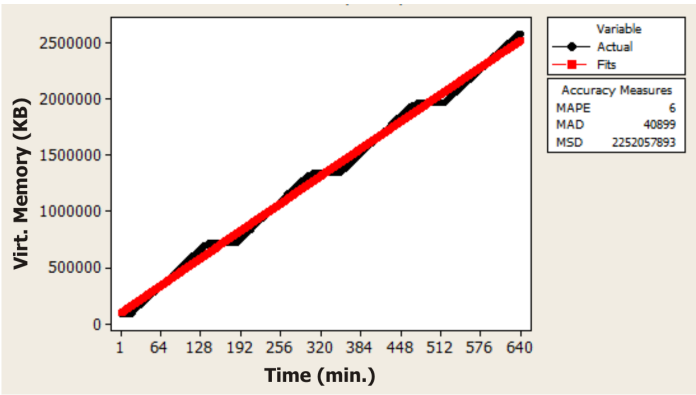

Figure 6.19 Quadratic Trend Analysis of Virtual Memory

Figure 6.19 shows the trend analysis for the growth of virtual memory utilization, fit by a quadratic function $\widehat{Y}_t = 94429 + 3825.3t - 0.0686t^2$. This analysis results in 809 minutes for the T_{CMU}, i.e., the predicted time to reach the 3 GB limit. Considering a prediction error of 5 minutes and the time spent during the rejuvenation action as 5 seconds, the safety margin (T_{safe}) was subtracted from T_{CMU}, so the rejuvenation was scheduled to $T_{rej} = 803,9$ minutes, counting up from the beginning of the experiment. After rejuvenation, the memory usage is reduced, and other trend analysis is carried out when the 80% limit is reached again.

The results show that the proposed rejuvenation triggering method implies in a higher system availability, when compared to the method that did not consider time trend analysis. The number of nines increases from 3.38 to 4.11 (see Table 6.8). In a time-lapse of one year, such difference means a decrease from 218 minutes to 40 minutes of downtime, i.e. the downtime was reduced about 80%.

Table 6.9 Comparison of virtual memory predictions and actual values

Time (min)	Predicted (KB)	Actual (KB)	Error (%)
120	608181	695624	12.57%
240	1069185	1064728	0.42%
360	1530189	1433776	6.72%
480	1991193	1977272	0.70%
600	2452197	2601880	5.75%

Table 6.9 presents the absolute percent error between the predictions and the actual values for the virtual memory utilization in this experiment. That error varies in this range from two to ten hours, but it is below 10% in most analyzed points and it may be enhanced by using the last prediction errors to adjust the related threshold.

The best approximations are obtained at four and eight hours of experiment, that are the points were the "fit" line intercepts the "actual" line in Figure 6.19. The largest error was 12.57% at 120 minutes. Such a maximum error in the predictions enable the use of this approach in other environments which have similar aging characteristics.

7

Conclusions

O importante é não parar de questionar. A curiosidade tem sua própria razão de existir.
The important thing is not to stop questioning. Curiosity has its own reason for existing.

—ALBERT EINSTEIN (Phrase)

The provisioning of services in a cloud computing environment requires high availability of both hardware and software components. Software aging may have a huge impact on cloud systems, due to their variety of components and interactions.

In this dissertation, we have investigated the evolution of resources utilization in a Eucalyptus-based cloud infrastructure over operation time. Data collected during experiments indicated the presence of software aging, mainly related to virtual and resident memory usage.

Some software aging effects were detected in Eucalyptus-based cloud computing infrastructure. Indicators of memory leak and memory fragmentation in Eucalyptus processes and other processes directly related to virtual machine management were found. Such problems may be harmful to system dependability, as well as they probably cause performance degradation, e.g. impacting the speed of virtual machines startup.

Crashes were observed when Eucalyptus processes reach about 3.0 GB of virtual memory use. The use of virtual memory falls when the service *eucalyptus-nc* is restarted. Therefore, the periodical reboot of the *eucalyptus-nc* process before reaching critical levels, in this case 3.0 GB, may be considered as a kind of incidental software rejuvenation. Systems using a 64-bit architecture will have a larger limit for virtual memory usage, so they had no

crashed during the observed period, but the aging process occurs is similar way to that seen in 32-bit machines.

Considering the studies conducted, a mechanism was adopted for avoinding aging phenomenom through mechanisms that reduce memory fragmentation and leaking.

An approach based on time series trend analysis to reduce downtime during the execution of rejuvenation actions was proposed too. The rejuvenation scheduling adopted here is thus primarily threshold based but aided by predictions. A private cloud environment built with Eucalyptus framework running a specific workload, which reached critical limits of virtual memory usage in the node controllers was monitored. A rejuvenation strategy is used to avoid system unavailability, by scheduling the process restart to a proper time before the system stops due to the memory utilization limit.

In order to minimize downtime caused by the action of rejuvenation, a time series was generated from sample data obtained at the beginning of the experiment. With the time series, we estimate in advance the time in which the rejuvenation action should be performed, considering that virtual memory usage follows a certain pattern of growth.

The experimental results show the accuracy of our strategy, as well as a decrease of 80% in the downtime of the adopted Eucalyptus cloud computing environment.

7.1 Statement of the contributions

As a result of the work presented in this dissertation, the following contributions can be highlighted:

- Software aging occurences in the Eucalyptus cloud computing environment were verified. Several features have degraded over the experiments, but some were more critical than others.

- An efficient rejuvenation action that reduces the use of the most critical resource, the virtual memory, avoiding the system stopping was proposed.

- It was proved that the use of time series is effective to predict the failure time and to reduce system downtime.

In addition to the contribution mentioned, some papers presenting the findings of this dissertation were produced:

1. Jean Araujo, Rubens Matos, Paulo Maciel and Rivalino Matias (2011). Software Aging Issues on the Eucalyptus Cloud Computing Infrastructure. In Proceedings of the *IEEE International Conference on Systems, Man, and Cybernetics (IEEE SMC 2011)*. Anchorage, Alaska, USA.

2. Jean Araujo, Rubens Matos, Paulo Maciel, Francisco Vieira, Rivalino Matias and Kishor S. Trivedi (2011). Software Rejuvenation in Eucalyptus Cloud Computing Infrastructure: a Method Based on Time Series Forecasting and Multiple Thresholds. In Proceedings of the *Third International Workshop on Software Aging and Rejuvenation (WoSAR'11)* in conjunction with *22nd annual IEEE International Symposium on Software Reliability Engineering (ISSRE'11)*, Hiroshima, Japan.

3. Jean Araujo, Rubens Matos, Paulo Maciel, Rivalino Matias and Ibrahim Beicker (2011). Experimental Evaluation of Software Aging Effects on the Eucalyptus Cloud Computing Infrastructure. In Proceedings of the *Industrial Track at ACM/IFIP/USENIX 12th International Middleware Conference*, Lisboa, Portugal.

4. Rubens Matos, Jean Araujo, Paulo Maciel, F. Vieira De Souza, Rivalino Matias and Kishor Trivedi (2011). Software Rejuvenation in Eucalyptus Cloud Computing Infrastructure: A Hybrid Method Based on Multiple Thresholds and Time Series Prediction. Journal *International Transactions on Systems Science and Applications - ITSSA*, Vol. 7, No. 3/4, December 2011, pp. 278-294

Other publication:

1. Jean Carlos Teixeira de Araujo and Paulo Romero Martins Maciel (2010). Redes de Petri na Modelagem de Sistemas Computacionais: Análise de Propriedades e Aplicações. A book chapter published in *VIII Escola Regional de Redes de Computadores (ERRC'10)*, Alegrete, Rio Grande do Sul, Brazil.

7.2 Future works

As future works, intends to monitor the impact that different rejuvenation policies have on the availability of similar cloud environments. Other possibility to future work, intends to adress the impact of such mechanisms on dependability measures of Eucalyptus platform.

To investigate the existence of software aging in file systems is another objective. The criteria will be based on open source options and great use in real environment, such as web servers and cloud computing platforms.

As another future work, intends to improve our approach using a larger range of time series models, including wavelets models. The minimization of downtime in different environments, such as data centers and traditional application servers is also planned.

Finally, intends to use the control charts (X, R, S, CUSUM and EWMA) to analyze and detect small variations in the utilization of resources that are almost imperceptible to the "naked eye".

Bibliography

Akaike, H. (1969). Fitting autoregressive models for prediction. *Annals of the Institute of Statistical Mathematics*, **21**(1), 243–247.

Amazon (2011a). Amazon Elastic Block Store (EBS). Amazon.com, Inc. Available in: http://aws.amazon.com/ebs.

Amazon (2011b). Amazon Elastic Compute Cloud (EC2). Amazon.com, Inc. Available in: http://aws.amazon.com/ec2.

Arcangeli, A., Eidus, I., and Wright, C. (2009). Increasing memory density by using ksm. In *Proc. of The Ottawa Linux Symposium (OLS '09)*, pages 19–28, Montreal, Quebec, Canada.

Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., and Zaharia, M. (2009). Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, UC Berkeley Reliable Adaptive Distributed Systems Laboratory.

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

Avritzer, A. and Weyuker, E. J. (1997). Monitoring smoothly degrading systems for increased dependability. *Empirical Softw. Engg.*, **2**(1), 59–77.

Baishnab, K. L., Dev, S., Choudhury, Z. H., and Nag, A. (2010). An efficient heap management technique with minimum fragmentation and auto compaction. In *Proc. of the 5th International Conference on Industrial and Information Systems - ICIIS*, Silchar, India. IEEE Computer Society.

Baker, H. (1995). *Memory management: international workshop, IWMM '95, Kinross, UK, September 27-29, 1995 : proceedings*. Lecture notes in computer science. Springer.

Bao, Y., Sun, X., and Trivedi, K. S. (2005). A workload-based analysis of software aging and rejuvenation. *IEEE Transactions on Reliability*, **54**, 541–548.

Bloomfield, P. (2000). *Fourier Analysis of Time Series: An Introduction*. Wiley Series in Probability and Statistics.

Blum, R. (2008). *Linux Command Line and Shell Scripting Bible*. Wiley Publishing, Inc.

Box, G. and Jenkins, G. (1970). *Time series analysis*. Holden-Day series in time series analysis. Holden-Day, San Francisco, CA.

Candea, G. and Fox, A. (2001). Designing for high availability and measurability. In *Proceedings of the 1st Workshop on Evaluating and Architecting System Dependability (EASY)*, page 42.

Carrozza, G., Cotroneo, D., Natella, R., Pecchia, A., and Russo, S. (2010). Memory leak analysis of mission-critical middleware. *The Journal of Systems and Software*, **83**, 1556–1567.

Castelli, V., Harper, R. E., Heidelberger, P., Hunter, S. W., Trivedi, K. S., Vaidyanathan, K., and Zeggert, W. P. (2001). Proactive management of software aging. In *IBM Journal of Research and Development*, volume 45, pages 311–332.

Chatfield, C. (1996). *The Analysis of Time Series: An introduction*. Chapman & Hall/CRC, New York, USA, 5th edition edition.

Cordeiro, T., Damalio, D., Pereira, N., Endo, P., Palhares, A., Gonçalves, G., Sadok, D., Kelner, J., Melander, B., Souza, V., and Mangs, J.-E. (2010). Open source cloud computing platforms. In *Proc. of 9th International Conference on Grid and Cloud Computing (GCC'2010)*, pages 1–5, Jiangsu, China.

Cully, B., Lefebvre, G., Meyer, D., Feeley, M., Hutchinson, N., and Warfield, A. (2008). Remus: High availability via asynchronous virtual machine replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 161–174, San Francisco, California.

Dandamudi, S. P. (2005). *Guide to Assembly Language Programming in Linux*. Springer.

Domingo, D. and Cohen, W. (2011). Systemtap 1.4 - SystemTap Beginners Guide: Introduction to SystemTap (for Fedora). Technical report, Red Hat, Inc.

Eigler, F. C. (2010). Systemtap tutorial. Technical report, Red Hat, Inc.

Eucalyptus (2009). *Eucalyptus Open-Source Cloud Computing Infrastructure - An Overview*. Eucalyptus Systems, Inc., 130 Castilian Drive, Goleta, CA 93117 USA.

Eucalyptus (2010a). *Cloud Computing and Open Source: IT Climatology is Born*. Eucalyptus Systems, Inc., 130 Castilian Drive, Goleta, CA 93117 USA.

Eucalyptus (2010b). Eucalyptus cloud computing platform - administrator guide. Technical report, Eucalyptus Systems, Inc. Version 1.6.

Eucalyptus (2012). Eucalyptus - the open source cloud platform. Eucalyptus Systems, Inc. Available in: http://open.eucalyptus.com/.

FlexiScale (2011). Flexiscale cloud comp and hosting. FlexiScale.com, Inc. Available in: http://www.flexiscale.com.

Furht, B. and Escalante, A. (2010). *Handbook of Cloud Computing*. Springer Science+Business Media, LLC.

Garg, S., van Moorsel, A., Vaidyanathan, K., and Trivedi, K. S. (1998). A methodology for detection and estimation of software aging. In *Proc. of the The Ninth International Symposium on Software Reliability Engineering (ISSRE'98)*, pages 283–292, Paderborn, Germany.

GoGrid (2011). Gogrid cloud hosting. GoGrid.com, Inc. Available in: http://www.gogrid.com.

Google (2011). Google app engine. Google.com, Inc. Available in: http://code.google.com/appengine.

Grottke, M., Vaidyanathan, K., and Trivedi, K. S. (2006). Analysis of software aging in a web server. In *IEEE Transactions on Reliability*, volume 55, pages 411–420.

Grottke, M., Matias, R., and Trivedi, K. (2008). The fundamentals of software aging. In *Proc 1st Int. Workshop on Software Aging and Rejuvenation (WoSAR), in conjunction with 19th IEEE Int. Symp. on Software Reliability Engineering (ISSRE'08)*, Seattle, WA.

Hastings, R. and Joyce, B. (1992). Purify: Fast detection of memory leaks and access errors. In *In Proc. of the Winter 1992 USENIX Technical Conference*, pages 125–138.

Herber, R. J. (1997). *UNIX System V (Concepts)*. The Collider Detector at Fermilab (CDF). Defunct, zombie and immortal processes.

Huang, Y., Kintala, C., Kolettis, N., and Fulton, N. D. (1995). Software rejuvenation: Analysis, module and applications. In *Proc. of 25th Symp. on Fault Tolerant Computing, FTCS-25*, pages 381–390, Pasadena, CA.

Iosup, A., Ostermann, S., Yigitbasi, N., Prodan, R., Fahringer, T., and Epema, D. (2011). Performance analysis of cloud computing services for many-tasks scientific computing.

IEEE Transactions on Parallel and Distributed Systems (TPDS), Special Issue on Many-Task Computing, **22**, 931–945.

Jacob, B., Larson, P., ao, B. H. L., and da Silva, S. A. M. M. (2009). *SystemTap: Instrumenting the Linux Kernel for Analyzing Performance and Functional Problems*. IBM RedBook, first edition. International Technical Support Organization.

Jones, M. T. (2008). Cloud computing with linux - cloud computing platforms and applications. page 12. IBM Corporation.

Kedem, B. and Fokianos, K. (2002). *Regression Models for Time Series Analysis*. John Wiley & Sons, Inc., Publication.

Kourai, K. and Chiba, S. (2011). Fast software rejuvenation of virtual machine monitors. *IEEE Trans. Dependable Secur. Comput.*, **8**(6), 839–851.

Laird, L. M. and Brennan, M. C. (2006). *Software Measurement and Estimation A Practical Approach*. John Wiley & Sons, Inc.

Lan, Z., Li, Y., Zheng, Z., and Gujrati, P. (2008). Enhancing application robustness through adaptive fault tolerance. In *Proc. of the NSFNGS Workshop in conjunction with 22nd IEEE International Symposium on Parallel and Distributed Processing (IPDPS'08)*, pages 1–5.

Laurie, B. and Laurie, P. (2002). *Apache: The Definitive Guide*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 3rd edition.

Lilja, D. J. (2000). *Measuring Computer Performance: A Practitioner's Guide*. Cambridge University Press, New York, NY.

LTTng (2012). Linux trace toolkit - next generation. LTTng Project. Available in: http://lttng.org.

Marks, E. A. and Lozano, B. (2010). *Executive's Guide to Cloud Computing*. John Wiley & Sons, Inc., Hoboken, New Jersey.

Marshal, E. (1992). Fatal error: How patriot overlooked a scud. page 1347. Science. vol. 255.

Matias, R. and Filho, P. J. F. (2006). An experimental study on software aging and rejuvenation in web servers. In *Proc. of 30th Annual Int. Computer Software and Applications Conference (COMPSAC'06)*, Chicago, IL.

Matias, R., Beicker, I., Leitao, B., and Maciel, P. (2010). Measuring software aging effects through os kernel instrumentation. In *Proc. Second International Workshop on Software Aging and Rejuvenation (WoSAR), in conjunction with 21th IEEE International Symposium on Software Reliability Engineering (ISSRE'10)*, San Jose, CA.

Mckinley, P. K., Samimi, F. A., Shapiro, J. K., and Tang, C. (2006). Service clouds: A distributed infrastructure for composing autonomic communication services. In *Proc. of the 2nd IEEE International Symposium on Dependable, Autonomic and Secure Computing (DASC'06)*, pages 341–348, Indianapolis, IN, USA.

Menken, I. and Blokdijk, G. (2009). *Cloud Computing Virtualization Specialist Complete Certification Kit - Study Guide Book and Online Course*. The Art of Service.

Microsoft (2011). Windows azure. Microsoft Corporation. Available in: http://www.windowsazure.com.

Microsystems, S. (2009). *Introduction to Cloud Computing Architecture*. Sun Microsystems, Inc., 1 edition.

Mihailescu, M., Rodriguez, A., and Amza, C. (2011). Enhancing application robustness in infrastructure-as-a-service clouds. In *Proc. First International Workshop on Dependability of Clouds, Data Centers and Virtual Computing Environments (DCDV 2011) in conjunction with The 41st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2011)*, Hong Kong, China.

Mitchell, M., Oldham, J., and Samuel, A. (2001). *Advanced Linux Programming*. New Riders.

Montgomery, D. C., Jennings, C. L., and Kulahci, M. (2008). *Introduction to Time Series Analysis and Forecasting*. Wiley series in probability and statistics.

Murari, K., D, J., Raju, M., RB, S., and Girikumar, Y. (2010). *Eucalyptus Beginner's Guide*, uec edition. For Ubuntu Server 10.04 - Lucid Lynx, v1.0.

Musa, J. D. (1998). *Software Reliability Engineering: More Reliable Software, Faster Development and Testing*. McGraw-Hill, 2 edition.

Ni, Q., Sun, W., and Ma, S. (2008). Memory leak detection in sun solaris os. In *Proceedings of the 2008 International Symposium on Computer Science and Computational Technology - Volume 02*, pages 703–707, Washington, DC, USA. IEEE Computer Society.

Nimbus (2012). Opennebula: The open source solution for data center virtualization. Nimbus.org Project. Available in: http://www.nimbusproject.org.

NIST (2011). National Institute of Standards and Technology, Information Technology Laboratory, U.S. Department of Commerce. Available in: http://csrc.nist.gov.

Nurmi, D., Wolski, R., Grzegorczyk, C., Obertelli, G., Soman, S., Youseff, L., and Zagorodnov, D. (2009). The eucalyptus open-source cloud-computing system. In *Proc. the 9th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid)*, pages 124–131, Washington, DC, USA. IEEE Computer Society.

Office, U. S. G. A. (1992). Patriot missile defense: Software problem led to system failure at dhahran, saudi arabia. Technical report. GAO/IMTEC-92-26.

OpenNebula (2012). Opennebula: The open source solution for data center virtualization. OpenNebula.org Project. Available in: http://opennebula.org.

Peng, J., Zhang, X., Lei, Z., Zhang, B., Zhang, W., and Li, Q. (2009). Comparison of several cloud computing platforms. In *Proc. of Second International Symposium on Information Science and Engineering (ISISE)*, pages 23–27, Shanghai, China. IEEE Press.

Prasad, V., Eigler, F. C., Keniston, J., Cohen, W., Hunt, M., and Chen, B. (2005). Locating system problems using dynamic instrumentation. In *Proceedings of Ottawa Linux Symposium*, Ottawa, Canada.

RackSpace (2011). Dedicated server, managed hosting, web hosting by rackspace hosting. RackSpace.com, Inc. Available in: http://www.rackspace.com/.

SalesForce (2011a). Social & mobile application development platform - force.com. SalesForce.com, Inc. Available in: http://www.force.com.

SalesForce (2011b). The social enterprise platform - salesforce.com. SalesForce.com, Inc. Available in: http://www.salesforce.com/platform.

Schwarz, G. (1978). *Estimating the Dimension of a Model*. Annals of Statistics.

Sempolinski, P. and Thain, D. (2010). A comparison and critique of eucalyptus, opennebula and nimbus. In *Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science*, CLOUDCOM '10, pages 417–426, Washington, DC, USA. IEEE Computer Society.

Shereshevsky, M., Crowell, J., Cukic, B., Gandikota, V., and Liu, Y. (2003). Software aging and multifractality of memory resources. In *Proc. Int. Conf. on Dependable Systems and Networks (DSN'03)*, pages 721 – 730, San Francisco, California.

Skotiniotis, T. and Chang, J. M. (2002). Estimating internal memory fragmentation for java programs. *Journal of Systems and Software*, **64**(3), 235 – 246.

Stanoevska-Slabeva, K., Wozniak, T., and Ristol, S. (2009). *Grid and Cloud Computing: A Business Perspective on Technology and Applications*. Springer Publishing Company, Incorporated, 1st edition.

Sun, D., Chang, G., Guo, Q., Wang, C., and Wang, X. (2010). A dependability model to enhance security of cloud environment using systemlevel virtualization techniques. In *Proc. First International Conference on Pervasive Computing, Signal Processing and Applications (PCSPA 2010)*, Harbin Institute of Technology, China.

SystemTap (2012). Systemtap tool. RedHat, Inc. Available in: http://sourceware.org/systemtap.

Thein, T. and Park, J. S. (2009). Availability analysis of application servers using software rejuvenation and virtualization. *Journal of Computer Science and Technology*, pages 339–346.

Trivedi, K. S., Vaidyanathan, K., and Goseva-Popstojanova, K. (2000). Modeling and analysis of software aging and rejuvenation. In *IEEE Annual Simulation Symposium*, pages 270–279, Greece.

Trivedi, K. S., Kim, D. S., Roy, A., and Medhi, D. (2009). Dependability and security models. In *Proc. of 7th International Workshop on the Design of Reliable Communication Networks (DRCN 2009)*, Washington, DC, USA.

Vaidyanathan, K. and Trivedi, K. S. (2005). A comprehensive model for software rejuvenation. *IEEE Transactions on Dependable and Secure Computing*, **2**, 124–137.

Vaidyanathan, K., Harper, R. E., Hunter, S. W., and Trivedi, K. S. (2001). Analysis and implementation of software rejuvenation in cluster systems. In *Proceedings of the 2001 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '01, pages 62–71, New York, NY, USA. ACM.

VMware (2007). *VMware Infrastructure Automating High Availability (HA) Services with VMware HA*. VMware. VMWARE Technical Note.

von Hagen, W. (2008). *Professional Xen Virtualization*. Wiley Publishing, Inc., Indianapolis, Indiana.

Xie, M., Poh, K., and Dai, Y. (2004). *Computing System Reliability: Models and Analysis*. Kluwer Academic Publishers.

Xu, Z. and Zhang, J. (2008). Path and context sensitive inter-procedural memory leak detection. In *Proc. of The Eighth International Conference on Quality Software*, pages 412–420, Washington, DC, USA.

Zhang, Q., Cheng, L., and Boutaba, R. (2010). Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications*, **1**(1), 7–18.

Appendices

A

Eucalyptus workload generator

A.1 Workload script

```
#!/bin/bash
# Script para reiniciar, matar e iniciar a VM

# Funcao que instancia 8 VMs da imagem ftp-server-img, com o tipo m1.xlarge
instanciaVMs(){
    i=0
    while [ $i -lt 8 ]
    do
     euca-run-instances -t m1.xlarge -z cluster-modcs -k admin emi-3FDE12D4
     i=`expr $i + 1`
    done
    sleep 10
    medeTempoBootVMs
}

matarVMs(){
    instancias=`euca-describe-instances | grep INSTANCE | awk '{print $2}'`
    euca-terminate-instances $instancias
}

reiniciarVMs(){
    instancias=`euca-describe-instances | grep INSTANCE | awk '{print $2}'`
    euca-reboot-instances $instancias
}
```

```
25
26 medeTempoBootVMs(){
27   booting=TRUE
28   flag=TRUE
29   while [ $booting = TRUE ]
30   do
31     tempoPrint=`date --rfc-3339=seconds`
32     running=`euca-describe-instances | grep running | wc -l`
33     if [ $running -gt 0 ]
34     then
35       if [ $flag = TRUE ]
36       then
37         echo "Pelo menos 1 VM foi iniciada: "$tempoPrint>>boot-vm.txt
38         echo "Pelo menos 1 VM foi iniciada: "$tempoPrint
39         flag=FALSE
40       fi
41     fi
42
43     # Se nao houver maquinas pendentes e forem menos de 8 em execucao,
         instancie novamente as que faltam
44     pending=`euca-describe-instances | grep pending | wc -l`
45     if [ $pending -eq 0 ]
46     then
47       if [ $running -lt 8 ]
48       then
49         num=`expr 8 - $running`
50         i=0
51         while [ $i -lt $num ]
52         do
53           euca-run-instances -t m1.xlarge -z cluster-modcs -k admin emi-3
                 FDE12D4
54           i=`expr $i + 1`
55         done
56       else
57         booting=FALSE
58         echo "Todas as VMs foram iniciadas: "$tempoPrint>>boot-vm.txt
59         echo "Todas as VMs foram iniciadas: "$tempoPrint
60       fi
61     fi
62     sleep 5
63   done
64 }
65
```

```
66 # Obtem o tempo inicial da execucao do script, no formato RFC3339: AAAA-
       MM-DD HH:MM:SS
67     tempoPrint=`date --rfc-3339=seconds`
68     tempoInicial=`date +%s`
69
70     echo "Inicio: "$tempoPrint >> historico-vm.txt
71     echo "Inicio - Execucao do script: "$tempoPrint
72     instanciaVMs
73
74 while [ True ]
75 do
76   tempoPrint=`date --rfc-3339=seconds`
77   tempo=`date +%s`
78
79   # REINICIAR TODAS AS VMS
80   # Se o tempo atual for menor que o tempo inicial + 5 horas, REINICIA
         todas as VMs
81   if [ $tempo -lt $(expr $tempoInicial + 18000) ]
82     then
83       #Comando para reiniciar todas as VMs
84       echo "Reinicio VMs.: "$tempoPrint >> historico-vm.txt
85       reiniciarVMs
86     fi
87
88   # MATAR TODAS AS VMS
89   # Se o tempo atual for maior que o tempo inicial + 5 horas, MATA todas
         as VMs
90   if [ $tempo -ge $(expr $tempoInicial + 18000) ]
91     then
92       #Comando para MATAR todas as VMs
93       echo "Kill VMs ....: "$tempoPrint >> historico-vm.txt
94       matarVMs
95
96       sleep 300
97
98       #Comando para iniciar todas as VMs
99       echo "Start VMs ...: "$tempoPrint >> historico-vm.txt
100      instanciaVMs
101      tempoInicial=`date +%s`
102    fi
103
104  sleep 600
105 done
```

B

Monitoring scripts

B.1 CPU utilization monitoring script

```
1  #!/bin/bash
2  # Script para monitoramento da utilizaÃ§Ã£o da CPU
3
4  # Escreve o cabecalho de identificacao dos dados
5  echo "%usr %sys %iowait %idle date time" >> monitoramento-cpu.txt
6
7  echo "%usr %sys %iowait %idle date time"
8
9  while [ True ]
10 do
11
12    # Armazena somente os campos de interesse
13    cpu=`mpstat 60 1| grep all | head -n1 | awk '{print $3,$5,$6,$11}'`
14
15    # Obtem o tempo atual, no formato RFC3339: AAAA-MM-DD HH:MM:SS
16    # O tempo corresponde ao retorno do mpstat,
17    # portanto  o uso de cpu eh a media do ultimo minuto
18    tempo=`date --rfc-3339=seconds`
19
20    # Separa data e hora do tempo obtido
21    data=`echo $tempo | cut -d\  -f1`
22    hora=`echo $tempo | cut -d\  -f2| gawk 'BEGIN{FS="-"}{print $1}'`
23
24    # Mostre na tela as informacoes capturadas pelos script
25    echo $cpu $data $hora
26
27    # Escreve no arquivo as informacoes do disco
28    echo $cpu $data $hora>> monitoramento-cpu.txt
```

```
29
30      #Executa o script a cada X unidade de tempo
31      #Comentado porque o mpstat ja espera os 60 segundos
32      #sleep 60
33
34  done
```

B.2 Disk utilization monitoring script

```
1   #!/bin/bash
2   # Script para monitoramento da utilização do disco
3
4   # Escreve o cabeçalho de identificação dos dados
5   echo "Used(KB)  Avail(KB) Use% Date Time" >> monitoramento-disco.txt
6
7   echo "Used(KB)  Avail(KB) Use% Date Time"
8
9   while [ True ]
10  do
11      # Obtem o tempo atual, no formato RFC3339: AAAA-MM-DD HH:MM:SS
12      tempo=`date --rfc-3339=seconds`
13
14      # Armazena somente os campos de interesse: Free, Available e %Used
15      disco=`df | grep /dev/sda | awk '{print $3,$4,$5}'`
16
17      # Separa data e hora do tempo obtido
18      data=`echo $tempo | cut -d\  -f1`
19      hora=`echo $tempo | cut -d\  -f2 | gawk 'BEGIN{FS="-"}{print $1}'`
20
21      # Mostre na tela as informaçãoes capturadas pelos script
22      echo $disco $data $hora
23
24      # Escreve no arquivo as informações do disco
25      echo $disco $data $hora >> monitoramento-disco.txt
26
27      # Executa o script a cada X unidade de tempo
28      sleep 60
29
30  done
```

B.3 Memory usage monitoring script

```
#!/bin/bash
# Script para monitoramento da utilização da RAM

# Escreve o cabeçalho de identificação dos dados
echo "Mem_used Mem_free Mem_buffers Mem_cached Swap_used Swap_free Date Time" >> monitoramento-memoria.txt
echo "Mem_used Mem_free Mem_buffers Mem_cached Swap_used Swap_free Date Time"

while [ True ]
do

    # Obtem o tempo atual, no formato RFC3339: AAAA-MM-DD HH:MM:SS
    tempo=`date --rfc-3339=seconds`
    memory=`free | grep Mem:`
    swap=`free | grep Swap:`

    # Armazena somente os campos de interesse: Free, Used, Buffers, ...
    memfree=`echo $memory | awk '{print $4}'`
    memused=`echo $memory | awk '{print $3}'`
    membuff=`echo $memory | awk '{print $6}'`
    memcache=`echo $memory | awk '{print $7}'`

    swapfree=`echo $swap | awk '{print $4}'`
    swapused=`echo $swap | awk '{print $3}'`

    # Separa data e hora do tempo obtido
    data=`echo $tempo | cut -d\  -f1`
    hora=`echo $tempo | cut -d\  -f2 | gawk 'BEGIN{FS="-"}{print $1}'`

    # Mostre na tela as informações capturadas pelos script
    echo $memused $memfree $membuff $memcache $swapused $swapfree $data $hora

    # Escreve no arquivo as informações da RAM
    echo $memused $memfree $membuff $memcache $swapused $swapfree $data $hora >> monitoramento-memoria.txt

    # Executa o script a cada X unidade de tempo
    sleep 60

```

```
38 done
```

B.4 *Eucalyptus-cloud* process monitoring script

```
1  #!/bin/bash
2  # Script para monitoramento do processo eucalyptus no Cloud Controller
3
4  # Escreve o cabecalho de identificacao dos dados
5  echo "%cpu %mem virt.mem res.mem data hora" >> monitoramento-processo-
       clc.txt
6
7  echo "%cpu %mem virt.mem res.mem data hora"
8
9  while [ True ]
10 do
11    # Obtem o PID do processo eucalyptus-cloud
12    pid=`ps aux | grep eucalyptus-cloud | grep "107 " | awk '{print $2}'`
13
14    # Obtem os campos de interesse
15    cpu=`pidstat -u -h -p $pid -T ALL -r 60 1 | sed -n '4p' | awk '{print
          $6,$12,$10,$11}'`
16
17    # Obtem o tempo atual, no formato RFC3339: AAAA-MM-DD HH:MM:SS
18    tempo=`date --rfc-3339=seconds`
19
20    # Separa data e hora do tempo obtido
21    data=`echo $tempo | cut -d\  -f1`
22    hora=`echo $tempo | cut -d\  -f2 | awk 'BEGIN{FS="-"}{print $1}'`
23
24    # Mostra na tela as informacoes capturadas pelos script
25    echo $cpu $data $hora
26
27    # Escreve no arquivo as informacoes do disco
28    echo $cpu $data $hora >> monitoramento-processo-clc.txt
29
30    # Executa o script a cada X unidade de tempo
31    sleep 60
32
33 done
```

B.5 *Eucalyptus-nc* process monitoring script

```
#!/bin/bash
# Script para monitoramento do processo eucalyptus no Node Controller

# Escreve o cabecalho de identificacao dos dados
echo "%cpu %mem virt.mem res.mem data hora" >> monitoramento-processo-nc.txt

echo "%cpu %mem virt.mem res.mem data hora"

while [ True ]
do

    # Obtem o PID do processo eucalyptus
    pid=`ps aux | grep eucalyptus | grep "106 " | awk '{print $2}'`

    # Obtem os campos de interesse
    cpu=`pidstat -u -h -p $pid -T ALL -r 60 1 | sed -n '4p' | awk '{print $6,$12,$10,$11}'`

    # Obtem o tempo atual, no formato RFC3339: AAAA-MM-DD HH:MM:SS
    tempo=`date --rfc-3339=seconds`

    # Separa data e hora do tempo obtido
    data=`echo $tempo | cut -d\  -f1`
    hora=`echo $tempo | cut -d\  -f2 | awk 'BEGIN{FS="-"}{print $1}'`

    # Mostra na tela as informacoes capturadas pelos script
    echo $cpu $data $hora

    # Escreve no arquivo as informacoes do disco
    echo $cpu $data $hora >> monitoramento-processo-nc.txt

    # Executa o script a cada X unidade de tempo
    sleep 60

done
```

B.6 Zombie process monitoring script

```
#!/bin/bash
# Script para monitoramento de processos zumbis

# Escreve o cabeçalho de identificação dos dados
echo "num_zumbis data hora" >> monitoramento-zumbis.txt

echo "num_zumbis data hora"

while [ True ]
do
    # Obtem o tempo atual, no formato RFC3339: AAAA-MM-DD HH:MM:SS
    tempo=`date --rfc-3339=seconds`

    # Armazena somente os campos de interesse
    num=`ps aux | awk '{if ($8~"Z"){print $0}}' | wc -l`

    # Separa data e hora do tempo obtido
    data=`echo $tempo | cut -d\  -f1`
    hora=`echo $tempo | cut -d\  -f2 | awk 'BEGIN{FS="-"}{print $1}'`

    # Mostre na tela as informações capturadas pelos script
    echo $num $data $hora

    # Escreve no arquivo as informações do disco
    echo $num $data $hora >> monitoramento-zumbis.txt

    # Executa o script a cada X unidade de tempo
    sleep 60

done
```

B.7 Memory leaking monitoring script

```
import subprocess, re, time

output_file = open("resultado_leak.csv","w")
PIPE = subprocess.PIPE

#mudar o tempo entre cada captura dos valores
intervalo_sleep = 60

```

```
#mudar para o nome do processo que se deseja monitorar
p = subprocess.Popen("ps -C apache2 --no-heading",shell=True,stdout=PIPE
    )

return_string = p.stdout.read()
pids = re.findall(r"\s(\d+)\s",return_string)

for pid in pids:
        output_file.write(pid+",")
output_file.write("\n")

while True:
        for pid in pids:
                num = subprocess.Popen("grep VmRSS /proc/"+pid+"/status"
                    ,shell=True,stdout=PIPE).stdout.read()
                valor = re.search(r"(\d+)\s",num)
                output_file.write(valor.group(0) + ",")
        output_file.write("\n")
        time.sleep(intervalo_sleep)
```

B.8 Memory fragmentation monitoring script

```
global alloc, fallback, fragmenting = 0

global who
probe kernel.trace("mm_page_alloc_extfrag"){
        who[execname(),pid(),pexecname(),ppid(),uid()] <<<1
        alloc = $alloc_order
        fallback = $fallback_order
        if(fallback<alloc){
                fragmenting++
        }
}

probe begin{
        printf("\nProbing...\n")
}

probe end{
        printf("Processo,Pai,UID,ocorrencias\n")
        foreach([process,pid,pexecname,ppid,uid] in who)
```

```
20                  printf("%s(%d),%s(%d),%d,%d\n", process,pid,pexecname,
                        ppid,uid,@count(who[process,pid,pexecname,ppid,uid]))
                        ;
21          printf("ocorrencia : %d\n", fragmenting)
22  }
```