ADVERTIMENT. La consulta d'aquesta tesi queda condicionada a l'acceptació de les següents condicions d'ús: La difusió d'aquesta tesi per mitjà del servei TDX (www.tesisenxarxa.net) ha estat autoritzada pels titulars dels drets de propietat intel·lectual únicament per a usos privats emmarcats en activitats d'investigació i docència. No s'autoritza la seva reproducció amb finalitats de lucre ni la seva difusió i posada a disposició des d'un lloc aliè al servei TDX. No s'autoritza la presentació del seu contingut en una finestra o marc aliè a TDX (framing). Aquesta reserva de drets afecta tant al resum de presentació de la tesi com als seus continguts. En la utilització o cita de parts de la tesi és obligat indicar el nom de la persona autora.

ADVERTENCIA. La consulta de esta tesis queda condicionada a la aceptación de las siguientes condiciones de uso: La difusión de esta tesis por medio del servicio TDR (www.tesisenred.net) ha sido autorizada por los titulares de los derechos de propiedad intelectual únicamente para usos privados enmarcados en actividades de investigación y docencia. No se autoriza su reproducción con finalidades de lucro ni su difusión y puesta a disposición desde un sitio ajeno al servicio TDR. No se autoriza la presentación de su contenido en una ventana o marco ajeno a TDR (framing). Esta reserva de derechos afecta tanto al resumen de presentación de la tesis como a sus contenidos. En la utilización o cita de partes de la tesis es obligado indicar el nombre de la persona autora.

WARNING. On having consulted this thesis you're accepting the following use conditions: Spreading this thesis by the TDX (www.tesisenxarxa.net) service has been authorized by the titular of the intellectual property rights only for private uses placed in investigation and teaching activities. Reproduction with lucrative aims is not authorized neither its spreading and availability from a site foreign to the TDX service. Introducing its content in a window or frame foreign to the TDX service is not authorized (framing). This rights affect to the presentation summary of the thesis as well as to its contents. In the using or citation of parts of the thesis it's obliged to indicate the name of the author

“We are born and reborn countless number of times, and it is possible that each being has been our parent at one time or another.”

-- Dalai Lama

Proactive Software Rejuvenation solution for web environments on virtualized platforms

Javier Alonso

Rejuvenation – the act of restoring to a more youthful condition

"You must wish to consume yourself in your own flame: how could you wish to become new unless you had first become ashes?"

- Friedrich Nietzsche from "Thus Spoke Zarathustra"

Proactive Software Rejuvenation solution for web environments on virtualized platforms.

By Javier Alonso López

Advisors:
Jordi Torres
Ricard Gavaldà

A dissertation submitted in partial fulfilment of the requirements for the degree of:

Doctor per la Universitat Politècnica de Catalunya

Barcelona, Spain
2011

UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

ACTA DE CALIFICACIÓN DE LA TESI DOCTORAL

Reunido el tribunal integrado por los abajo firmantes para juzgar la tesis doctoral:

Título de la tesis: Proactive Software Rejuvenation solution for web environments on virtualized platforms ....

Autor de la tesis: Javier Alonso López..

Acuerda otorgar la calificación de:

☐ No apto

☐ Aprobado

☐ Notable

☐ Sobresaliente

☐ Sobresaliente Cum Laude

Barcelona, …………… de…................……………. de ....….

El Presidente	El Secretario	
.... (nombre y apellidos)	.... (nombre y apellidos)	
El vocal	El vocal	El vocal
.... (nombre y apellidos)	.... (nombre y apellidos)	.... (nombre y apellidos)

Acknowledgements

This work has been partially supported by the Ministry of Science and Technology of Spain and the European Union (FEDER Funds) under contract TIN2004-07739-C02-01, TIN2007-60625 and TIN2008-06582-C03-01, and by the European Network of Excellence CoreGRID (Contract IST-2002-004265), EU PASCAL2 Network of Excellence, and by the Generalitat de Catalunya (2009-SGR-980 and 2009-SGR-1428).

A María Ángela y Jose Luís,

Mis Padres.

Abstract

The availability of the Information Technologies for everything, from everywhere, at all times is a growing requirement. We use Information Technologies from common and social tasks to critical tasks like managing nuclear power plants or even the International Space Station (ISS). However, the availability of IT infrastructures is still a huge challenge nowadays. In a quick look around news, we can find reports of corporate outage, affecting millions of users and impacting on the revenue and image of the companies.

It is well known that, currently, computer system outages are more often due to software faults, than hardware faults. Several studies have reported that one of the causes of unplanned software outages is the software aging phenomenon. This term refers to the accumulation of errors, usually causing resource contention, during long running application executions, like web applications, which normally cause applications/systems to hang or crash. Gradual performance degradation could also accompany software aging phenomena. The software aging phenomena are often related to memory bloating/ leaks, unterminated threads, data corruption, unreleased file-locks or overruns. We can find several examples of software aging in the industry.

The work presented in this thesis aims to offer a proactive and predictive software rejuvenation solution for Internet Services against software aging caused by resource exhaustion. To this end, we first present a threshold based proactive rejuvenation to avoid the consequences of software aging. This first approach has some limitations, but the most important of them it is the need to know *a priori* the resource or resources involved in the crash and the critical condition values. Moreover, we need some expertise to fix the threshold value to trigger the rejuvenation action. Due to these limitations, we have evaluated the use of Machine Learning to overcome the weaknesses of our first approach to obtain a proactive and predictive solution.

Finally, the current and increasing tendency to use virtualization technologies to improve the resource utilization has made traditional data centers turn into virtualized data centers or platforms. We have used a Mathematical Programming approach to virtual machine allocation and migration to optimize the resources, accepting as many services

as possible on the platform while at the same time, guaranteeing the availability (via our software rejuvenation proposal) of the services deployed against the software aging phenomena.

The thesis is supported by an exhaustive experimental evaluation that proves the effectiveness and feasibility of our proposals for current systems.

Agradecimientos

Este trabajo es el resultado de muchas horas de esfuerzo y dedicación. Sin embargo este no habría sido posible sin la ayuda, consejos, apoyo y sobre todo ánimos de muchas personas que a lo largo de 4 largos años me han ayudado a crecer como investigador, pero sobre todo como persona. Es el momento de darles las gracias en estas hojas, aunque se merecen mucho más.

Quiero agradecer a mi familia su apoyo incondicional durante este tiempo. Sin vosotros no habría sido posible, ni esto, ni muchas otras cosas buenas que me han pasado. En especial a mis padres por su esfuerzo diario para que me pudiera dedicar a este trabajo. A mi hermana Elena por su apoyo, incluso sin que ella sea consciente. A vosotros tres, nunca podré agradeceros todo lo que me habéis dado. Gracias.

A mis directores: Profesor Jordi Torres y Profesor Ricard Gavaldà. La noche y el día y no necesariamente en este orden ni el inverso. Que cada cual decida. Sus personalidades complementarías me han aportado un equilibrio que sin él, este trabajo no habría podido ser el que es, seguramente, sería mucho peor. Gracias a los dos, por vuestros consejos, por aguantar mi impaciencia, mis frustraciones y por animarme siempre. Aquí no tenéis un colega, tenéis un amigo.

A Luis Silva, sin él este trabajo habría sido imposible. Fue él quien me descubrió el mundo de la alta disponibilidad. Un hombre brillante y exigente. Fue un placer trabajar con él, incluso cuando chocaban nuestros caracteres. Fue con él con quien pude conocer Portugal, y con ello quizás cambiar mi vida. Lástima que nuestros caminos se separaran, espero que el futuro los vuelva a cruzar.

También quiero agradecer de forma especial a Artur Andrzejak, Rean Griffith, Josep Ll. Berral (gran investigador de corazón) y Dimiter Avresky su ayuda y apoyo para llevar a cabo el trabajo que contiene esta tesis.

No quiero olvidar a mis compañeros de *eDragon* (ya extinto, pero que nos marco a todos): David Carrera, Ramon Nou, Vicenç Ferrer, Iñigo Goiri, Ferran Julià y un largo etcétera. A TODOS GRACIAS por los buenos ratos pasados, por las conversaciones y reflexiones sobre el mundo de la investigación, ha sido un placer coincidir con todos

vosotros.

Tampoco voy a olvidar a mis compañeros del café de las 5pm: Jordi Guitart, Ramon Canal, Llorenç Cruz y Alex Pajuelo. Sin vosotros los días habrían sido más aburridos. Hemos pasado muchos ratos divertidos y las penas en grupo son menos penas. GRACIAS.

Para acabar quiero agradecer a Bea Otero, David López, Pau Artigas y Guille Pérez estar en mi vida. Conversar con vosotros sobre lo humano y lo divino. Ha sido un placer y conoceros un verdadero honor. Espero que nuestras charlas se sigan dando, da igual donde nos lleve el futuro.

Acknowledgements

This work is the result of many hours of effort and dedication. However this has not been possible without the help, advice, support and encouragement of many people around that over 4 long years have helped me grow as a researcher, but above all as a person. It's time to thank in this page, although they deserve more.

I want to thank my family for their unconditional support during this time. Without your support it would not have been possible. Especially my parents for their daily effort that it allow me to devote all my time to this work. Thank to my sister Elena for her support, even though she is not always conscious. To you three, I can never thank all that you have given me. Thanks.

Thanks to my advisors: Professor Jordi Torres and Professor Ricard Gavaldà. Night and day and not necessarily in that order or reverse. Let everyone decide. Complementary personalities that they have given me a balance that without it, this work would not have been what it is, surely, would be much worse. Thank you both for your advice, for putting up with my impatience, my frustrations and always encouraging me. Here you do not have a colleague, you have a friend.

Thanks to Luis Silva, without whom this work would have been impossible. It was he who discovered the world of high availability. A brilliant man and demanding. It was a pleasure working with him, even when our characters bumped. It was with him that I met Portugal, and thus change my life. Too bad our separate ways, I hope that future cross again.

I also special thanks to Artur Andrzejak, Rean Griffith, Josep Ll. Berral (awesome researcher at heart) and Dimiter Avresky. For their help and support to carry out the work contained in this thesis.

I do not want to forget my co-*eDragon* (now extinct group, but in our heart): David Carrera, Ramon Nou, Vicenç Ferrer, Iñigo Goiri, Ferran Julià and others. MANY THANKS for the good times past, for discussions and reflections on the world of research. It has been a pleasure to meet you all.

Nor will I forget my co 5pm coffee: Jordi Guitart, Ramon Canal, Llorenç Cruz and

Alex Pajuelo. Without our talks, the days would have been more boring. We spent many great moments and penalties are less penalties in group. THANKS.

To finish I want to thank Bea Otero, David Lopez, Pau Artigas and Guillermo Perez to be in my life. Talk to you about the human and divine have been a pleasure and an honor to meet you. I hope our talks will continue in the future, no matter where the future will put us.

Contents

Abstract i

Agradecimientos v

Acknowledgments vii

Table of contents xv

List of Figures xix

List of Tables xxii

1 Introduction 1

1.1 Availability on IT infrastructures 3
1.2 The Cost of the Complexity 5
1.3 Software Errors 6
1.4 Software Aging Phenomena 8
1.5 Thesis Contributions 10
1.5.1 Proactive Software rejuvenation framework 10
1.5.2 A framework for software aging prediction based on Machine Learning 11
1.5.3 Autonomic High Availability and Resource Usage Optimization 13
1.6 Thesis Organization 13

2 Proactive Software Rejuvenation framework 15

2.1 Introduction 17
2.2 Rationale for a new scheme of Software rejuvenation 18
2.2.1 How to achieve our goals 20
2.3 Virtualized Clustering Rejuvenation Framework Description 22
2.3.1 Monitoring Infrastructure 24
2.3.2 Aging and Anomaly Detector 25
2.3.3 Rejuvenation Infrastructure 27
2.4 Experimental Evaluation 28
2.4.1 Experimental Environment 28

2.4.1.1 TPC-W benchmarking 28
2.4.1.2 Webservice Container Axis v1.3 29
2.4.1.3 OGSA-DAI middleware 29
2.4.2 Experimental Setup 30
2.4.3 Experimental Results 30
2.4.3.1 Virtualized Clustering Rejuvenation Framework Overhead 30
2.4.3.2 Average Performance under Virtualization Overhead . . 34
2.4.3.3 Virtualized Clustering Rejuvenation Framework Effectiveness 37
2.4.3.4 Downtime Achieved by the Virtualized Clustering Rejuvenation Framework 46
2.4.3.5 Session-data issue and the overhead to maintain it on Tomcat 51
2.4.3.6 VM-ClusteringRejuv effectiveness in Cluster Environments 52
2.5 A proposal Software rejuvenation framework for Legacy Application Servers 57
2.5.1 Crash-only Concept 58
2.5.2 Crash-only and masking failure Architecture for Legacy Application servers 59
2.5.3 Discussion of the solution 62
2.6 Related Work 63
2.7 Summary 65

3 A framework for software aging prediction based on Machine Learning 67
3.1 Introduction 69
3.2 Modelling Assumptions and Prediction Strategy 71
3.2.1 Motivating Examples 72
3.2.1.1 Example: Nonlinear Resource Behavior 72
3.2.1.2 Example: Different Viewpoints on a Resource 74
3.2.2 Prediction Assumptions 76
3.3 Experimental Setup 78
3.3.1 Machine Learning Evaluation Process 78
3.3.1.1 Training Process 79
3.3.1.2 Validation Process 80
3.3.2 Experimental Environment 81
3.4 Time-based prediction & its Evaluation 84
3.4.1 Linear Regression 84
3.4.2 M5P 85
3.4.3 Experimental Evaluation 85
3.4.3.1 Deterministic Software aging 86
3.4.3.2 Dynamic and Variable Software aging 87

3.4.3.3 Software Aging Hidden within Periodic Resource Behavior . . . 90
3.4.3.4 Dynamic and Variable Software aging due to Two Resources . . . 92
3.4.4 Understanding M5P generated Models . . . 94
3.5 "Red-light" Alarm-based Prediction & its Evaluation . . . 95
3.5.1 J48 . . . 96
3.5.2 Naive Bayes . . . 96
3.5.3 IBk . . . 97
3.5.4 Experimental Evaluation . . . 97
3.5.4.1 Deterministic Software aging . . . 97
3.5.4.2 Dynamic and Variable Software aging . . . 100
3.5.4.3 Software Aging Hidden within Periodic Resource Behavior . . . 105
3.5.4.4 Dynamic and Variable Software aging due to Two Resources . . . 105
3.5.4.5 Learned Lessons . . . 110
3.6 Adaptive and Predictive Software Rejuvenation Framework . . . 111
3.6.1 Monitoring Subsystem . . . 111
3.6.2 Analysis Subsystem . . . 112
3.6.3 Planning Subsystem . . . 112
3.6.4 Execution Subsystem . . . 113
3.7 Adaptable Monitor to Determine Software Aging Root Cause Failure . . . 113
3.7.1 Technology Used . . . 114
3.7.1.1 Aspect Oriented Programming . . . 114
3.7.1.2 Java Management Extensions . . . 115
3.7.2 Architecture Description . . . 115
3.7.2.1 Aspect Component . . . 116
3.7.2.2 JMX Monitoring Agents . . . 117
3.7.2.3 JMX Manager Agent . . . 117
3.7.2.4 External Front-end . . . 117
3.7.3 Root Cause Determination Strategy . . . 117
3.7.4 Experimental Evaluation . . . 118
3.7.4.1 Experimental Setup . . . 118
3.7.4.2 Experimental Results . . . 120
3.8 Related Work . . . 126
3.8.1 Related Work on Prediction . . . 126
3.8.2 Related Work on Monitoring . . . 127
3.9 Summary . . . 129

4 Autonomic High Availability and Resource Usage Optimization 131
4.1 Introduction . . . 133
4.2 Virtualization Manager Framework Description . . . 133

4.2.1 Failure Predictor 134
4.2.2 High Availability and Resource Optimization Scheduler 135
4.2.2.1 Catcher Manager 135
4.2.2.2 Decision Manager 137
4.2.2.3 Reconfiguration and HA Manager 138
4.2.3 High Availability Load Balancer 140
4.3 Problem Formulation and Discussion 140
4.3.1 Problem Statement 140
4.3.2 Decision Manager Model Definition 141
4.3.2.1 The Objective Functions 142
4.3.2.2 Constraints 144
4.3.2.3 Restrictive Model (RM) 148
4.3.2.4 Mixed Model (MM) 149
4.3.3 Framework Availability Discussion 151
4.4 Experimental Evaluation 152
4.4.1 Experimental Setup 152
4.4.1.1 Analyzing the maximum capacity of our experimental environment 154
4.4.2 Experimental Results 155
4.4.2.1 Impact of the Framework Infrastructure 156
4.4.2.2 Impact of the Live-Migrations 157
4.4.2.3 Model Comparison 158
4.5 Related Work 163
4.6 Summary 165

5 Conclusions and Future work 167
5.1 Conclusions 169
5.1.1 Proactive Software rejuvenation framework 169
5.1.2 A framework for software aging prediction based on Machine Learning 169
5.1.3 Autonomic High Availability and Resource Usage Optimization . 170
5.2 Future Work 171

List of Acronyms 175

A Prediction Framework: System and Derived Metrics used 177
A.1 Detailed description of variables used in the Training Process 177

B Extended Prediction Algorithms comparison 183
B.1 Introduction 183
B.2 Deterministic Software aging 183
B.3 Dynamic and Variable Software aging 183
B.4 Dynamic and Variable Software aging due to Two Resources 185

Bibliography **186**

List of Figures

1.1 (a) Error type division in middle 80's. (b) Error Type division in 2005 . . 7
1.2 Error propagation chain 8

2.1 Virtualized Clustering Rejuvenation Process 21
2.2 Virtualized Clustering Rejuvenation Architecture 23
2.3 Virtualized Clustering Rejuvenation Components Behavior 26
2.4 Comparing the Throughput of Tomcat with Axis on top of OS, on top of XEN, and on top of XEN with VM-ClusteringRejuv 32
2.5 Comparing the Throughput of TPC-W on top of OS, on top of XEN, and on top of XEN with VM-ClusteringRejuv 32
2.6 Comparing the Throughput of OGSA-DAI on top of OS, on top of XEN, and on top of XEN with VM-ClusteringRejuv 33
2.7 Throughput Comparison: VM-ClusteringRejuv evaluation with Tomcat/Axis 38
2.8 Response Time Comparison: VM-ClusteringRejuv evaluation with Tomcat/Axis 39
2.9 Throughput Comparison: VM-ClusteringRejuv evaluation with TPC-W Benchmark 40
2.10 Response Time Comparison: VM-ClusteringRejuv evaluation with TPC-W Benchmark 40
2.11 OGSA-DAI unstable response time behavior under constant burst workload 41
2.12 OGSA-DAI unstable Throughput behavior under constant burst workload 42
2.13 OGSA-DAI unstable Response Time behavior under constant workloads . 42
2.14 OGSA-DAI memory consumption under burst workload 43
2.15 Throughput Comparison: VM-ClusteringRejuv evaluation with OGSA-DAI Benchmark 44
2.16 Response Time Comparison: VM-ClusteringRejuv evaluation with OGSA-DAI Benchmark 44
2.17 OGSA-DAI Benchmark memory usage with VM-ClusteringRejuv Active 45
2.18 Client perceived availability for different restart mechanisms with Tomcat/Axis 47
2.19 Client perceived availability for different restart mechanisms with OGSA-DAI 49

2.20 Overhead comparison of session replication with session objects of 8kb with Tomcat/Axis . . . 52
2.21 Throughput of Server S1 (Tomcat/Axis) in a cluster, suffering from severe aging. . . . 53
2.22 Throughput of Server S2 (Tomcat/Axis) in a cluster, suffering from severe aging. . . . 54
2.23 Cluster Throughput comparison, with and without VM-ClusteringRejuv, using Tomcat/Axis . . . 55
2.24 Cluster Throughput comparison, with and without VM-ClusteringRejuv and automatic restart, using Tomcat/Axis . . . 55
2.25 Impact Evaluation of a "clean" restart in front of "blind" restart and "blind" reboot. . . . 56
2.26 Crash-Only and masking failure Architecture . . . 60

3.1 MTTR Description in a reactive rejuvenation mechanism . . . 69
3.2 MTTR Description in our Threshold-based rejuvenation mechanism . . . 70
3.3 MTTR Description in our Prediction-based rejuvenation mechanism . . . 71
3.4 Progressive memory consumption of the Java Application. . . . 74
3.5 The Tomcat memory consumption from system and heap perspectives . . 75
3.6 Detailed Architecture of the Prediction Framework . . . 77
3.7 Machine Learning Model Selection process . . . 79
3.8 Experimental environment components. . . . 81
3.9 The MAE, S-MAE and PRE/POST MAE calculation process in dynamic experiments . . . 88
3.10 Time to crash predicted vs. Tomcat Memory Evolution using M5P . . . 89
3.11 Time predicted vs. Java Heap Tomcat Memory Evolution . . . 91
3.12 Time predicted and resource evolution during the two-resource experiment 93
3.13 M5P Model in the experiment 5.4 . . . 94
3.14 The three different prediction classes . . . 96
3.15 Alarm classification results for 75 EBs workload, (a)J48, (b) Naive Bayes, and (c) IBk . . . 99
3.16 Alarm classification results for 150 EBs workload, (a)J48, (b) Naive Bayes, and (c) IBk . . . 100
3.17 J48 Model generated using the Training data set in Dynamic and Variable Software aging experiment . . . 101
3.18 The last 4 tuples of validation experiment. . . . 102
3.19 Alarm classification results for a dynamic and variable software aging scenario, (a)J48, (b) Naive Bayes, and (c) IBk. . . . 104
3.20 Alarm classification results for a 100EBs of workload and software aging within Pattern behavior, (a)J48, (b) Naive Bayes, and (c) IBk. . . . 106
3.21 Alarm classification results for last 2 hours and 30 minutes of 100EBs workload and software aging within Pattern behavior, (a)J48, (b) Naive Bayes, and (c) IBk. . . . 107

3.22 Alarm classification results for 100EBs workload and dynamic software aging trend and two resources involved, (a)J48, (b) Naive Bayes, and (c) IBk. 109
3.23 Framework components 111
3.24 Components of Monitoring Framework 116
3.25 Resource Consumption vs Component Usage map 118
3.26 Throughput of TPC-W under a dynamic workload 120
3.27 Injection in component A (100KB) 122
3.28 Detail of injection in 4 components 123
3.29 Resource Consumption vs. Component Usage map composed by JMX Manager Agent 124
3.30 Determination of Root failure in 4 different injections 125

4.1 Provider Architecture Description 134
4.2 Conceptual Provider Architecture Description 135
4.3 Communication process between framework components 136
4.4 Matrix evaluation example conducted by RHAM 139
4.5 Recovery architecture 141
4.6 Experimental Scenario Proposed 153
4.7 TPC-W Throughput with one live-migration 158
4.8 TPC-W Throughput with two live-migrations 158
4.9 Total number of a) Services, b) Migrations and c) Replicas by every model 161
4.10 Experiment execution using MM: Amount of Services and VMs 163
4.11 Comparison of VM allocation of *MM* (a) and $C = 0$ Model (b) 164

List of Tables

1.1 The Direct Costs of Downtime 6

2.1 Detailed experimental Setup Description 31
2.2 Comparing the Overhead in Web services Environment using Tomcat/Axis 33
2.3 Comparing the Overhead in Web Application Environment using TPC-W 34
2.4 Comparing the Overhead in Grid Environment using OGSA-DAI 34
2.5 Measured virtualization overhead in the time interval 10 minutes with OGSA-DAI 36
2.6 Measured virtualization overhead in the time interval 10 minutes with Tomcat/Axis 37
2.7 Minimal required performance q_{min} depending on the $MTTF$ t for $rejuv_{virt} = 0$ and $rejuv_{phys} = 12.5$ seconds 37
2.8 Detailed values from 1 hour of Execution with OGSA-DAI 45
2.9 Comparing Downtime and Failed Requests with different restart options in AXis 48
2.10 Comparing Downtime and Failed Requests with different restart options in OGSA-DAI 49
2.11 Analysis of Availability 50
2.12 Reboot times from different Systems 58

3.1 Detailed experimental Setup Description 82
3.2 Variables used in every experiment to build the model 83
3.3 MAE obtained under constant software aging experiment 87
3.4 MAE obtained under dynamic and variable software aging experiment . . 90
3.5 MAE obtained in the periodic pattern experiment 92
3.6 Confusion Matrix with 75EBs: J48/NB/IBk 98
3.7 Confusion Matrix with 150EBs: J48/NB/IBk 98
3.8 Confusion Matrix of Dynamic and Variable Software aging: J48/NB/IBk . 101
3.9 Confusion Matrix of Dynamic and Variable Software aging: J48/NB/IBk with 14% of orange instances and 11% of red instances 103
3.10 Confusion Matrix of software aging hidden within Periodic Pattern: J48/NB/IBk 105

3.11 Confusion Matrix of Dynamic and Variable Software aging due to two resources: J48/NB/IBk 106
3.12 Confusion Matrix of Dynamic and Variable Software aging due to two resources equilibrating instances: J48/NB/IBk 108
3.13 Detailed experimental Setup Description 119
3.14 Detailed overhead introduced by our monitoring framework 120

4.1 Description of Testbed machines and their roles 152
4.2 Description of Virtual Machines Type used 153
4.3 Maximum number of Services accepted by different solution scenarios . . 155
4.4 Virtual Machines and service creation 157
4.5 Maximum services accepted by the models 160
4.6 Availability achieved by the models 162

B.1 MAE S-MAE PRE and POST MAE obtained under constant software aging experiment by a diverse set of ML algorithms 184
B.2 MAE S-MAE PRE and POST MAE obtained under dynamic and variable software aging experiment by a diverse set of ML algorithms . . 184
B.3 MAE S-MAE PRE and POST MAE obtained under dynamic and variable software aging due to two resources experiment by a diverse set of ML algorithms 185

Chapter 1

Introduction

In current industrialized society, we are used to using Information Technologies (IT)[1]for everything, from everywhere. We use IT from common and social tasks like being in touch with our friends through social networks, using the Internet Services for sending our tax report to avoid bothering queues, and organizing our holidays from our living room. IT is even used to manage critical tasks like managing nuclear power plants or even the International Space Station (ISS). Nonetheless, the most worrying is that the demands of society and industry continue to grow day by day, making IT essential in our lives.

1.1 Availability on IT infrastructures

The new requirements on availability and ubiquity of IT infrastructures makes that the enterprise IT departments are forced to continuously adapt themselves to new needs as they appear. In particular, availability of the information and services all the time and from everywhere is today a growing common requirement. To achieve these new challenges demanded by the industry and society, new IT infrastructures have had to be created and designed. Applications have to interact among themselves and with the environment to achieve these new goals, resulting in complex IT infrastructures.

However, the availability of the IT infrastructures is still a huge challenge nowadays. In September 13, 2010, the JP Morgan Chase bank (the second-largest U.S. bank) on-line services were unavailable for three days. In a first moment (Monday, September 13, evening) Chase officials said that the service would be restored Wednesday early morning, though, the customers still had delays on on-line services Wednesday evening. This computer outage affected 16 millions of on-line customers and business, preventing them to access their own bank accounts, order transfers or other operations. The importance of the outage impact on the JP Morgan Chase Bank image could be measured in quantitative numbers: Currently (2010), some 51% of consumers prefer online banking to other methods, up from 44% in 2008, according to a study issued August, 2010 by Mercator Advisory Group, a consulting firm that specializes in the financial-services industry. Or qualitatively, observing some of the opinions from customers: "A system outage of this length communicates to me that they really don't have a handle on their systems," said a Chase customer in Chicago who does all of his banking on-line. Another customer said "My relationship with Chase is now under reconsideration." Still another customer who runs an Internet domain name service and does his business banking with Chase, said he

[1]Information Technologies (IT) or Information and Communication Technologies (ICT) are synonym, using the US acronym option IT in this document.

was upset by the lack of communication from the bank. "The only thing on the website was this dark image that said something to the effect of the fact that it was temporarily down, but there was no indication of when it was coming back up," he said.[2]

This is not an isolated case. During last years, several banks and other industries have suffered outages and IT availability related problems. Last February, 2010, Bank of America, the largest U.S. bank, on-line service was off line for 12 hours, affecting near of 24 million on-line customers, 80% of total on-line customers. Last January, 2010, because of mainfraime outage, HSBC customers were unable to use cash machines and online banking services. Moreover, JP Morgan Chase bank suffered another 15-hours outage last August, 2010.

These downtime problems are not just from financial system: last year, 2009, T-Mobile Sidekick was unavailable to use for a whole week and lost amount of data from customers. Even Facebook site suffered those problems. This October 5, 2010, Facebook site was slow or even unavailable for some of its 500 millions of users.

Actually, according to Dunn & Bradstreet, 49% of Fortune 500 companies experience at least 1.6 hours of downtime per week. That translates into more than 80 hours annually. This includes software crashes, required system reboots and normal maintenance. How would the customers react to an 80-hour outage?

Such cases make companies invest an important part of their IT budget on servers looking for an improvement on the performance and availability of their software systems. However, the utilization of these servers was quite low, around 5% to 20% [29]. For this reason, the renaissance of virtualization became a reality. Introducing virtualization in data centers [47, 85] in order to consolidate several services in the same physical machine, has become in the last years a growing trend. Virtualization technology opens new possibilities like the capability for different operating systems to coexist, applications or services sharing the same physical resources, without disturbing each other. The usage of virtual machines (VMs) and virtual machine monitors (VMMs) makes that possible.

Following this trend, the usage of virtualization allows us to see a set of physical machines as an one single conceptual machine with all resources merged. Using that *"machine abstraction"*, we can deploy several virtual machines with different operating systems and applications on top of these virtualized platforms or data centers, without taking into account the underlying hardware and the physical details.

Even though these virtualized platforms help to improve the security, reliability, and availability of services, while reducing the cost and management, the complexity, due to the industry/society demanded requirements, of the deployed systems is still growing.

[2] Sillicon Valey Mercury News.com.a [URL] http://www.mercurynews.com/news/ci_16086560?source=rss

This fact increases the difficulty of managing them. Moreover, our current and growing reliance on these IT systems to manage critical and ordinary tasks in our lives requires these systems not only to offer an acceptable performance but also continuous availability. To meet these social and industrial closed to 100% availability requirements, more skilled developers and administrators are needed to maintain these complex and heterogeneous systems, resulting in a large fraction of the total cost of ownership (TCO) of these systems. However, the complexity is achieving such a level that even the best administrators can hardly cope with it. For example, from 1994 until 2001, the Linux kernel source alone increased by a factor of ten, in only 7 years. For this reason, autonomic computing seems to be the only possible solution [69]. We need more self-managed systems to overcome the increasing and non-human affordable complexity.

1.2 The Cost of the Complexity

As the system complexity is increasing day by day, the number of failures due (directly or indirectly) to this complexity has also been increasing, resulting in undesirable behaviors, poor levels of service, and even total outages. The need to prevent or gracefully deal with (unplanned and planned) outages of business and critical systems is clear, given the industry huge loss due to the downtime per hour. A study [64] showed the average downtime or service degradation cost per hour for an average typical enterprise is around US$125,000. Moreover, outages have a negative impact on the company image that could affect profits indirectly as we have presented before with customers' opinion of JP Morgan Chase bank on-line services. This type of facts could lead customers from JP Morgan Chase bank to live other banks or even, a loss of future new customers. Table 1.1 summarizes some of the downtime (direct) costs according to the business.

This increasing system complexity and the failures related with it, have also changed the companies' IT budget distribution. Fifteen years ago, the companies spent around 75% of their budget on new hardware and software, and 25% on repairs, operations and maintenance. But this trend has changed dramatically in last years, spending around 70% - 80% in repairations, maintenance and operations related (directly or indirectly) with the complexity [64].

However, the system complexity is not the unique reason of downtimes. If we analyze the literature about causes of downtime, we can divide the unplanned downtimes reasons in three main categories: Human or operator errors, software errors, and hardware errors. According to [98][3], the distribution of errors in these categories was (approximately)

[3]The newest reference we could find where the error classification is presented.

Industry	Average Downtime cost per hour
Brokerage Services	$6.48 million
Energy	$2.8 million
Credit card	$2.58 million
Telecomm	$2 million
Financial	$1.5 million
Manufacturing	$1.6 million
Financial institutions	$1.4 million
Retail	$1.1 million
Pharmaceutical	$1.0 million
Chemicals	$704,000
Health care	$636,000
Media	$340,000
Airline reservations	$90,000

[a] Sources: Network Computing, The Meta Group, and Contingency Planning Research. Extracted from [81]

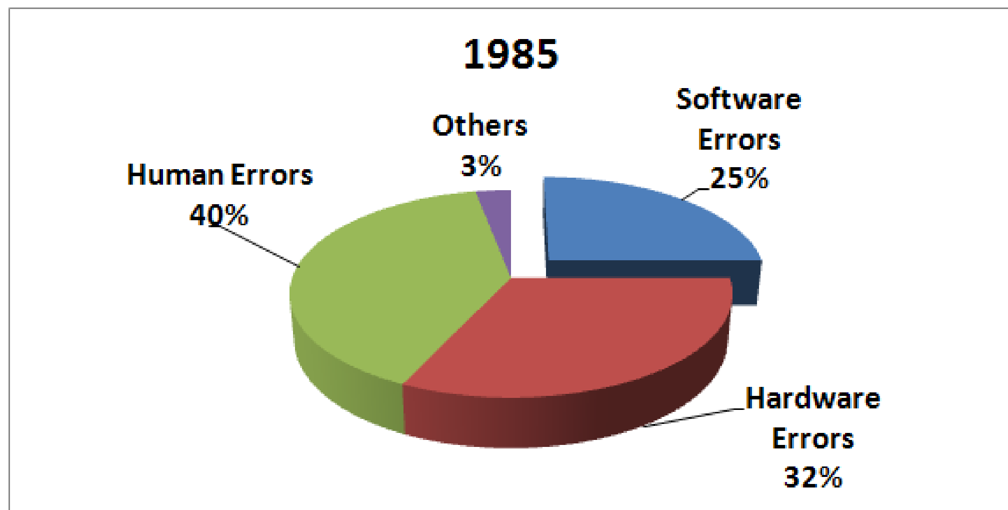
Table 1.1 The Direct Costs of Downtime

40%, 40% and 20%, respectively, in 2005. If we observe in detail the evolution of this distribution from 1985 to 2005 [56, 57, 97, 94, 98], we can see that the proportion of hardware errors has decreased (from 32% in the 80s to 15-20% by 2005). The number of operator errors remains fairly stable along time: Although system complexity has grown and keeps growing, and this would suggest an increase in human errors during the management tasks, the operators currently have better administration tools to automate some parts of their task. Nevertheless this 40% shows clearly that there is important room for improvement.

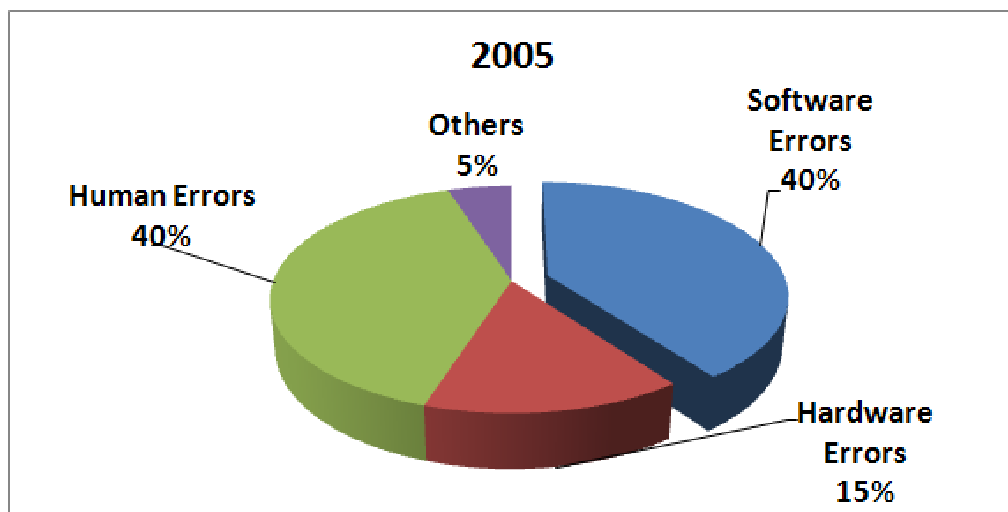
On the other hand, the software errors have been increasing along the time (from 25% in 80s to 40% today), due mainly to the complexity of the current software and the heterogeneous environments where the systems have to work. Figure 1.1 summarizes this error evolution during the last twenty-thirty years in the IT systems.

1.3 Software Errors

While it is true that more sophisticated development/testing and debugging tools are appearing to help developers avoid software errors like [83, 96], they still appear and have an important impact over the application availability, becoming the first reason of



(a)



(b)

Figure 1.1 (a) Error type division in middle 80's. (b) Error Type division in 2005

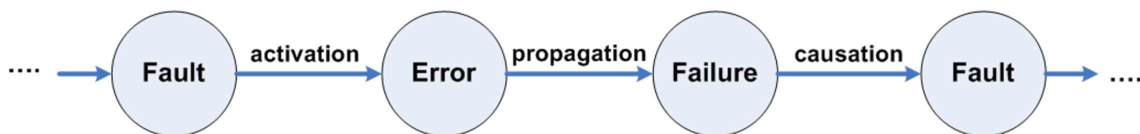


Figure 1.2 Error propagation chain

unplanned downtimes. In fact, fixing all faults during testing and debugging phase is a titanic task with an unaffordable cost, because these tools often cannot access third-party modules code.

Even more, the transient and intermittent faults are too difficult to fix because it is highly complicated to reproduce them. Design faults are too often dormant and they activate only under unknown or rare circumstances becoming in nature transient or intermittent software faults.

In the previous text we have been using the terms failures, errors, faults as synonyms. But to be precise about causes, effects, and observable effects, we will use the terms fault, error/bug, and failure following the definitions in [19]. Briefly, a fault is the cause of an error; it is active when it causes an error, otherwise it is a dormant fault. An error or bug is a part of the total state of the system that may lead to its subsequent service failure. And a failure is a visible erroneous operation of the system, probably due to an error or a set of them. Figure 1.2 shows the error propagation chain. We can observe that when a fault is activated, it causes an error. This error or an accumulation of errors could propagate a failure in a component or a system. This failure could cause a permanent or transient fault in other service or component that receives service from the given system, building *the chain of threats*.

1.4 Software Aging Phenomena

In our work, we are mainly focused on software errors; human/operator errors and hardware errors are out of scope of our work. For this reason, we need to analyze the causes of software failures. Several studies [63, 113, 43] have reported that one of the causes of the unplanned software outages is the **software aging phenomena**. This term **refers to the accumulation of errors, usually causing resource contention, during long running application executions**, like web applications or grid jobs, which normally cause applications/systems hang or crash [60]. **Gradual performance degradation could also accompany software aging phenomena**. This phenomena becomes critical in 24x7 applications [35].

The software aging phenomenon is often related to memory bloating/leaks, untermi-

nated threads, data corruption, unreleased file-locks or overruns (as examples).

This phenomena is likely to be found in any type of software that is complex enough, but it is particularly troublesome in long-running applications. It is not only a problem for desktop operating systems: it has been observed in telecommunication systems [20], web servers [15, 86, 59], enterprise clusters [35], OLTP systems [34] and spacecrafts systems [111]. This problem has even been reported in military systems [82] with severe consequences such as loss of lives.

Trying to fix the problem in development/testing phase could be a heavy task with an unaffordable cost. For this reason, the applications have to deal with the software aging problem in production and operation stage. Due to this scenario, software rejuvenation techniques [63] become necessary to mitigate or avoid the consequences of software aging. The software rejuvenation techniques are based mainly on the restart of the service to come back to a state without aging. The software rejuvenation techniques do not avoid the software aging problem, they only mitigate the symptoms.

Some relative recent experimental studies have proved that rejuvenation can be a very effective technique to avoid failures even when it is known that the underlying middleware [106] or the protocol stack [27] suffers from clear memory leaks.

Software rejuvenation strategies can be divided into two basic categories: Time-based and Proactive/Predictive-based strategies. It is important to remark that not all proactive solutions are predictive, but all predictive solutions are proactive.

In time-based strategies, rejuvenation is applied regularly and at predetermined time intervals. In fact, time-based strategies are widely used in real environments, such as web servers [115, 52]. One of the last time-based strategies can be found at International Space Station (ISS). Every two months the File Server Station Support Computer (FS SSC) has to be rebooted[4](regularly) to mitigate of the problems derived from a memory leak, which avoids to run correctly different applications. The last reboot, while this document was wrote, was conducted December 12, 2010.

In Proactive/Predictive rejuvenation, system metrics are continuously monitored and the rejuvenation action is triggered when a crash or system hang up due to software aging seem to approach. This approach is a better technique, because if we can predict/anticipate the crash and apply rejuvenation actions only in these cases, we reduce the number of rejuvenation actions with respect to the time-based approach. Moreover, the time-based approaches are unable to deal with dynamic or changing software aging phenomena. The time-based approaches apply a recovery action regularly, however, what happens if a crash (due to the software aging) occurs between two consecutive rejuvenation actions? The

[4]Information extracted from NASA website ISS daily reports. [URL] http://www.nasa.gov/directorates/somd/reports/iss_reports/index.html

system would crash and the time-based rejuvenation would be useless.

In view of this scenario, the thesis focuses on offering a **proactive and autonomic software rejuvenation technique** for Internet Services[5]in the presence of **software aging due to resource consumption**. Furthermore, **our proposal has to be effective in** the new and every day more extended **virtualized platform**s.

The global contribution of this thesis is divided in three main contributions: 1) proposing a clean and proactive rejuvenation technique that allow us to apply the rejuvenation technique based on thresholds according to the resource state, 2) adding a predictive characteristic to our proactive rejuvenation mechanism, and presenting an evaluation of some Machine Learning algorithms to help us in the task to predict the time to crash due to the software aging. Finally, 3) the last contribution of the thesis is the design, development and evaluation of a framework to manage virtualized environments to guarantee the availability of the services deployed. Our approach offers high availability against software aging in a transparent way for the clients. Furthermore, our framework also optimizes the resource utilization, accepting as many services as possible in the platform.

1.5 Thesis Contributions

1.5.1 Proactive Software rejuvenation framework

The first part of the thesis is focused on offering a clean and non-intrusive proactive software rejuvenation mechanism. Our approach is based on monitoring the Quality of Service (QoS) metrics (like throughput or latency) of a service and the system metrics. When a resource or other metric (even it could be a merge of various metrics) monitored achieves a determined threshold, the framework has to trigger the rejuvenation action by itself without or with minimum human intervention. Moreover, the rejuvenation action has to be *clean*. That is, the outage is small or even zero during the rejuvenation process, if it is possible.

On the other hand, the solution has to be Internet Service non-intrusive and easy to deploy. If a solution needs to re-engineer the current applications or web server containers, the solution has a limited spectrum of action, or even useless in most cases. For this reason, we decided to use virtualization technology and a clustered-based solution to avoid this problem, achieving very promising results in quite different scenarios: from web applications to grid services, through webservices, as we present later, in Chapter 2. Our

[5]Internet Services cover but not limited to web services, Grid services, web applications, etc...

solution is able to avoid outages during the rejuvenation action, taking into account the state of the services.

Furthermore, virtualization technology helps us to reduce the deployment cost and reduce the hardware needed reducing the complexity of our proposal.

The work performed in this area has resulted in the following publications:

[108] Luis Silva, Javier Alonso and Jordi Torres. **Using virtualization to improve Software rejuvenation**. IEEE Transactions on Computers, Vol. 58, No 11, 2009

[107] Luis Silva, Javier Alonso and Jordi Torres. **Using virtualization to improve Software rejuvenation**. In Proceedings of the 2007 IEEE International Symposium on Network Computing and Applications (NCA'07) (Best Application Paper Award), 2007

[4] Javier Alonso, Luis Silva, Artur Andrzejak, Paulo Silva and Jordi Torres. **High-available grid services through the use of virtualized clustering**. In Proceedings of the 8th IEEE/ACM International Conference on Grid Computing (GRID'07), 2007

[8, 9] Javier Alonso, Jordi Torres, and Luis Silva. **Carrying the Crash-Only Software Concept to the Legacy Application Servers**. In Proceedings of the CoreGRID Workshop on Programming Models Grid and P2P System Architecture Grid Systems, Tools and Environments 12-13 June 2007, Heraklion, Crete, Greece.

- Also published on Springer Book "Making Grids Work", pp. 165-174, ISBN: 978-0-387-78447-2, 2008.

[10, 11] Javier Alonso, Jordi Torres, Luis Silva and Paulo Silva. **Dependable Grid Services: A case Study with OGSA-DAI**. In Proceedings of the First CoreGrid Symposium, Rennes, France, 2007.

- Also published on Springer Book "Towards Next Generation Grids", pp. 292-300, ISBN: 978-0-387-72498-0, 2007.

1.5.2 A framework for software aging prediction based on Machine Learning

The second contribution of this thesis is the detailed evaluation of a set of Machine Learning algorithms to predict the time to crash of a system based on easy collected

system and QoS metrics at runtime. In this work, we understand the concept of the time to crash as the period of time the system could run acceptably before crash.

Our first contribution was based on offering a self-healing proactive solution for Internet Services based on simple metric thresholds. However, this approach had some weaknesses: we need to know *a priori* the metric related with the software aging to fix the threshold value, we need an expert to determine the value of the threshold and even, the threshold could result in many false positives. For example, if the system reaches the threshold, we cannot know if there is a trend to continue going up until crash or the resource consumption becomes stable after that.

Machine Learning (ML) can help us to cope with these weaknesses. ML lets the system learn from past executions and decide by itself which metrics are relevant or not and what values define the crash. So, we need some prediction method to make more effective our proactive rejuvenation action presented in our first contribution, resulting in a predictive rejuvenation mechanism. ML can help us to build this predictive approach.

The work performed in this area has resulted in the following publications:

[1] Javier Alonso, Josep Ll. Berral, Ricard Gavaldà and Jordi Torres. **Adaptive on-line software aging prediction based on Machine Learning**. In Proceedings of The 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2010), 2010. (Student Travel Grant Award)

[5] Javier Alonso, Ricard Gavaldà and Jordi Torres. **Predicting web server crashes: A case study in comparing prediction algorithms**. In Proceedings of The Fifth International Conference on Autonomic and Autonomous Systems (ICAS2009), 2009.

[2] Javier Alonso, Josep Ll. Berral, Ricard Gavaldà and Jordi Torres. **J2EE Instrumentation for software aging root cause application component determination with AspectJ**. On Proceedings of the 15th IEEE Workshop on Dependable Parallel, Distributed and Network-Centric Systems (DPDNS2010), 2010. Held in conjunction with the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS2010), 2010.

[6, 7] Javier Alonso, Jordi Torres, Rean Griffith, Gail Kaiser and Luis Silva. **Towards Self-Adaptable monitoring framework for Self-healing**. In Proceedings of the 3rd CoreGrid Workshop on Middleware, 2008.

- Also published on Springer Book "Grid and Services Evolution". CoreGrid Series

Vol n.11. ISBN:978-0-387-85965-1, 2009.

1.5.3 Autonomic High Availability and Resource Usage Optimization

Our final contribution is a whole framework ready to manage by itself (without or with minimum human intervention) virtualized platforms composed by several physical nodes over them, running tens, hundreds or thousands of services within virtual machines. This framework uses the previous contributions to offer a transparent high availability against software aging to the services deployed, guaranteeing the optimization of resources available in the environment as much as possible. We understand the optimization of resources in this work such as accepting as much services as possible, using all available resources to improve the potential revenue of the provider (the owner of the infrastructure), offering an extra value to his/her customers, guaranteeing the availability of their services without any work from their (customers) part: a transparent service.

We have presented a model with two main approaches or policies to manage the framework and we have evaluated the approaches to determine which one could be the best.

The work performed in this area has resulted in the following publication:

[3] Javier Alonso, Iñigo Goiri, Jordi Guitart, Ricard Gavaldà and Jordi Torres. **High Availability on Virtualized Platforms with Minimal Physical Resource Impact**. Submitted to Revision.

1.6 Thesis Organization

The rest of this document is composed as follows: Chapter 2 presents the first contribution of this thesis, the proactive software rejuvenation technique developed. After that, Chapter 3 describes the evaluation of Machine Learning techniques in several complex software aging scenarios. Chapter 4 presents our framework to manage virtualized platforms. Finally, Chapter 5 presents the conclusions extracted from this work and depicts the future research lines to continue it.

Chapter 2

Proactive Software Rejuvenation framework

2.1 Introduction

One of the possible definitions of availability is *the quality of being at hand when needed*[1]. When we talk about the computer systems, we understand the quality attribute of systems to be up. In the current world where everything has to be measured and compared, we need some means to evaluate the availability because currently, companies sign agreements based on these type of metrics. We need some measure to quantify the level of availability, or using other words, how much time the system is available. Furthermore, we need some way to compare high availability solutions, to decide which is better. We need some metric to compare and evaluate availability achieved by two systems that can carry out the same task. Currently, academia and industry have widely accepted the following formula [81] as the standard to calculate the percentage of availability of a system:

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

Where $MTTF$ is the Mean Time To Failure and $MTTR$ is Mean Time To Recovery. Following this formula, there are two main approaches to improve the availability of a system: Extending the MTTF or reducing the MTTR. The system components will fail in time; it is difficult to influence how often they fail. The unique way to improve the MTTF is developing components without faults or developing better debugging and testing tools. Although more sophisticated developing/testing tools are appearing to help developers avoid software faults [96, 83], faults still appear and have an important impact on the application availability. In fact, fixing all faults during the testing and debugging phase is a titanic task with an unaffordable cost, because these tools often cannot access third-party modules. Moreover, even using a good development process, it is hard to reduce the number of bugs below 0.1 bugs per 1000 lines of code [62]. So, even the best development and testing processes do not avoid faults completely in deployment phases.

On the other hand, as it was well explained in [49], it is easier and definitely more valuable to approach the goal of 100% availability by reducing the MTTR instead of making efforts to increase the MTTF of systems. If we cut down the MTTR, we can mitigate the impact of an outage to the end-user. This is of utmost importance in the internet world.

Focusing on the goal of decreasing the MTTR as much as possible, we propose a proactive rejuvenation mechanism that can be applied in off-the-shelf application servers without re-engineering the applications or the middleware. Such as rejuvenation technique should minimize the MTTR of the applications. Moreover, we need some

[1]Based on the dictionary of Princeton University: http://wordnet.princeton.edu/ [consulted May, 2010]

autonomic mechanism to be more agile and quick to apply the solution so as to reduce the MTTR as much as possible. The main idea is to offer a proactive and automatic software rejuvenation mechanism to avoid the unplanned outages due to the software aging.

We propose a proactive rejuvenation mechanism instead of a time-based rejuvenation mechanism because we understand that the rejuvenation process has an impact on the performance of the services. For this reason, we have to reduce as much as possible the number of rejuvenation cycles. If we believe that the rejuvenation process has zero impact, then the time-based rejuvenation process based on replicas (like our approach) could be enough. Although, the time-based techniques are not efficient if the failure occurs between two rejuvenation processes.

In this context, we looked into the current state of the art in terms of virtualizaton technology [47, 104] and we realized that it can be a good recipe to optimize the software rejuvenation process. Encouraged by this concept, we have done a prototype implementation of our VM-based rejuvenation approach followed by an experimental study, achieving promising results.

2.2 Rationale for a new scheme of Software rejuvenation

We have defined a set of guidelines that characterize our software rejuvenation mechanism.

1. **Our rejuvenation mechanism should be easily applied in off-the-shelf application servers without re-engineering the applications or the middleware**. A solution based on a new application server architecture or a solution based on re-engineering of the currently deployed and running applications, in our opinion, becomes useless because it will be not used at all, because the organizations would have to invest an important part of their budget to move their applications and application servers to the new design. For this reason, our first guideline is to offer an approach to be used on off-the-shelf application servers, without re-engineering the applications. It becomes useful in a wide range of current scenarios.

2. **The mechanism should provide a very fast recovery to cut down the MTTR to the minimum; if possible we should achieve a zero downtime even in case of restart**. To achieve that hard goal, we need to offer a highly efficient and fast recovery solution. The best way to reduce the MTTR is to advance the recovery action over time. So, we have decided to apply planned restarts based on the status

of the system recognizing the software aging phenomena, i.e., a proactive software rejuvenation.

3. **The mechanism should not lose any in-flight request or session data at the rejuvenation time; the end-user or the interacting application should see no impact at all when there is a server restart**. This is a very important goal. If during the rejuvenation phase the system loses (new or on-going) requests, then some of the end-users will see the impact of the rejuvenation action, which will undermine their confidence level in the application and consequently in the company behind it. Furthermore, it may even cause some data inconsistency in the applications. The on-going requests could cause data inconsistency or misbehavior from the client perspective if the system fails during the execution of their requests. For example: We want to buy a flight ticket to go to enjoy our deserved Christmas holidays. However, when we submit our credit card information to pay for the tickets, the system/application can experience an error or short outage. The end-user does not know if the purchase was successful or not and hence, becomes exasperated and frustrated.

4. **The software infrastructure should automate the rejuvenation scheme in order to achieve a self-healing system**. The solution has to work autonomously. We want to offer a self-healing solution. The rejuvenation mechanism has to decide by itself when to apply the rejuvenation action. This approach is quite interesting because we avoid potential human errors and at the same time we increase the autonomy of the infrastructure to be available as much time as possible. This point could be discussed because it is true that several administrators prefer advices more than automatic mechanisms. However, in order to reduce the complexity of the system managements and reduce the possibility of another source of operator errors (one of the main sources of unplanned outages), our approach becomes as automatic as possible.

5. **The mechanism should not introduce a visible overhead during the runtime phase**. Our solution has to offer an increase of the availability of the application without a noticeable performance impact. If a solution that increases the availability of the application, but at the same time introduces a significant penalty on the performance, the solution could become useless because it will reduce the number of end-users that the application could serve per second.

6. **The scheme should not require any additional hardware: it should work well in a single-server as well as in a cluster configuration**. A clear recipe to avoid

downtime when there is a server restart is to use a hardware-supported cluster configuration: when one server is restarted there is always a backup server that assures the service. However, even in these cluster configurations the server restart has to be carefully done to avoid losing the work-in-progress in that server. Using a cluster for all the application servers in an enterprise represents a huge increase in terms of budget: in [81] it is argued that the adoption of a cluster may represent an increase of 4 of 10 times more in the budget, when compared with a single-server configuration. If we increase the number of server machines and load-balancers we also increase the cost of management and hence the TCO (Total Cost of Ownership). In order to achieve a small MTTR with small cost (money and management) during the recovery action even in a single-server configuration we decided to exploit the use of virtualization. With this technique we are able to optimize the rejuvenation process without requiring any additional hardware. Obviously, it can either work on cluster configuration or single-server applications.

7. **The mechanism should be easy to deploy and maintain in complex IT systems**. Our rejuvenation mechanism is totally supported by software, and can be easily deployed in existing IT infrastructures. Furthermore, the use of virtualization technology helps to reduce the complexity of the deployment of our solution.

Following all these guidelines seems to be very demanding but still achievable. If the application of our rejuvenation scheme could be automated then we will have made a some contribution towards the development of a self-healing system. This is one of our ultimate goals.

2.2.1 How to achieve our goals

The first step in our approach is to achieve a very effective software aging detection mechanism, in order to apply the recovery technique in advance over time (proactivity). This is achieved by using external surveillance of QoS metrics and internal monitoring of system parameters. The surveillance of external QoS metrics has proved to be relevant in the analysis of webservers and services [84] and it has been highly effective in the detection of software aging and fail-stutter failures [106]. This is tightly combined with the monitoring of system internal metrics. These monitoring probes should be timely in their reaction and able to detect potential anomalies, before the occurrence of a system failure. Even if this detection would lead to some false-alarms it would be important to avoid the occurrence of unplanned crashes.

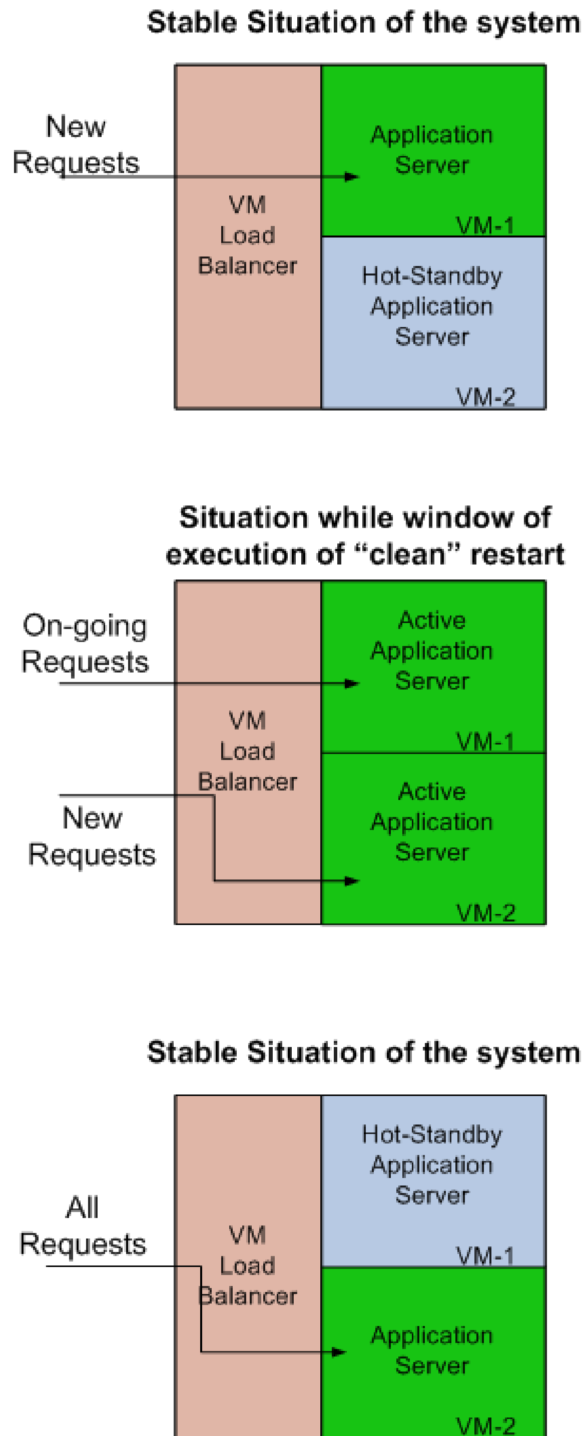


Figure 2.1 Virtualized Clustering Rejuvenation Process

When a potential anomaly is detected we should apply an automatic recovery action. In most of the cases we apply a rejuvenation action. To avoid losing on-going and new requests when there is a planned restart or rejuvenation we make use of a service backup that has been running in the same server machine of the primary service. When the framework triggers a rejuvenation action, we do not restart the main server right away. First, we start the standby service, all the new requests and sessions are sent by the framework to the service backup. The session state, if there is, is migrated from the primary service to the secondary service. The framework monitors the primary service and waits for all the on going request to be finished. When we are able to do this we can restart the main service without losing any in-flight request or session state. So, the old-primary service becomes the new secondary service, waiting to be used in a future new rejuvenation cycle. We will explain more in detail which components of the framework participate and how they contribute to this process. Figure 2.1 shows that process we call it "*clean restart*".

To allow the execution of two replicas of the same service in the same machine we will make use of a virtualization layer and we will launch a minimum of three virtual machines per physical server machine. In this way we are not increasing the cost of the infrastructure since we are exploiting the concept of virtualization and server consolidation [47, 48].

Since in our technique we inherit some of the concepts of server clustering together with virtualization we decided to name this approach by *Virtualized Clustering Rejuvenation (VM-ClusteringRejuv)*.

The use of virtualization in every server that is running a service may be seen as a source of performance degradation since we are introducing a new abstraction layer. However, the results obtained with our proposal are promising and clearly show that the use of virtualization is positive.

2.3 Virtualized Clustering Rejuvenation Framework Description

In this section we present our proposal framework to offer high availability and a huge type of services, mainly focusing on web services, web applications and Grid applications. We target our rejuvenation mechanism to any application server. It can be Websphere, Bea Weblogic, JBoss, Tomcat, Microsoft .Net or others. The clients communicate with the server through TCP-IP, HTTP or SOAP. All the persistent state of the application is maintained in a database that should be made reliable by some RAID scheme. Exception is made to some important state of the application that is maintained in session-objects.

This state is important for the end-users and cannot be lost in a restart application. The session-objects maintain the browsing information of the end-user and they allow the application to know in every moment the current state of the end-user. We do not require any restructuring of the applications nor any change of the middleware container. The scheme should work seamlessly with any server in the market. As explained, our software solution for offering high availability assumes the adoption of a virtualization layer on the machine that provides the service. In our prototype and experiments we have used XEN[126], although it is also possible to use other virtualization middleware like VMWare [118], Virtuoso[117], among others.

Our solution requires the creation of three virtual machines: one virtual machine (*VM1*) with our own load balancer module ; a second virtual machine (*VM2*) where the main service runs; and a third virtual machine (*VM3*)where we have a replica of the service which works as a hot standby service. Figure 2.2 represents the conceptual architecture of our approach. We note that our approach is focused on offering software rejuvenation to the Internet services. So, we are not considering the fact (possible) that the aging phenomena was on the virtualization middleware or even on the guest operating system installed on the virtual machine.

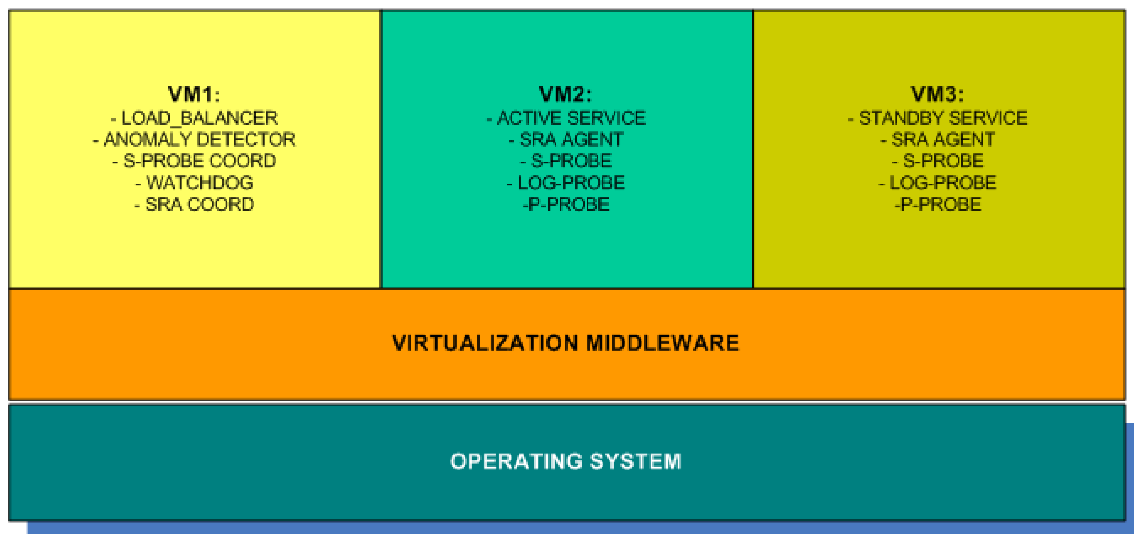


Figure 2.2 Virtualized Clustering Rejuvenation Architecture

The *VM-ClusteringRejuv* is divided in three main components: the *Monitoring Infrastructure*, the *Aging/Anomaly Detector* and the *Rejuvenation Infrastructure*. All these components will be described below. Briefly, as their names say, the main task of *Monitoring Infrastructure* is monitor the QoS metrics of the service and at the same time the system metrics of the service from the operating system; The *Aging/Anomaly Detector* has to be able to detect aging process based on the monitored metrics, and even detect

anomalies. The anomaly detection is a complementary capability to be able to, basically, detect application-level anomalies, protocol errors, log errors and threshold violations of the system parameters. Finally, the *Rejuvenation Infrastructure* is the responsible of applying a clean rejuvenation action guaranteeing the guidelines described before.

2.3.1 Monitoring Infrastructure

The *Monitoring Infrastructure* is composed of the *S-Probe Coordinator* (S-Probe Coord), the *watchdog*, the *S-Probe*, *LOG-Probe* and the *P-Probe*.

The *P-Probe* is a small proxy installed in front of the application server, it filters some error conditions and collects some fine-grain performance metrics, like throughput and latency of the service. This *P-Probe* is able to distinguish latency variations per service-component that is externally accessible by the end-users. As in our experiments we have used as application server the Apache Tomcat Server [16], a Java-based container, we used the *servlet-filter* technology [46] to implement this functionality.

The *Servlet-filter* technology allows us to put a non-intrusive filter per application or even service component deployed on the container (Tomcat in our case). When a filter is installed in front of a service, the container redirects every request to the filter before it is redirected to the application itself. At the same time, when the request is answered, the response message is sent through the filter. Using this technology, the filter is able to calculate the latency of a request, the performance of the service and even the filter is able to check the content of the request and response. This option allows us to check if there is an application anomaly not visible at system metrics level or in application logs like HTTP errors. This technology is allowed in any Java-based application server. Although, the filter technology is not exclusive in Java-Based containers. Currently, most all the containers allow the use of this technology, implying that our solution could be used in every commercial or open source container.

The *LOG-Probe* is a simple software module that performs some anomaly analysis at the logs of the service. Basically, the *LOG-Probe* conducts a search in the log files to find some error or dangerous warning messages.

The *S-Probe* is responsible on monitoring several system metrics like CPU, memory usage, application memory usage, swap space, disk usage, number of threads, I/O traffic, connections to the database, among other system parameters. The *S-Probe* is based on Nagios Monitoring tool [89]. In particular, we have used some Nagios Addons [89] to collect the system metrics easily. We have had to develop our own Addons for Nagios to collect some system metrics that are not allowed in standard Nagios distribution or Addons that are available.

The *Monitoring Infrastructure* design requires that we have to install all of these modules (*P-Probe*, *LOG-Probe* and *S-Probe*) in every Virtual Machine where a service is deployed. In our case in *VM2* and *VM3*.

All metrics and monitoring data collected by the monitoring modules installed in *VM2* and *VM3* are sent to the *S-Probe Coordinator*. The *S-Probe Coordinator* is the core of the *Monitoring Infrastructure*. The *S-Probe Coordinator* receives the data from every Probe (S-Probe, P-Probe and LOG-Probe) installed in the Virtual Machines and the Coordinator stores all this information to be used by the *Aging/Anomaly Detector*.

Finally, we have one last module: the *watchdog*. The *watchdog* has the responsibility to check if a service is alive or not. By simply, pinging to the server. In our current version, we have used the *ldirectord* tool [76]. *Ldirectord* is a perl-based module that executes HTTP probing to some static HTML file and it is used to determine potential outages. However, the *ldirectord* is not able to differentiate between an outage and an overloaded service. We have enhanced this tool in the time-out policies to avoid the occurrence of false-alarms.

2.3.2 Aging and Anomaly Detector

The *Monitoring Infrastructure* provides the data feed for the *Aging and Anomaly Detector* module. This module uses some detection techniques based on data collected by different sensors:

- System-level metrics collected by the *S-Probes* installed.
- Communication protocol errors like HTTP errors, TCP-IP errors or communication timeouts that are collected by the *watchdog* and the *P-Probes*.
- Error codes and anomaly detection in logs performed by the *LOG-Probes*.
- Application specific sensors developed to detect HTML errors in message responses conducted by the *P-Probes*.
- The Performability metrics like the throughput and latency as well as the fine-grain latency per service-component, collected by the *P-Probes*.

Figure 2.3 presents the framework components communication to help the reader understand the rejuvenation framework components and their behavior, which we will present below. The green components were fully developed by us and the red (with lines) components are based on pre-existing solutions and modified by us to adapt them to our requirements.

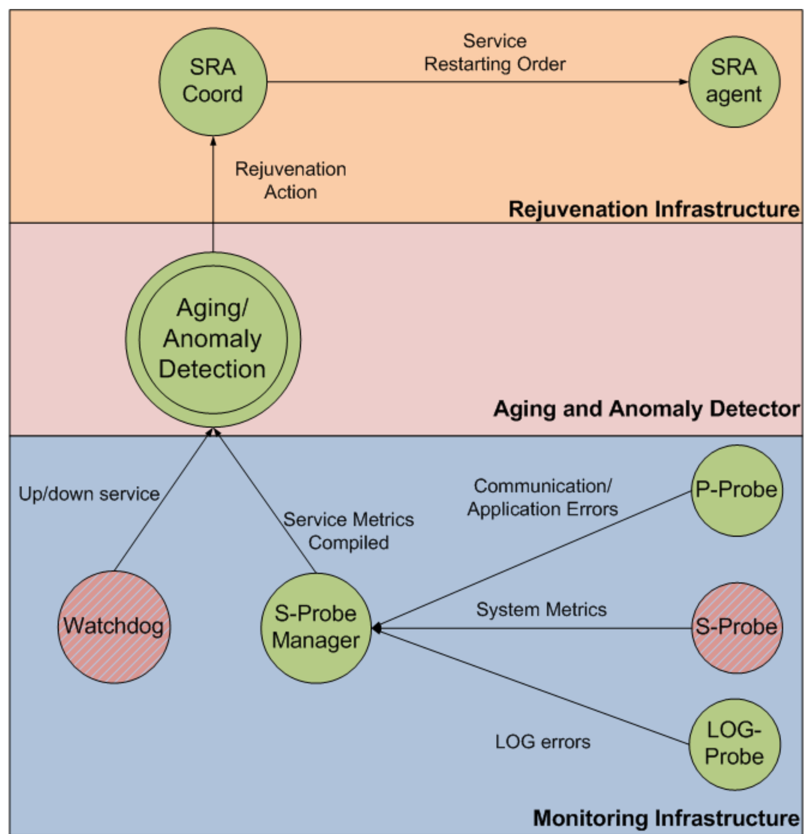


Figure 2.3 Virtualized Clustering Rejuvenation Components Behavior

The *Aging/Anomaly Detector* is the core of our system. To evaluate the effectiveness of our "clean restart" mechanism to apply the rejuvenation process, the *Aging/Anomaly detector* was based on simple threshold techniques based on the conditions observed on the target servers. The surveillance of external QoS metrics has proved relevant in the analysis of web-servers and services [84] and it has been highly effective in the detection of software aging phenomenon [17] in a previous study that was presented in [106]. To evaluate the effectiveness of the rejuvenation technique we have used similar approach as in these papers. It is important to remark the importance of this module. The effectiveness of this module determines the level of proactivity of our proposal. If the Aging/Anomaly Detector fails to detect the onset of aging the system will crash, something that we want to avoid. At this level, we propose a threshold-based approach which collects QoS metrics and triggers the software rejuvenation action when the threshold is violated. In this way, if the threshold is good, we can reduce the MTTR as much as possible, even possibly zero.

2.3.3 Rejuvenation Infrastructure

The *rejuvenation Infrastructure* is composed of the *Load Balancer*, the *Software Rejuvenation Action Coordinator (SRA-Coord)* and the *Software Rejuvenation Action Agents (SRA-Agents)*. The *Load Balancer* and the *SRA-Coord* are installed in *VM1* and the *SRA-Agents* are installed in *VM2* and *VM3*. The *SRA-Coord* works in coordination with the *SRA-Agents* installed in every virtual machine. When this module starts a planned restart to clear up the potential anomaly of the system the *Load Balancer* should apply a careful migration mechanism between the primary and the backup service, as we described earlier in Figure 2.1. This process is implemented by the up-front *Load Balancer* that assures a transparent migration of the client applications that are accessing that service. This operation should also preserve data consistency and assure that no in-flight request should be lost during the migration phase. For this reason we have a window of execution where we have both servers running in active mode: the hot-standby service is made active and will receive the new requests while the old service should finalize the requests that are still on-going. This process does not lose any in-flight requests and we assure the data consistency.

To implement the *Load Balancer* we used the *Linux Virtual Server* (LVS) [79]. LVS is a transport layer load balancer for Linux environments. LVS allows us to know if there are in-flight requests in a server. Using this information, the *SRA-Coord* can know when the two service running window has finished and apply the service restart process. Furthermore, LVS uses routing tables to redirect the requests to the right server. The *SRA-Coord* is ready to modify the tables to activate the migration of new requests to the

secondary service.

When the *SRA-Coord* determines that the aging service has to be restarted to become the new hotstandby service, it sends the order to the *SRA-Agent* installed in the same virtual machine. The *SRA-Agent* has the task to restart the service.

The *SRA-Agent* is an unique component of our infrastructure service dependent. The *SRA-Agent* has to be able to stop and start the service on demand of the *SRA-Coord*. However, the deployment of our framework does not require any change to the service or the middleware containers at all. The solution is also neutral to the virtualization layer. In fact, our approach could be used even with physical servers instead of virtual machines. However, we decided to use virtualization technology to offer a highly resource efficient solution consolidating the services. Furthermore, a resource-efficient solution becomes an energy efficient solution, leading to a greener computing infrastructure, that is quite important these days.

2.4 Experimental Evaluation

In our experiments we have evaluated the effectiveness of our rejuvenation framework to apply the "clean restart" accomplishing the seven guidelines defined. Furthermore, we have evaluated our approach with three different environments to show how the rejuvenation framework could be used in different environments without modifying the Internet Services. We have conducted our experiments in three environments: web applications, web services and Grid services.

2.4.1 Experimental Environment

To study the behavior of our rejuvenation scheme we have used three different application benchmarks: TPC-W [112] for web applications, Axis v1.3 [14] for web services and OGSA-DAI middleware [92] for Grid Services. We have chosen these three environments because all three applications represent the typical environments where software aging phenomena can appear: complex environments and long-running applications.

2.4.1.1 TPC-W benchmarking

TPC-W benchmark is a multi-tier e-commerce site that simulates an on-line book store. We have used the version developed on Java servlets and MySQL [88] as a Database server, and as application server we have used Apache Tomcat [16]. TPC-W allows us to

run different experiments using different parameters and under a controlled environment. TPC-W clients, called Emulated Browsers (EBs), access the web site (simulating an on-line book store) in sessions. A session is a sequence of logically connected requests (from the EB point of view). Between two consecutive requests from the same EB, TPC-W undergoes a thinking phase, representing the time between the user receiving a web page s/he requested and generating the next request. Moreover, following the TPC-W specification, the number of concurrent EBs is kept constant during the experiment.

TPC-W benchmark does not suffer from any software aging problem, at least not observed/reported yet. For this reason, we decided to modify TPC-W to simulate the aging-related errors consuming resources until their exhaustion. We have implemented a small fault-injector that works as a resource parasite: it consumes system resources in competition with the application. Although, the fault-injector supports several resources like CPU, memory, disk or threads, we have only used the memory consumption option with an aggressive configuration to speed-up the effects of (synthetic) aging in our experiments. This parasite consumes memory during time periods, injecting memory leaks on the system. The fault injection tool is similar to the one described in [58]. In our experiments we have used an aggressive and constant configuration of memory leak rate to speed-up the effects of the aging. We inject a memory leak per request.

2.4.1.2 Webservice Container Axis v1.3

To evaluate our approach in Service Oriented Architecture (SOA) environments, we have selected Tomcat/Axis server. We have chosen Tomcat/Axis since we already knew from a previously study published [106] that Axis v1.3 is suffering from memory leaks, and then by software aging phenomena. We have implemented a simple shopping store with a database back-end, using MySQL. The client application may search for products, add them to a shopping cart and run for checkout. This application is accessed through SOAP communication protocol. The Axis v1.3 software aging phenomena shows a slow but constant performance degradation depending on the workload level.

2.4.1.3 OGSA-DAI middleware

To evaluate our *VM-ClusteringRejuv* framework we selected as benchmark Grid Services the OGSA-DAI middleware [92]. OGSA-DAI is a package that allows remote access to data resources (files, relational and XML databases) through standard front-end based on Web services. The software includes a collection of components for querying, transforming and delivering data in different ways, and a simple toolkit for

developing client applications. OGSA-DAI provides a way for users to Grid-enable their data resources.

The front end of OGSA-DAI is a set of Web services that in the case of WSI requires a SOAP container to handle the incoming requests and translate them into the internal OGSA-DAI engine. This SOAP container is Tomcat/Axis 1.2.1. As we described earlier, this SOAP container suffers from memory leak that causes software aging. However, OGSA-DAI is used by several Grid projects[93] and a large community, hence this aging is a major concern, from the point of view of availability. The Axis implementation is highly prone to internal memory leaks. When the service developers of OGSA-DAI make use of session-scope in their Grid services those memory leaks result in software aging, with performance degradation and even system crashes.

2.4.2 Experimental Setup

In our experiments we have used a cluster of 12 machines: 10 running the client benchmarks for TPC-W and Tomcat/Axis. However, we only used 3 running the client benchmark application for OGSA-DAI middleware experiments. We used less machines in the OGSA-DAI scenario because the workload supported by OGSA-DAI is lower than TPC-W or Tomcat/Axis. One Database server (for TPC-W and Tomcat/Axis, Katrina or Wilma, and Tania in the case of OGSA-DAI experiments). Our main server (Tania or Nelma in the case of TPC-W and Tomcat/Axis and Katrina for OGSA-DAI experiments) running XEN and the three virtual machines. All machines are interconnected with a 100Mbps Ethernet switch. The detailed description of the machines is presented in Table 2.1. The reason to use different machines in the experiments of TPC-W/Tomcat/Axis and OGSA-DAI is due to the requirements of the applications.

We used XEN version 3.0.2 configured with 3 virtual machines: 2 virtual machines with 700MB memory for the application servers and a third virtual machine with 256MB memory to run the main cores of the *VM-ClusteringRejuv* presented in Figure 2.2.

2.4.3 Experimental Results

We have conducted a set of extensive experiments to evaluate the effectiveness of our approach to reduce the MTTR and achieve the seven guidelines described in Section 2.2.

2.4.3.1 Virtualized Clustering Rejuvenation Framework Overhead

As we described earlier, our solution runs over a virtualization layer. This new abstraction layer added to the stack could increase the system overhead. For this reason,

	Katrina & Wilma	Tania & Nelma	Client Machines
CPU	Dual AMD64 Opteron (2GHz)	Dual Core AMD64 Opteron 165 (1,8GHz)	Intel Celeron (1GHz)
Memory	4GB	2GB	512MB
Hard Disk	160GB(SATA2)	160GB(SATA2)	
Swap Space	8GB	4GB	1024MB
Operating System	Linux 2.6.16.21-0.25-smp	Linux 2.6.16.21-0.25-smp	Linux 2.6.15-p3-Netbook
Java JDK	1.5.0_06, 64-bits Server VM	1.5.0_06, 64-bits Server VM	1.5.0_06-b05, Standard Edition
Tomcat JVM heap size	512MB or 1024MB[a]	512MB	
Other software	Apache Tomcat 5.5.20, Axis1.3, MySQL 5.0.18 or Axis 1.2.1[a], OGSA-DAI WSI 2.2	Apache Tomcat 5.5.20 or MySQL 5.0.18[a]	

[a] Specifically used in OGSA-DAI experiments

Table 2.1 Detailed experimental Setup Description

our first experiment was focused on estimating the overhead introduced by our solution.

Our first experiment was to measure the performance penalty due to the use of a virtualization layer (in our experiments, XEN) and our virtualized clustering technique.

We executed several short runs of 15 minutes each in a burst workload mode. From these runs we removed the initial 5 minutes, considering it as the warm-up interval. So, the total run length was 10 minutes each. Although, in the case of TPC-W evaluation, we needed to change the code of the TPC-W clients to eliminate the random "Thinking time" between requests. The TPC-W specification includes a random time between 7 and 70 seconds following a exponential distribution. In this run to evaluate the overhead introduced by our proposal, we set the "thinking time" to a fixed value: 7 seconds.

Figures 2.4, 2.5 and 2.6 present the throughput comparison between the throughput of every application benchmark on top of the operating system, on top of XEN virtualization middleware and on top of XEN virtualization middleware with *VM-ClusteringRejuv* full installed in all three scenarios: Tomcat/Axis, TPC-W and OGSA-DAI respectively. As can be seen there is some overhead of using XEN and our *VM-ClusteringRejuv* framework, when compared with a simple run on top of the operating system.

However, a simple eye evaluation is not enough to evaluate the real penalty introduced by the virtualization layer and the framework developed to monitor and rejuvenate the service when it is needed. For this reason, we present a more precise comparison in terms

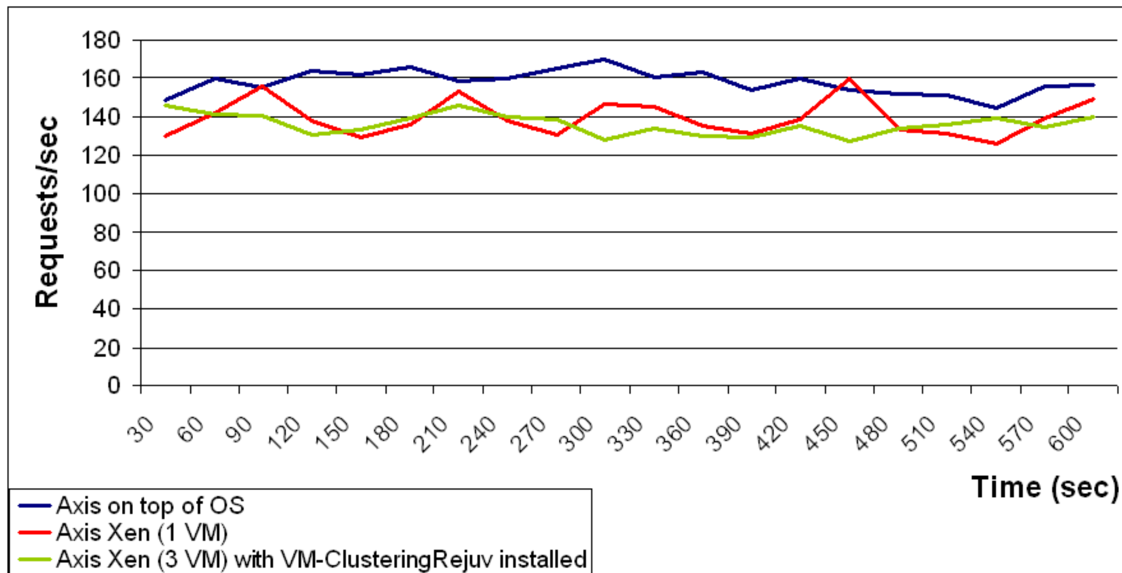


Figure 2.4 Comparing the Throughput of Tomcat with Axis on top of OS, on top of XEN, and on top of XEN with VM-ClusteringRejuv

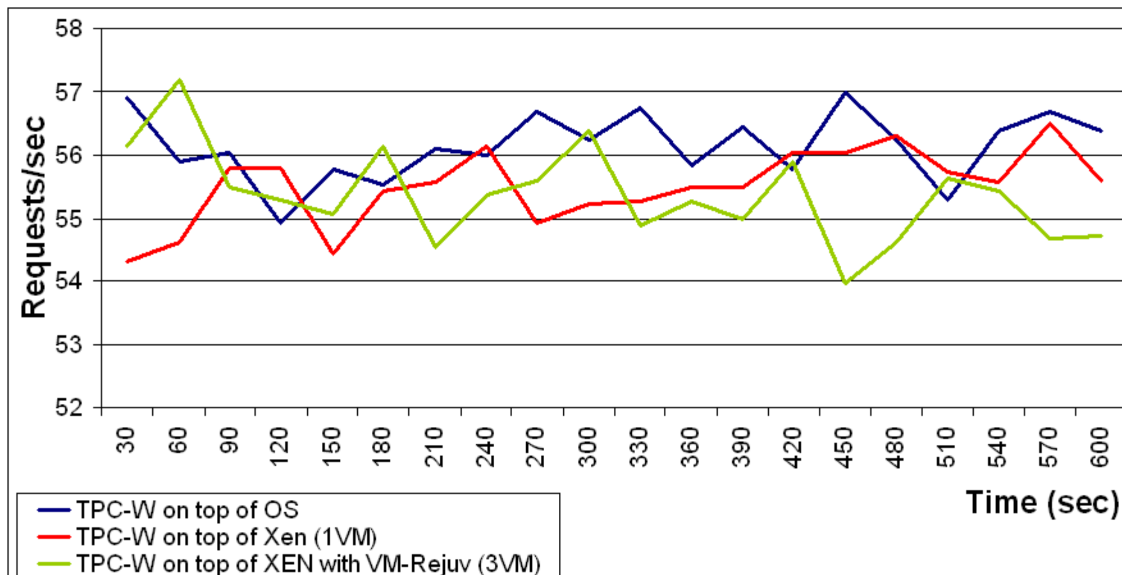


Figure 2.5 Comparing the Throughput of TPC-W on top of OS, on top of XEN, and on top of XEN with VM-ClusteringRejuv

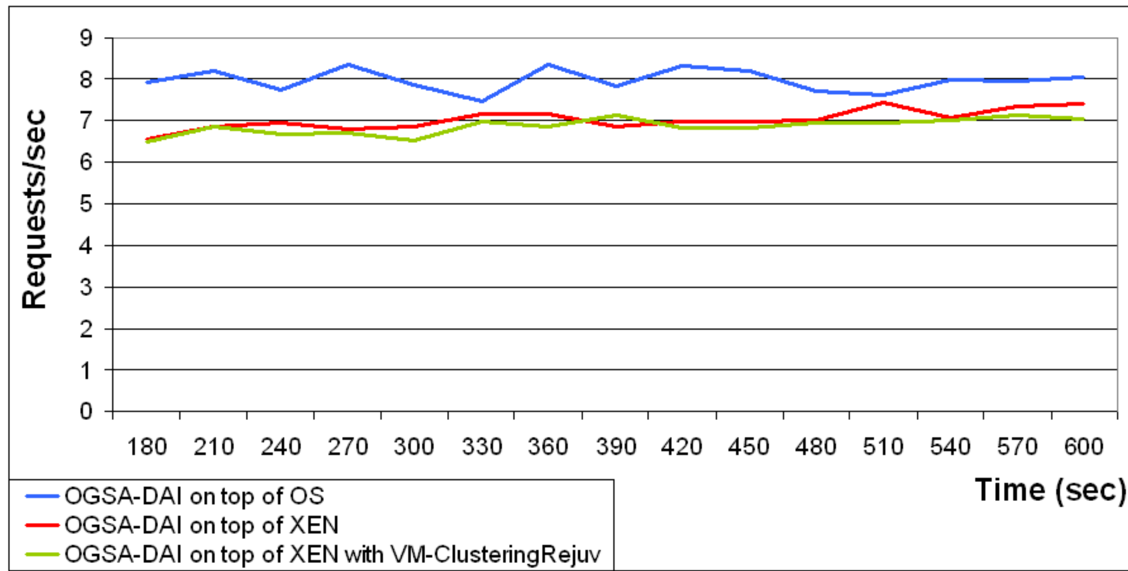


Figure 2.6 Comparing the Throughput of OGSA-DAI on top of OS, on top of XEN, and on top of XEN with VM-ClusteringRejuv

of average throughput and total number of requests. From that data we can measure an overhead in terms of total number of requests served in that time-interval.

	Average Throughput (requests/sec)	Average Throughput (with 95% confidence)	Total number of Requests	Overhead
Axis on top of OS	157.9	$155.08 < \mu < 160.72$	94,743	-
Axis on top of XEN	139.2	$135.11 < \mu < 143.35$	83,541	12%
Axis on top of XEN with VM-ClusteringRejuv	136.1	$133.66 < \mu < 138.50$	81,651	14%

Table 2.2 Comparing the Overhead in Web services Environment using Tomcat/Axis

Table 2.2 shows the results of Tomcat/Axis benchmark. In the case of Tomcat/Axis benchmark we observe that XEN virtualization layer introduces an overhead of 12% while our mechanism adds an additional overhead of 2%. We can observe how it is the virtualization layer the most expensive in terms of performance. However, the penalty is acceptable if the framework allows us to reduce the underutilization of the servers described in the introduction of this document using virtualization and at the same time offers an important improvement of availability of the services as we will show later in this chapter.

The results of TPC-W are rather different. In this run, as we explained earlier, we modified the TPC-W clients to fix the "thinking time" to 7 seconds. Table 2.3 presents

the results obtained. As can be seen, the overhead of using XEN is almost negligible. The reason of this result is because the application clients are not using a continuous burst distribution thanks to the "thinking time", resulting on an insignificant overhead introduced by the virtualization and the framework.

	Average Throughput (requests/sec)	Average Throughput (with 95% confidence)	Total number of Requests	Overhead
TPC-W on top of OS	56.13	$55.89 < \mu < 56.36$	33,682	-
TPC-W on top of XEN	55.51	$55.24 < \mu < 55.77$	33,308	1.1%
TPC-W on top of XEN with VM-ClusteringRejuv	55.36	$55.03 < \mu < 56.68$	33,217	1.3%

Table 2.3 Comparing the Overhead in Web Application Environment using TPC-W

	Average Throughput (requests/sec)	Average Throughput (with 95% confidence)	Total number of Requests	Overhead
OGSA-DAI on top of OS	7.943	$7.75 < \mu < 8.13$	2383	-
OGSA-DAI on top of XEN	7.136	$7.01 < \mu < 7.25$	2141	10.1%
OGSA-DAI on top of XEN with VM-ClusteringRejuv	6.966	$6.89 < \mu < 7.03$	2090	12.3%

Table 2.4 Comparing the Overhead in Grid Environment using OGSA-DAI

Finally, Table 2.4 presents the results obtained in the Grid environment using OGSA-DAI middleware. By comparing the total number of requests we can see that virtualization overhead introduces a 10% of overhead and our solution adds a 2% additional overhead.

All of these results have been achieved with a burst workload distribution or at least the maximum workload bearable. So, they should be seen by the reader as the maximum observable overhead.

2.4.3.2 Average Performance under Virtualization Overhead[2]

One could argue that the virtualization middleware introduces a visible overhead in the performance of Grid and Web services. Due to the introduction of significant overhead we decided to study and quantify the advantages of virtualization. We have developed an

[2] This section was developed thanks to the support of Dr. A. Andrzejak who help us on the mathematical definition

accurate mathematical study to measure the advantages *versus* the disadvantages of using virtualization.

The usage of a virtualization layer obviously decreases application performance, while on the other hand it helps to reduce the Mean Time To Recovery (MTTR) in case of a fault or rejuvenation need. This gives rise to a question: what is more beneficial for the long-term performance: given a large time interval with many rejuvenation cycles, will an application A_{phys} running on a physical server completes more requests than an identical application A_{virt} running in a virtual machine, or other way around?

To treat this problem we introduce the following definitions. Assume that under the full load (burst request distribution) A_{phys} can serve at most R_{phys} requests per second which we call its *instantaneous performance* (and could be called the maximum throughput), and let R_{phys} depend on the time since last rejuvenation. Under the same conditions let $R_{virt} = R_{virt}(t)$ be the number of requests per second served by A_{virt}.

The *average performance* $\overline{R}_{phys}$ is defined and computed by summing up the number of served requests over a large time interval and dividing it by the length of this interval (analogously for $\overline{R}_{virt}$).

The Mean Time To Failure (MTTF) t gives us the expected time until an application must be rejuvenated from the last one, and we assume that it is equal for both A_{phys} and A_{virt}. This assumption, in our opinion makes sense since both are the same application and we assume that the fault which causes the software aging phenomena and finally the failure is independent of the virtualization layer. However, each application has a different MTTR which we designate as $rejuv_{phys}$ and $rejuv_{virt}$, respectively. We introduce the ration $q = \frac{R_{virt}}{R_{phys}}$ called the *performance ratio*, which is less than 1 due to virtualization overhead. Even if the instantaneous performances vary over time, we might assume that q is constant since both servers age similarly.

First, to compute the *average performance* under a burst distribution with a fixed request rate above $Max(R_{phys}, R_{virt})$ it is sufficient to consider just one failure and recovery cycle while assuming that MTTF t is not an expectation but a fixed time.

We obtain them:

$$\overline{R}_{phys} = \frac{\int_0^t R_{phys}(x)dx}{t + rejuv_{phys}} \tag{2.2}$$

$$\overline{R}_{virt} = \frac{\int_0^t R_{virt}(x)dx}{t + rejuv_{virt}} \tag{2.3}$$

The major question to be answered is: what is the minimum level q_{min} of the performance penalty under which the average performance of the virtualized application

A_{virt} is still matching the average performance of A_{phys}?

By setting $\overline{R}_{phys} = \overline{R}_{virt}$ and solving by q, we find out that under above conditions:

$$q_{min} = \frac{t + rejuv_{virt}}{t + rejuv_{phys}} \tag{2.4}$$

These solutions look different if the applications are not running under their respective full load. For example, if the request rate never exceeds R_{virt}, then there is no difference in the throughput between applications. In the later case q_{min} might take any value above 0 and the ratio of the *average performances* depends only on the ratio of the MTTR's $rejuv_{virt}$ and $rejuv_{phys}$. Furthermore, if the requests arrive according to some more complex distribution, e.g. the Poisson distribution, then only time intervals with arrival rates above the momentary levels $R_{virt}(t)$ and $R_{phys}(t)$ contribute to the differences between average performances.

To compute q_{min} in the experimental setting of the OGSA-DAI server and Axis, we have compared the throughput of the non-virtualized case ("On top of OS") *versus* the virtualized case without the rejuvenation framework ("On top of XEN") and with the framework ("On top of XEN with VM-ClusteringRejuv"). To safeguard against the effects of the initialization phase and the aging effects we compared the service rates after the 5 minutes since start for the duration of 10 minutes.

Table 2.5 and Table 2.6 shows the *performance ratio* q for Axis and OGSA-DAI, respectively.

	Total number of Requests	Overhead	Performance ratio q
OGSA-DAI on top of OS	2383	-	-
OGSA-DAI on top of XEN	2141	10.2%	0.8984
OGSA-DAI on top of XEN with VM-ClusteringRejuv	2090	12.3%	0.8771

Table 2.5 Measured virtualization overhead in the time interval 10 minutes with OGSA-DAI

To confront these values with the computed minimal required performance q_{min} to match the average performance in both modes, we use as values of the MTTR parameters $rejuv_{virt} = 0$ and $rejuv_{phys} = 12.5$ seconds. Under the assumption of a variable MTTF t we obtain the results of q_{min} presented in Table 2.7. It shows that only for very short rejuvenation cycles (below 125 seconds) the average performance in the virtualized case does not fall behind the case of A_{phys}. Since the aging effects are noticeable after much longer time (depending on the application from tens of minutes until hours or days), the

	Total number of Requests	Overhead	Performance ratio q
Axis on top of OS	94,743	-	
Axis on top of XEN	83,541	12%	0.881
Axis on top of XEN with VM-ClusteringRejuv	81,651	14%	0.861

Table 2.6 Measured virtualization overhead in the time interval 10 minutes with Tomcat/Axis

$MTTF\,t$	75	100	125	150	500	2000	3000
q_{min}	0.857	0.889	0.909	0.923	0.976	0.994	0.996

Table 2.7 Minimal required performance q_{min} depending on the $MTTF\ t$ for $rejuv_{virt} = 0$ and $rejuv_{phys} = 12.5$ seconds

average performance of A_{virt} is in general worse than the one of A_{phys}.

However, for the cost of 14% (Tomcat/Axis) or 12.3% (OGSA-DAI) average performance - with the worst case only in the burst mode - we completely eliminate outages caused by software aging, which is a fair trade. After this evaluation of the performance penalty versus availability, we can conclude the benefits of using Virtualization to increase the availability of the services in face of the software aging phenomena.

2.4.3.3 Virtualized Clustering Rejuvenation Framework Effectiveness

The next step was to evaluate the effectiveness of our automated rejuvenation mechanism. We used the three application benchmarks (Tomcat/Axis, TPC-W and OGSA-DAI) and we configured the rejuvenation mechanism to be triggered when there is some threshold violation in one of the external QoS metrics. In this case we observe the throughput of the application: if the application starts to get slower over time there is a high probability we are facing a fail-stutter behavior [17]. The fail-stutter behavior could be defined by a system/component that it has not absolutely failed but its performance is less than that of its performance specification or achievable.

Web Services environment

In the case of Axis we already knew that version 1.3 is suffering from severe memory leaks [106]. This memory leak causes Axis to crash in less than 5 hours, using ten simultaneous clients in burst mode. We measured the throughput in time-runs of 4 hours

when using the application and no rejuvenation action at all, comparing the two scenarios where we deployed our rejuvenation framework, triggering the rejuvenation action when there was a violation of the throughput SLA (Service Level Agreement). In these two rejuvenation scenarios we setup the SLA values to 50% and 75%. When the observed service throughput decreased to lower than 50% or 75% of the maximum value, the framework triggers the rejuvenation action. We calculated the maximum value using the average throughput achieved during the next 10 minutes after the starting time fixed in 5 minutes of the execution.

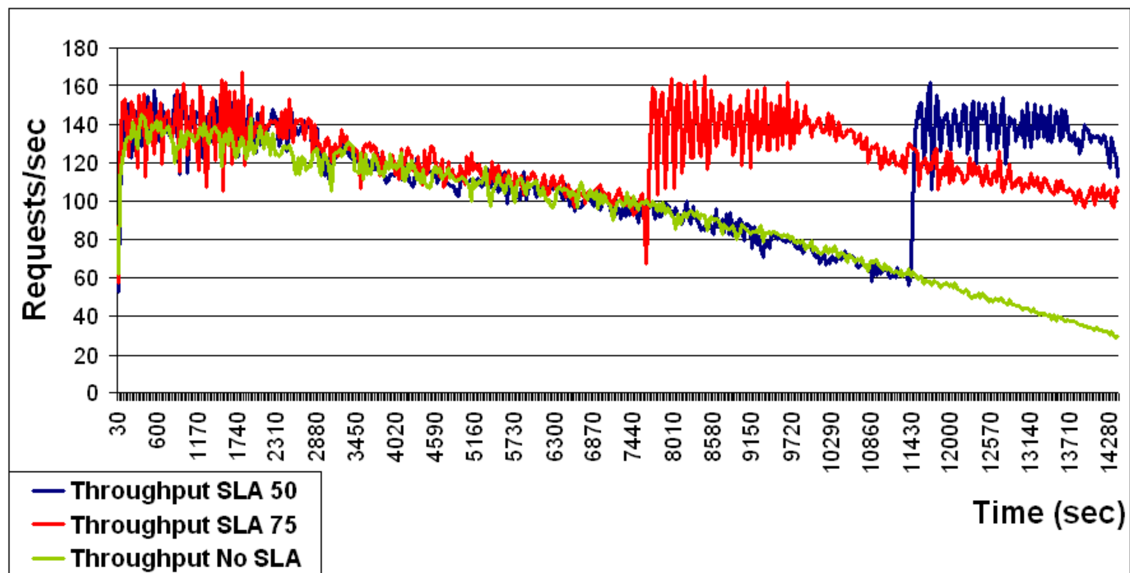


Figure 2.7 Throughput Comparison: VM-ClusteringRejuv evaluation with Tomcat/Axis

Figure 2.7 shows the results obtained in these experiments and the effectiveness of our autonomous rejuvenation system. When the throughput goes down achieving a determined threshold (50% or 75%), the framework is ready to trigger a rejuvenation action, avoiding the future service crash. We observe clearly the rejuvenation action result, making possible that the service recovers its previous maximum performance every time the rejuvenation is triggered like instant 7500 seconds, when the application performance goes down under the 75% of the maximum performance. We can observe how that our approach avoids the possibility that performance achieves zero throughput during the rejuvenation action. At the same time, we can observe how if we do not apply our rejuvenation mechanism the service would crash approximately after 4.5 hours of execution.

However, the throughput is not the unique external metric whose its behavior improves significantly using our framework. Figure 2.8 presents the response time of Tomcat/Axis

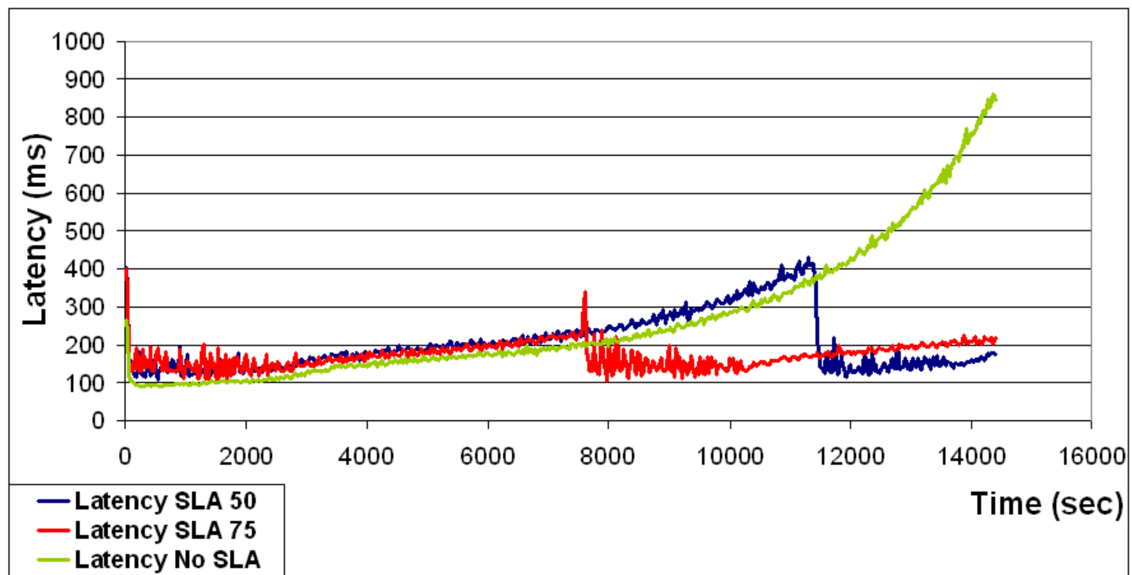


Figure 2.8 Response Time Comparison: VM-ClusteringRejuv evaluation with Tomcat/Axis

service in the same experiment. We observe how the response time goes up as the throughput goes down. We can observe how our mechanism recovers the minimum response time like the previous throughput experiment. However, if we do not apply our mechanism the response time grows until it becomes unacceptably large, before even that service crashes, making the service useless.

Web Applications environment

We conducted the same experiment and the same threshold definition values with TPC-W benchmark application to evaluate the effectiveness of our proposed solution. In this case, as we have presented earlier, we have used a parasitic application to aggressively consume memory to simulate a scenario where TPC-W suffers from a software aging due to a memory leaks. In this experiment we injected a memory leak of 1kb per request: an aggressive leak. Our idea was that the software will start aging and we will observe how our mechanism works in face of an aggressive leak. Figure 2.9 presents how the TPC-W application dies after 5310 seconds of execution due to our parasitic application.

However, our autonomous and proactive software rejuvenation scheme is able to avoid a crash and keep a sustained level of performance, within the throughput SLA defined. In the same way as Tomcat/Axis behavior, the response time of the service also improves thanks to our scheme, shown clearly in Figure 2.10. Figure 2.10 does not show the response time of the service until the crash moment because the degradation level of the

Figure 2.9 Throughput Comparison: VM-ClusteringRejuv evaluation with TPC-W Benchmark

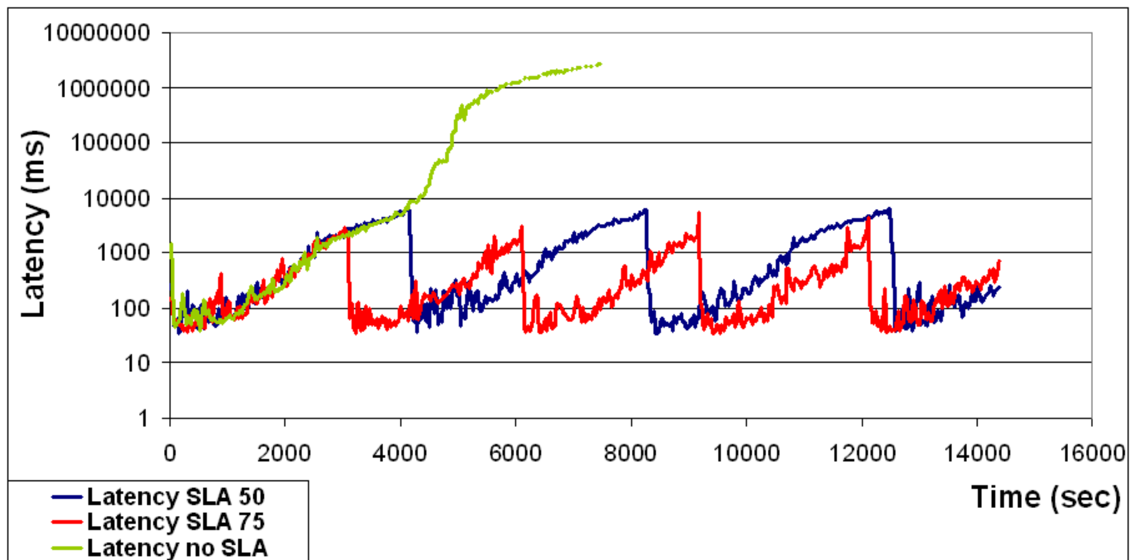


Figure 2.10 Response Time Comparison: VM-ClusteringRejuv evaluation with TPC-W Benchmark

service do not allow to collect it.

Grid Services environment

The case of OGSA-DAI is quite different than the other two previous situations. We started by conducting several experiments to observe the behavior of OGSA-DAI and the underlying system metrics. We concluded that external QoS metrics used in the previous experiments were very unstable and it would be quite difficult to apply some threshold analysis to trigger automatic recovery action. Figures 2.11 and 2.12 show the response time and the throughput, respectively, of OGSA-DAI under a constant burst workload during one hour of execution.

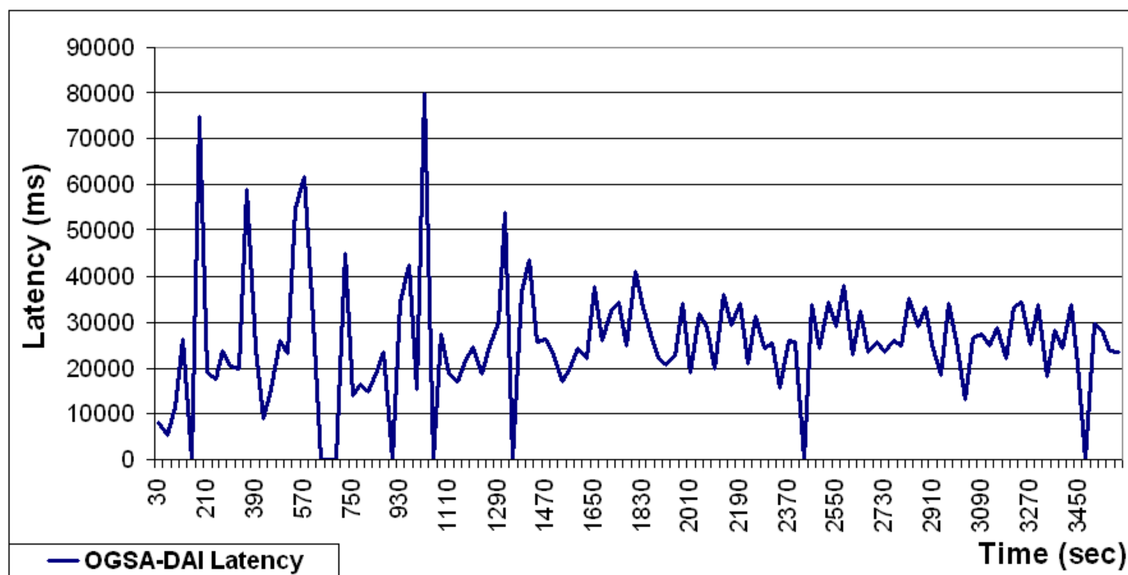


Figure 2.11 OGSA-DAI unstable response time behavior under constant burst workload

Our first thought was that the instability of the external QoS metrics was due an aggressive workload. For this reason, we repeated the experiment of one hour of execution using a lower and constant workload. In Figure 2.13, we present the latency of the OGSA-DAI server when applying different constant workloads of 1 request every 10, 20 or 30 seconds, per client. We observe how even under a constant and low workload the response time of the OGSA-DAI using session scope is completely useless to fix a effective threshold to trigger the rejuvenation action.

After that, we studied in detail the underlying system metrics and we observed that memory usage was increasing with time until the maximum memory usage allowed by the configuration was achieved. When this happens the OGSA-DAI had an unpredictable

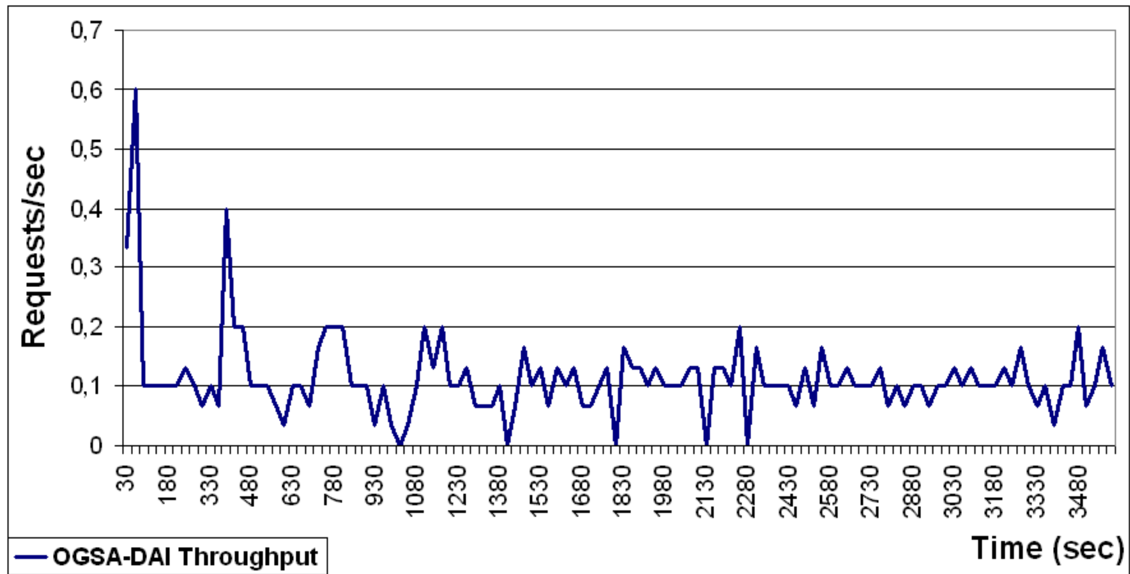


Figure 2.12 OGSA-DAI unstable Throughput behavior under constant burst workload

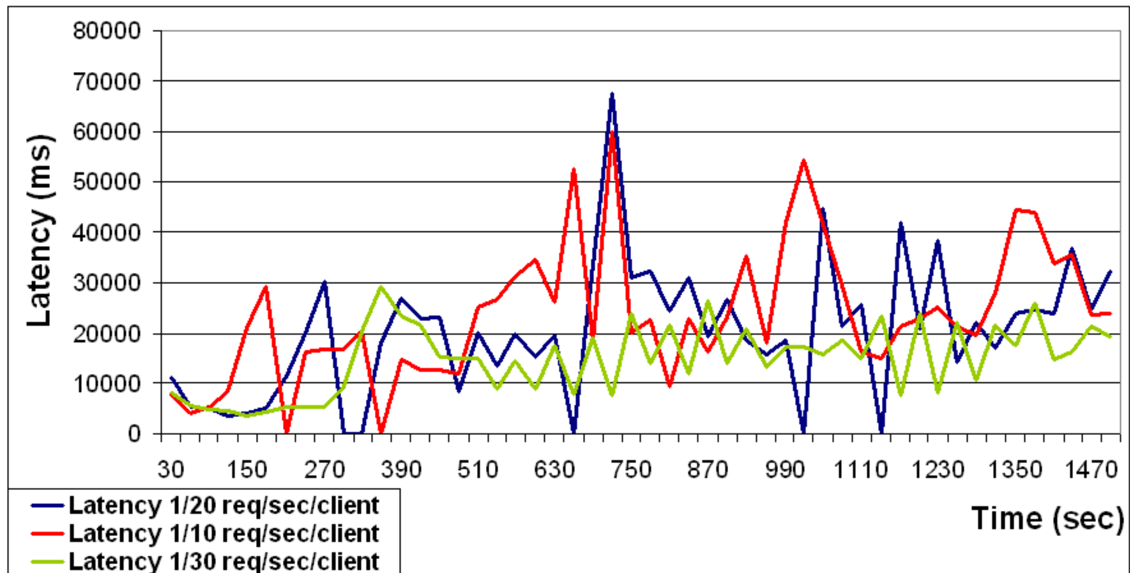


Figure 2.13 OGSA-DAI unstable Response Time behavior under constant workloads

behavior with very unstable performance and sometimes it resulted in system crashes. This behavior of memory consumed by OGSA-DAI is shown in Figure 2.14.

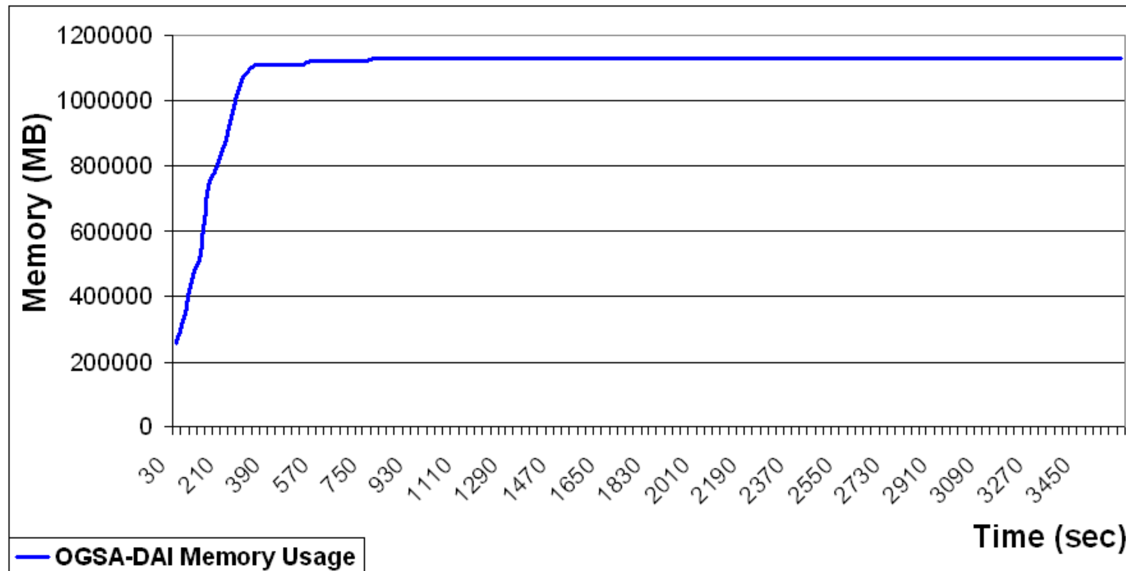


Figure 2.14 OGSA-DAI memory consumption under burst workload

Since we knew the cause of the anomaly was an internal memory leak on Axis layer, we decided to apply a threshold trigger associated with memory usage. Thereby, we have configured our *S-Probe* to obtain the memory usage every 5 seconds and when it would get higher than a certain threshold we would apply a rejuvenation action in the main Grid Service to avoid a possible failure or crash.

We run two configurations: OGSA-DAI without our solution to obtain the reference curves; and then OGSA-DAI with our solution. In this latter case, we have configured the threshold to trigger when memory usage gets to a certain limit: 50% of maximum memory allowed. Both experiments were executed with a burst workload configuration.

Figures 2.15 and 2.16 show the throughput and the latency results, respectively. We can observe that our solution was able to achieve a higher throughput and a lower latency. The reason for that improvement on QoS metrics is because our approach avoids the resource (memory in this case) saturation reducing the misbehavior caused by that fact.

In Figure 2.16 it is clear that the usage of our Virtualized Clustering Rejuvenation approach was able to provide a very stable latency, while in the normal case the OGSA-DAI (with session scope configuration) presented a very unstable latency and in some cases we even registered some missed requests due to the overload memory resource.

The difference in the throughput/latency behavior and values achieved is explained since a normal execution of OGSA-DAI consumes the memory quickly and after that the throughput goes down as we can observe around second 30 in Figure 2.15. Our solution

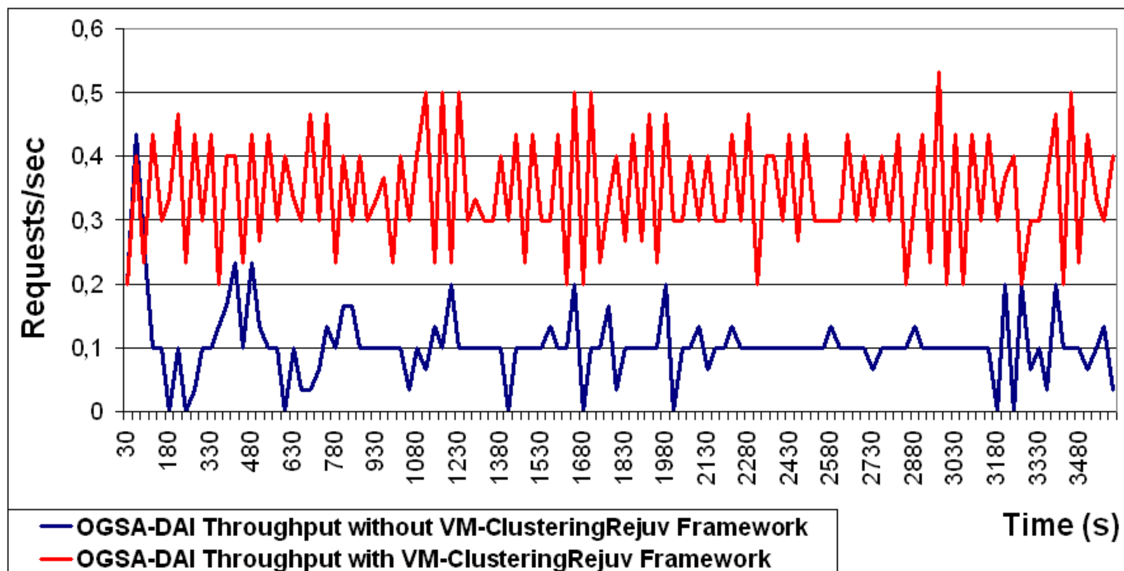


Figure 2.15 Throughput Comparison: VM-ClusteringRejuv evaluation with OGSA-DAI Benchmark

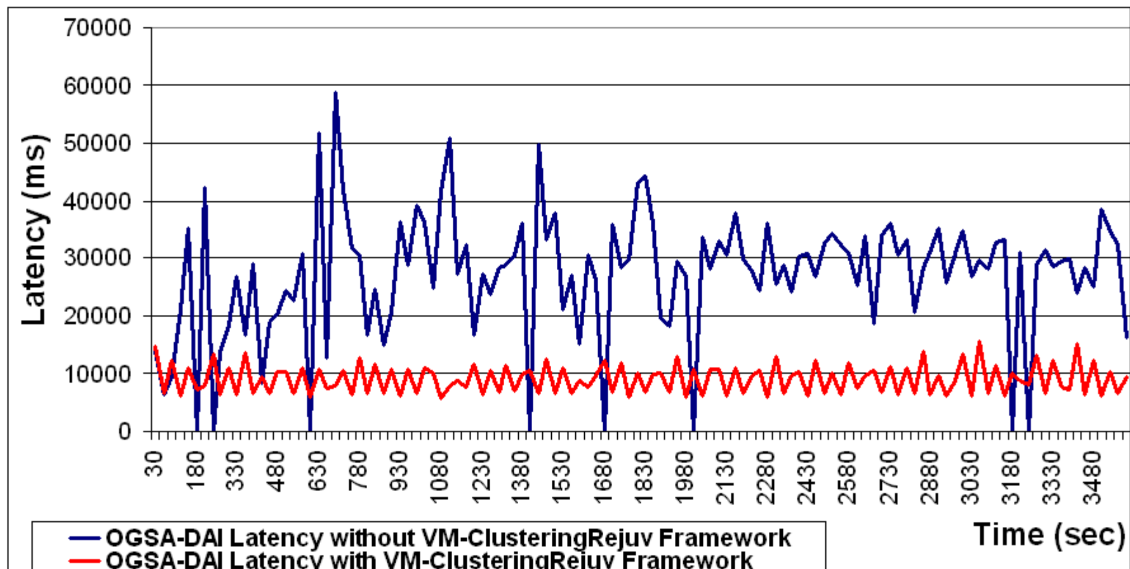


Figure 2.16 Response Time Comparison: VM-ClusteringRejuv evaluation with OGSA-DAI Benchmark

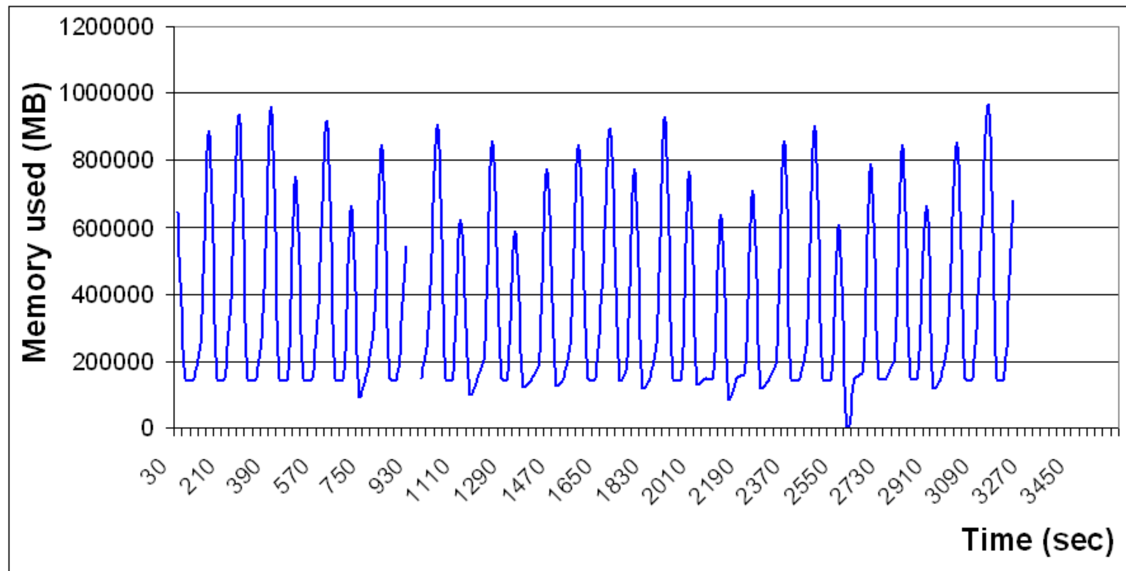


Figure 2.17 OGSA-DAI Benchmark memory usage with VM-ClusteringRejuv Active

avoids that the OGSA-DAI reaches the maximum memory allowed as is presented in Figure 2.17, improving the results of the OGSA-DAI behavior clearly.

	Average Latency	Average Throughput	Average Memory Used	Missed Requests
OGSA-DAI without VM-ClusteringRejuv	29013.2ms	0.105 req/sec	1100784.4 Bytes	25
OGSA-DAI with VM-ClusteringRejuv	9049.5ms	0.345 req/sec	313384 Bytes	0

Table 2.8 Detailed values from 1 hour of Execution with OGSA-DAI

Table 2.8 presents more detailed numbers in this experiment.We can see three main things:

- Our approach was able to avoid any missed request. Without our framework we have observed 25 missed requests during the time-frame of the experiment due to the overloaded memory resource.

- Our approach was able to reduce the average Latency by a factor of 3.

- Our approach was able to improve the average Throughput by a factor of 3.

Even if we have to pay the overhead of virtualization we were able to improve the performance of OGSA-DAI by a considerable factor, just by applying some planned and *clean* restarts to avoid performance degradation or misbehavior.

With all above experiments and results we are able to argue the cost-effectiveness of our solution for autonomic highly-available services in different but representative scenarios.

2.4.3.4 Downtime Achieved by the Virtualized Clustering Rejuvenation Framework

After showing effectiveness of our solution to obtain a highly available and highly-stable services, we wanted to evaluate the potential downtime for the service when a recovery action was executed as well as the number of failed requests and the potential loss of session data during the rejuvenation process. In other words, we want to measure the Mean Time To Recovery (MTTR) achieved by our solution. Because the MTTR represents the downtime perceived by the users during the rejuvenation process.

These results would be of extreme importance, according to guidelines defined earlier in this chapter. The idea of this experiment is to compare four different recovery techniques: our approach, a service restart, a virtual machine restart and a full physical machine restart, in order to analyze the downtime obtained by every approach and evaluate the real impact on the availability of the service of every solution (some of them commonly used in real environments).

We run four different experiments in the same target machine using the Tomcat/Axis and OGSA-DAI application:

- One run where we applied a rejuvenation action at the time of 300 seconds;
- Another run where we triggered a service (OGSA-DAI or Tomcat/Axis) restart at exactly that time.
- A third run, where we applied a restart in the XEN Virtual Machine where the main server was running, at the same time (300 seconds).
- And finally a last one, where we executed a full machine reboot at that execution time.

We decided to use only Tomcat/Axis and OGSA-DAI because both cases provide real software aging scenarios and we wanted to evaluate the behavior of the framework in face of real scenarios.

Web Services environment

In the case of Tomcat/Axis, the results are presented in Figure 2.18. It presents an execution window of 600 seconds. We adopted that figure style to show the client perception about the availability of the service when a "restart" is done. Furthermore, this style was extracted from George Candea work, to evaluate the availability of his micro-rebooting technique, published in [33].

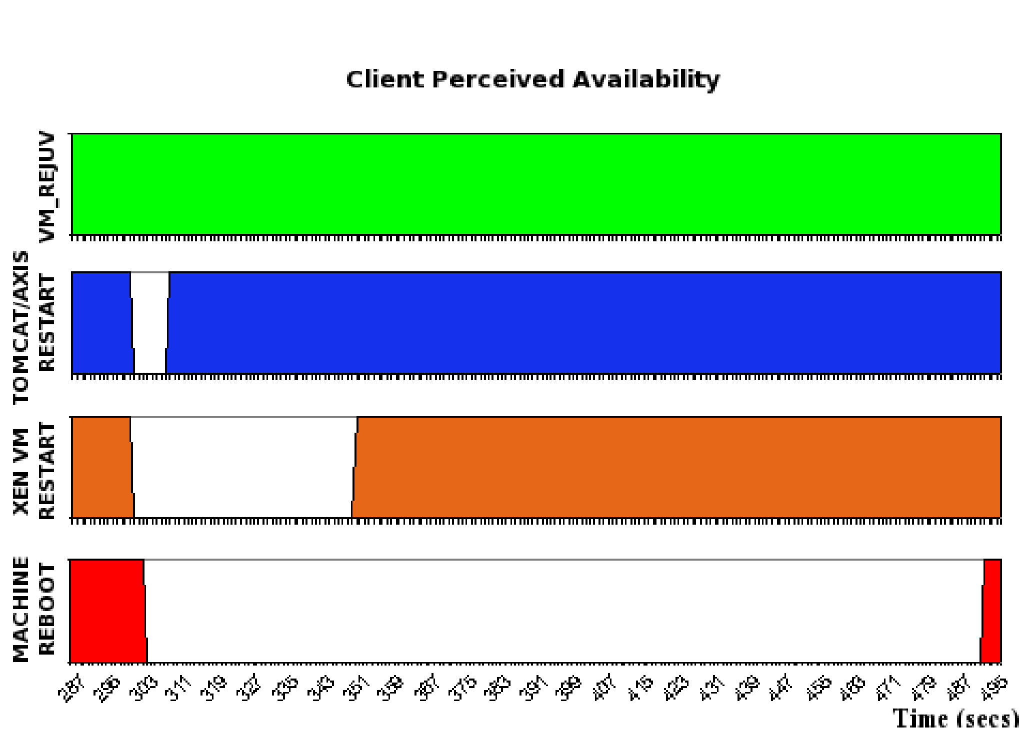


Figure 2.18 Client perceived availability for different restart mechanisms with Tomcat/Axis

As can be seen our rejuvenation scheme achieves a zero downtime. The clients do not even notice there was a rejuvenation action in the active server. In the other cases there was some visible downtime for the end-users: a Tomcat/Axis restart produced a visible downtime of 12 seconds; the XEN VM restarted resulted in a visible downtime of 56 seconds and finally a machine reboot represented a downtime of about 200 seconds.

The next step was to analyze the number of failed requests caused by one restart action.

Table 2.9 presents the average throughput during the 10 minutes of experiment, considering there was one restart action triggered, the total number of requests during the experiment, the total number of failed requests, the number of slow requests and the perceived downtime, as measured from the client's side.

The definition of "slow" requests corresponds to those requests that have a response time higher than 8 seconds. This value is defined based on the maximum time that a human client could wait for a web response before giving up.

	VM-ClusteringRejuv	Tomcat/Axis Restart	XEN Restart	Machine Reboot
Avg[a]. Throughput (req/sec)	143.8	134.7	128.8	97.3
Total Requests	86313	80847	77292	58401
Failed Requests	0	476	536	1902
Slow Requests	0	10	0	0
Downtime (msec)	0	12490	56143	200722

[a] Average

Table 2.9 Comparing Downtime and Failed Requests with different restart options in Axis

As can be seen, a server restart using our rejuvenation scheme resulted in zero failed requests: no work-in progress request was lost due to the recovery action. This shows clearly how we achieve the guidelines we proposed earlier, in detail guideline 3. Our proposed implementation provides a clean restart of the server with no perception for the end-user and no impact on the application consistency.

One important curiosity was that there were no slow requests when we applied a XEN Virtual Machine restart or a node restart. Those operations did cause refused connections that were counted as failed requests. The time-out mechanism when the node is available (scenarios Tomcat/Axis Restart and XEN Restart) is quite different from when it is not available (scenario Machine Reboot). This is related to the TCP-IP mechanism for time-outs, and explains why the number of failed requests in scenario "Machine Reboot" was not proportional to the downtime.

GRID Service environment

The OGSA-DAI results are presented in Figure 2.19, which presents an execution window of 600 seconds. Following the same format as before, we present the client perception. We observe that when our solution is triggered there is no perceived downtime: from the point of view of the clients the Grid service is always available. This is achieved due to the execution window where both grid services (primary and hot-standby) are running simultaneously.

The restart of the OGSA-DAI server had a visible downtime of 25 seconds. A restart of the XEN Virtual Machine resulted in a downtime of 92 seconds. Finally, a restart of the full operating system incurred a downtime of 279 seconds.

The next step was to measure the impact of every "restart" process according to the number of requests lost. In this case, there is no sense in talking about slow request

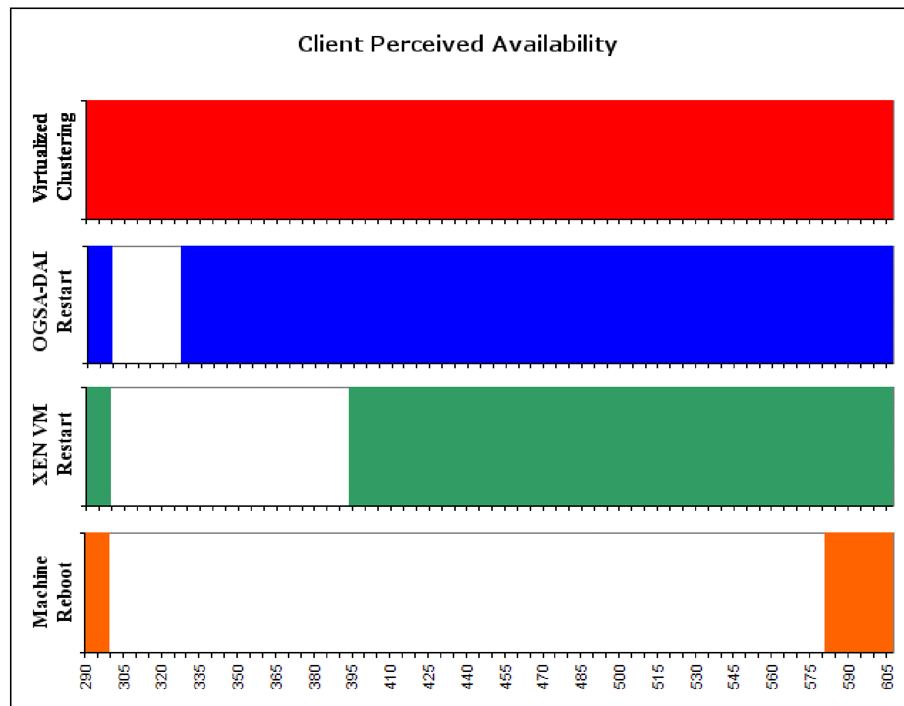


Figure 2.19 Client perceived availability for different restart mechanisms with OGSA-DAI

	VM-ClusteringRejuv	OGSA-DAI Restart	XEN Restart	Machine Reboot
Avg[a]. Throughput (req/sec)	1.205	1.165	1.085	0.647
Total Requests	723	699	651	388
Failed Requests	0	70	758	2353
Downtime (msec)	0	25479	92514	279454

[a] Average

Table 2.10 Comparing Downtime and Failed Requests with different restart options in OGSA-DAI

concept because in Grid environments the interlocutor of OGSA-DAI would be another machine, not human, and we do not have any threshold to measure slow requests. The results are presented in Table 2.10. We included the average throughput obtained in every experiment during a test-run of 10 minutes. Assuming that there is one unique restart action in those 10 minutes we present the total requests during the execution, the number of failed requests, the average throughput in that test-run of 10 minutes and the observed downtime.

When applying our mechanism we were able to achieve the smallest MTTR possible: zero. The Grid service was always available and no downtime was perceived by the client applications. We were also able to register zero failed requests. Other restart solutions incurred several failed requests (from 70 up to 2353) and a visible downtime.

An brief discussion of results

In both cases, our main achievement in the experiments was a zero downtime for our rejuvenation mechanism. This result has a strong impact if we want to do some simple calculations on the resulting availability when we apply prophylactic restarts (rejuvenation actions) to combat software aging phenomena. Table 2.11 presents the maximum downtime allowed to guarantee different percentages of availability in a service.

Percentage uptime	Percentage downtime	Downtime per year	Downtime per year (msec)
98%	2%	7.3 days	63072x10^4
99%	1%	3.65 days	31536x10^4
99.8%	0.2%	17 hours, 30 minutes	63000x10^2
99.9%	0.1%	8 hours, 45 minutes	31500000
99.99%	0.01%	52.5 minutes	3150000
99.999%	**0.001%**	**5.25 miuntes**	**315000**
99.9999%	0.0001%	31.5 seconds	31500

Table 2.11 Analysis of Availability

Without considering other sources of unplanned hardware and software failures, taking into account the software aging source of unplanned outages: Following the Table 2.11, if we want to achieve a 99.999% (five "nines") availability in a whole year in a single server machine with Tomcat/Axis installed, then we have the following limits for the number of restarts: we can only do 1 node reboot in that whole year; or 5 restarts at XEN Virtual Machine level; or alternatively if it works out, 25 restarts in the Tomcat/Axis

application level. In the case of OGSA-DAI the numbers are similar: 1 one reboot in a whole year, 3 restarts at XEN level, 12 OGSA-DAI restarts.

If the target application server is installed with our *VM-ClusteringRejuv* framework then we can apply huge number of planned restarts per year within the five nine figure. The unique constraint is the minimum acceptable interval between restarts. Of course, this is only truth against software aging scenarios. Because our approach does not avoid other software failures or even hardware failures, which have to be taken into account to evaluate the real "number of nines" achieved by a system. So, our analysis is only focused on aging scenarios.

All Those experiments have shown that our virtualized clustering rejuvenation approach is a promising quite solution to achieve ahigh availability for quite different services, with zero downtime and zero failed requests and without incurring any additional cost in terms of servers and IT infrastructure.

2.4.3.5 Session-data issue and the overhead to maintain it on Tomcat

In this sub-section we present some results about the session-replication scheme of Tomcat. Tomcat application server offers a set of capabilities to guarantee the session data of requests in case of failure on the server. Mainly, there are three mechanisms: via local/remote file, via relational data base or via clustering. The first two ways, as their name suggests store the session data on file or database respectively. The last one is quite different. The clustering mechanism is designed to work in cluster configurations, with several servers running the same application to increase the service performance. The tomcats in the same logical cluster share their session data. So, if one Tomcat server of the cluster creates a new session object, then this Tomcat replicates this session object in the rest of Tomcat servers on the cluster. We decided to evaluate this mechanism to see if it is able to be used jointly with our VM-ClusteringRejuv mechanism to guarantee the session objects in web environments.

It is important to remark that this type of mechanisms can be used in other commercial web application servers like Jboss or IIS.

To avoid losing session data, we need to replicate the session-objects from the active to the backup server. This replication scheme would bring some overhead, but it is of great importance if we do not want to lose any session data when there is a restart operation.

We have conducted an experiment of 15 minutes with Tomcat/Axis configured with the session objects of 8kb (typical session-object size is less than this value). The results are presented in Figure 2.20. The Figure presents two scenarios: one with session replication mechanism activated and another without it.

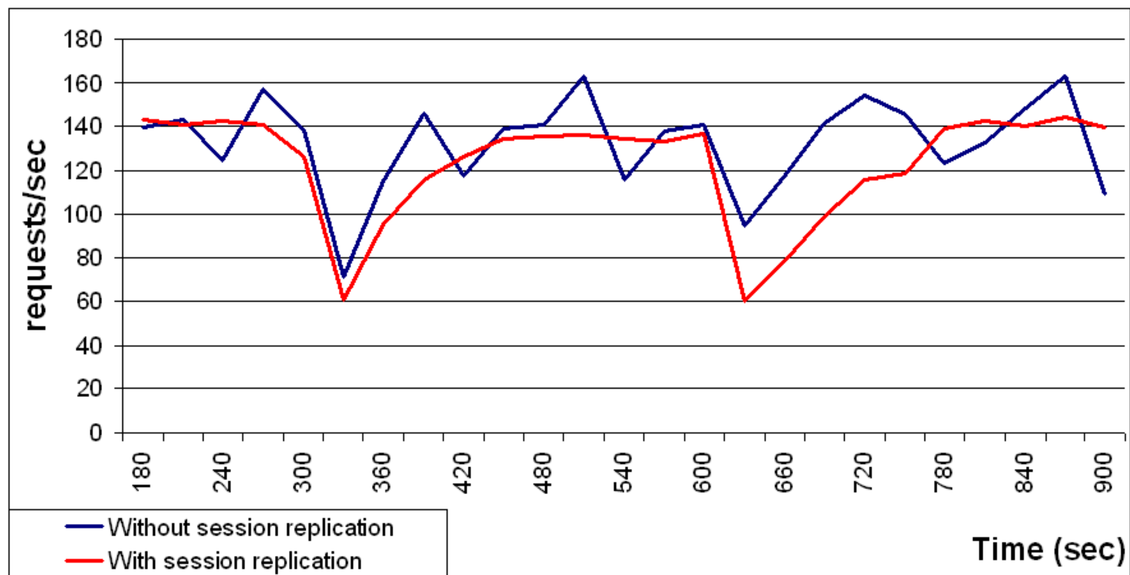


Figure 2.20 Overhead comparison of session replication with session objects of 8kb with Tomcat/Axis

We just presented the last 750 seconds of the experiment, since we removed the warm-up time.

In the first case we did not lose any session data and the application served a total of 117,750 requests, during those 15 minutes. In the second case we observed 15 session errors and the application was only able to serve 107,301 requests. This represents an overhead of 8.8%. With session objects of 1k we got an overhead of 4.9%. To summarize: If we want to maintain the consistency of the application through the rejuvenation process we need to maintain the session state of the clients. So, there is some overhead to guarantee the session state and this is the price to pay if we want to maintain it during our rejuvenation process. The session state is critical for stateful applications like online stores as Amazon or others.

2.4.3.6 VM-ClusteringRejuv effectiveness in Cluster Environments

So far we have been testing our rejuvenation in single servers. One pertinent question still remains: is our scheme any useful when we have a cluster configuration?

When we use a cluster we have a load-balancer box and support for IP fail-over when there is a server crash. And here is the critical point: clusters are usually used to tolerate failures, by using redundancy. Our rejuvenation scheme is meant to avoid failures due to software aging. So it has a high potential of being used in cluster configurations.

Suppose we have an application server that degrades over time, due to aging. If we

replicate that application through N servers, it will degrade in all the servers, probably with different decay functions, but it will decay in all of them. If we use our rejuvenation scheme we can mitigate the impact of these fail-stutter failures. We also have a way to restart a server without losing any in-flight request, something that sometimes is not achieved when IT managers apply planned restarts in servers belonging to a cluster.

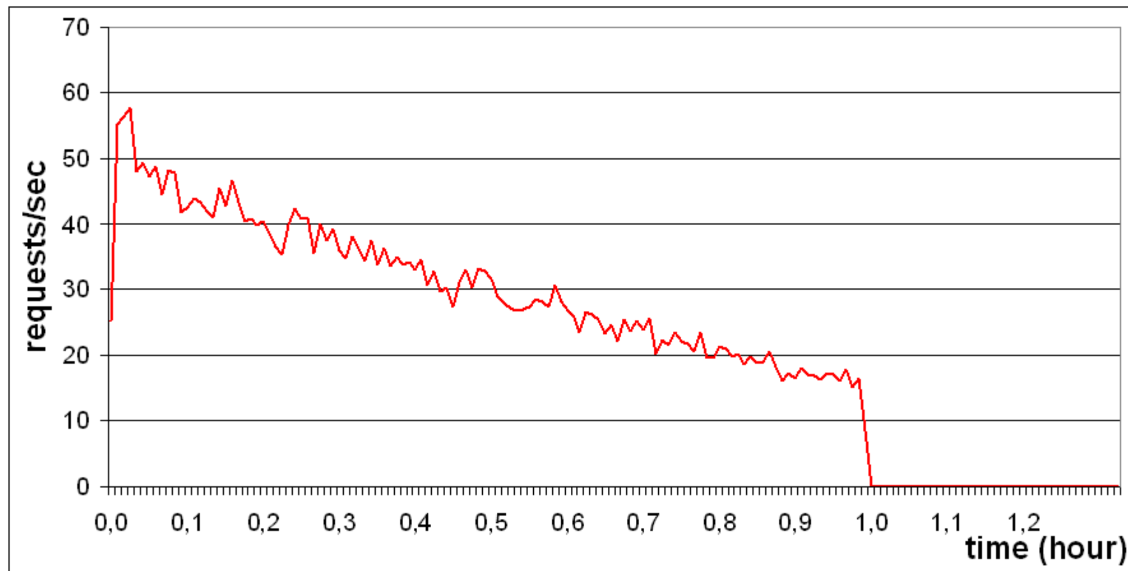


Figure 2.21 Throughput of Server S1 (Tomcat/Axis) in a cluster, suffering from severe aging.

To demonstrate the potential of our rejuvenation scheme in clusters we have set up the following experiment: we ran the Tomcat/Axis application in a cluster configuration (using 3 virtual-machines in XEN: one LB and two active servers). In order to speed up the visualization of the aging, we configured the JVM to 64 Mb. The throughput curves of both servers are presented in Figures 2.21 (Server S1) and 2.22(Server S2). Since both servers run the same application they will suffer from aging at a similar pace, if the load-balancer is using a round-robin strategy.

During this experiment Server S1 had a crash after 3,630 seconds of execution. Upon the failure of S1, LVS migrates all the requests to S2. We decided not to restart S1 automatically to observe the behavior of S2 without interference. We can observe an important fact when S1 crashes and all requests are migrated to S2. We can observe how at that moment the throughput of S2 reaches zero, before it recovers a bit the throughput. This is due to the fact that suddenly, S2 has to absorb double workload. Server S2 had a crash after 4650 seconds. In Figure 2.23 we can see the performance decay in the overall throughput of the cluster, down to the zero level. We can also see the overall throughput when using our rejuvenation scheme (with a SLA verification of around 75%

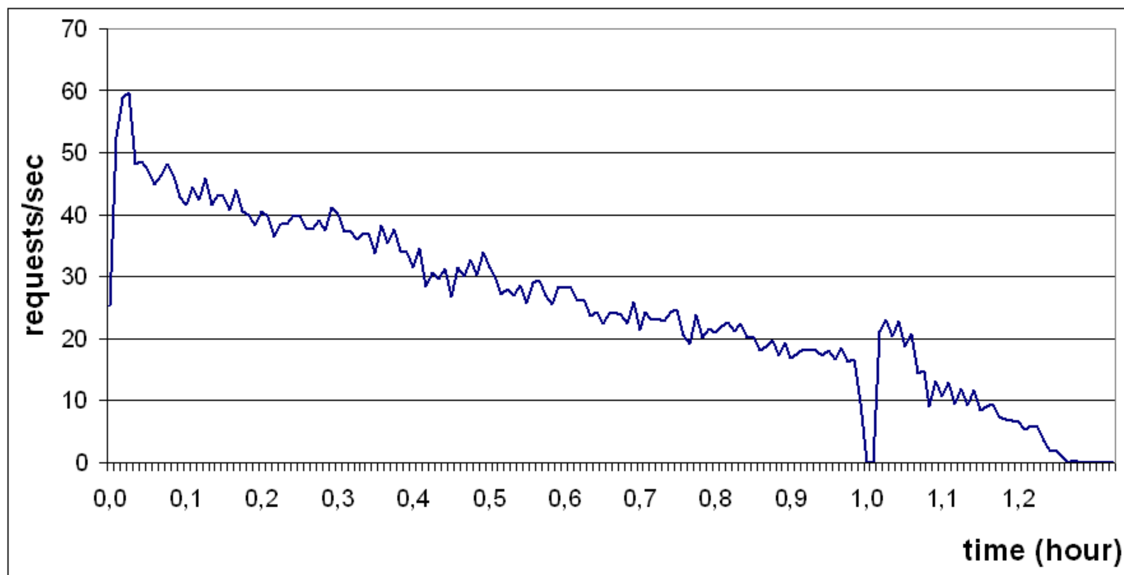


Figure 2.22 Throughput of Server S2 (Tomcat/Axis) in a cluster, suffering from severe aging.

of the throughput). As can be seen, our scheme is able to maintain a sustained level of performance: the cluster did not face any performance failure or a complete crash (as in the default case). There were no failed requests when using rejuvenation in this experiment, while in the default fail-over mechanism of LVS we noticed 2,223 failed requests. During the 4,800 seconds of the experiment the default configuration of the cluster was able to serve a total of 231,044 requests from the clients. By using our rejuvenation scheme the cluster managed to serve 339,186 requests. This means that our mechanism promoted an increase of about 46% in the performance of the application.

We have done a second experiment where we included an automatic restart in the default cluster configuration: when the Load Balancer detects a crash it restarts the failed server automatically. The results of 5 hours of execution are presented in Figure 2.24. It presents the cluster throughput when we use our rejuvenation scheme compared to the default configuration with automatic restart. Without our rejuvenation there were several crashes in the cluster and we lost 2,640 requests. With our rejuvenation scheme we had no failed requests. Comparing the difference in the total number of requests served we have also observed an improvement of 66% in the overall performance. The improvement was calculated taking into account the number of requests processed by the web server with and without our framework during the same period of time. We assume that the use of our framework provides the maximum number of processed requests possible.

In our opinion this is a sound result since it proves the high potential of using our rejuvenation scheme in cluster configurations: it does not only avoid crashes due to aging,

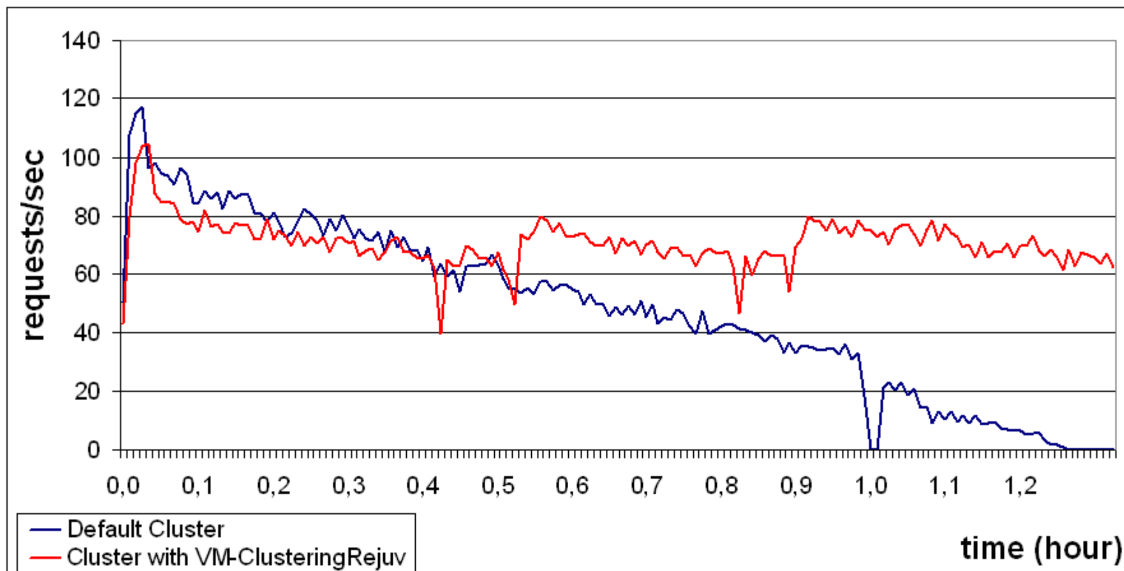


Figure 2.23 Cluster Throughput comparison, with and without VM-ClusteringRejuv, using Tomcat/Axis

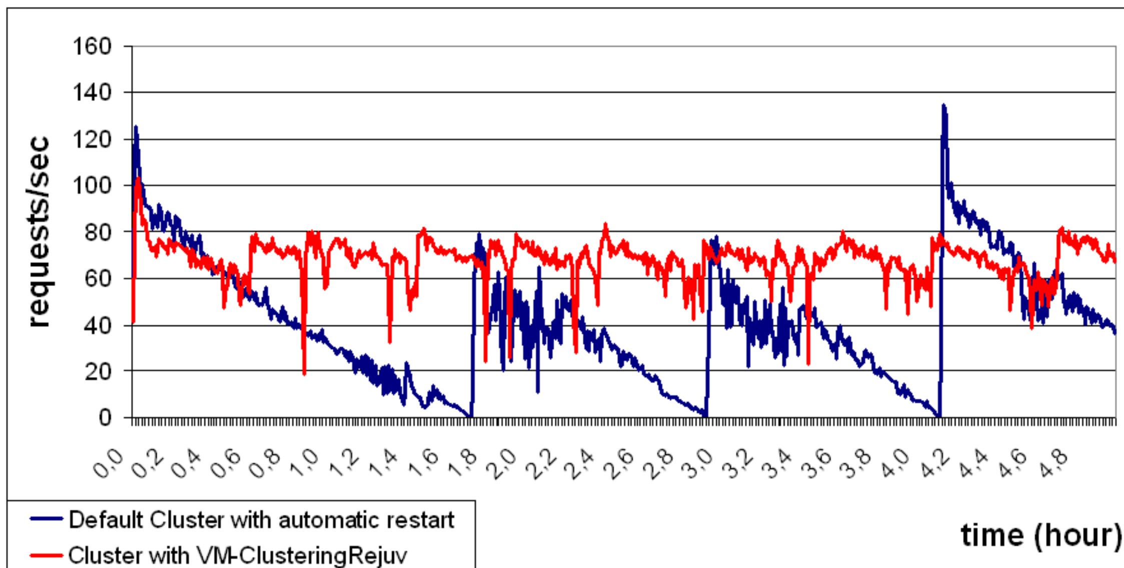


Figure 2.24 Cluster Throughput comparison, with and without VM-ClusteringRejuv and automatic restart, using Tomcat/Axis

but also increases the performance of the applications running in the cluster.

How important is our "clean" restart in comparison with "blind" restarts?

In this experiment we decided to compare the importance of applying our "clean" restart mechanism when compared with typical procedures that resemble a "blind" restart: where once there is a decision to trigger a rejuvenation action the server is restarted without any extraordinary concern. In this experiment we used a single server running XEN and we have configured a cluster with two active servers. Our goal was to compare the effect of a "clean" restart from our VM-ClusteringRejuv scheme with a "blind" restart and a "blind" reboot of an active server. In the configuration of VM-ClusteringRejuv we needed to use 5 virtual-machines: one for the LB, two for the active servers and other two for the hot-standby servers. In the two other scenarios we just needed to use 3 virtual machines: one for the LB and two for the active servers. Results are presented in Figure 2.25.

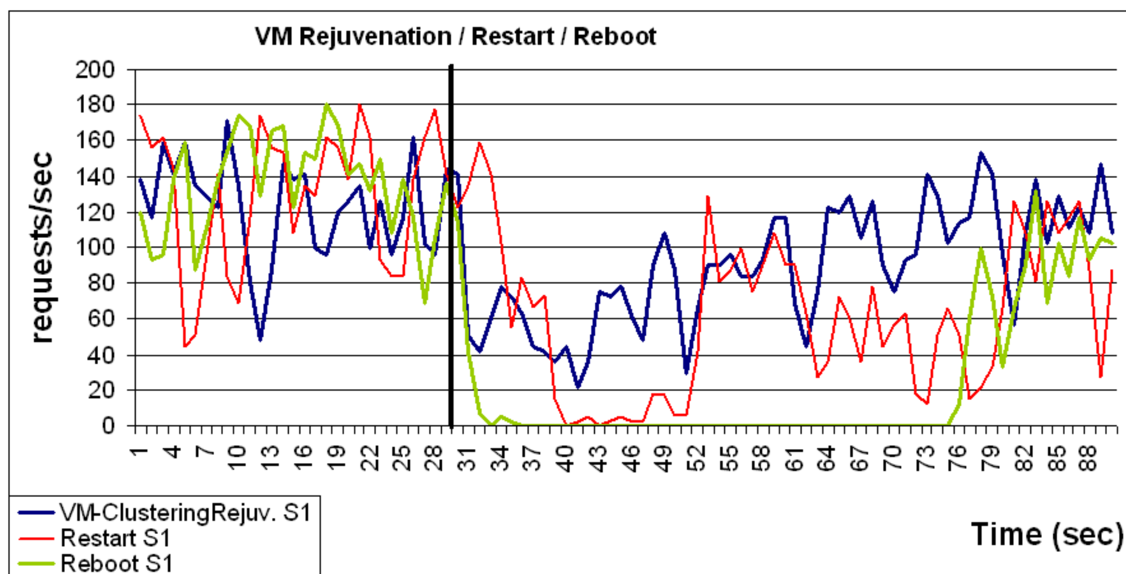


Figure 2.25 Impact Evaluation of a "clean" restart in front of "blind" restart and "blind" reboot.

Figure 2.25 only shows a "zoom-in" of the results of around 90 seconds. After 30 seconds we applied a restart action in server S1 using our VM-ClusteringRejuv scheme, a "blind" restart of server S1, and a "blind" reboot of server S1. In our scheme we had zero failed requests as opposed to the other schemes: the "blind" restart produced 244 failed requests while the "blind" reboot led to 341 failed requests. These failed requests happened even when we had a cluster configuration.

It can also be seen in the Figure 2.25 that there was a "window" with a very low throughput right after the "blind" restart. It was much more visible after the "blind" reboot. That happens because when server S1 is restarted there are several requests that get an exception and the LB applies a fail-over to server S2. However, the XEN layer gives more CPU to server S1 during the restart phase. During that phase S2 does not get enough CPU and some of the requests are not executed or executed with extra delay. And this is why we see that fall in the throughput curves.

2.5 A proposal Software rejuvenation framework for Legacy Application Servers

After presenting our main contribution in the area of proactive software rejuvenation mechanism and showing its effectiveness, we decided to go one step ahead and try to avoid even sudden crashes and their potential consequences.

As it seems that it is unfeasible to build systems that are guaranteed to never crash or suffer transient failures, G. Candea and A. Fox proposed the crash-only software concept [32]. The crash-only software is based on the idea of designing software systems that handle failures by simply restarting them without attempting any sophisticated recovery. Since, some authors have proposed the idea of developing new Internet Services using the concept of crash-only software. The crash-only concept can be seen like a generalization of the transaction model taken from the data storage world. The crash-only system failures have similar effects over the data storage systems. However, what happens with the recently designed and deployed Internet Services, usually made-up by cooperative legacy servers?

In this section, we discuss the possibility to migrate the crash-only software concept to these legacy servers for Internet Services. We focus on application servers because traditionally, these servers have the business logic of the applications and are more sensitive to the transient failures or potential crashes.

As an example, we expose the classical online flight ticket store. We have a multi-tier application environment made-up at one web server for static web pages, an application server for business logic and a database server for storing all ticket information flight.

We can define the session in this environment as the interval between the time that the user logs-in and finally, logs-out. During this interval the user can search for flights, add some of them to his/her shopping basket and pay for them. If there was a crash during the session, the more sensitive point of the multi-tier application would be the application server, because the web server only manages static information and database

servers have their traditional after crash recovery systems. The application servers have the responsibility to manage the session state. This session state is needed for the session's success because the session state contains information useful to the session's subsequent states. For this reason, if there was a crash in a legacy application server, a simple and blind restart could be dangerous to the consistency of the business logic state.

Due to the crash-only constraints we restrict our proposal to application servers with external session state storage, which meets crash-only laws like SSM [78] or Postgres database system [110].

The main goal is to migrate the ideas proposed by authors of the crash-only concept to the current legacy application servers in order to obtain a "crash-safe" and "fast" recovery system.

The second goal of our proposal is to hide any possible crash from the end-users improving the crash-only software concept which achieves crash-safe and fast recovery systems although it does not avoid the occasional unavailability time.

2.5.1 Crash-only Concept

The crash-only concept is based on the idea that all systems suffer transient failures and crashes hence it is useful to develop techniques that could overcome a potential crash or transient failure. Furthermore, normally the system crash recovery time is lower than the time needed to apply a clean reboot using the tools provided by the application itself. Table 2.12 illustrates this reasoning.

System	Clean Reboot	Crash Reboot
RedHat 8 (with ext3fs)	104 sec	75 sec
JBoss 3.0 application server	47 sec	39 sec
Windows XP	61 sec	48 sec

[a] Extracted from [32]

Table 2.12 Reboot times from different Systems

The reason for this approach is the desire to improve the system performance. The systems usually store potential non-volatile information on volatile devices, like RAM, to be faster and obtain higher performance. For this reason, before a system can be rebooted, all this information has to be saved on non-volatile devices like hard disks to maintain data consistency. However, in the case of a crash reboot all this information is lost and potential inconsistency state may result after reboot.

Based on these potential problems, crash-only software is software where a crash behaves as a clean reboot. Every crash-only system has to be made of crash-only

components and every crash-only component has only one way to stop - by crashing the component- and only one way to start the component: applying a recovery process. To obtain crash-only components, authors defined five properties that the component has to include:

1. *All important non-volatile state is managed by dedicated state stores.* The application becomes a stateless client of the session state stores, which helps and simplifies the recovery process after a crash. Of course, this session state store has to be crash-only, otherwise the problem has just moved down to the other place.

2. *Components have externally enforced boundaries.* All components have to be isolated from the rest of the components to avoid faulty propagation. There are different and potential ways to isolate components. One of the most successful ways for isolating components which has become popular in the last few years is virtualization [47].

3. *All interactions between components have a timeout.* Any communication between components has to have a timeout. If no response is received after that time, the caller can assume that the callee is failing and the caller can start a crash and recovery process for the failing component.

4. *All resources are released.* The resources cannot be tied up indefinitely thus it is necessary to either guarantee the resources be free after a limited time or the component using the resources crashes.

5. *Requests are entirely self-describing.* It is needed that all requests are entirely self-describing to make the recovery process easier. After a crash, the system can continue from where the previous instance left off. It is necessary to know the time to live (TTL) of the request and the indempotency property.

Trying to describe in fine detail all philosophy of Crash-only software is out of scope of this document. In order to obtain more information, we recommend the reader to read the ROC project motivation [97] and visit the website [103] and other authors' papers around this concept.

2.5.2 Crash-only and masking failure Architecture for Legacy Application servers

Our architecture is focused on a given set of application servers: application servers with external session state storage. The reason is because we cannot force an application

server to manage its internal session objects to external storage without changing the code. We want to propose a solution which avoids modifying the legacy application server code because this can be a titanic work or even impossible if the software is not available.

The architecture is made-up by three main components: the Requests Handler (RH), the Storage Management Proxy (SMP) and the Recovery Manager (RM) as shown in figure 2.26.

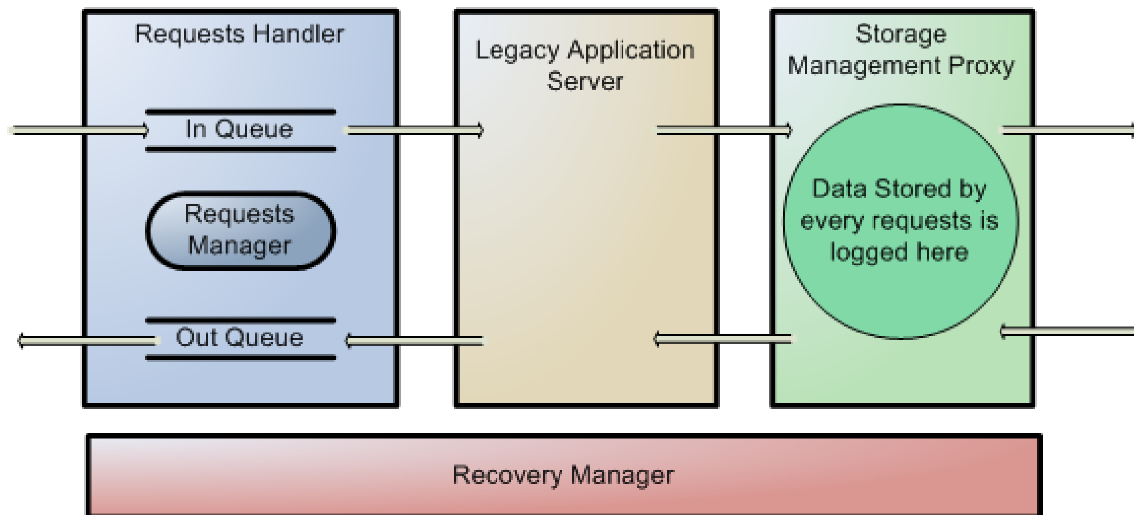


Figure 2.26 Crash-Only and masking failure Architecture

The Requests Handler has the responsibility to capture all HTTP requests that are sent to the application server from a potential end-user or a web server. Two queues and one requests manager form the RH. There is one requests queue, a responses queue, and a requests manager to manage and synchronize them.

When a new request is sent to the application server, RH handles the request and copies it to the *requests in queue* and then the request is redirected to the application server without any modification. When the application server sends a response to the end-user or web server, the Requests Handler captures the response and copies the response to the response queue and redirects the response without changes to the end-user.

To achieve this behavior the Requests Handler has to work in the same network domain as the application server and has to be configured to work in a promiscuous mode, and by using the application server IP it can capture all the application server requests and discard all not relevant requests (e.g. non-HTTP requests). Capturing the response is quite more complicated because, the Request Manager has to save all IP sources from all requests without response in the requests queue and any packet sent by the application server to one of the IP's from the requests queues will be captured. When the response is captured, the request associated to the response is removed from the queue. We base our

work on the idea that the requests and responses can be joined if we understand that the requests from one end-user are sequential and the responses too, for this reason, using the source IP of the request and the destination IP of the response can be enough to join one response with its request.

The Requests Handler has more important tasks other than only preserving a copy to know what in-flight requests are there in the application server. It has also the responsibility of detecting potential failures (transient failures and crashes).

Based on the idea proposed in [32] and [22], every time one request is sent to the application server from the Requests Handler, a timeout is activated. If the request timeout finishes without response, the Requests Handler tries to ping over the application server and if there is not an answer, the Requests Handler assumes that the application server has crashed and it starts a recovery action. Furthermore, to detect fine-grain potential failures like application failures, the Requests Handler reads every response content to try to detect potential transient or intermittent failures (e.g. any HTTP 400 error) and notifies this fact to the developers or administrators and applies a recovery action discarding the response message failure to avoid the potential concerns to the end-user. The Storage Management Proxy (SMP) is based on the idea proposed in [24]. [24] proposes a new ODBC for communications with data base servers which understands all SQL statements as a transaction and modifies the statements syntax to achieve a successful behavior. This ODBC is integrated in Phoenix APP [21], to achieve a system with crash-safe processes.

We have simplified the idea proposed in [24]. Our system only stores information of the communications between the application server and any data storage device (database servers or session state storage). The information obtained from these communications will be used only in the recovery process and during a correct behavior of the application server. We can say that SMP is only a logging system that saves all requests and responses' information. If a crash happens during a transaction process in the database server, the transaction will rollback and the SMP will write this event only to keep track of what happened during this interaction between the application server and the database server. Finally, the Recovery manager has the responsibility to coordinate the recovery process. To define the recovery process, we have designed an architecture and a process to avoid the potential problems of the recovery process described in [23]: exactly-once execution, (where the latter means, no output to the user is duplicated to avoid confusion), the user provides input only once (to avoid user irritation) and the user attempt is carried out exactly once. If the Requests Handler detects a failure a signal is triggered to the Recovery manager to start a recovery process. First, the RM notifies this situation to the RH and SMP. When the RH receives the notification, it stops to redirect requests and redirect

responses to and from the application server and it only receives requests from end-users. At the same time, the SMP avoids all communications between the application server and the data storages. After these both processes are concluded, the RM recovers (crash reboot) the application server (e.g. kill -9 Pid-process) and restarts the application server waiting for a new application server instance. When the new instance is running, the RM notifies the fact to the RH and SMP. The RH redirects all requests without response (in-flight requests) to the new instance of the application server again, so the user does not need to provide the input again. If at any time, the RH receives a response without request associated to it, the response is discarded to avoid potential duplicates to the user in order to avoid confusion.

At the same time the SMP is monitoring the communications between the application server and data base server or third services, avoiding potential duplicated database or session object modifications. When SMP detects a duplicated communication, the packet is not redirect to the storage if the performance of this communication was successful, otherwise the communication is redirected. On the other hand, if the communication was not successful, we use the SMP response saved to build a new (old) response and send it to the application server as if it'd been stored. Thanks to this control of the duplicated communications of the SMP we avoid potential problems presented in [23].

The idea of this coordination between RH and SMP using the RM is to avoid the potential crash hazards presented in [22, 23]. In these papers, the authors present an architecture based on interact contracts. These contracts are thought to apply safe-recovery mechanisms after a crash, replaying all requests without response before the crash and avoid uncomfortable behaviors of the application servers. However, authors present a solution that has an important constraint for all components: even internal components have to keep the contracts to maintain the coherence of the architecture. We have used the idea presented in these papers to present the architecture to achieve crash-only software legacy application server and mask the failure to the user.

2.5.3 Discussion of the solution

In this section, we propose a new architecture based on two proposals to achieve crash-safeness and fast recovery. Our goals proposed at the beginning of this section were to migrate the crash-only software concept to achieve a legacy application server with the same characteristics as a crash-only designed system. Moreover, we proposed the improvement with regards to results presented in [33] where when using the crash-only concept, the system had 78 missed requests during a micro-rebooting process based on crash-only software. We want to design an environment to avoid these missed requests,

reducing to zero missed requests if it is possible. We understand a legacy application server as an indivisible component to make possible the crash-only concept migration, because the crash-only software is made-up by crash-only subcomponents. Based on this premise, it is easy to understand the reasoning of how our architecture preserves the properties of the crash-only software. As we have mentioned, we have restricted our study to the "stateless" application servers: The session state is preserved in external session state storages, accomplishing the first property (1). The architecture alone cannot achieve the second property (2), though we can use virtual machines (VM) to run the legacy application server and the rest of our proposal's components: one VM for each component to guarantee the isolation between components. The third property (3) is guaranteed by the RH, the SMP components and the behavior of the application servers. All communications have a timeout configured at least at OSI 4-layer (e.g. TCP/IP protocols). The fourth property is more difficult to accomplish. Working with legacy application servers, this property has to be delegated to the Operating system, which guarantees that all resources are leased. Finally, the HTTP requests, the traditional type of message for application servers, which are completely self-described. The secondary goal is preserved thanks to the queues inside the Requests Handler. During the crash and the recovery process, the application server is unavailable for the end-users or third applications. However, our Requests Handler continues capturing requests for the application server like a proxy and when the recovery process finishes, these requests waiting on the queue will be redirected to the application server like nothing happened. This process can mask the crash for the end-users in most cases like a previous work [107] where the solution masked potential service degradation and reboot process. Our proposal also avoids error messages if it is possible, because the RH parses the requests to try to detect fine-grain application failures or the application server failures (e.g. 400 and 500 HTTP errors) avoiding that end-users may observe neither these transient failures nor temporary system unavailability. In our proposal we get crash-safe self-recovery legacy application servers and even our solution offers a "fast" recovery process. We can confirm that the Requests Handler simulates a non-stop service which is the maximum speed of the recovery, and though the end-users will suffer response delays during the recovery process, we think that that penalty delay is proportional to the advantages of the proposal.

2.6 Related Work

Last twenty-five years have seen a considerable amount of research on the topic of fault-tolerance and high-availability. Techniques like server redundancy, server fail-

over, watchdogs, replication schemes, checkpointing and rollback have been extensively studied in the literature. However, most of these techniques only act when a failure has occurred. If the system anomaly does not result in a crash but rather in a decrease in the performance levels or some instability in the quality of Service (QoS) metrics most of those techniques are not even triggered. For this reason, the proactive software rejuvenation becomes an improvement to avoid crashes before they happen.

Two basic approaches for rejuvenation have been proposed in the literature: time-based and proactive rejuvenation. Time-based rejuvenation is widely used today in some real production systems, such as web-servers [15]. Apache Web server [15], the current most popular web server [91], implements a time-based rejuvenation mechanism to avoid the symptoms and consequences of software aging. This mechanism causes a restart of the user response threads of the server when the threads have attended a determined number of requests. However, this mechanism has been studied in detail in [59] and the authors demonstrate that the configuration to activate the rejuvenation mechanism reduced the performance of the server. On the other hand, proactive rejuvenation has been studied in several works. Castelli et al. [35] presents a methodology for proactive management of software systems which are prone to aging, more specifically due to resource exhaustion. They try to predict the time to crash due to the aging phenomena using time-series analysis. However, their approach does not avoid the downtime during the rejuvenation process, but thanks to the prediction they reduce the MTTR. Furthermore, they present analytical models based on stochastic reward nets to evaluate the impact of different rejuvenation policies on the availability of cluster systems.

Some recent experimental studies have proved that rejuvenation can be very effective technique to avoid failures even when it is known that the underlying middleware [106] or the protocol stack [27] suffers from memory leaks like our technique.

Several studies published in the literature tried to define some modeling techniques to find the optimal time for rejuvenation [115, 52, 114, 13].

On the other side, ROC project [103] from Stanford University presented the concept of micro-rebooting in order to reduce the rejuvenation overhead and the MTTR: instead of applying restarts at the application level like our solution, they only reboot a very small portion of components, the potential guilty components. Their main contribution was to decrease the MTTR of the applications and the results were very promising [33]. As an example, they were able to reduce the downtime due to their micro-rebooting to only 600 milliseconds. They reduced the downtime two orders of magnitude in comparison with an application restart (take 20 seconds). However, in our approach we were able to obtain a MTTR equal to zero.

There exist some traditional techniques for survivability of application-services such as server replication, server migration and reconfiguration [123]. These approaches are not directly applicable to legacy applications running on top of commercial middleware. Some session-storage mechanisms [78] will also be developed to maintain the critical state of the applications when a recovery action is applied.

We want to note that in the coupe of this chapter we have presented several related works. For this reason, we have omitted them in this section. We wanted to avoid to repeating again the same references increasing the difficulty to the reader.

2.7 Summary

In this chapter, we have presented a highly effective software solution against software aging phenomena, fail-stutter and performance failures. Our solution makes use of a virtualization layer that although it introduces some overhead it pays-off in terms of consolidation and ease of deployment.

We have proved after an exhaustive experimental evaluation the effectiveness of our framework in the case of three different Internet environments: Webservices, web applications and Grid. Furthermore, although we have showed the effectiveness of our approach in a concrete Web Services server, web application server and Grid service, we are convinced that similar results can also be observed in other application examples that present latent errors.

Another important achievement of this chapter and the framework presented within is the fact our solution is completely independent of the server under test. We can use our framework with legacy software without requiring any change.

The results presented in this paper encourage us to consider that this approach for high-availability can be partial contribution for the deployment of dependable systems.

Furthermore, we have discussed the power of crash-only concept to be the core of a wrapper to involve legacy application servers to become crash-only servers to avoid the problems of the unplanned crashes due to the software aging phenomena.

The work described in this chapter has resulted in publications presented in Section 1.5.1.

Chapter 3

A framework for software aging prediction based on Machine Learning

3.1 Introduction

As we presented in the previous chapter, our main goal is to reduce as much as possible the MTTR: offering a proactive recovery framework against software aging symptoms. Our mechanism was triggered by a simple rule: threshold violation. This threshold was fixed based on an instantaneous value of a metric. If the metric overcomes this value the software rejuvenation action is triggered.

Traditional reactive techniques (e.g. fault-tolerant or clustering) based on redundancy and replication causes a downtime based on the time to detect the failure and the time to repair the system (the sum of both delays composes the MTTR presented in Formula 2.1), as shown in Figure 3.1. This downtime could be reduced significantly using proactive techniques like our proactive recovery framework.

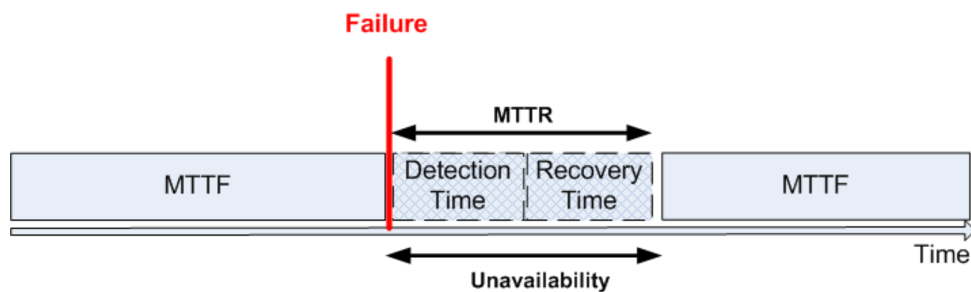


Figure 3.1 MTTR Description in a reactive rejuvenation mechanism

Although, our approach allows us to increase the availability to near 100% in several aging scenarios, its effectiveness is limited. First of all, our approach is focused on aging scenarios, so our approach is not focused on other software or hardware failures. It is based on the correcteness of the threshold definition and the software aging trend. Our technique defines a threshold based on the value of a metric or a set of them, and we trigger our recovery mechanism when this threshold is violated.

This approach allows us to start the recovery action before the failure occurs, reducing the MTTR, and hence increasing the availability of the system. However, our threshold-based mechanism introduces an important and hard constraint to become useful. We need to know which metric or a set of them are causing the software aging or the crash *a priori*. This constraint is too hard. Because, it is impossible to know *a priori* which resource or system metric causes the software aging. This fact makes it necessary to monitor all metrics allowed in a system and apply thresholds to all of them. This approach is quite unaffordable and becomes useless in a production environment. It is also possible that the cause of the crash or software aging would be a combination of metrics. Moreover, we need some human expertise to determine the threshold values and the resources we have

to monitor.

On the other hand, there is other constraint to take into account, though less important for the effectiveness of the approach. We can detect that a metric or a set of them have violated a given threshold, but we can not anticipate how much time the system could be working before crash, even using our mechanism. Even, an expert or the system administrator has to fix the threshold value and depending on this value, the rejuvenation framework would do better or worse. Besides, during a given window of time the active and the hot-standby server are running simultaneously before we restart the active, closing the rejuvenation cycle. This window of time length depends on the workload and application type, going from seconds to minutes or hours. If the software aging is too aggressive for the time window length, the active server could crash within the window of time, resulting in a lapse of visible downtime, as it is presented in Figure 3.2. Increasing the visible MTTR, although less than the MTTR caused by reactive solutions.

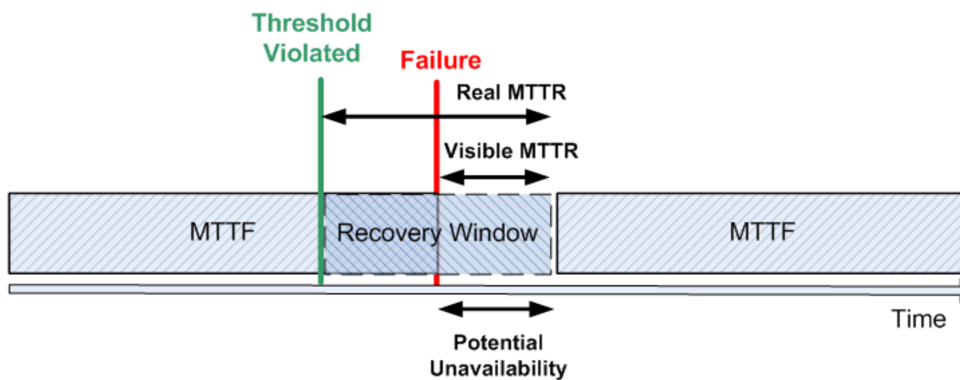


Figure 3.2 MTTR Description in our Threshold-based rejuvenation mechanism

Furthermore, our threshold-based approach could induce false-positives. If a metric value violates the threshold due to a sudden and temporary workload peak, it is not able to anticipate the real trend of the metric. It is only based on a simple and instantaneous state of the system. This fact triggers an unnecessary rejuvenation action. This is could be important only if the rejuvenation action has an important cost (in time, money or other relevant metric). In our case, as we have showed our rejuvenation action has no-cost for the application or customers.

If we want to overcome the constraints described before, increasing the effectiveness of our previous presented rejuvenation action, we have to be able to know/predict with an acceptable error the time to crash or at least when failure is approaching. Then, we can start our rejuvenation mechanism to avoid potential crashes during the rejuvenation window phase. So, if we are able to predict the time to crash, we could reduce the crash possibility during the recovery window achieving 0 downtime (if it is possible) because

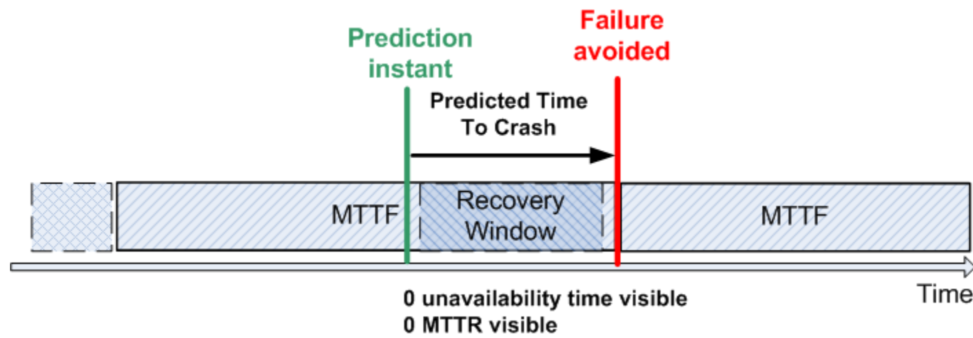


Figure 3.3 MTTR Description in our Prediction-based rejuvenation mechanism

we will be able to know (with an acceptable error) how much time the server could be running before crash as it is described in Figure 3.3.

However, predicting the time until resource exhaustion due to software aging is not an easy task. The progressive resource consumption over time could be non-linear, or the degradation trend could change over the time [52, 114, 115]. Software aging phenomena could be related to the workload, or even the type of the workload. Software aging could also be masked inside a periodic resource usage pattern (e.g. leaving a fraction of memory used allocated only after a periodic load peak). Another situation that complicates resource exhaustion prediction is that the phenomenon could look very different if we change the perspective or granularity used to monitor the resources; we will provide specific examples of this later on. This could be relevant, specially, when we are working with virtualized resources, as we do in our framework. Furthermore, another difficulty for software aging prediction is that it can be due to two or more resources simultaneously involved in the service failure [34].

Based on the two main limitations of our first approach described in previous chapter: a) the ignorance (*a priori*) of the metric/resource (and its trend) involved on the software aging phenomena and b) the limitations due to the arbitrary and fixed threshold, we decided to conduct a detailed study to evaluate the use of powerful of Machine Learning (ML) algorithms to predict the time to crash (TTC) of a system which suffers from software aging phenomena. Machine Learning algorithms allow us to determine by themselves which metrics/resources are more relevant/related with the software aging, learning from previous executions.

3.2 Modelling Assumptions and Prediction Strategy

We are focusing our software rejuvenation approach to solve a subset of software aging phenomenon. We have focused our research on software aging caused by resource

exhaustion (the most common cause of software aging), which causes finally a crash. We understand resource in this context such as any measurable system or application metric such as memory, CPU, threads or even application/system throughput or response time. It is important to remark that throughput or response time are not resources at all, though they are measurable and they could be exhausted (maximum acceptable response time or the maximum throughput achievable by the system), making clear the system/application crash.

In a perfect and easy world, resource exhaustion due to software aging or to some other factor would be a linear phenomenon with respect to time. The time to crash could be then predicted easily by:

$$T_{crash} \simeq \frac{Rmax_i - R_{i,t}}{s_i}, \tag{3.1}$$

Where $Rmax$ is the maximum available amount of resource i, $R_{i,t}$ is the amount of resource i used at instant t, and s_i is the consumption speed per time unit of resource i. However, this is a very simplistic approach. First, in this approach we are assuming the consumption rate is constant over time. As shown in Vaidyanathan et al. [114, 115], the consumption rate could depend on the workload. Moreover, we assume we know the resource i involved on the causing crash (due to the software aging) a priori.

But system/application crash can have several unknown reasons. It could originate from an interaction of several resources, or we might not be monitoring all the significant metrics from the system, or consumption rate could change over time, and could also depend on the, possibly changing, workload. Or even, the perspective in monitoring the resource is not the appropriate one; this is mainly true when we are working with virtualized resources. On the other hand, the system actions themselves (such as garbage collection) can mask or mitigate the software aging process, adding more life time to the system.

3.2.1 Motivating Examples

To understand the complexity and problems we can build a resource consumption model. The reader may refer to two examples that we found when we tried to model the Java Memory exhaustion of a J2EE application server.

3.2.1.1 Example: Nonlinear Resource Behavior

Given a deterministic and progressive software aging that consumes a resource at constant rate, our first approach to predict the time until resource exhaustion would

compute the rate (constant in this case) and apply Formula 3.1. One simple method to automatically obtain this slope is Linear Regression. Linear Regression has been used in several works, like [39], to predict the resource consumption under normal circumstances. It is a powerful tool in the area of capacity planning. The linear regression and statistical analysis to estimate the resource consumption trend have also been proposed to predict software aging [52]. However, linearity is a strong assumption. Even if the resource consumption by the faulty application is linear *a priori*, the system may exhibit nonlinear behaviors.

For example, we performed an experiment in which we injected memory leaks at regular rates under a constant workload in a Java Application (a Tomcat web server, as we will describe in Section 3.3). Figure 3.4 presents the memory actually used during the experiment with a progressive memory consumption and constant workload (dark line). We let the application run until the server failure due to memory exhaustion. We observe that even if our memory leak injection has a constant rate, the complexity of the underlying system introduces a nonlinear behavior.

In more detail: In Java applications, the Java Heap Memory is divided into three main zones: Young, Old and Permanent. When a Java Object is created, it is stored in the Young zone. When the Young Zone is full, the alive objects are moved to the Old zone. This zone stores objects that have been alive for a long time. The Heap management system defines an initial size for this zone, a fraction of the maximum memory available for the application. When the Old zone is full, the Heap Management System resizes it, allocating more memory to it if available. In Figure 3.4, we observe this resizing (grey line) at three epochs of the execution: at 2150 seconds, at 4350 seconds, and (less visible) at 5150 seconds. At the same time that the Old zone is resized, some objects are moved to the Permanent Zone, or are freed if they are not referenced by others. We can see that the monitoring perspective is crucial here. In the figure, the dark line is the percentage of memory used from the system perspective. The Heap resizes the Old zone and releases some part of the memory allocated by the application. The perspective from the system (dark line) is that the Java application is using a constant amount of memory for a while, but that is only part of the truth: the application has released a part of memory (grey line) but the system does not yet see that fact. That fact is only visible if we monitor the Heap internal behavior, hence obtaining a more accurate system view. We come back to this interesting fact in the next example.

In the example, the normal Heap size behavior allows the application to run for about 16 extra minutes, over what we would predict from the initial consumption rate. Furthermore, this extra time is strongly dependent on the aggressiveness of the memory

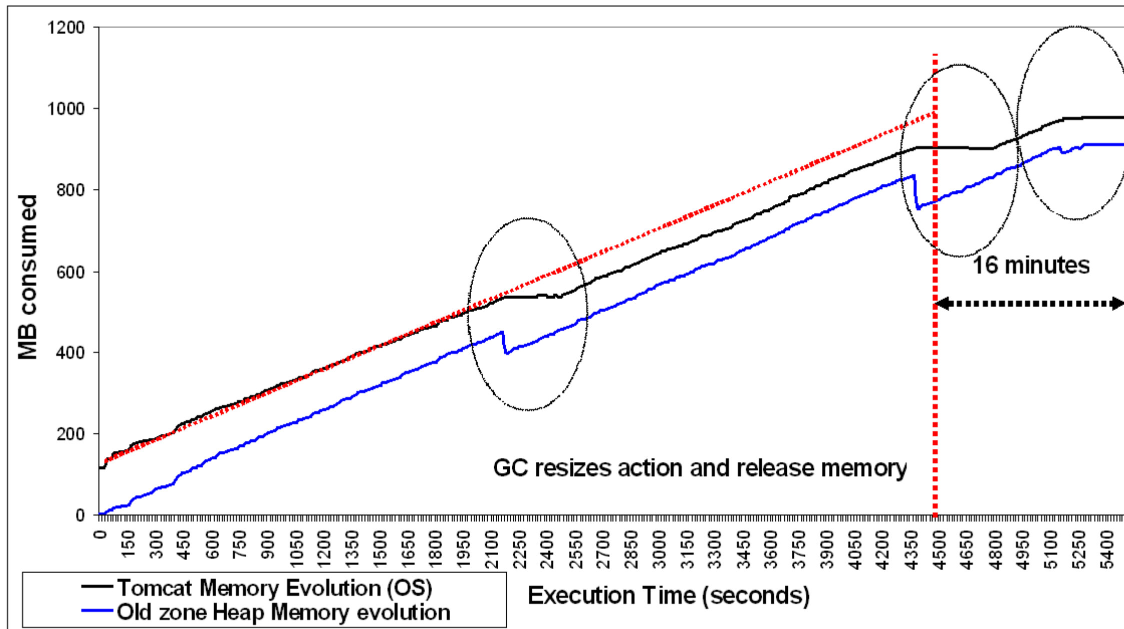


Figure 3.4 Progressive memory consumption of the Java Application.

leak and the workload; less aggressive leaks or lighter loads increase this extra time, hence the prediction error of a naive prediction strategy.

3.2.1.2 Example: Different Viewpoints on a Resource

As discussed in the previous example, resource behavior can look quite different depending on our monitoring strategy. This is most dramatic when we are working with virtualized resources, such as (in some sense) the Java Heap Memory. Memory usage by a Java application looks quite different if we monitor at the operating system (OS) level or at the Java Virtual Machine (JVM) level.

We conducted a simple experiment to show this duality. We run a web application (presented in Section 3.3 in detail) under a constant workload. We have modified the application to force three different phases in the application execution. Every phase lasts 20 minutes. The application has a normal behavior for 20 minutes, after which it consumes memory abnormally during 20 minutes, and after which the application releases the memory acquired during the previous phase, returning to the initial state. We repeat this periodic behavior/pattern every hour in a 5 hour duration. However, this periodic and repetitive resource consumption pattern is not shown by a simple monitoring over the memory used by the Java application from the OS perspective, but it is observable from JVM perspective. In Figure 3.5 we present both perspectives of the same resource during the same experiment. The waves (the grey line) represent the sum of the memory

used by the Young and Old zones. The Permanent zone is not depicted because it is constant during the experiment. We can observe how the memory allocated by the Java application looks constant from the OS perspective (dark line). In a Linux system, when an application frees up some memory, this free zone is added to a free zone lists but the system does not recover this memory automatically: it only recovers it when it is required by other applications. Moreover, in this case, there is another factor at work: the JVM memory management reserves a maximum memory (defined by the user via input parameter), so this action does not permit that the operating system to recover the free zone. Due to this behavior, if we monitor the OS memory consumed by an application it may look constant over time, but if we observe the Java Heap Memory, the application is consuming and releasing memory.

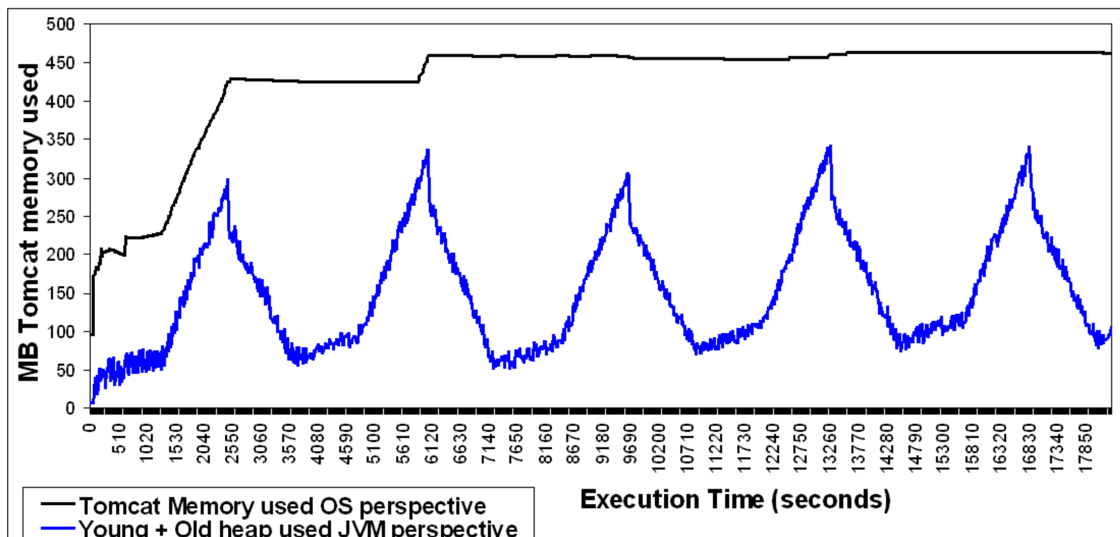


Figure 3.5 The Tomcat memory consumption from system and heap perspectives

These examples indicate the difficulty of building an accurate prediction system. We need detailed and sophisticated monitoring tools to obtain detailed information of the resources. Even if we collect all metrics, we need a lot of human expertise to decide which are the relevant ones, and to actually build the model taking into account the system's complex behavior. This is probably unaffordable in most cases, and anyway useless in ever-changing environments where hardware and software are updated frequently. For this reason, we propose Machine Learning and Data Mining as an alternative (or at least, a complement) to explicit expert modelling. Machine Learning can be used to automatically build models of complex systems from a set of (possibly tens or hundreds of) apparently independent metrics.

3.2.2 Prediction Assumptions

Our proposal is thus to use Machine Learning (ML) to predict the time to crash due to resource exhaustion. Due to the complexity of modeling these complex environments and with low knowledge a priori about them, we decided to use ML to automatically build the model from a set of metrics easily available in any system like CPU utilization, system memory, application memory, Java memory, threads, users, jobs, etc. The potential of Machine Learning methods is their ability to learn from previous executions what are the most important variables to take into account to build the model. In fact, the software aging could be related with other reasons, in addition to resource exhaustion. The techniques used in our work could predict the crash due to software aging if we collect the metrics (from resources or not) related with the software aging.

Note that our approach is valid when an impending failure is somehow foreseeable from the system metrics, not for sudden crashes that happen with no warning. These would require completely different techniques, such as static analysis of the code to reveal dangerous logical conditions, which we do not address here.

The rationale that supports our approach is that while a global behavior of the software aging phenomena may be highly nonlinear, it may be composed (or approximated) by a reasonable number of linear patches, i.e., it may be piecewise linear. This rationale has been proposed in other related works like [114, 115]. This may well be the case for many system behaviors of the kind we want to analyze, where the system may be in one of a relatively small number of phases, each of which is essentially linear. This approach was extracted after analyzing the examples presented before. Although the software aging phenomena could be presented as a progressive resource exhaustion along time, this progressive consumption could have different trends over time or even could be masked by the application resource consumption pattern.

Our idea is that Machine Learning algorithms evaluated predict time to crash if the state of the system (including workload) does not vary in the future. However, if the situation changes (the consumption speed changes) the model has to be able to recalculate the time to crash under the new circumstances. For this reason, we added a set of derived metrics as variables to achieve a more accurate prediction. The most important variable we add is the consumption speed from every resource under monitoring (threads, system memory, web application memory and every Java Heap zone: Young and Old). To calculate the consumption speed (s_i in Formula 3.1), we decided to use an *average speed* to avoid too much fluctuations in the measure. We decided to use a sliding window average (also called moving average). The sliding window average collects the last X speed (speed is calculated as the difference between previous and current resource/metric

value) observations from the resource and calculates its average, so as to smooth out noise and fluctuation. The choice of X is a certain trade-off: a long window is more noise tolerant, but also makes the method slower to reflect changes in the input. It must be set by considering the expected noise and the frequency of change in our scenario.

Our process to build our prediction models is depicted in Figure 3.6.

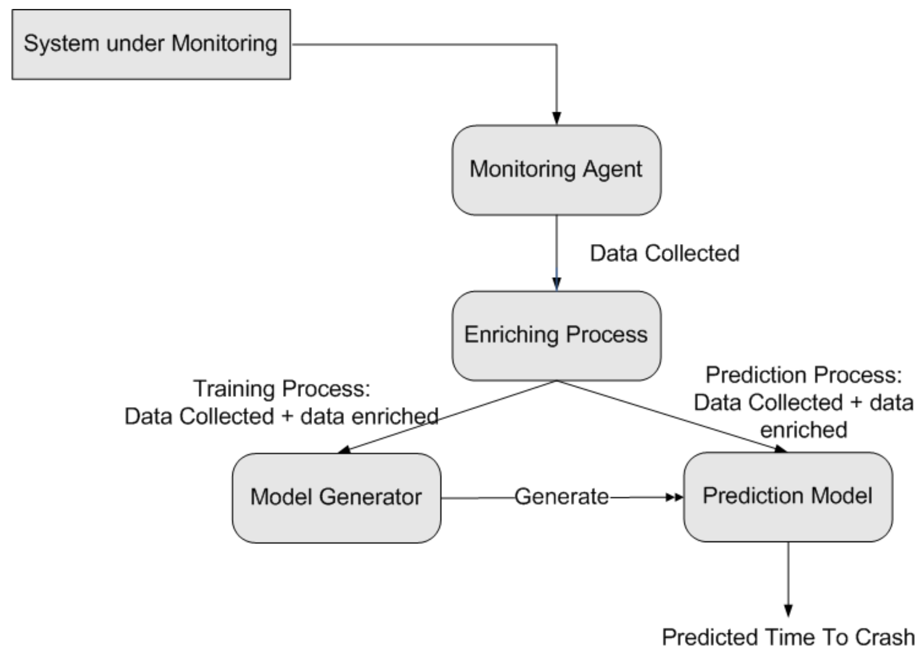


Figure 3.6 Detailed Architecture of the Prediction Framework

The *Monitoring Agent* is responsible of collecting system metrics. We have decided to collect a small set of metrics easily available from any operating system. The metrics are: Throughput, Response time, workload, System load, disc usage, swap space used, number of processes, the number of threads, free system memory, memory occupied by the running application/s[1], the number of http connections received, and the number of connections to the database. To collect all of this information, we have modified Nagios [89]. It is a well-known and industry-used monitoring tool that allows one to execute monitoring gadgets in the system every n seconds (in our case, we chose $n = 15$ seconds) and store the results in a file or database.

The framework must be thought to eventually run on-line. Although, so far we have implemented only an off-line process. More precisely, we collect all the data from one full experiment using the monitoring tool. After that, using an expert, we identify when a crash occurs in the dataset. The new dataset is used as input for the *Enriching process*, which has the main task to add variables derived from the system metrics monitored. The

[1] it will be Tomcat, in our experiments.

most important derived metric is the sliding window average described earlier for each metric or resource and its consumption speed. In Appendix A, we present a detailed description list of the system and derived metrics used in our experiments. The idea is to be able to detect potential changes of the consumption of any resource. Furthermore, we add some more expert information used to train the Machine Learning system. The *Enriching process* also adds a column indicating, for each measurement (i.e. observation or instance) in the dataset, the real time to crash at that moment. We know that value because an expert identified the crash in the execution data collected.

The resulting data set is used to build the model, which would be validated using different data sets. The framework has been designed to allow us enough flexibility to change the predictive model easily. Besides, it is prepared to develop a set of models and choose the best one according to different parameters selecting the more accurate result as well as having a prediction board where every model follows different strategies. For example, in [128], it is proposed an approach based on a general consent. They used four different prediction models and compare their predictions. If at least 3 of them agree in the same prediction, this is the output of the *prediction box*.

3.3 Experimental Setup

In this section we describe the Machine learning evaluation process conducted to select the best ML algorithm to predict an imminent crash and the experimental setup used to conduct the experiments needed to collect the training and validation datasets needed by the ML evaluation process.

3.3.1 Machine Learning Evaluation Process

In this subsection we present the Machine Learning evaluation process conducted. This process is different according to the goals in mind. When we want to estimate the prediction accuracy error, we may have two different goals in mind:

- *Model Selection*: Comparing different Machine Learning algorithms in order to choose the (approximately) best one.
- *Model Assessment*: Having chosen a final algorithm, estimate its generalization error on new data.

According to the goal (model selection or assessment) in mind, different tasks could be conducted. However, in a rich-data scenario, the best approach (if there is enough

data) to both model selection and model assessment goals is to divide the data set into three disjunct parts:

- Training data set needed to build (or fit) the models.
- Validation data set (or development data test set) used to estimate the test error for model selection.
- Test data set used for assessment of the generalization error of the finally chosen model.

The main goal of this chapter is to conduct an evaluation of different Machine Learning algorithms to chose the best one. Later, in Chapter 4, we are going to use the best algorithm found during current chapter to predict the time to crash and trigger the rejuvenation action in real environments. Due to this goal, we have conducted a model selection. Model selection process requires that ML algorithms given are trained with the training data set, and later compared according to the estimated test error using a completely different data set, called validation data set or development data test set. The training and validation process is presented in Figure 3.7 and described in detail below.

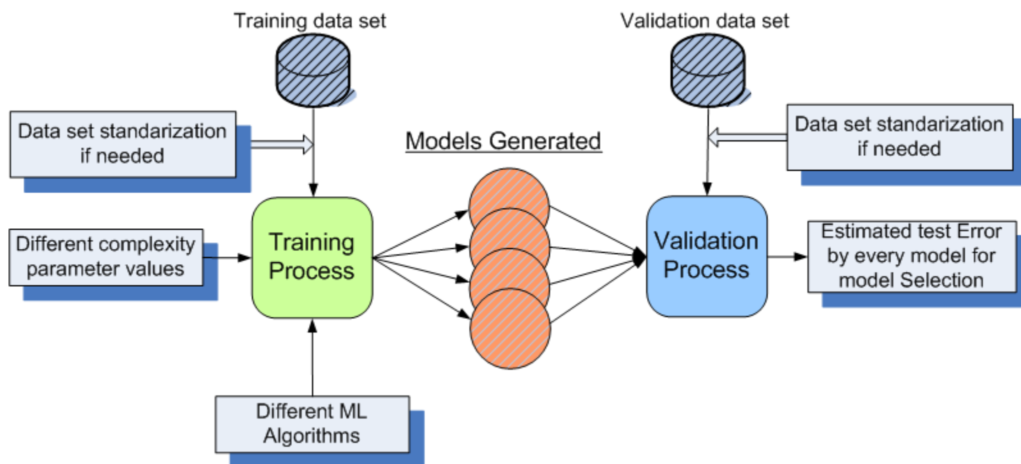


Figure 3.7 Machine Learning Model Selection process

3.3.1.1 Training Process

Our prediction model is trained using failure executions. The Training process does not really take into account the absolute value of the times, but the number of checkpoints (instances) we collect. If the resource degrades 100 times slower than in other environments, but we still measure the same number of checkpoints, the measurements of

the degrading resource will be the same, only spaced apart or a factor of 100 in time, the ML algorithms used in our experiments will build the same model, and the predictions will be equally good.

We have evaluated two different approaches: Time-based Prediction and "Red-light" alarm-based prediction. Some of the algorithms evaluated in both approaches have a set of attributes which can significantly influence the model building process and the prediction accuracy; mainly M5P, J48 and IBk. These algorithms are described later. On the other hand, Linear Regression and Naive Bayes used in our experiments do not have any complexity parameter. Based on a suitable model selection, we have trained every algorithm using different values of these attributes, called *complexity parameters* to avoid the name used by our machine learning package used. So, from every algorithm used we have built several (15 variations) models to evaluate not only what algorithm is the best one, but also what configuration of the algorithm is the best to our purposes. Besides the *complexity parameters*, some ML algorithms need to standardize or normalize the values before conducting the training, validation or testing. In our case, IBk is more sensitive of this fact. For this reason, we have normalized the values of the data sets using the standardization approach. This process consists in subtracting from the value its average value and divide the result by its standard deviation.

3.3.1.2 Validation Process

The training data set and validation data set are completely disjoint sets, using different previous executions. This approach is the common way to train and validate the models in Machine Learning, although there are other approaches such as percentage split or cross validation which use a subset of the training data set for validation.

In our experiments we showed the results of the best model. Although, in some of them we also present the results of other models for comparison proposes. Moreover, in every experiment we indicate the value of the complexity parameter used to obtain the results.

In order to select the best model, we have used different metrics such as Confusion Matrices (in the case of classifiers) and Mean Absolute Error (in the case of predictors) in order to compare different models. The comparison conducted is described in detail later in every evaluation conducted.

3.3.2 Experimental Environment

In this subsection we describe the experimental setup used in all experiments presented below. The main goal of our experiments is to evaluate what ML algorithm evaluated is the best to predict the time to crash. The experimental environment simulates a real web environment, composed of the web application server, the database server and the client machines. Finally, to simulate the client workload we have a machine with the client simulator installed.

We have used a multi-tier e-commerce site that simulates an on-line book store, following the standard configuration of TPC-W benchmark [112]. We have used the Java version developed using servlets and MySQL [88] as database server. As application server, we have used Apache Tomcat [16]. TPC-W allows us to run experiments using different parameters and under a controlled environment. These capabilities are perfect to conduct the evaluation of our approach to predict the time to crash. Details of machine characteristics are given in Table 3.1. Figure 3.8 presents the experimental environment components involved in the experiments.

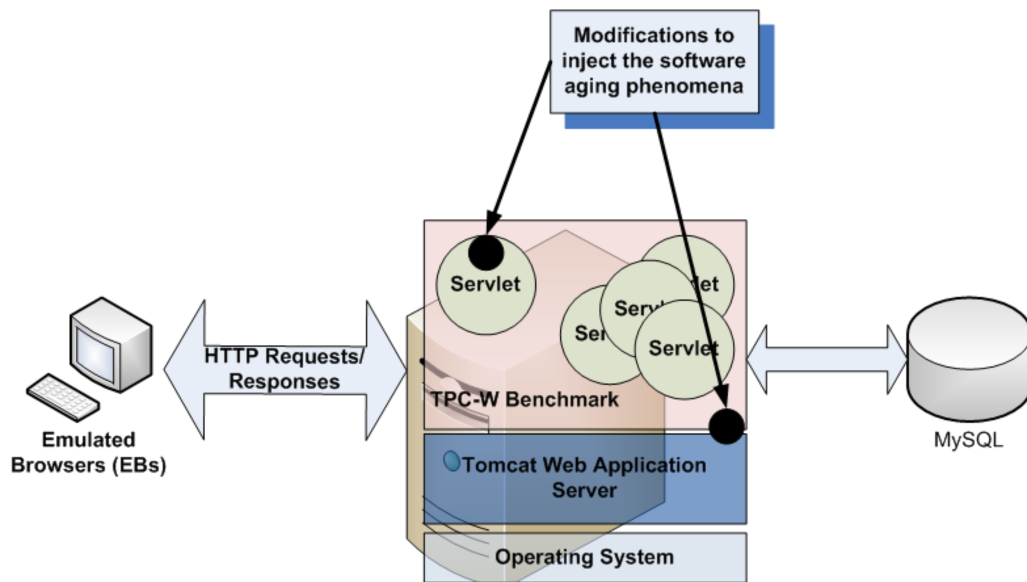


Figure 3.8 Experimental environment components.

TPC-W clients, called Emulated Browsers (EBs), access the web site (simulating an on-line book store) in sessions. A session is a sequence of logically connected (from the EB point of view) requests. Between two consecutive requests from the same EB, TPC-W computes a thinking time, representing the time between the user receiving a web page she/he requested and deciding the next request. In all of our experiments we have used the default configuration of TPC-W. Moreover, following the TPC-W specification, the

number of concurrent EBs is kept constant during experiments.

	Clients	Application Servers	Database server
Hardware	2-way Intel XEON 2.4 GHz with 2 GB RAM	4-way Intel XEON 1.4 GHz with 2 GB RAM	2-way Intel XEON 2.4 GHz with 2 GB RAM
Operating System	Linux 2.6.8-3-686	Linux 2.6.15	Linux 2.6.8-2-686
JVM	-	jdk1.5 with 1GB heap	-
Software	TPC-W Clients	Tomcat 5.5.26	MySQL 5.0.67

Table 3.1 Detailed experimental Setup Description

To simulate the aging-related errors consuming resources until their exhaustion, we have modified the TPC-W implementation. In our experiments we have played with two different resources: Threads and Memory, individually or merged. To simulate a random memory consumption we have modified a servlet (***TPCW_search_request_servlet***).

We have added a piece of code within the servlet to compute a random number between 0 and N. This number determines how many requests can use the servlet before the next memory leak is injected. Therefore, the variation of memory leak depends on the number of clients and the frequency of servlet invocations.

According to the TPC-W specification, this frequency depends on the workload chosen. Hence under high workload our servlet injects memory leaks quickly. However under low workload, the leak rate is lower. But, on the long term the average extra leak rate would depend on the average of random variable N, with fluctuations that become less relevant when averaged over time. Therefore, we could thus simulate this effect by varying N. We have decided to stick to only one relevant parameter, N.

On the other hand, to simulate a thread consumption in the servlet we use two parameters: T and M. At every injection, our modification injects a random number of threads between 0 and M, and determines how much time occurs until the next injection, a random number (in seconds) between 0 and T. Thread injection is independent of the workload (since injection occurs independently of the running applications), while memory leak is workload dependent (because it occurs when a certain application component is executed). Figure 3.8 presents the modifications conducted to simulate the aging phenomena in TPC-W benchmark.

These two errors help us to validate our hypothesis under different scenarios. TPC-W has three types of workload (Browsing, Shopping and Ordering). In our case, we have conducted all of our experiments using shopping distribution.

Table 3.2 presents the variables used to build every model used in our experiments presented in Sections 3.4 and 3.5. In these two sections, we present the same four

	Experiment 1	Experiment 2	Experiment 3	Experiment 4
Throughput(TH)	X	X	X-X [a]	X
Workload	X	X	X-X	X
Response Time	X	X	X-X	X
System Load	X	X	X-X	X
Disk Used	X	X	X-X	X
Swap Free	X	X	X-X	X
Num. Processes	X	X	X-X	X
Sys. Memory Used	X	X	X-	X
Tomcat Memory Used	X	X	X-	X
Num. Threads	X	X	X-X	X
Num. Http Connections	X	X	X-X	X
Num. Mysql Connections	X	X	X-X	X
Max. MB Young/Old (2)[b]		X	X-X	X
MB Young/Old Used (2)		X	X-X	X
% Used Young/Old Used (2)		X	X-X	X
SWA[c] Young/Old variation(2)		X	X-X	X
SWA variation (3)[d]	X	X	X-[h]	X
SWA variation /TH (2)[e]	X	X	X-	X
SWA variation /TH (2)[f]		X	X-X	X
1/SWA (3)[d]	X	X	X-[h]	X
1/SWA (2)[f]		X	X-X	X
Young/Old Used/SWA (2)		X	X-X	X
Resource Used(R)/SWA (3)[d]	X	X	X-[h]	X
(1/SWA variation)/TH (2)[e]	X	X	X-	X
(1/SWA variation)/TH (2)[f]		X	X-X	X
(R/SWA variation)/TH (2)[e]	X	X	X-	X
(R/SWA variation)/TH (2)[f]		X	X-X	X
SWA Resource Used (4)[g]	X	X	X-X	X
Time to Failure	X	X	X-X	X

[a] Exp. 5.3 Complete-Exp. 5.3 Feature Selection
[b] (X) number of variables represented
[c] Sliding Window Average (SWA)
[d] For Num. Threads, Tomcat Memory Used and System Memory Used
[e] For Tomcat Memory Used and System Memory Used
[f] For Young Zone Used and Old Zone Used
[g] For Response Time, Throughput, System Memory Used and Tomcat Memory Used
[h] Removed only Tomcat Memory Used and System Memory Used variables related

Table 3.2 Variables used in every experiment to build the model

experiments, but they are evaluated with different Machine Learning algorithms. In Table 3.2, we indicate the variables used (X) in every experiment. In Experiment 3, we have applied a feature selection, for this reason, the Experiment 3 column presents two X's. The second X indicates if the variable was used after the feature selection.

Moreover, in Appendix A, the reader can find a detailed description of every variable used to train the models and predict the Time To Crash (TTC).

3.4 Time-based prediction & its Evaluation

Our main goal is to evaluate the effectiveness of Machine Learning algorithms to predict the Time To Crash (TTC) (due to resource exhaustion causing the software aging phenomena) based on a limited set of system metrics easily available in any operating system. Among many Machine Learning algorithms and models available, we have chosen to evaluate two of the most well-known algorithms: Linear Regression and Model trees. Although it is not the scope of this document to present in detail these Machine Learning algorithms, we consider it important to present them briefly to understand their strengths and weaknesses. These two algorithms are included in the popular WEKA [121] machine learning and data mining package. Finally, if the reader is interested to learn more details about these models used in our experiments or other Machine Learning algorithms, we invite him/her to consult [124].

We have used Model Trees as our prediction algorithm because it follows our prediction rationale presented earlier: while a global behavior of software aging may be highly non-linear, it may be composed of (or approximately) a reasonable number of linear patches. Specifically, we have used an implementation of Model Trees called M5P extracted from the WEKA distribution. M5P offers a good trade-off between accuracy and training. Its accuracy is better than more sophisticated methods with small training data sets, as we discuss briefly in Appendix B. On the other hand, our final idea is to move our solution to an on-line predictive recovery framework. Our approach needs quick predictions with low computational cost and M5P accomplishes these constraints. Finally, other important point for choosing M5P is that M5P models are human interpretable. We will return to this fact, in Section 3.4.4 to show the relevance of this feature.

3.4.1 Linear Regression

A linear regression of variables $x_1, \ldots, x_n$ is a function of the form $\alpha_0 + \sum_{i=1}^{n} \alpha_i \cdot x_i$, for some set of coefficients $\alpha_0, \ldots, \alpha_n$. It will work well to predict some variable y when (and only when) y depends linearly on the x_i variables. The α coefficients are calculated

from the training data, trying to minimize the sum of the squares of these differences over all the training instances.

Linear regression is an efficient, simple method for numeric prediction, and it is widely used today to help for capacity planning analysis [129] or detecting anomalies on the system [39]. However, linear models have the disadvantage of relying too much on linearity. If the training dataset used exhibits a nonlinear behavior, the best-fitting straight line will be found, where "best" is interpreted as the least mean-squared difference. But it may not fit the data very well, so the prediction could be useless to reflect nonlinear behaviors.[2].

3.4.2 M5P

M5P is introduced by Wang and Witten [120], who described it as "a rational reconstruction of C5, a method proposed by Quinlan (1992) for inducing trees of regression models". Additionally, M5P includes techniques borrowed from the highly successful CART method by Breiman et al. (1984) [31] for dealing with enumerated values. According to [120], it outperformed the original C5 on the scarce evaluation data available.

An M5P model consists of a binary *decision tree* whose inner nodes are labelled with tests of the form “variable $<$ value?”, and each leaf is labeled with a linear regression model (possibly using all variables available in the training dataset). Intuitively, M5P will work well if the input data set space can be divided into a small set of pieces such that the label variable depends approximately linearly on input variables in each piece.

The *complexity parameter* used to influence on the building process in the case of M5P is the minimum number of instances used to build every leaf of the tree.

3.4.3 Experimental Evaluation

In this section, using the sliding window presented earlier and the Machine Learning algorithms described, we have conducted a set of experiments to evaluate the effectiveness of our approach for complex software aging scenarios. We have used Mean Absolute Error (MAE) to measure the prediction accuracy: this is the average of the absolute difference between true values and predicted values. So, we are using absolute errors. An error of 200 seconds over a time-to-failure of 1000 seconds is not equivalent to an error of 120 seconds over 600 seconds. Although, in both cases the percentage error is 20%. However the second is much more critical (and bigger) for taking recovery actions.

[2]Extracted from [124].

However, predicting exactly the time to crash is probably too hard, even as a baseline. We have used another measure called the Soft Mean Absolute Error (S-MAE): We decided that if the model predicts a time to crash within a margin of 10% of the real time (named *security margin*), we count it as zero error. For example, if the real time until crash is 10 minutes, we assume 0 error if the model predicts between 11 minutes and 9 minutes. If the system predicts say 13 (or 7 minutes), we would count a 2-minute error (the absolute error). Of course, thresholds other than 10% are possible. It is clear that S-MAE is always smaller than MAE. Moreover, we have trained our models to be more accurate when the crash is imminent. For this reason, we have calculated the MAE for the last 10 minutes of every experiment (POST-MAE) and for the rest of experiment (PRE-MAE). The idea is that our approach should have lower MAE in the last 10 minutes than the rest of the experiment, showing that the prediction becomes more accurate when it is more needed to apply the recovery action.

S-MAE also forces the model to become more accurate when the crash is near. A secure margin of 10% over 10 minutes until crash allows us an acceptable error/margin of 1 minute. However, when the crash is in 20 minutes, the margin is 2 minutes. It makes sense since we want to become more accurate when the crash is imminent.

3.4.3.1 Deterministic Software aging

Our first approach was to evaluate the chosen ML algorithms to predict the time to crash due to deterministic software aging. We decided to inject a 1MB of memory leak with $N = 30$ (see Section 3.3). We trained our models, generated using the algorithms selected with previous 4 executions with 25 EBs, 50EBs, 100EBs and 200EBs, becoming 2776 instances from 32 variables. Linear Regression used 20 variables. M5P model generated was composed by a tree with 33 leaves and 30 internal nodes, using 10 instances to build every leaf. We have used 10 instances as the *complexity parameter* value because it is the best value to achieve the best prediction accuracy result in comparison with other values like 1, 5, 20, 30, 50, 100 and 200.

In this first experiment, we did not add the heap information (the information related with the memory consumption of the heap zones Old, young and permanent) as described in Table 3.2 because we wanted to use the usual information extracted from the system point of view, without any expert information. This experiment was thought as a base case to compare both algorithms. The four training experiments were executed until the crash of Tomcat, to let the models learn the behavior of the system under a deterministic software aging. Finally, to evaluate the accuracy of the models and their capabilities to adapt themselves to new scenarios, we evaluated the model built with these four

experiments using two new experiments with different workload (75EBs and 150EBs).

	Lin. Reg	M5P
75EBs MAE	19 min 35 secs	15 min 14 secs
75EBs S-MAE	14 min 17 secs	9 min 34 secs
150EBs MAE	20 min 24 sec	5 min 46 secs
150EBs S-MAE	17 min 24 secs	2 min 52 secs
75EBs PRE-MAE	21 min 13 secs	16 min 22 secs
75EBs POST-MAE	5 min 11 secs	2 min 20 secs
150EBs PRE-MAE	19 min 40 secs	6 min 18 secs
150EBs POST-MAE	24 min 14 secs	2 min 57 secs

Table 3.3 MAE obtained under constant software aging experiment

In Table 3.3 we present the results obtained. We can observe how M5P obtains better results than simple linear regression. M5P handles better the trend changes due to the Heap Memory Management actions, even when we don't add the specific information to build the model, as presented in the examples in Section 3.2.2.

From this experiment we can extract a first preliminary conclusion. We can observe how even the simplest software aging scenario M5P obtains better results than Linear Regression. We attribute the superiority of M5P over Linear Regression to the fact that the aging phenomena is definitively non linear, but possibly composed of several approximately linear regions. So, in the rest of the experiments we will evaluate in highest detail the accuracy of M5P in different and complex scenarios, but do not completely forget the Linear Regression.

Furthermore, in Appendix B, we present detailed results obtained using other more sophisticated Machine Learning algorithms in every experiment conducted in this Section.

3.4.3.2 Dynamic and Variable Software aging

Our next experiment was to evaluate our model to predict progressive but dynamic software aging under constant workload.

We trained the Machine Learning models with only 4 executions (1710 instances from 50 variables, adding the Java Heap information variables): one hour execution where we did not inject any memory leak and three executions where we injected 1MB memory leak with constant ratio ($N = 15$,$N = 30$ and $N = 75$ in every respective execution, see Section 3.3). Models were trained to determine as an infinite time to crash as 3 hours (10800 secs) using the training data set without injection. These models were validated over an experiment where we changed the ratio every 20 minutes. During the first 20

minutes we did not inject any memory leak. After that, during the next 20 minutes we injected a memory leak following a ratio of $N = 30$. After that, we increase the injection to $N = 15$ and finally, 20 minutes later, we reduced the software aging following a $N = 75$ and we left constant this ratio until crash. The experiment was running for 1 hour and 47 minutes.

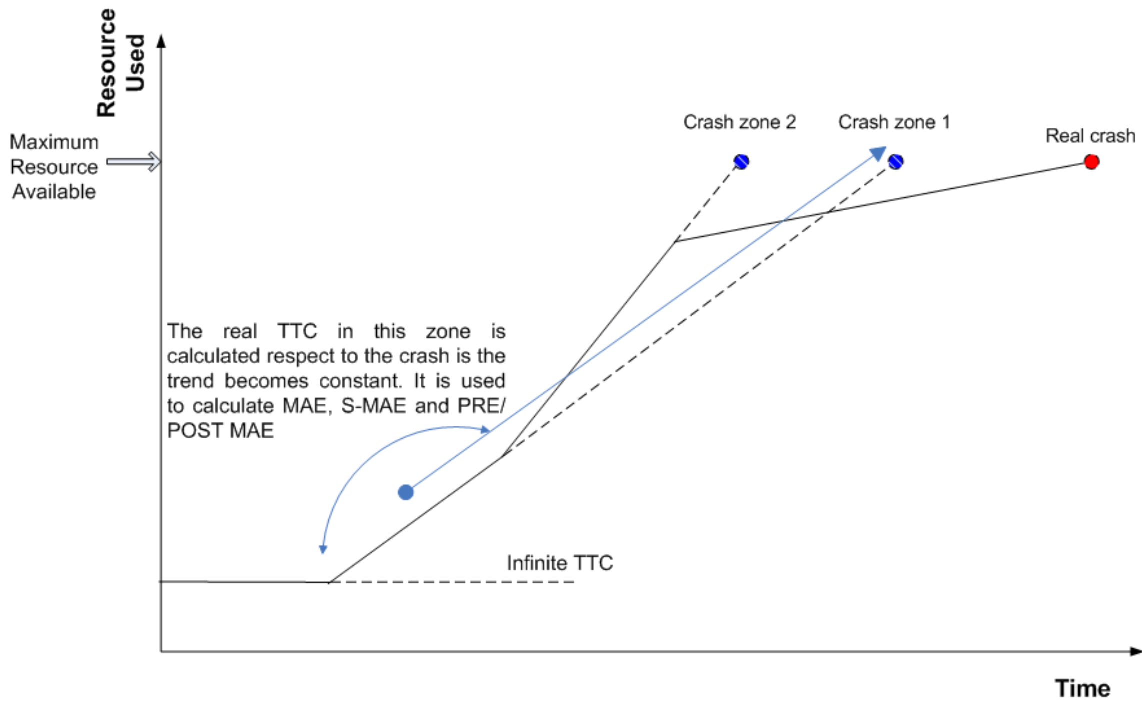


Figure 3.9 The MAE, S-MAE and PRE/POST MAE calculation process in dynamic experiments

We determine the real time until crash during the experiment as the real time until crash if the trend was constant (without changes) until crash. This "real time to crash" is used to calculate the MAE and the S-MAE (and of course PRE-MAE and POST-MAE). Figure 3.9 represents this way to calculate the MAE and S-MAE in dynamic and variable experiments.

So, apart from evaluating the prediction accuracy, we also wanted to evaluate the capability of the model to overcome software aging trend changes and react before it, so if the consumption speed goes down, the time until exhaustion increases and vice versa. We used 5 instances to build every leaf in the case of M5P. In this scenario, a value of 5 or 1 of a complexity parameter we obtain the best results on the average. However, using a value of 100 we reduce the error of the POST-MAE, but increasing significantly the MAE, S-MAE and PRE-MAE values. For this reason, we have used M5P with a complexity parameter of 5.

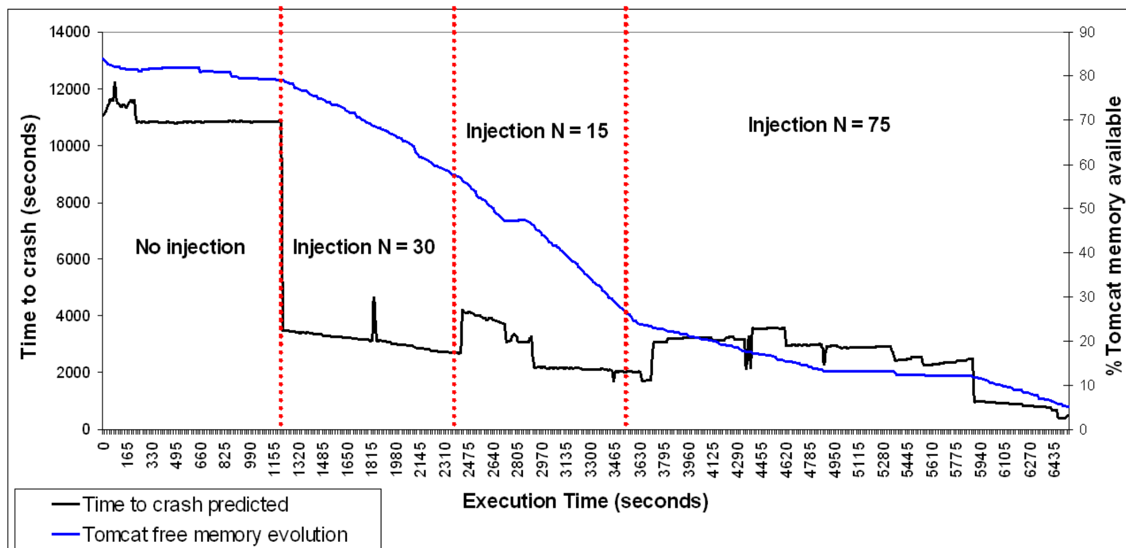


Figure 3.10 Time to crash predicted vs. Tomcat Memory Evolution using M5P

In Figure 3.10, we present the predicted time to crash using M5P(dark line) vs. Tomcat memory evolution (grey line) during the execution. First, we have trained our model to declare that the time to crash is 3 hours (standing for “very long” or “infinite”) when there is no aging; indeed, during the first 20 minutes the model predicts a 3-hour time to crash, meaning that no aging is occurring. After 20 minutes, we start injecting leaks; we can see that the Tomcat free memory starts decreasing gradually but predicted time to crash decreases drastically. The reason for this drastic adaptation is due to the construction of M5P or any regression tree. The starting injection phase causes a change of branch of the tree from non-aging ($>$ 3 hours to crash) to an aging branch, based on the aging trend. After another 20 minutes, the injection rate increases. The sliding window introduces some delay in detecting the trend changes. In our experiments 12 instances of monitoring (marks). We have used 12 value because after several validations, it seems the best option according to our experiments. So 12 marks * 15 seconds per monitoring instance, 180 seconds is the maximum delay to detect a reasonable change of trend. But a more important fact happens in the beginning of the third phase of the experiment: A flat zone caused by the Heap Management process, as discussed before.

This fact makes the model to start to overestimate the time to failure, because it is seeing a lower-than-real consumption rate, which translates to a larger-than-real predicted time. However, when the flat region ends (as visible in the the Tomcat memory plot), the model reacts quickly to adapt the time to crash close to the real value (around 2850 seconds after the start of the experiment). In the fourth phase, we reduce the rate and the model quickly adapts the time predicted to the new circumstances, increasing the

time until crash. Prediction in this region is not quite accurate, though: as the injection rate is so slow, the model has troubles detecting it, and it keeps the prediction almost constant. Furthermore, we can observe a second resizing (from 4850 seconds to 5900 seconds). During this phase, again the model is not accurate (as it does see a constant Tomcat memory usage), but when the Tomcat memory is out of the flat zone, again the model reacts reducing the time to crash. This behavior shows the adaptability to changes of M5P.

The MAE, S-MAE and PRE/POST-MAE obtained by M5P are presented in Table 3.4. The Linear Regression has a really unacceptable MAE results, achieving values near to hours of error. If we analyze in detail the results from Linear Regression, we can find that the errors are becoming from dramatically large outliers, more than a generalized error. These results are presented in Appendix B (Table B.2).

	M5P
MAE	16 min 23 secs
S-MAE	13 min 3 secs
PRE-MAE	17 min 13 secs
POST-MAE	8 min 14 secs

Table 3.4 MAE obtained under dynamic and variable software aging experiment

3.4.3.3 Software Aging Hidden within Periodic Resource Behavior

Our next step was to evaluate the models in front of a deterministic software aging masked by a periodic pattern of memory acquisition followed by memory release. Aging here means that not all memory allocated during the memory acquisition phase was later released, so memory leaks accumulate over time. Our experiment was the same we conducted in the second motivating example. But, now we modify the release phase to guarantee that after these 20 minutes some memory was retained, so a crash was bound to happen after several periodic phases. The workload was constant with 100EBs. As we can observe, the memory leak in fact is quite constant but we introduce a periodic behavior pattern introducing some noise (the released phases and non-injection phases). The injection phase follows $N = 30$ and release phase follows $N = 75$, allocating and releasing 1MB each time. We trained the models using the same training set as in Section 3.4.3.2. So, the training set does not have any execution with release phase or periodic patterns. Our idea is that ML algorithms can manage the periodic pattern and

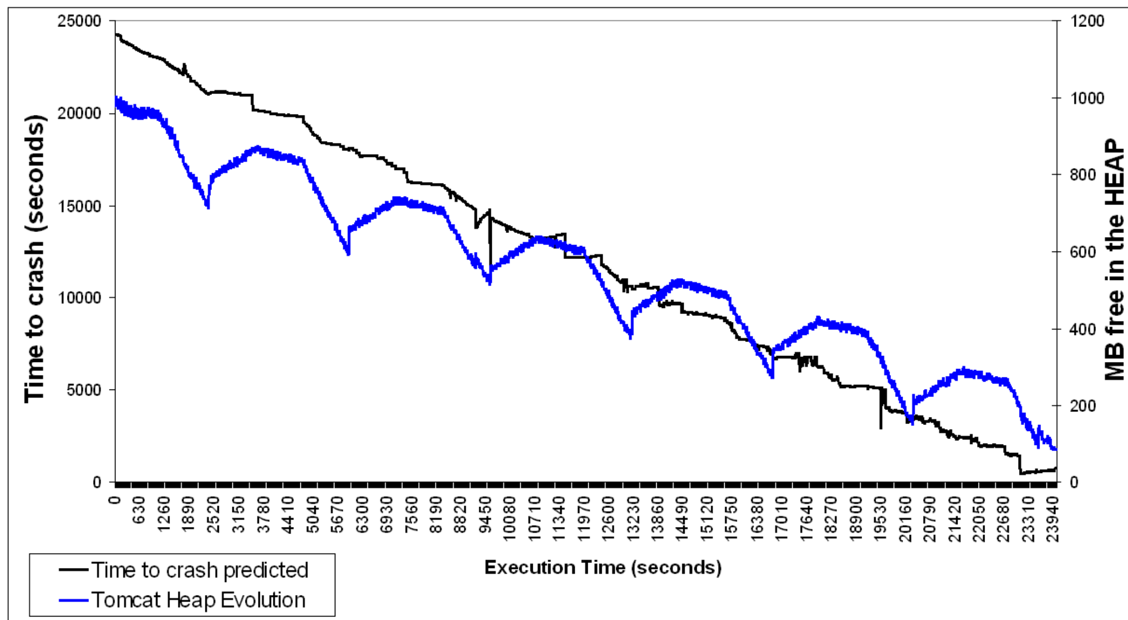


Figure 3.11 Time predicted vs. Java Heap Tomcat Memory Evolution

extract from that, the real trend (the random injection between 0 to N). However, we obtained poor results in our first approach in all of them, and after some inspection we noted that the models were paying too much attention to irrelevant attributes. We decided to apply an expert feature/variable selection following the conclusion extracted in [61], where the authors concluded that selection of a good subset of variables increases the prediction accuracy. We decided to re-train the model only with the variables related with the Java Heap evolution, removing the rest of memory-related variables, obtaining only 34 variables. The new model generated by M5P was formed by 17 inner nodes and 18 leafs. In Figure 3.11 we can observe clearly how the time to crash is linear and how the Java Heap memory is being consumed and released in every phase until the memory resource is exhausted.

Table 3.5 shows the MAE, S-MAE, PRE-MAE and POST-MAE obtained by M5P and the Linear Regression. We note that M5P can manage the periodic pattern, unlike Linear Regression which obtains much worse results even after variable selection. However, we notice that, in this case, M5P has problems to be accurate in the last 10 minutes of the experiment.

This experiment has opened a new research line we are currently working on. The feature selection applied here was by an human expert. We are working to evaluate different mathematical techniques such as Lasso Regression [41],[100],[30],[28] to apply automatic (without human intervention) feature selection in order to reduce the number of parameters under monitoring. This would reduce the performance overhead introduced

	LinReg	M5P
MAE	15 min 57 secs	3 min 34 secs
S-MAE	4 min 53 secs	21 secs
PRE-MAE	16 min 10 secs	3 min 31 secs
POST-MAE	8 min 14 secs	5 min 29 secs

Table 3.5 MAE obtained in the periodic pattern experiment

by the monitoring tools and more importantly, it will reduce the model complexity.

3.4.3.4 Dynamic and Variable Software aging due to Two Resources

After evaluating our approach to manage the aging due to a single resource, our next step was to evaluate it to determine the time until crash when the Tomcat server is suffering a software aging caused by two resources simultaneously.

The two resources involved in the experiment were Memory and Threads, and every phase was around 30 minutes. The experiment was conducted as follows: we have a first phase with no injection (both resources), after that we start to inject both resources: the memory rate injection was $N = 30$, while the thread rate injection was $M = 30$ and $T = 90$. After that, we increased the memory rate injection to $N = 15$ and the thread rate injection was reduced to $M = 15$ and $T = 120$. After that, in the last phase, we change again both rates, reducing the memory rate ($N = 75$) and increasing the thread rate injection ($M = 45$ and $T = 60$).

One important point is that when a Java Thread is created, a system thread is assigned to it until it dies. Moreover, every Java Thread has an impact over the Tomcat Memory, because the Java thread consumes a bit of Java memory in its creation. So, although the two causes of aging are unrelated on the surface, they are related after all; we believe this may be a common situation, and that it may easily go unnoticed even to expert eyes.

In Figure 3.12 we can observe the thread consumption and memory consumption evolution and the four phases clearly of the experiment. The MAE and S-MAE obtained by M5P (complexity parameter of 20 instances per leaf) in this experiment was: 16min. 49secs. and 13min. 17secs respectively: this is about 11% S-MAE error, given that the experiment took 1 hour and 55 minutes until crash. Furthermore, the PRE-MAE and POST-MAE was: 18 min. 12 secs. and 2 min. 4 secs. M5P is able to predict with great accuracy the time to crash, when it is near. This fact is important because in our opinion it is most relevant that the prediction was more accurate when the crash is imminent. For example, if the predictor said that the crash is in 5 hours but the reality is 5 hours

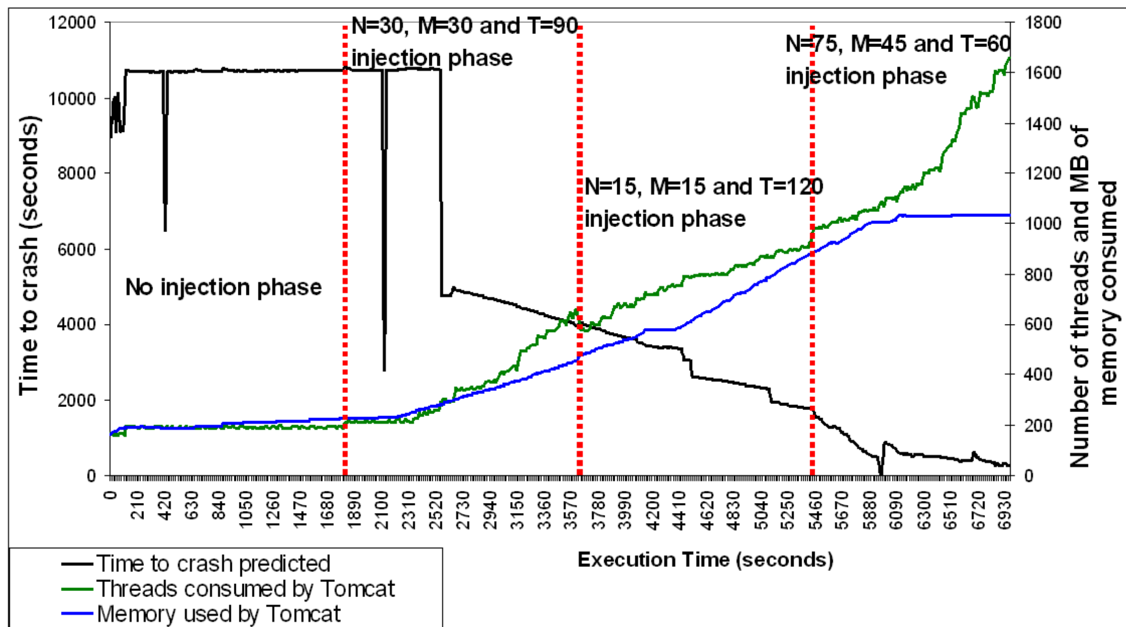


Figure 3.12 Time predicted and resource evolution during the two-resource experiment

and 15 minutes, the error is not relevant. However, if the predictor said the crash is in 25 minutes and the error is 10 minutes, this is relevant to the availability of the service. Most importantly, *the model never was trained using executions where both resources were injecting errors simultaneously*. We trained the model with several executions at different (constant) workload and different (constant) injection rates ($N = 15, 30, 75$, $M = 15, 30, 45$ and $T = 60, 90, 120$, so 6 executions, 2752 instances and 50 variables), but in all of them only one resource was involved in the execution. The model generated was composed of 35 internal nodes and 36 leaves. These results show the promising adaptability of this approach to new situations not seen during previous training. Linear Regression results become unacceptable under MAE again, and for the same reason of the experiment with only one dynamic resource involved: outliers. The results are presented in Appendix B.

In the future work section (Chapter 5), we want to evaluate different approaches to reduce the noise of outliers. We have observed how these rare values make that the prediction results were unacceptable to guarantee the availability of the system via a predictive recovery mechanism. Moreover, it could be interesting to evaluate other metrics to measure the prediction accuracy.

```
M5 pruned model tree:
(using smoothed linear models)

SYSTEM_MEMORY <= 1306.417 :
|   TOMCAT_MEMORY <= 57.022 :
|   |   THREADS <= 2172.5 :
|   |   |   SYSTEM_MEMORY <= 1293.5 : LM1 (130/2.001%)
|   |   |   SYSTEM_MEMORY >  1293.5 :
|   |   |   |   THREADS <= 1992.5 : LM2 (69/5.792%)
|   |   |   |   THREADS >  1992.5 : LM3 (54/4.469%)
|   |   THREADS >  2172.5 : LM4 (70/23.57%)
|   TOMCAT_MEMORY >  57.022 :
|   |   THREADS <= 2461.5 :
|   |   |   TOMCAT_MEMORY <= 86.58 :
|   |   |   |   THREADS <= 2387.5 :
|   |   |   |   |   TOMCAT_MEMORY <= 58.861 : LM5 (34/8.189%)
|   |   |   |   |   TOMCAT_MEMORY >  58.861 :
|   |   |   |   |   |   SYSTEM_MEMORY <= 817.5 : LM6 (37/2.903%)
|   |   |   |   |   |   SYSTEM_MEMORY >  817.5 : LM7 (95/4.705%)
|   |   |   |   THREADS >  2387.5 : LM8 (45/3.514%)
|   |   |   TOMCAT_MEMORY >  86.58 : LM9 (82/6.961%)
|   |   THREADS >  2461.5 : LM10 (83/6.487%)
SYSTEM_MEMORY >  1306.417 :
|   THREADS <= 1467.5 :
[.....]
```

Figure 3.13 M5P Model in the experiment 5.4

3.4.4 Understanding M5P generated Models

Finally, we want to remark another interesting point. After conducting all those experiments, we decided to inspect the models generated by M5P to check whenever the model could give clues to determine the root cause of failure or at least the resources related with the resource exhaustion which causes the system or application crash. Figure 3.13 shows a section of the tree built by M5P in the last experiment (two resources involved). As we can observe, the root node contains the system memory attribute (*SYSTEM_MEMORY*); if the used system memory is over 1306 MB, the second variable inspected is the number of threads in the system. But if system memory is below 1306 MB, the most relevant variable is Tomcat Memory and the only third is the number of threads. These first two levels of the tree indicate how relevant are the memory usage and the threads in the aging process. This gives administrators or developers a clue on the root cause of the failure which causes the software aging. So, interpreting the models generated via Machine Learning algorithms has an additional interest besides prediction.

3.5 "Red-light" Alarm-based Prediction & its Evaluation

In the previous section we have evaluated the effectiveness to predict the time to crash due to resource exhaustion using the Machine Learning algorithm M5P. Furthermore, we have shown the effectiveness of M5P to determine the most relevant metrics which cause the software aging. However, predicting numerically exactly or an approximation of the time to crash is probably too hard, even as a baseline. For this reason, we changed the prediction perspective. We are going to concentrate on the problem of detecting approaching and imminent crashes ("orange alerts" and "red alerts") rather than trying to produce accurate time-to-failure estimation. The alarm-detecting is the crucial element to apply the self-healing or demand the human operators attention. In this section we conduct an evaluation of a three well-known classifiers incorporated into the WEKA package [121]: J48, Naive Bayes and IBk. We have chosen these algorithms because they are the most commonly used in the Machine Learning area as a baseline. Other important reason for choosing these three algorithms is that they have a low computational cost with Naive Bayes being the cheapest and IBk the most expensive.

The classifiers are algorithms used to predict discrete values which represent classes, while M5P or Linear Regression are used to predict continuous values.

Our idea is to use ML classifiers to detect approaching (orange alert) and imminent (red alert) crashes. To allow for some flexibility, we distinguished three rather than two periods: Tuples in the training dataset were labeled Red, Orange, or Green as it is presented in Figure 3.14. Tuples in the last 10 minutes before the crash were labeled Red, those in the 10 minutes before the Red zone were labeled Orange, and all others (i.e., at least 20 minutes before the crash) were labeled Green. We still are primarily interested in distinguishing Red from non-Red, but we will not count Red tuples classified as orange as very severe errors. Red tuples classified as Green are probably the most expensive mistake, as they mean an imminent unpredicted crash. Green tuples classified as Red are false positives: they could mean starting preventive measures before they are strictly necessary or even without being necessary at all, which may be a nuisance but not as much as a crash.

We used WEKA's default options in all three algorithms. We remark our use of off-the-shelf classifiers to indicate that we included hardly no domain knowledge into the prediction system and that, obviously, there is ample room for improvement by designing ad-hoc algorithms.

The results are shown in the form of confusion matrix, indicating how many examples

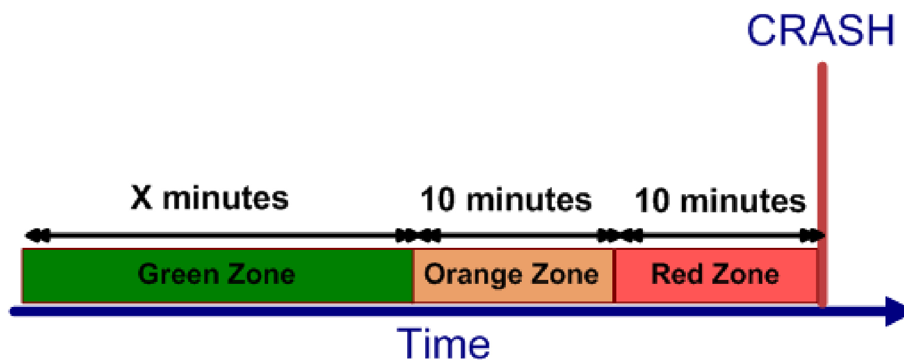


Figure 3.14 The three different prediction classes

of each class (red, orange, green) are classified as in each class. These matrices provide better information than simply the number of errors. Perfect predictions are those in the matrix diagonal. E.g., Red-as-Orange confusions are wrong, but intuitively not as wrong as Red-as-Green.

3.5.1 J48

J48 [101] is an implementation of the Quinlan's C4.5 decision tree inducer [102] for generating a pruned or unpruned C.4.5 decision tree. C4.5 is an extension of Quinlan's earlier ID3 algorithm. The decision tree generated is used for classification. J48 builds decision trees from a set of labeled training data using the concept of information entropy. It uses the fact that each attribute of the data can be used to make a decision by splitting the data into smaller subsets. J48 examines the normalized information gain (difference in entropy) that results from choosing an attribute for splitting the data. [3].

J48 complexity parameter is not an unique parameter like M5P. In this case, we have two parameters to build different models: *confidence factor* (C) and *minNumObj* (N). The first parameter allows us to influence on the pruning process. Lower value, more pruning. On the other hand, *minNumObj* works like the parameter of M5P, defines the minimum number of instances to build a leaf.

3.5.2 Naive Bayes

NaiveBayes is a probabilistic classifier based on Bayes' rule with strong naive independence assumptions. The classical naive Bayes model imposes a conditional independence constraint, namely, that the predictor variables, say, $x_1, ..., x_k$, are

[3]Extracted from the content of, and more information in website: http://www.opentox.org/ [consulted July, 2010]

conditionally independent given the response variable y. Extracted information and more details could be found in [127].

Naive Bayes does not have any complexity parameter.

3.5.3 IBk

IBk is a k-nearest neighbor (kNN) classifier. kNN is a method for classifying objects based on closest training examples in the feature space. It is a type of instance-based learning, or lazy learning where the function is only approximated locally and all computation is delayed until classification. A majority vote of an object's neighbors is used for classification, with the object being assigned to the class most common amongst its k (positive integer, typically small) nearest neighbors. [3]

IBk has as a complexity parameter the value of k.

3.5.4 Experimental Evaluation

To evaluate the effectiveness of the alarm-prediction methods we decided to use the new algorithms selected under the same set of experiments we used to evaluate M5P in front of Linear Regression (see Section 3.4).

The evaluation method to decide which algorithm is better is based mainly on the confusion matrix. We have also evaluated where the prediction errors are produced. If the method predicts Orange-as-Green in the frontier between both areas, it is not a big error, however if the method predicts Orange in the middle of a Green zone, this could trigger an unnecessary recovery action. Even more, if the prediction is Red-as-Green, the recovery action would not be triggered, becoming an imminent crash.

The confusion matrices presented in this document shows the best results obtained by every algorithm. Later, we compare the three best models to chose the best one.

3.5.4.1 Deterministic Software aging

As presented in Section 3.4.3.1, the first experiment checks our selected ML algorithms under a deterministic software aging situation. For more details of the experiment, see Section 3.4.3.1.

The results are summarized in Table 3.6 for workload with 75EBs and in Table 3.7 for 150EBs for three prediction algorithms. J48 complexity parameters were $C = 0.25$ and $N = 2$. The IBk complexity parameter k was 5.

As we can observe in Tables 3.6 and 3.7, the three algorithms have prediction errors. First, it is important to remark that for us a prediction error such as Red-

	Real Values			
		Green	Orange	Red
Predicted Values	Green	421/334/419 [a]	0/0/0	0/0/3
	Orange	32/114/35	1/1/20	0/5/2
	Red	1/6/0	19/19/0	19/14/14

[a] X/Y/Z, X J48, Y NB and Z IBk

Table 3.6 Confusion Matrix with 75EBs: J48/NB/IBk

	Real Values			
		Green	Orange	Red
Predicted Values	Green	210/166/189 [a]	8/0/0	0/0/1
	Orange	0/44/5	12/19/8	2/2/4
	Red	0/0/16	0/1/12	17/17/14

[a] X/Y/Z, X J48, Y NB and Z IBk

Table 3.7 Confusion Matrix with 150EBs: J48/NB/IBk

as-Orange or Orange-as-Red is acceptable, because both areas are dangerous for the availability of system. Following this approach we define a False Positive if the prediction model predicts Green-as-Red or Green-as-Orange and False Negative as Red-as-Green or Orange-as-Green.

In the case of 75EBs workload, we observe that:

- J48 and NB are very good to predict the dangerous zones (Red or Orange) but they are not able to distinguish between them. They predict quite all Oranges-as-Red (19 over 20). But they are very good to predict Red zone.

- IBk is perfect to predict the Orange zone, but it has a hard problem. It predicts 3 Red-as-Green (false negative), which makes that the system does not trigger the recovery action because the prediction says that the system is working perfectly (Green zone).

- The three algorithms are not bad to predict the Green zone: being J48 the best with only 32 Green-as-Orange (False positives) and NB the worst with 114 false positives.

In the case of 150EBs workload, we observe that:

- The three algorithms are good to predict Red zone, but again IBk has one Red-as-Green prediction: A dangerous prediction for the availability of the system.

- The three algorithms are better to predict Orange Zone, although J48 has some problems with 8 Orange-as-Green.

- The three algorithms are better than previous case to predict the Green zone, maintaining the same accuracy order.

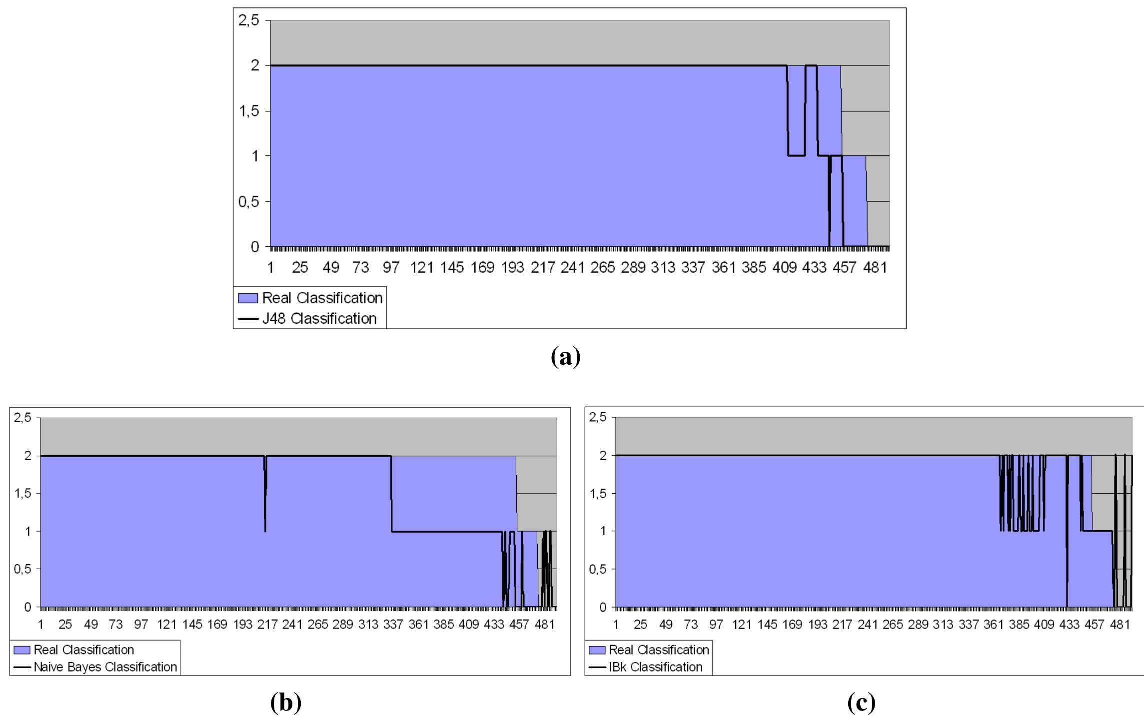


Figure 3.15 Alarm classification results for 75 EBs workload, (a)J48, (b) Naive Bayes, and (c) IBk

Although confusion matrices are a good way to analyze the predictors' accuracy, in our case, it is also important knowing when the classification errors are made. It is not the same if the error is the frontier between two areas or is in the middle of an area. For this reason, we analyzed where the errors were produced.

In Figure 3.15 we present alarm-prediction of three algorithms under evaluation. To help in the process of understand these figures, the blue space represents the real classification where a 2 means the green zone, 1 is the orange zone, and 0, the red zone. The x-axis represents the instances of time. Every mark represents a prediction. We conduct a prediction every 15 seconds. And the black line represents the prediction classification. In this way, we can observe where (according to the time to crash) the prediction fails and the level of the fail.

Figure 3.15 offers a complementary and valuable information to the confusion matrices previously presented. If we observe the confusion matrices alone we could

conclude, that J48 and NB algorithms achieve more or less the same level of accuracy. However, under this new perspective of the same numbers, we can conclude that J48 is a better option for the workload of 75EBs. We can observe how Naive Bayes (NB) and IBk have unacceptable prediction. Naive Bayes (NB) starts to predict Orange zone too early, quite 40 minutes before the real Orange zone. IBk is worse because during the real Red zone, IBk predicts some Green zones, becoming in false negatives which could seem that the system is working normally, when the crash is imminent.

In Figure 3.16, we present the prediction of three algorithms under 150EBs of workload. Again, we can observe how J48 is better option. In Figure 3.16a, J48 starts to predict Orange alarm a bit later (2 min and 10 seconds later), but it avoids false positive and false negatives. When the J48 classifies Orange zone never comes back to Green zone, avoiding this dangerous behavior. However, in Figures 3.16b and 3.16c, we can observe how Naive Bayes and IBk produce false positives (IBk mainly) and false negatives (Naive Bayes and IBk).

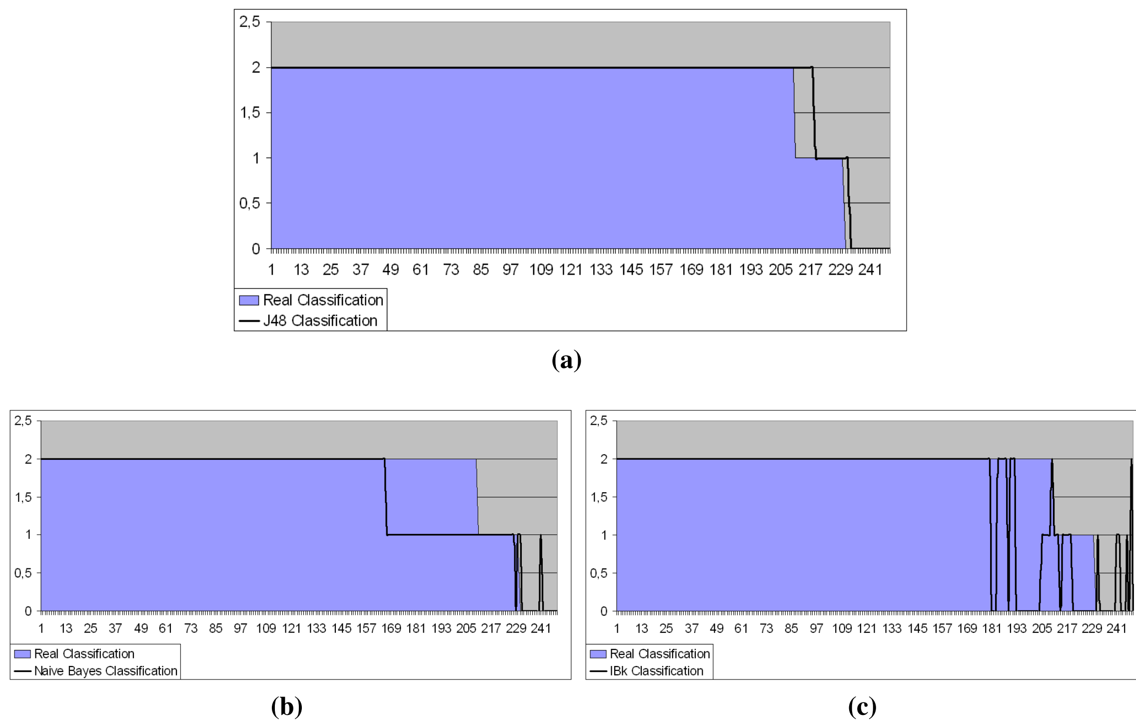


Figure 3.16 Alarm classification results for 150 EBs workload, (a)J48, (b) Naive Bayes, and (c) IBk

3.5.4.2 Dynamic and Variable Software aging

In this section we present the results using the classifiers selected under the same scenario presented in Section 3.4.3.2. We trained the model using the same experiments

and the same number of instances as used with M5P or Linear Regression. In Table 3.8 we present the confusion matrix obtained for every algorithm with poor results. The algorithms predict Green in essentials all instances. In detail, only J48 is able to predict twice orange zone instead of red zone while Naive Bayes and IBk once.

		Real Values		
		Green	Orange	Red
Predicted Values	Green	397/397/397 [a]	20/20/20	17/18/18
	Orange	0/0/0	0/0/0	2/1/1
	Red	0/0/0	0/0/0	0/0/0

[a] X/Y/Z, X J48, Y NB and Z IBk

Table 3.8 Confusion Matrix of Dynamic and Variable Software aging: J48/NB/IBk

```
J48 pruned tree
------------------

TOMCAT_MEMORY_EWMA <= 94.253333: NF (1630.0/2.0)
TOMCAT_MEMORY_EWMA > 94.253333
|   MEMORY_SYSTEM_EWMA <= 726.833333
|   |   OLD_PERCENTAGE_USED <= 99.904373: P (27.0)
|   |   OLD_PERCENTAGE_USED > 99.904373
|   |   |   NORMALIZED_TOMCAT_MEMORY_VARIATION_EWMA <= 0.001038
|   |   |   |   SYSTEM_LOAD <= 0.13: P (7.0)
|   |   |   |   SYSTEM_LOAD > 0.13
|   |   |   |   |   NORMALIZED_OLD_MEMORY_VARIATION <= 0.000342: F (30.0)
|   |   |   |   |   NORMALIZED_OLD_MEMORY_VARIATION > 0.000342
|   |   |   |   |   |   TOMCAT_MEMORY <= 94.92: P (4.0)
|   |   |   |   |   |   TOMCAT_MEMORY > 94.92: F (5.0)
|   |   |   NORMALIZED_TOMCAT_MEMORY_VARIATION_EWMA > 0.001038: NF (2.0)
|   MEMORY_SYSTEM_EWMA > 726.833333: NF (5.0)

Number of Leaves  :     8

Size of the tree :  15
```

Figure 3.17 J48 Model generated using the Training data set in Dynamic and Variable Software aging experiment

In this case, we avoid to present the figures to know where the errors are. The confusion matrix is clear to show us the unacceptable prediction level from the three Machine Learning algorithms evaluated. The reason of these poor results could become from an inappropriate training dataset, or an outlier validation dataset or because the system behavior is unpredictable. The last option could be ruled out because M5P obtained quite good results.

Our first step was to analyze the models generated by J48 and Naive Bayes, the unique two models human-friendly. Figure 3.17 presents the J48 model. There, we observed

that J48 starts to predict orange (P in the model, as it is presented in Figure 3.17) when the memory consumed by Tomcat (TOMCAT_MEMORY_EWMA) was over 94.25%. Before it, the models predict green. The Naive Bayes model had similar conclusion. After that, we analyzed the validation data set observing an important fact. Only the last two instances of the validations data achieve the boundary fixed by the training data set to pass from Green to Orange zone as it is presented in Figure 3.18.

```
Plot : Master Plot
Instance: 432
        SYSTEM_LOAD : 0.14
        TOMCAT_MEMORY : 94.31
        OLD_PERCENTAGE_USED : 98.88235174408001
        OLD_MEMORY_VARIATION : 0.09166937934027779
        NORMALIZED_OLD_MEMORY_VARIATION : 8.041173626340157E-4
        MEMORY_SYSTEM_EWMA : 677.6666666666666
        TOMCAT_MEMORY_EWMA : 93.90166666666666
        NORMALIZED_TOMCAT_MEMORY_VARIATION_EWMA : 1.052631578947367E-4
        FAULT : F

Plot : Master Plot
Instance: 433
        SYSTEM_LOAD : 0.16
        TOMCAT_MEMORY : 94.55
        OLD_PERCENTAGE_USED : 99.14155586575846
        NORMALIZED_OLD_MEMORY_VARIATION : 8.890999688042535E-4
        MEMORY_SYSTEM_EWMA : 675.6666666666666
        TOMCAT_MEMORY_EWMA : 94.10333333333334
        NORMALIZED_TOMCAT_MEMORY_VARIATION_EWMA : 1.0503472222222168E-4
        FAULT : F

Plot : Master Plot
Instance: 434
        SYSTEM_LOAD : 0.14
        TOMCAT_MEMORY : 94.68
        OLD_PERCENTAGE_USED : 99.29519481496541
        NORMALIZED_OLD_MEMORY_VARIATION : 0.0020041869165372196
        MEMORY_SYSTEM_EWMA : 674.1666666666666
        TOMCAT_MEMORY_EWMA : 94.27333333333335
        NORMALIZED_TOMCAT_MEMORY_VARIATION_EWMA : 1.9883040935672714E-4
        FAULT : F

Plot : Master Plot
Instance: 435
        SYSTEM_LOAD : 0.15
        TOMCAT_MEMORY : 94.89
        OLD_PERCENTAGE_USED : 99.48745471898818
        NORMALIZED_OLD_MEMORY_VARIATION : 0.05900743272569381
        MEMORY_SYSTEM_EWMA : 672.6666666666666
        TOMCAT_MEMORY_EWMA : 94.455
        NORMALIZED_TOMCAT_MEMORY_VARIATION_EWMA : 0.006055555555555574
        FAULT : F
```

Figure 3.18 The last 4 tuples of validation experiment.

This clue indicates that the boundary fixed by training data set was not good. We analyzed the training data set from this new perspective, to know what was the cause to fix this boundary. The training data set had 1710 instances from which only 2.5% was orange and other 2% was red instanteces. As a consequence, ML algorithms pay too much attention to the green zone compared to the rest of zones, so the models generated

were ready to predict mainly green.

We decided to enrich the file with more orange and red instances, repeating the orange and red zone of every experiment several times until achieving 20% of orange instances and 17% of red instances from the whole new training dataset composed by a total of 2565 instances.

		Real Values		
		Green	Orange	Red
Predicted Values	Green	364/270/336 [a]	0/1/4	0/0/1
	Orange	33/126/51	20/19/16	16/17/17
	Red	0/1/10	0/0/0	3/2/1

[a] X/Y/Z, X J48, Y NB and Z IBk

Table 3.9 Confusion Matrix of Dynamic and Variable Software aging: J48/NB/IBk with 14% of orange instances and 11% of red instances

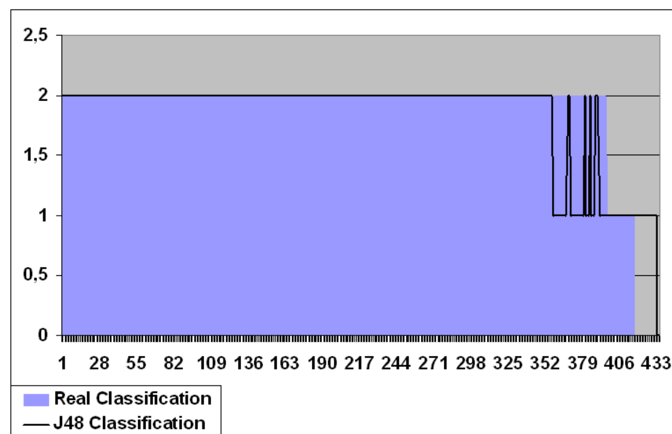
Table 3.9 presents the new results obtained, observing a clear improvement with regard to the results presented in Table 3.8. Only repeating the instances from orange and red zones and playing with the complexity parameters of J48 and IBk. In the case of J48, $C = 0.0000001$ and $N = 100$ and IBk case, $k = 10$, we increase the prediction accuracy.

We can observe clearly how J48 is able to differentiate dangerous zones (red and orange) from green zone. On the other hand, NB and IBk are also good to determine the red and orange zones. We can observe how we have introduced a penalty on green zone. We have increased the error of green zone prediction to increase the accuracy of the dangerous zones, more important for our final main goal: trigger the rejuvenation action when it is really needed.

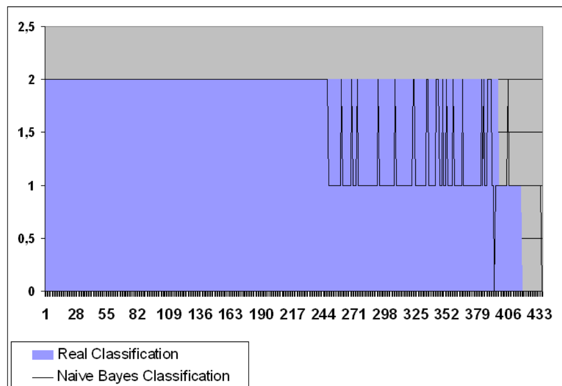
The technique of repeating important (for us) instances has a limit in our case. For example, if we decided to increase again the number of instances to 20% of red instances and 25% of orange, obtaining a new training data set composed by 2565 instances, we obtained the same results presented in Table 3.9. There is a limit in this strategy according to the current training data set.

Now, using these new acceptable results, we analyze when the errors are presented in Figure 3.19.

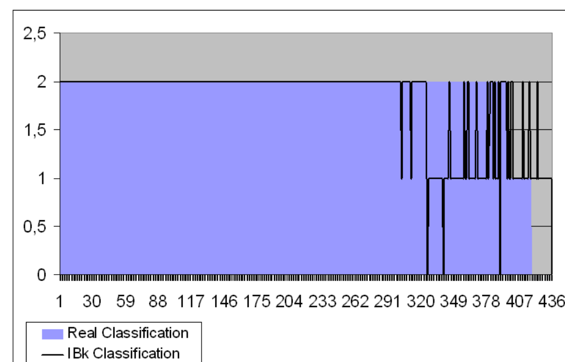
Analyzing in detail the confusion matrices and the Figure 3.19, we can observe how the three algorithms produce false positives and false negatives. Figure 3.19 reveals clearly how in this case, though we have improved the accuracy significantly, and we can conclude that J48 is able to predict dangerous (red and orange alarms) state with good accuracy becoming useful to guarantee the availability of the system.



(a)



(b)



(c)

Figure 3.19 Alarm classification results for a dynamic and variable software aging scenario, (a)J48, (b) Naive Bayes, and (c) IBk.

3.5.4.3 Software Aging Hidden within Periodic Resource Behavior

After evaluating the algorithms under constant and dynamic software aging scenarios, we conducted the software aging hidden within periodic resource behavior experiment. We used the same scenario presented in Section 3.4.3.3.

Our first test was to evaluate the classifiers without any feature selection, applying the raw datasets. In Table 3.10 we present the confusion matrix obtained.

		Real Values		
		Green	Orange	Red
Predicted Values	Green	1564/1563/1560 [a]	6/20/12	0/11/4
	Orange	0/1/13	14/0/8	15/2/4
	Red	0/0/1	0/0/0	4/6/11

[a] X/Y/Z, X J48, Y NB and Z IBk

Table 3.10 Confusion Matrix of software aging hidden within Periodic Pattern: J48/NB/IBk

We can observe how J48 ($C = 0.25$ and $N = 2$) obtains better results: it determines the Green zone and not too bad to predict the Orange and Red zones. Misclassified Red tuples are classified as Orange (15 over 19), and this zone is also dangerous for the system availability. However, IBk and Naive Bayes produce several false negatives: NB predicts 11 Red-as-Green and IBk ($k = 5$), 4.

Figure 3.20 presents the whole experiment results and Figure 3.21 presents the last 2 hours and 30 minutes of the same experiment where the errors are more visible. It is in this figure where we can observe clearly the prediction problems of Naive Bayes and IBk. We can observe how IBk models is unstable in its predictions going from orange to green several times, making impossible to take any decision about the recovery action. In the case of Naive Bayes, the problem is that the model is unable to predict the orange zone and starts to predict red zone too late to trigger, with security, the recovery action.

On the other hand, J48 has some problems but less important. It starts to predict the Orange and Red zones with a small delay. But this delay could be acceptable because we have time to run the rejuvenation action.

3.5.4.4 Dynamic and Variable Software aging due to Two Resources

Using the same experimental environment described in Section 3.4.3.4, using a variable and dynamic software aging due to two resources: memory and threads, which change their trend every 20 minutes. We evaluated the three algorithms selected.

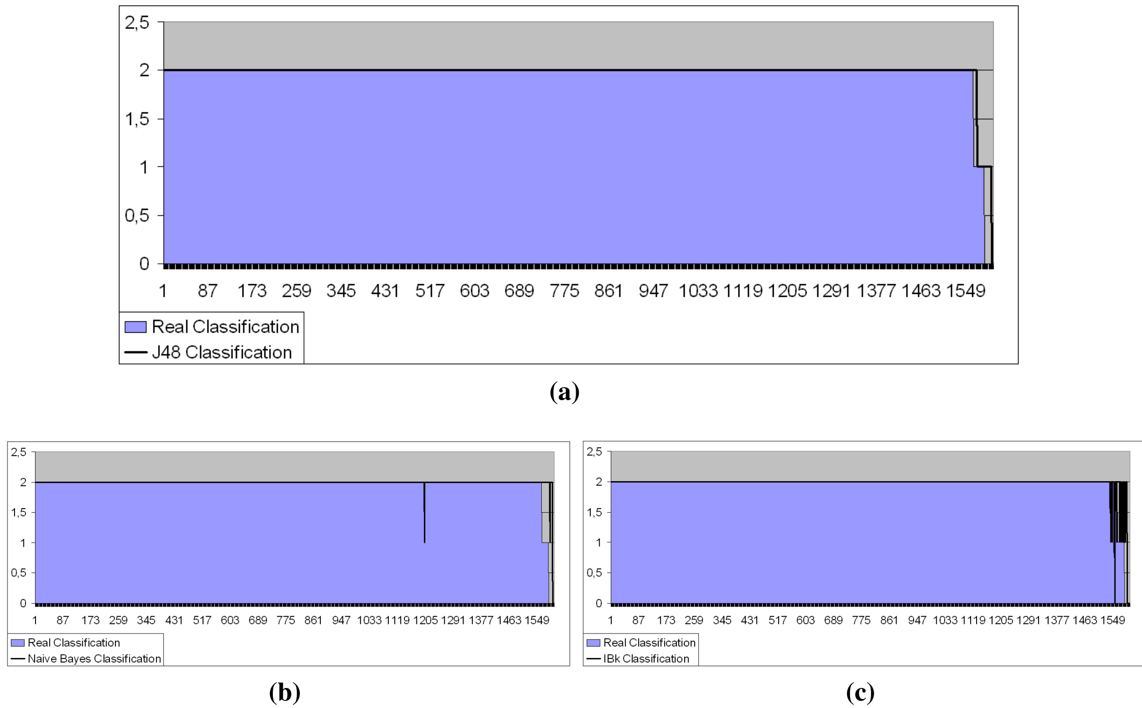
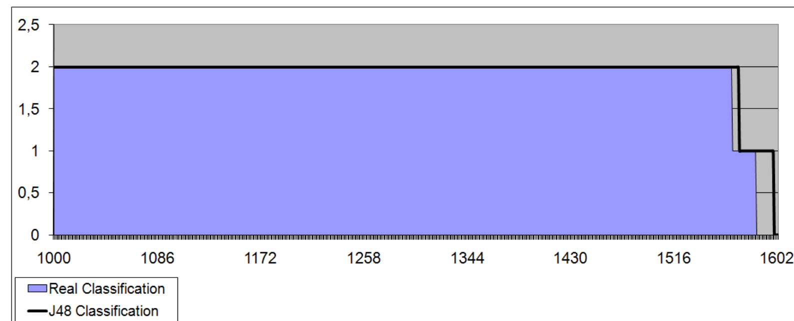


Figure 3.20 Alarm classification results for a 100EBs of workload and software aging within Pattern behavior, (a)J48, (b) Naive Bayes, and (c) IBk.

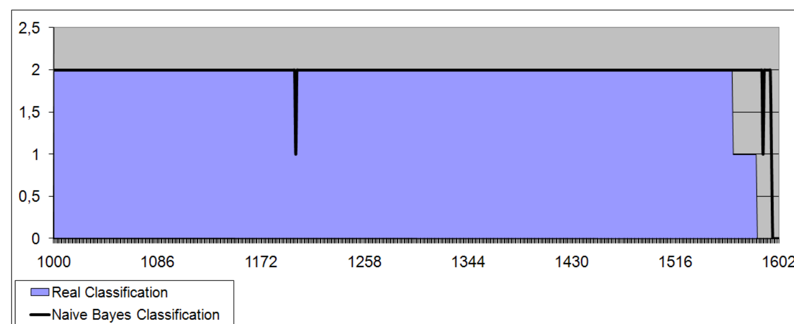
		Real Values		
		Green	Orange	Red
Predicted Values	Green	419/58/411 [a]	7/0/8	15/0/8
	Orange	3/364/14	0/20/12	0/16/11
	Red	3/3/0	13/0/0	5/4/1

[a] X/Y/Z, X J48, Y NB and Z IBk

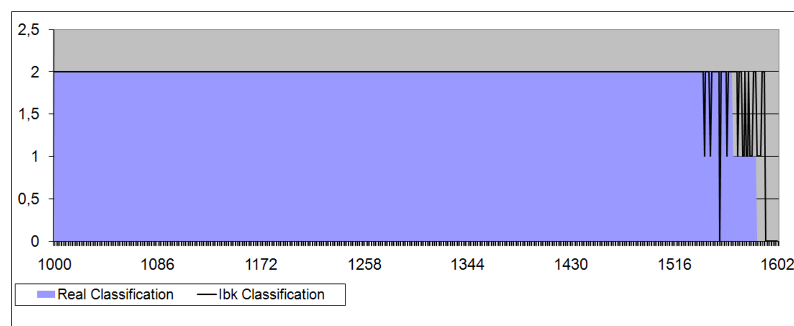
Table 3.11 Confusion Matrix of Dynamic and Variable Software aging due to two resources: J48/NB/IBk



(a)



(b)



(c)

Figure 3.21 Alarm classification results for last 2 hours and 30 minutes of 100EBs workload and software aging within Pattern behavior, (a)J48, (b) Naive Bayes, and (c) IBk.

Table 3.11 presents the confusion matrix of three algorithms in this scenario. The first conclusion we can extract from these numbers is that three algorithms have important errors in at least one of the zones. J48 ($C = 0.5$ and $N = 2$) and IBk (k=5) are good to determine the green zone, but not for orange or red zones. On the other side, Naive Bayes (NB) is good to determine the orange zone and not too bad to determine the red zone, because, it is predicting mainly orange zone, at least. But NB has severe problems with the green zone.

However, we decided to apply the technique previously described to balance the number of green, orange and red instances. In the original training data set the 2.5% of instances were red and another 2.6% were orange. We moved on to 20% in both cases. Table 3.12 presents the results obtained under the new more balanced training data set during the validation phase.

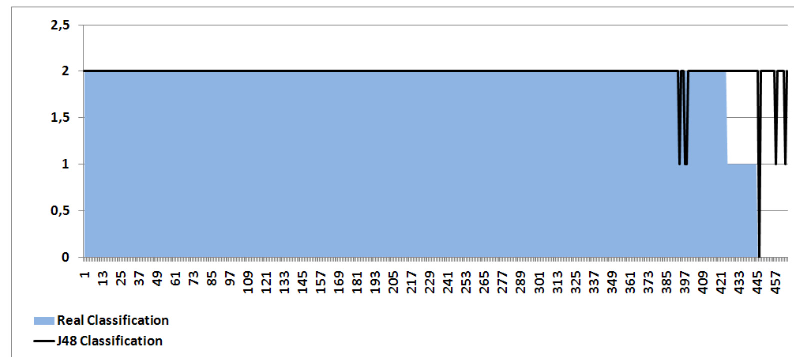
		Real Values		
		Green	Orange	Red
Predicted Values	Green	422/79/385 [a]	20/0/0	17/0/0
	Orange	3/338/40	0/20/20	2/15/15
	Red	0/8/0	0/0/0	1/5/5

[a] X/Y/Z, X J48, Y NB and Z IBk

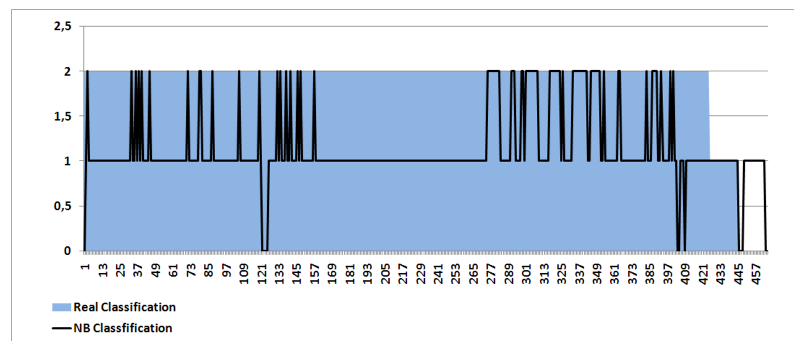
Table 3.12 Confusion Matrix of Dynamic and Variable Software aging due to two resources equilibrating instances: J48/NB/IBk

We can observe clearly the improvement of IBk ($k = 20$) and Naive Bayes to predict the dangerous zones (orange and red). However, we can observe how J48 has increased its error in these zones. On the other hand, we can observe how J48 is very good to predict the green zone.

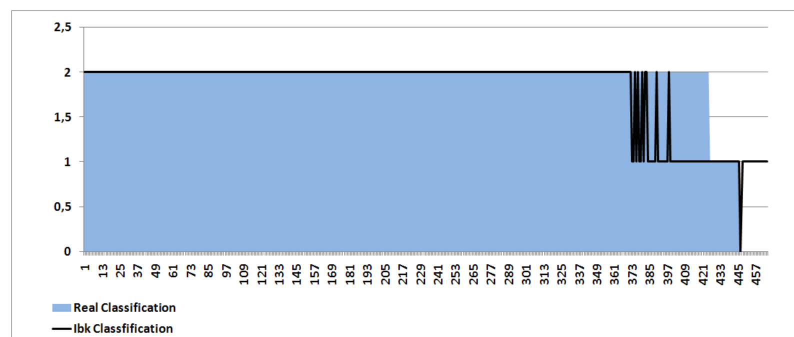
Figure 3.22 shows the results of three algorithms according to the errors are done and presented in table 3.12. Figure 3.22a shows clearly the effectiveness of J48 to determine the green zone, however, this figure also shows the unacceptable and dramatical results obtained to predict the orange and red zones. Finally, in Figures 3.22b and 3.22c we can observe how with the results obtained by Naive Bayes and IBk ($k = 20$), we cannot conclude when we are in the green vs. orange/red zones. But, we can detect clearly the dangerous zone. This shows how the results from the confusion matrices could be dangerous alone to evaluate the prediction accuracy of a model because NB shows clearly its instability during the experiment. In this case, the best option could be a consensus board of the three algorithms. We can use J48 to determine the green zone. However, when IBk and NB have an agreement continued along the time we can determine that we



(a)



(b)



(c)

Figure 3.22 Alarm classification results for 100EBs workload and dynamic software aging trend and two resources involved, (a)J48, (b) Naive Bayes, and (c) IBk.

are out of the green zone, coming in the dangerous(orange or red) zone.

3.5.4.5 Learned Lessons

To conclude the classifiers evaluation, we can observe how in all experiments the overall result was that J48 is the best option or at least it is a not-really bad option. However, it has several problems to become useful in a real on-line prediction system. So, our conclusion is that a simple classifier is not enough to predict with acceptable accuracy. In fact, J48 is a model tree like M5P, but simpler: without Linear Regressions in the leaves.

The experiments conducted in this section also reveal the importance of the training and validation data files. It is important to use as rich as possible training data sets (with a lot of previous executions) to increase the accuracy of the predictors. As is to be expected, if the Machine Learning algorithm has more data to extrapolate the prediction function, better predictions will be obtained. Moreover, it is important to train the models according to the prediction we really want to perform, i.e. carefully choose the error function or chosen the correct value for the *complexity parameter*. In our case, we want to be more accurate when the crash was imminent (orange or red), for this reason we introduced in the training data set repeated tuples from the last 20 minutes of every execution which makes up it. However, in our case, we improved the accuracy but not enough to obtain acceptable and useful results.

Regarding the importance of the training phase, we have evaluated some algorithms using different values of the complexity parameter: M5P, J48 and IBk. In all of them, this value has a very limited impact on the results. The most important fact to influence on the prediction accuracy in the case of classifiers is a training data set balanced between green, orange and red. This fact is due to the stability of the data obtained and even more due to the fact that the trees and models built by the ML algorithms are small. The k complexity parameter from IBk seems the most relevant of them. However, we cannot find one unique good value for all experiments. In any case, we have presented the best results obtained and the value of the complexity parameter used of every algorithm to allow any other researcher, engineer or practitioner to repeat the experiments.

Finally, these experiments state the effectiveness and high accuracy achieved by M5P using the same training and validation datasets. This fact ratifies our prediction hypothesis presented in Section 3.2.2.

3.6 Adaptive and Predictive Software Rejuvenation Framework

In this section we present our rejuvenation framework in detail, which is the merge of the threshold-based rejuvenation framework presented in Chapter 2 and the Machine Learning predictive engine described in previous sections. Figure 3.23 presents the basic architecture and components of the framework. It has been designed taking into account the fact that we are focusing on self-healing for web applications. We consider these applications formed by a web application server where the applications are deployed, and a centralized or distributed data base.

The framework is based on the MAPE Architecture defined in [69], which defines a cyclic system with four stages: Monitoring, Analyzing, Planning, and Executing. Our framework covers all four stages.

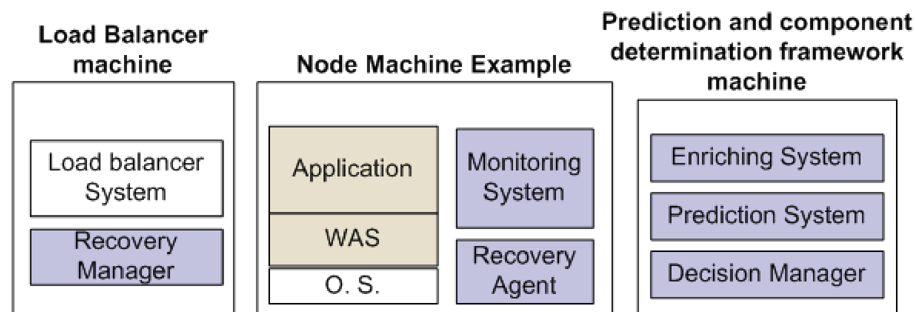


Figure 3.23 Framework components

3.6.1 Monitoring Subsystem

The monitoring subsystem's main task is to obtain the system activity report metrics (SARs) at regular time intervals. The monitoring subsystem is implemented by the *monitoring agent* which is installed in every web application server machine that we want to include in our self-healing framework, as shown in Figure 3.23.

The monitoring agent collects metrics from the whole system and specifically related to the web application server under monitoring such as memory used by the application server, CPU used, threads running on the system, throughput, response time, etc. The current version of the system-level monitoring agent is based on the well-known and used monitoring tool called Nagios [89]. It offers a scalable and easy to upgrade monitoring framework used in several computer centers. We have added some monitoring plug-ins to add more metrics to obtain more and detailed data from the system. In detail, currently we collect 17 system metrics: the first 17 rows from Table 3.2.

The monitoring subsystem captures the state of the system every N seconds. In our experiments, N was fixed to 15 seconds and the data, in the form of a tuple which represents the system state, is sent to the Analysis subsystem.

3.6.2 Analysis Subsystem

The Analysis subsystem is divided in two modules: *the enriching system* and *prediction system* (Figure 3.23). In our current prototype version of the framework we have a centralized Analysis subsystem, although definitely it should be easy and desirable to implement a decentralized analysis subsystem in the future, to reduce the implied bottleneck. The enriching system receives the monitoring data and generates a subset of derived metrics to add to the tuple of the raw metrics. The derived metrics (such as resource consumption speed) are needed to build a more detailed server state photo usable by the Machine Learning Algorithms. In fact, we are not adding *new* information, we are transforming the data to help in the learning process of the Machine Learning algorithms.

After the enriching process has processed the captured metric tuples, the new generated tuple is sent to the prediction system. The prediction system is a trained model obtained (offline) by one or a set of Machine Learning algorithms (currently, M5P). This trained model has the responsibility to predict the time to crash due to resource exhaustion.

3.6.3 Planning Subsystem

When the time to crash prediction for the web application server has been estimated, it is sent to the Planning subsystem. The Planning subsystem is implemented by the Decision Manager. Currently, our Decision Manager simply triggers a recovery action when the time to crash is below X minutes. Here, X minutes is fixed according to three main values: The error of the predictor, the time needed to start the hot stand by server and a security margin. The first is determined from experience (e.g. training phase) and the other two by the system administrators.

The usage of M5P model allows us to know (approximately) the time to crash. If we know the time needed to start the hot-stand by server, we can avoid to have both virtual machines running simultaneously.

Our plans for the future include investigating more elaborate scheduling and planning techniques to trigger different recovery actions at possibly different times, to maximize server usage/utility.

3.6.4 Execution Subsystem

The Execution subsystem is responsible of applying the recovery action decided by the planning subsystem. The Execution subsystem is composed of the Recovery Manager and the Recovery Agent. The Execution subsystem is based on a hot-standby server and a load balancer that redirects the requests from the primary to the secondary server, avoiding to miss any on-going or new request. The recovery manager is installed in the Load Balancer machine and the recovery agent is installed in every node under the framework control. The recovery system is based on the software rejuvenation mechanism presented in Chapter 2.

3.7 Adaptable Monitor to Determine Software Aging Root Cause Failure

In proactive/predictive rejuvenation strategies, system metrics are continuously monitored and the rejuvenation action is triggered when a crash or hang of the system due to the resource exhaustion (the cause of the software aging) is an evident probability: using predictions or thresholds.

The effectiveness of these proactive strategies is mainly based on the accuracy of the monitoring system used to collect the system metrics. However, the monitoring systems mainly collect system metrics understanding the applications as *black boxes*, becoming impossible to know which is the root cause of the software aging. We call root cause failure the application component, usually a piece of software which is responsible of the resource exhaustion. We understand application component as the smallest piece what the application could be divided into. For example: objects, servlets, EJB's or others, depending on the technology used to develop the application. Traditionally, monitoring systems are based on knowing the resource or resources involved in the software aging, however they cannot offer any clue or help to determine the piece of software where the bug is. For this reason, the main rejuvenation strategy is applying a reboot or application restart. This approach has an important impact over the application availability. New techniques have been proposed to reduce the Mean Time to Recover (MTTR) such as Micro-rebooting [32]. Micro-rebooting reduces dramatically the MTTR because it only reboots the faulty component, achieving outages of less than 600 milliseconds.

For this reason, determining the root cause component becomes critical to apply these surgical techniques. However, if we want to determine which component/s are involved in the software aging we need a monitoring framework which allows us to know how many,

the quantity (the usage trend) and the type of resources are used and not released by every component.

In this section we present a framework based on Aspect Oriented Programming (AOP) [71] to monitor the resources used by every application component. Our approach is focused, but not limited, on J2EE architectures, but the same idea could be moved to other languages like C++ [80]. We have focused on J2EE infrastructures, because they are the most currently widespreaded to develop web applications.

Our approach is based on the idea to offer a monitoring solution without need to modify the application or web application server (WAS) source code. AOP makes it possible to inject our solution to J2EE architectures in runtime. This feature makes our solution adds a very limited overhead to the original application. Furthermore, our approach allows us to know, with great detail, the resources used by every component and the frequency of resource consumption by every component. We collect all of these metrics from every component under monitoring to determine with high accuracy the real root cause component of the software aging phenomena. The idea is to use this framework to establish which component or set of components are consuming more resources to build a resource-component consumption map to help developers and administrators to determine the software aging root cause failure. Moreover, our proposal could be used in development and testing application life-cycle phases to detect misbehaviors, anomalies or to help to optimize the resource usage by the application.

3.7.1 Technology Used

Before presenting the monitoring architecture proposed, we present the technologies used: Aspect Oriented Programming (AOP) and Java Management Extensions (JMX)[67]. Although it is out of scope of this document to present in detail both technologies, it is necessary to describe them briefly to make clearer the solution presented.

3.7.1.1 Aspect Oriented Programming

The AOP paradigm isolates the main business logic of the application from secondary functions like logs or authentication. This paradigm increases the modularity, separating concerns, specifically cross-cutting concerns. *Aspects* is the name of the main concept of the AOP technology. The aspects are composed of two elements: *Advices* and *Join Points*. The advices are the code that is executed when the aspect is invoked: The advice has access to the class, methods and/or fields of the module which the advice invokes. The

Join Point is the definition to indicate when the advice will be invoked. We can see the Join Point like a trigger: when the condition is true the Advice is invoked. For this technology's implementation, we have chosen AspectJ [70] because it is a well-known, widely used and mature technology. In addition, this technology offers a simple and powerful definition of Aspects like Java class, so the learning curve is quite smooth for experienced Java developers. The AOP paradigm is not limited to Java Applications, we can find AOP solutions for C# or C++ like AspectC# [72] and AspectC++ [18] respectively.

Furthermore, AOP offers other important and interesting capability for our purposes. AOP allows us to inject code at compile, load or runtime. The framework is injected at runtime without needing access to the source code, becoming useful for third-part J2EE applications or even legacy Java Applications. The AOP injection process is based on preprocessors (if we have access to the source code) or uses the bytecode to weave the Aspects in compiling time or if the weave is per-class, could be done in loading time. Finally, the runtime weaving needs special environments and not all solutions offer this option. More information could be found in [87].

3.7.1.2 Java Management Extensions

The JMX technology offers a set of capabilities to manage and monitor any system component: from devices to Java objects. The JMX is based on 3-level architecture: Probe level, Agent level and Remote Management Level. The Probe level is composed of the probes (called MBeans). Every MBean represents a Java object. The Agent level or MBeanServer is the core of the JMX technology and acts as intermediary between MBeans and the external applications. Finally, the Remote Management Level allows external applications to communicate with the MBeanServer via JMX connectors or protocol adapters. So, JMX allows us to connect and communicate with Java objects (MBeans) at runtime without modifying the application sourcecode allowing us to interact with them, transparently.

3.7.2 Architecture Description

After the presentation of technologies used to develop our approach, it is time to present the architecture of our solution. We can divide our approach in four main components: The Aspect Component (AC), the JMX monitoring Agents, the JMX Manager Agent and the External Front-end. Figure 3.24 shows the components that compose our solution.

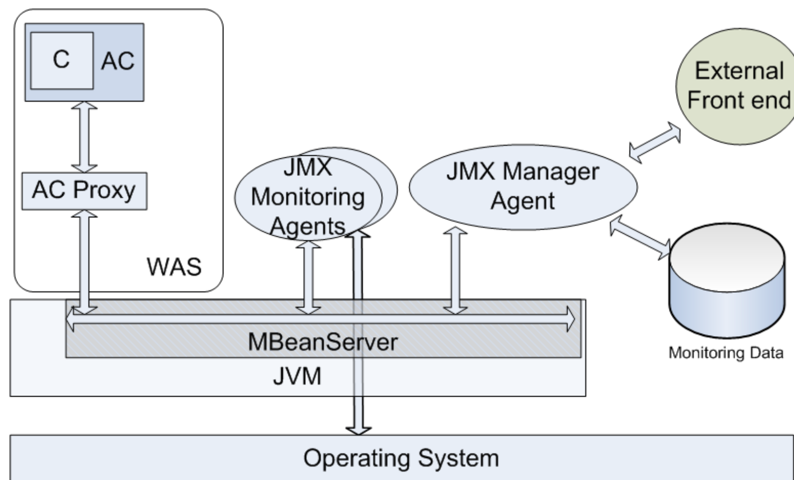


Figure 3.24 Components of Monitoring Framework

3.7.2.1 Aspect Component

Aspect Component is composed of the two elements: The Aspect Component (AC) and the Aspect Component Proxy (AC Proxy). Every application component has an AC associated (thanks to the Join Point definition). We understand an application component to be any application class. The AC has two advices: before and after the application component execution. The idea is to measure every system resource before and after a component is used. In this way, we can know how much resource has been used by the component. If the component has a resource consumption bug, the resource available after the execution will be lower than before. To achieve this, the AC communicates (using MBeanServer) to the JMX Monitoring Agents to know the resource status when it is demanded. Currently, our architecture is based on a limited set of Monitoring Agents by every resource under monitoring. We have decoupled the JMX Monitoring Agents to the AC (thanks to JMX technology) to increase the adaptability and flexibility of the solution. Currently, if a Monitoring Agent is modified or changed, we do not need to change the AC at all. The MBeanServer capabilities allow the AC to discover new or updated JMX Monitoring Agents. The AC Proxy is responsible for creating a communication channel between the AC and the JMX Manager Agent. This channel allows the JMX Manager Agent to interact with the AC: from asking some information like how many requests have been processed by the component to activating/deactivating the AC at runtime. The JMX technology offers a great flexibility and adaptability because we can change the ACs or add new ones and the JMX Manager can discover them by itself and vice versa.

3.7.2.2 JMX Monitoring Agents

JMX Monitoring Agents have the responsibility to access the operating System and collect resource metrics, AC on demand. They usually will be executed before and after every access to every application component under monitoring. Currently, we have developed a limited number of monitors only to show the effectiveness of the approach (i.e. number of threads or I/O file blocking). The JMX Monitoring Agents ask for the Operating System to know the resource status at the request instant. This information is sent to the Aspect Component (AC) to be managed.

3.7.2.3 JMX Manager Agent

JMX Manager Agent is the core of our proposal, having the responsibility to collect the metrics by component and build the resource-component map. Furthermore, it has the responsibility to activate or deactivate ACs on demand. For example, to reduce the overhead of the solution or to focus the monitoring on a set of determined objects. The JMX Manager Agent builds the resource-component map and offers a first analysis to establish the most possible root cause component of the software aging in advance. If a software aging has been detected while monitoring the system metrics using a traditional monitoring tool, we can use the JMX Manager Agent (and the rest of our framework) to determine (or at least help) the component or components involved in the software aging.

3.7.2.4 External Front-end

The *External Front-end* is a simple front-end to allow administrators to communicate with the JMX Manager Agent to know the status of the components in real time or activate new ACs or new JMX Monitor Agents and obtain more details of the application behavior.

3.7.3 Root Cause Determination Strategy

The main idea is that the component is more aging-related when the component resource consumption and the usage frequency is high. Of course, current root cause strategy is very simple and has to be refined in the future. In Figure 3.25 we present the core of our current map theory. If a component is highly used and the resource usage is high (accumulated along the time) the component increases its probability to become the main aging-component of the application. For example, if we have four components in our application: A, B, C and D. A and B have a memory leak of 100KB in each execution and C and D have a memory leak of 10KB. A and B will be in the right zone of the vertical-axis and C and D in left zone. If A is more used than B then A is in the bottom of the

right size (the most suspicious zone) and B in the top. In the same way we can locate the C and D components. Using this analytic approach of the components behavior, the JMX Manager Agent builds the map of root cause aging failure. We have used this approach because the software aging can be often observed as an accumulation of aging-errors that usually are consuming resources along time until their exhaustion. For this reason, we want to know which component is consuming more resources, so which component is more correlated with the resource exhaustion.

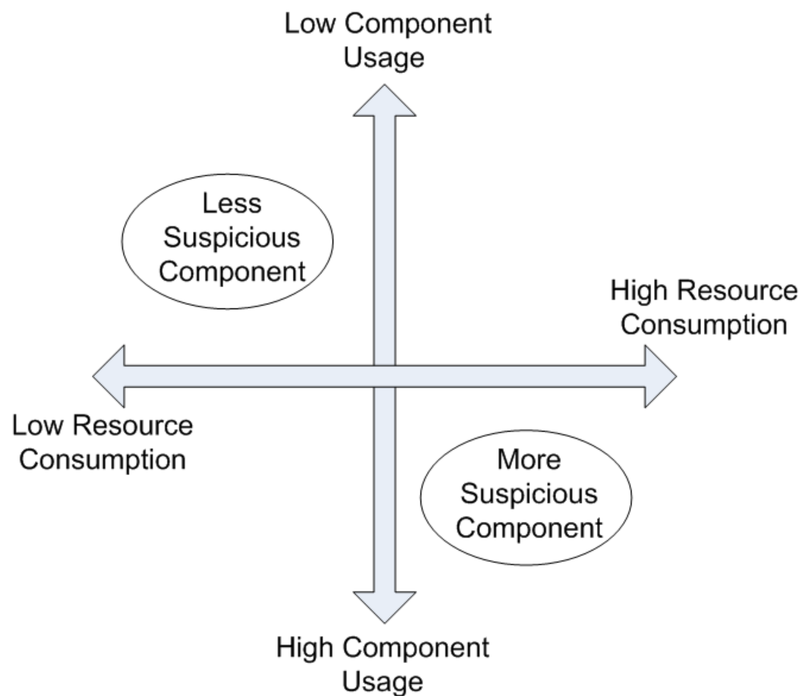


Figure 3.25 Resource Consumption vs Component Usage map

3.7.4 Experimental Evaluation

We have conducted a set of experiments to evaluate the effectiveness of our approach. Our idea is to test our prototype to determine the cause (the faulty component) of a memory leak.

3.7.4.1 Experimental Setup

In this section we describe the experimental setup used in all experiments. The experimental environment is composed by the web application server, the database server and the client's machine. We have used the same scenario used in the previous prediction evaluations. However, we believe that it is important to recall briefly the description.

In our experiments, we have used a multi-tier e-commerce site that simulates an on-line book store, following the standard configuration of TPC-W benchmark [112]. We have used the Java version developed using servlets and using Mysql [88] as database server. As application server, we have used Apache Tomcat [16]. TPC-W allows us to run different experiments using different parameters and under a controlled environment. These capabilities allow us to conduct the evaluation of our approach to predict the time to crash. Details of machine characteristics are given in Table 3.13.

TPC-W clients, called Emulated Browsers (EBs), access the web site (simulating an on-line book store) in sessions. A session is a sequence of logically connected (from the EB point of view) requests. Between two consecutive requests from the same EB, TPC-W computes a thinking time, representing the time between the user receiving a web page s/he requested and deciding the next request. In all of our experiments we have used the default configuration of TPC-W. Moreover, following the TPC-W specification, the number of concurrent EBs is kept constant during the experiment.

	Clients	Application Servers	Database server
Hardware	2-way Intel XEON 2.4 GHz with 2 GB RAM	4-way Intel XEON 1.4 GHz with 2 GB RAM	2-way Intel XEON 2.4 GHz with 2 GB RAM
Operating System	Linux 2.6.8-3-686	Linux 2.6.15	Linux 2.6.8-2-686
JVM	-	jdk1.5 with 1GB heap	-
Software	TPC-W Clients	Tomcat 5.5.26	MySql 5.0.67

Table 3.13 Detailed experimental Setup Description

To simulate the aging-related errors consuming resources until their exhaustion, we have modified the TPC-W implementation. In our experiments we have played with Memory resource. To simulate a random memory consumption we have modified a servlet which computes a random number between 0 and N. This number determines how many requests use the servlet before the next memory consumption is injected. Therefore, the variation of memory consumption depends on the number of clients and the frequency of servlet visits. According to the TPC-W specification, this frequency depends on the workload chosen. This makes that with high workload our servlet injects quickly memory leaks, however with low workload, the consumption is lower too. But, again, the average consumption rate would depend on the average of this random variable, with fluctuations that become less relevant when averaged over time. Therefore, we could thus simulate this effect by varying N, and we have decided to stick to only one relevant parameter, N. This error helps us to validate our framework under different scenarios. TPC-W has three types of workload (Browsing, Shopping and Ordering). In our case, we have conducted

all of our experiments using shopping distribution.

3.7.4.2 Experimental Results

We have conducted a set of experiments to determine the effectiveness of our approach to monitor the resources consumed by every application component under the experimental environment described before.

Framework Overhead

As we presented before, we have injected a set of components (Aspects) to the application code increasing the number of instructions executed by the computer. So, our monitoring framework has an impact over the performance of the applications. Our first experiment was to evaluate the performance penalty introduced by our framework in an one hour execution of TPC-W with two workload changes. The first two minutes (the warm-up) the workload was 50 Emulated Browsers. During the next 30 minutes the workload was increased to 100 EBs and finally, the last 30 minutes, the workload was 200 EBs. In Figure 3.26 we can observe the throughput obtained by the original TPC-W and TPC-W under monitoring with our infrastructure.

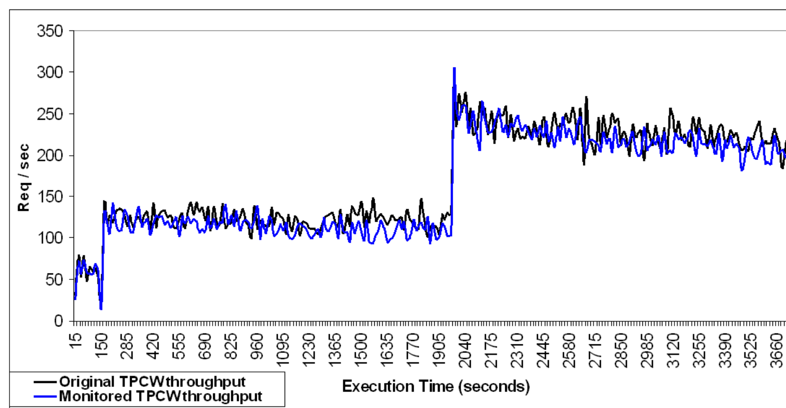


Figure 3.26 Throughput of TPC-W under a dynamic workload

In Table 3.14 we present the results in detail. We can observe how the overhead is quite acceptable, assuming that we are monitoring all application components.

	Average Throughput	Average Latency
Non Active Monitoring Scenario	171.17 req/sec	44 milliseconds
Active Monitoring Scenario	162 req/sec	67.5 milliseconds

Table 3.14 Detailed overhead introduced by our monitoring framework

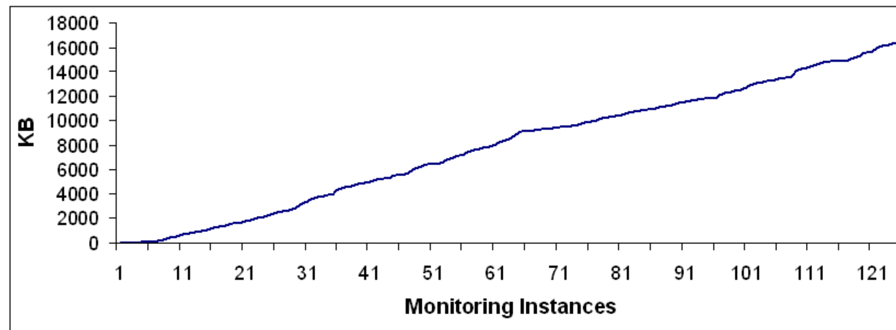
The penalty overhead is quite promising: only 5% of overhead in throughput perspective, monitoring all TPC-W application components. In this experiment we did not inject any memory leak. The response time penalty is quite complicated to evaluate because TPC-W uses a thinking time to simulate the time used by users to read the webpage which is generated at random by same interval probability distribution. For this reason, two executions have different response times. However, the workload (requests by time unit) is constant on average along time.

Effectiveness to determine a memory leaking component
After evaluating the overhead introduced by our framework we decided to test our approach under a software aging due to a memory leak in one component of TPC-W. The memory leak was injected as it was described before and we introduced 100Kb of memory leak. We have developed a JMX Monitoring Agent which allows us to know the real size of a Java Object. The real size of a Java Object includes the size of the objects referenced by the object under monitoring. However, if we follow this way, we start a recursive process for this reason, we don't follow the references from referenced objects, avoiding a recursively process. We only monitor one level of references of every object. This approach limits the effectiveness of the monitor but reduces the overhead introduced by the monitor. Furthermore, in J2EE applications, all objects inherit from superclass and if we apply recursively the process to calculate the size of the object, we find that one object has a indirect relationship with practically all objects of the application. Thanks to this monitoring agent we can know the memory object size at every moment.

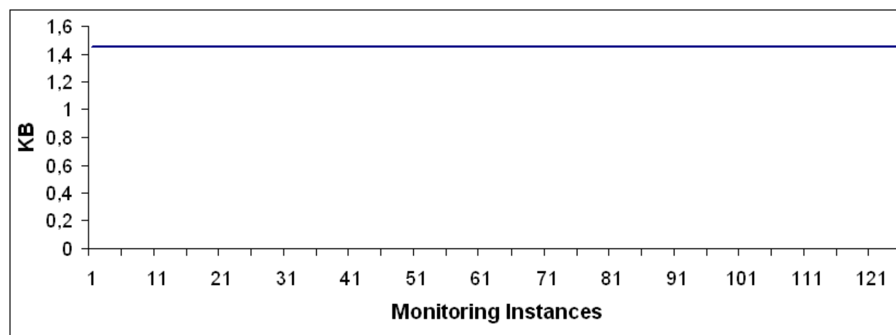
We conducted one hour execution injecting 100KB with $N = 100$ in component A (see Figure 3.27[4]) and the rest of components are not modified. We can observe clearly how the component A is growing in memory size due to the memory leak, becoming clear which is the guilty component of the software aging. While the rest of component sizes are constant a long the experiment consuming a few Kbs, the Object A size is growing from few Kb to MBs consumed during the experiment. In this point our simple mechanism is quite clear, only one component has more memory than the rest of them, concluding that A has the 100% of the responsibility of the software aging.

Effectiveness to determine set of memory leaking components
The next experiment presents the effectiveness of our approach to determine how four components are guilty of the software aging, but with different level of responsibility. We have conducted a new one hour execution, however this time, four objects (A, B, C and D)

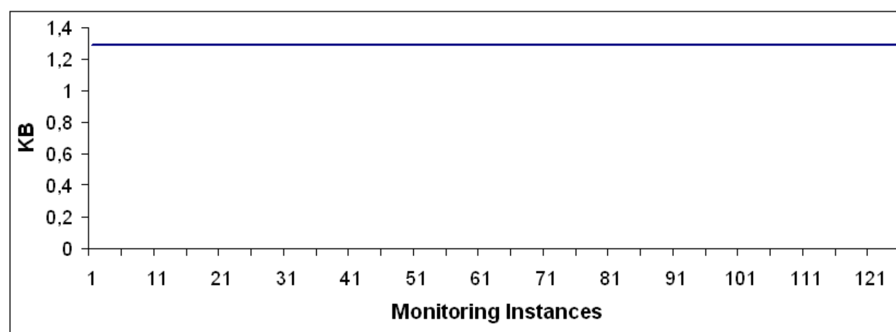
[4]Every monitoring instance is collected every 15 seconds



(a) Object A



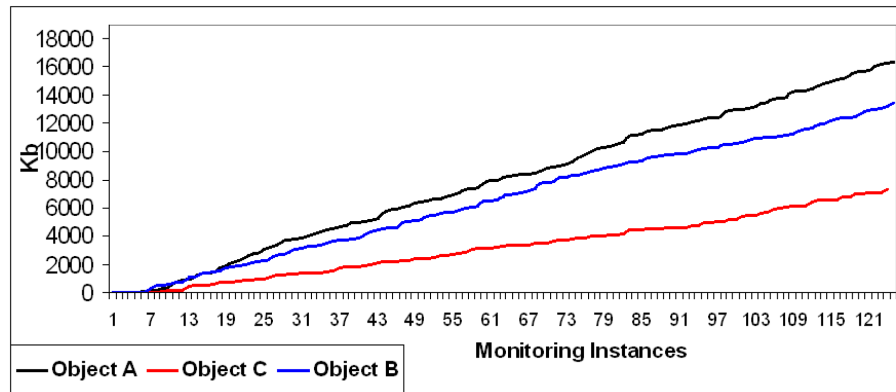
(b) Objects B and C



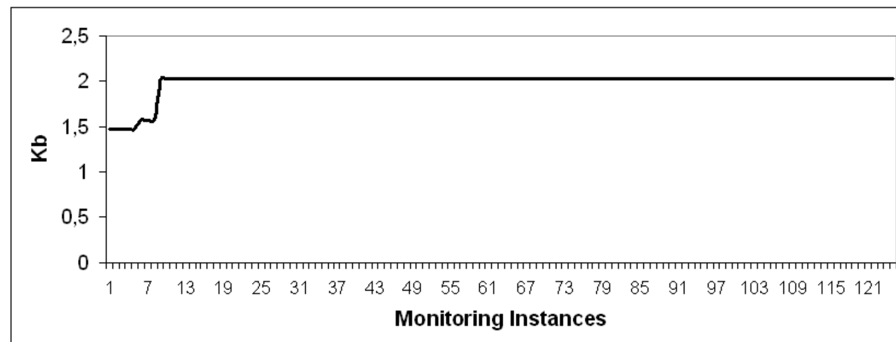
(c) Objects D, E and F

Figure 3.27 Injection in component A (100KB)

have been modified to inject 100KB following the fault injection process described in the experimental setup section. In Figure 3.28, we only present the four guilty components to reduce the image. We can observe how the four object sizes are increasing along the experiment but at different rate due to the injection mechanism in deed: all objects follow the same injection rate configuration $N = 100$, but the frequency with which they are used by the clients (EBs) is different.



(a) Objects A, B and C



(b) Object D

Figure 3.28 Detail of injection in 4 components

We can observe how components A and B have similar memory size after the same period, however object A has 2MB more than B. This fact indicates that A and B have more or less the same usage frequency (of course, A more than B) by the users. For this reason, in average they have the same memory leak. They will be in the bottom right zone of the Figure 3.25. However, we can observe how object C uses less memory, so, it will be in the top right zone. Finally, we can observe how object D never injected a memory leak because the frequency usage by the users is too low to cause the injection. For this reason object D memory usage is maintained constant. It will be in the top right zone. Following our approach objects A and B will be the most suspicious components, after that, object C and finally, object D. Figure 3.29 shows the composited map by JMX Manager Agent.

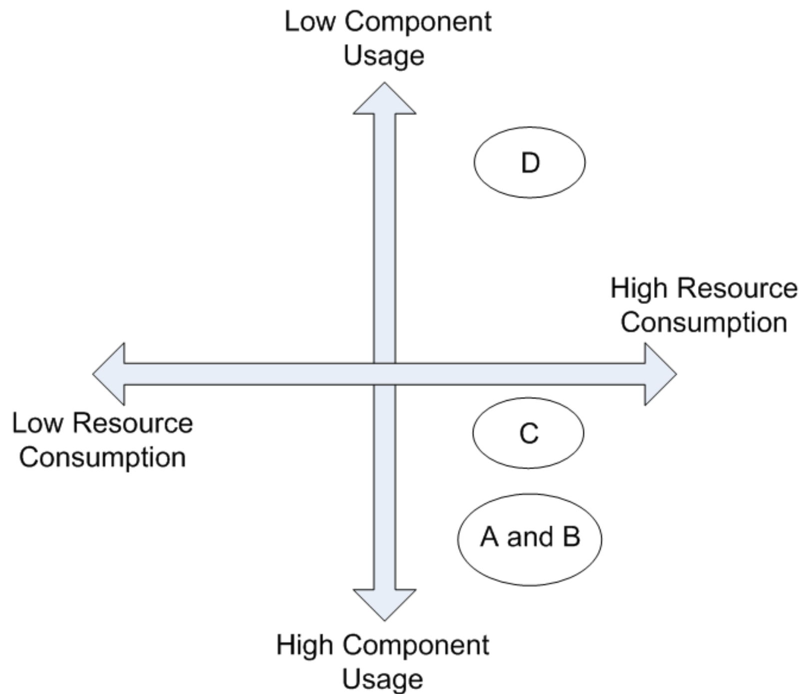
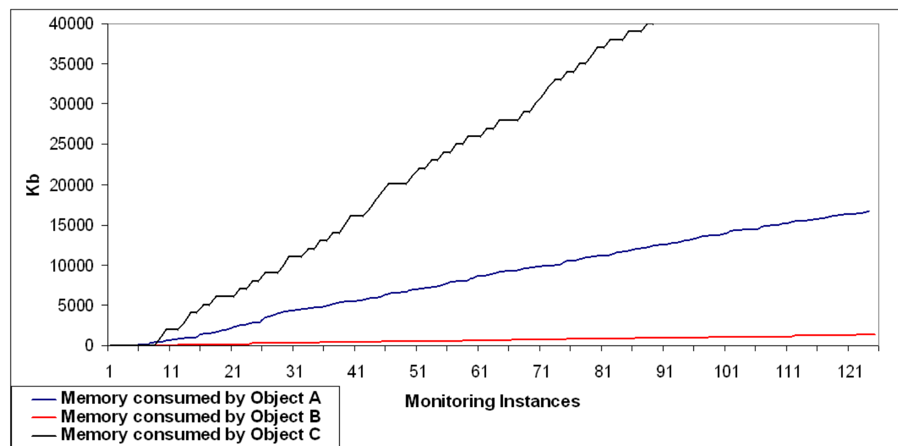


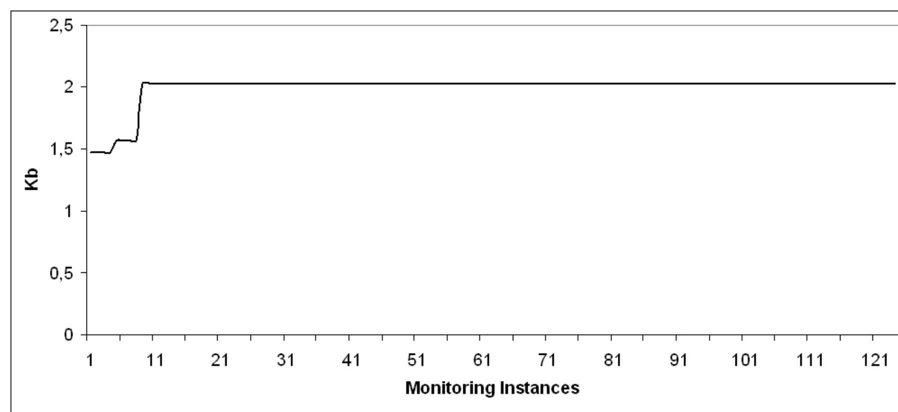
Figure 3.29 Resource Consumption vs. Component Usage map composed by JMX Manager Agent

Effectiveness to determine the root cause failure under different injection sizes
Our approach allows to know how the components are consuming the memory along the execution. After that, we decided to repeat the last experiment, but in the new experiment we injected only 10KB in Object B, 1MB in Object C and D while object A becomes the same with 100KB. The idea is showing how the memory leak and the usage of the component has an impact over the suspicious level of the component or components to be the root cause of software aging.

In Figure 3.30 we show the memory consumed by the four objects. We can see again how object D results in constant object size (2KB approximated) because it is used infrequently. On the other hand, in the last experiment, object C was in third position following our root cause determination approach. However, in the new experiment, increasing the size of the memory leak (from 100Kb to 1MB) has becoming on the most important reason of resource exhaustion. Object A continues being an important factor of the software aging (second position) and Object B, has also important impact over the memory consumption but now with a lower memory leak (from 100KB to 10KB), it is in third position.



(**a**) Objects A, B and C



(**b**) Object D

Figure 3.30 Determination of Root failure in 4 different injections

3.8 Related Work

3.8.1 Related Work on Prediction

The idea of modeling resource consumption and forecast system performance from here is far from new. A lot of effort along this line has been concerned with capacity planning. In [40], an off-line framework is presented to develop performance analysis and post-mortem analysis of the causes of Service Level Objective (SLO) violations. It proposes the use of TANs (Tree Augmented Naive Bayesian Networks), a simplified version of Bayesian Networks, to determine which resources are most correlated to performance behavior. In [130], Linear Regression is used to build an analytic model for capacity planning of multi-tier applications. They show how Linear Regression offers successful results for capacity planning and resource provisioning, even under variable workloads. All of these research works have a common characteristic: they are modeling non faulty systems. The performance of these systems is linear on allocated resource. However, they are not taking into account what happens if the system has any fault accumulating errors which consume the allocated resource in a non-linear way.

Other works such as [114, 77] present different techniques to predict resource exhaustion due to a workload in a system that suffers software aging. In these two works they present two different approaches: in [114], authors use a semi-Markov reward model using the workload and resource usage data collected from the system to predict resource exhaustion. In [77], authors use time-series ARMA models from the system data to estimate the resource exhaustion due to workload received by the system. However, these works assume a general trend of the software aging, not a software aging that could change with time. Another important difference with our work is that they apply the model over the known resource involved with the software aging. This is a priori hard assumption in any system. If we know the resource involved we can use simple thresholds to avoid the crash. But our approach is more generic because we allow Machine learning algorithms to learn, by themselves, what is the resource (or combination of resources) involved in the software aging.

In [59], authors conducted a set of analysis to calculate the trend of the aging using a set of different non-parametric statistical methods. After that they used time-series analysis to predict future values of any resource and calculate if the resource could be depleted or not. This work has several similarities with the work presented in this chapter, however the approach used is completely different. They use statistical approaches and we are using Machine Learning techniques. One of the most relevant points of this work was that authors analyzed the influence of seasonal patterns of the resource consumption

and how these patters could be used to predict future values.

Another interesting approach was presented in [12], where the authors present an evaluation of three well-known Machine Learning algorithms: Naive Bayes, Decision Trees and Support Vector Machines to evaluate their effectiveness to model and predict deterministic software aging. However, the authors did not test their approach against a dynamic setting or multi-resource exhaustion causing the crash.

Some interesting works have addressed the important task of predicting failures and critical events specifically in computer systems. In [105], the authors present a framework to predict critical events in large-scale clusters. They compare different time-series analysis methods and rule-based classification algorithms to evaluate their effectiveness when predicting different types of critical events and system metrics. Their conclusion is that different predictive methods are needed according to the element that we want to predict. But, as we showed M5P could obtain good results, even with more than one resource involved. [39] presents an on-line framework that determines whether a system is suffering an anomaly, a workload change, or a software change. The idea is to divide the sequence of recorded data into several segments using the Linear Regression error. If for some period it is impossible to obtain any Linear Regressions with acceptable error at all, the conclusion is that the system is suffering some type of anomaly during that period. Their approach is complementary with ours, because the underlying assumption is that, except on transient anomalies and between software changes, the system admits a static model, one that depends on the workload only and does not degrade or drift over time. On the other hand, we concentrate on systems that can degrade, i.e., for which a model valid now will not be valid soon, even under the same workload.

3.8.2 Related Work on Monitoring

Traditionally, the monitoring tools have been focused on collecting a set of external data from the system like performance, memory consumption, response time, threads used, etc. All of them treat the applications or web application servers (WAS) as a *black boxes*. As an example, we find several commercial and free solutions like Nagios [89] or Ganglia [51]. Both these systems allow detecting failures in our systems when the failure happens. The detection is based on rules defined by the human system administrators following their experience.

However, the effectiveness of these solutions is limited. These tools detect failures but cannot determine where the error is located. The administrator could find the resource or resources involved with the software aging phenomena thanks to these tools, but s/he cannot apply or fix the problem because s/he cannot know where the error is exactly

placed.

In the last years, new development applications have introduced tracing code with the objective of helping the system administrators and developers to determine where the error is at the same moment when the failure happens. However, this approach only offers a post-mortem analysis of the root cause of failure. Moreover, these solutions are not portable to other applications because they are developed ad-hoc and they require a reengineering work in order to adapt applications to obtain tracing features.

Concerning root cause determination, several approaches have also been presented. The Pinpoint project [38] collects end-to-end traces through the application server with the main goal to determine the most probably component cause of the failures in the system. For this purpose, they use statistical models. They find the components more related with faulty transactions. The idea is collecting all components used by every request and the result of the request (failure or success). This information is used to build a matrix to determine the guilty component or components. Their approach is quite similar to our approach, however they cannot deal with software aging phenomena because they cannot know the resources used by every component. Moreover, the Pinpoint solution has another important limitation: the coupled components. If two components are used always together (very common in J2EE applications) in faulty traces, the Pinpoint framework determines both components with the same probability to be the root cause of failure. However, our approach is ready to deal with this situation, understanding every component as independent one.

On the other hand, the Magpie system [25] collects the resource consumption of each component to model with high accuracy the system behavior, even in distributed systems. The Magpie approach is the most similar to ours in order to determine the root cause failure; however, the difference is that we are working at application level and Magpie works at operating system level. Also, the Magpie needs to modify the operating system architecture and our solution is completely independent of the source code increasing the flexibility and adaptability of our solution.

The use of Aspect Oriented Programming to monitor the applications is a hardly explored solution. Currently, the most mature solution in this area is Glassbox [53], which offers a fine-grain monitoring tool. This approach is focused mainly on execution time of every component allowing to detect a big set of failures. However, it is not creating a relationship between the application components and the resources available or used, which is needed to determine the root cause failure of software aging phenomena, not a bug on the component code where Glassbox is useful. Another interesting approach is TOSKANA [45]. It provides an AOP solution for kernel functions but again it is focused

on other metrics more than aging-metrics related.

For all of this, include monitoring systems which can collect external and internal data from complex systems at runtime is needed to determine the resources used by every component, as well as to integrate itself in the system without re-engineering the application or obtain the source code. It is also necessary that these monitoring systems are adaptable and flexible to allow activation or monitoring level change (from application overview to component or even method level or vice versa) at runtime.

3.9 Summary

In this chapter we have presented mainly two contributions. First, we have conducted an extensive evaluation of Machine Learning algorithms to predict the time to crash or at least the proximity of the downtime. The time to crash due to software aging (due to resource consumption) could be easy in a first overview, however the resource consumption trend could be not linear or moreover, could change along the experiment due to changes on the workload or the type of the workload. Due to these potential difficulties, predict the time to crash becomes in a heavy task. We have used Machine Learning to overcome the challenges presented by the software aging.

We have proposed a Machine Learning approach to build automatically models from the system metrics available in any system expecting that it can deal with the complexity of the resources behavior and the complexity of the environments. We have based our approach in feeding our model with a set of derived variables of which the most important is the consumption speed of every resource, which is smoothed using averaging over a sliding window of recent instantaneous measurements. We have evaluated our approach to predict the effect of software aging errors which gradually consume resources until its exhaustion, in a way that cannot be attributed to excess load. We have conducted a set of experiments to evaluate the M5P in front of Linear Regression in different and complex software aging scenarios to predict the time until crash. We have evaluated the M5P effectiveness in front of software aging phenomena which changes its trend along the experiment, playing with two different resources, individually or jointly. Moreover, we have showed clearly the capacity of M5P to adapt itself in front of scenarios where the model was never trained. We trained M5P using software aging experiments with one unique resource involved. After that, we validated the model in front of a software aging due to two resources involved simultaneously. After this evaluation, we have suggested a new potential approach to help to determine the root cause software aging, interpreting the models generated by M5P.

On the other hand, for predicting the exact (numeric) time until crash is too hard constraint, we decided to relax that requirement and evaluate ML classifiers to try to determine if the system is working correctly (green zone), approaching crash (orange zone) or near an imminent crash (red zone). This approach may be sufficient for many scenarios where we do not need the precision of a numeric prediction.

From our experiments we can conclude that Machine learning approach has important advantages. Although there is an open research field to improve the accuracy of the prediction and classification algorithms. However, M5P seems a good accurate predictor, while there is more research to do.

Second, we have also presented a complete self-healing framework, which predicts the time to crash of an application server. When a crash approaches, triggers a clean recovery mechanism. Our framework is ready to work with other ML algorithms easily.

Third, we have presented a monitoring framework for detecting software aging root causes, using Aspect Oriented Programming and Java Management Extensions technologies. Our methodology allows monitoring the applications without altering their source code and injecting the observers in runtime, being able to connect or disconnect them on demand. In our case study, we focused on a specific kind of software aging: memory leaks. We presented a theoretic map where, depending on the components observed behavior, we can determine the component that with high probability is the root cause of the resource consumption. Our experimentations showed that our method allows determining, given an aging-error, the most suspicious components, helping designers and system operators to look for the real cause and then fix the problem.

The work described in this chapter has resulted in publications presented in Section 1.5.2.

Chapter 4

Autonomic High Availability and Resource Usage Optimization

4.1 Introduction

Virtualization technologies improve the security, reliability and availability of services, reducing the cost and management. However, they are not enough to maximize the resource usage by themselves. On the other side, the services deployed on virtualization layers (within virtual machines) have to continue deal with the growing complexity and its derived problems described in previous chapters. In these scenarios, software aging continues to be an important factor of unplanned downtime because the virtualization is orthogonal to the services complexity. Moreover, this new abstraction layer can be a new source of aging or complexity problems.

In this chapter, we present a framework that is able to manage and reconfigure if it is needed (with minimum human intervention) a virtualized platform composed by several physical machines. It has been developed to assist platform providers to manage their virtualized platforms, achieving two main goals: a) offering a high availability mechanism for web services deployed on the platform, and b) maximizing/optimizing the resource usage as much as possible. We define the optimization of the resource usage as accepting as many services as possible on the platform. From the provider perspective, the resources are optimized if they are in use, for this reason, our idea is to accept as many service as possible guaranteeing that all possible resources are used. In our scenario, we understand service maximization and resource usage optimization (or resource optimization) as synonyms.

The framework offers an user-transparent high availability mechanism against software failures. The solution proposed reserves a part of the resources to guarantee the services availability, but the rest of resources have to be optimized, to accept the maximum number of services. We use mathematical programming [125] to guarantee the maximization of the number of services deployed on the virtualized platform while the availability is guaranteed. So, our framework offers a trade-off between high availability and the services maximization. We reduce the maximum services allowed by the platform physical resources, to guarantee the availability of the services deployed.

The high availability rejuvenation mechanism is an evolution of the rejuvenation framework presented in Section 3.6.

4.2 Virtualization Manager Framework Description

In this section, we present the three main components of our framework to achieve the two goals presented in previous section: the *High Availability and the Resource*

Optimization Scheduler (HARO-Scheduler, the *Failure Predictor (FPdr)*, and the *High Availability Load Balancer (HA-LB)*. Figure 4.1 presents how our framework components would be installed in a virtualized platform.

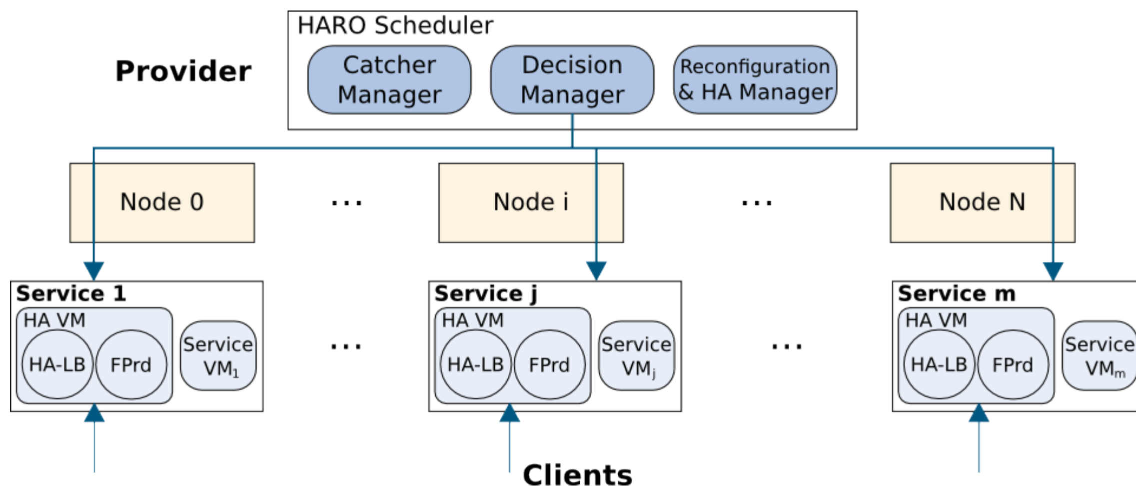


Figure 4.1 Provider Architecture Description

From the provider perspective, every customer service is composed by a VM where the service is deployed (Service VM) and an user-transparent VM for software rejuvenation purposes (*HA VM*). The *HA VM* is composed by two software components: the *Failure Predictor (FPdr)* and the *High Availability Load Balancer (HA-LB)*. As Figure 4.3 shows, the *FPdr* has the responsibility to predict the time to crash of the *Service VM*. This predicted time is sent to the *HARO-Scheduler* which reconfigures the VMs position to maximize the number of services if it is possible or decides to apply a recovery process on any faulty service to guarantee the availability. The Recovery process is basically conducted by the *HA-LB*. The details of the architecture and the processes conducted by it are presented below.

4.2.1 Failure Predictor

The *Failure Predictor (FPdr)* has the responsibility to predict the time to crash (TTC) due to software aging errors of the service associated. The prediction algorithm used in our framework is M5P [120] and its accuracy and details were presented in Section 3.4 to predict the time to crash due to software aging phenomena. The prediction process is based on a set of system/service metrics collected by the *FPdr*. Then, it calculates the TTC periodically to catch any change on the aging trend. After that, it sends the TTC predicted to the *HARO-Scheduler*.

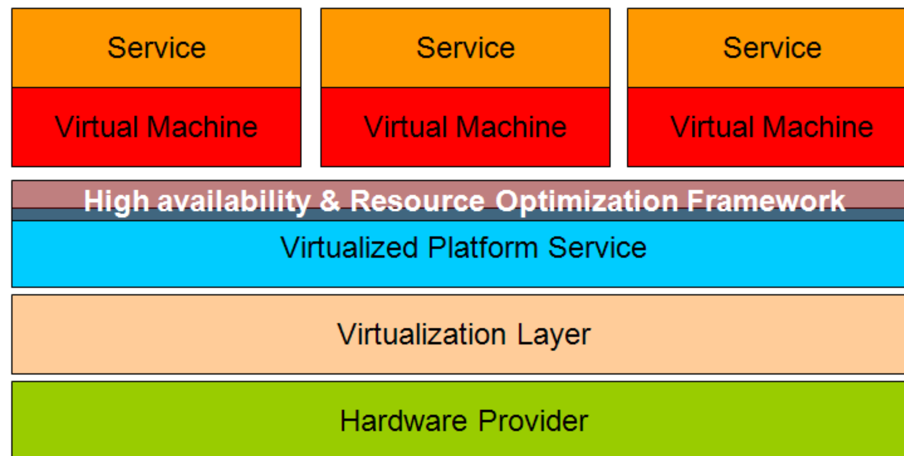


Figure 4.2 Conceptual Provider Architecture Description

4.2.2 High Availability and Resource Optimization Scheduler

The HARO-Scheduler is the core of our framework. It decides the correct moment to rejuvenate a service and how to allocate the VMs to accept the maximum number of services on the platform according to the resources available. The HARO-Scheduler is composed by three modules: *the Catcher Manager*, *The Decision Manager* and *the Reconfiguration and HA Manager*. Figure 4.3 shows the interaction of every component.

4.2.2.1 Catcher Manager

It is aware of all the VMs running on the platform. From the *CM* perspective all VMs are equals: *Service VM* and *HA VM*.

If a customer wants to deploy a new service on the virtualized platform, the request is also processed by *CM*. However, it does not decide if the request is accepted or not. *CM* maintains a list of running and waiting VMs.

Our high availability mechanism is based on replacing faulty services by new ones: *Replica VMs* . *CM* knows the running replicas on the platform. A *Replica VM* is an exact copy of the *Service VM* to be substitute (called *Service VM*). *Replica VMs* are only created to substitute a faulty *Service VM* and to guarantee the service availability.

On the other hand, the *CM* maintains another list with a set of attributes from all running/waiting VMs (*Service VMs*, *Replica VMs* and *HA VMs*). These attributes define the *size* of the VMs. They are basically defined by the customer (i.e. the number of virtual CPUs, the memory requested or the network interfaces needed). These values are used to compute the *size* of the VM. The used virtualization technology defines the maximum number of virtual resources allowed to create according the the physical resources available. The maximum number of VMs in a single physical node is based on

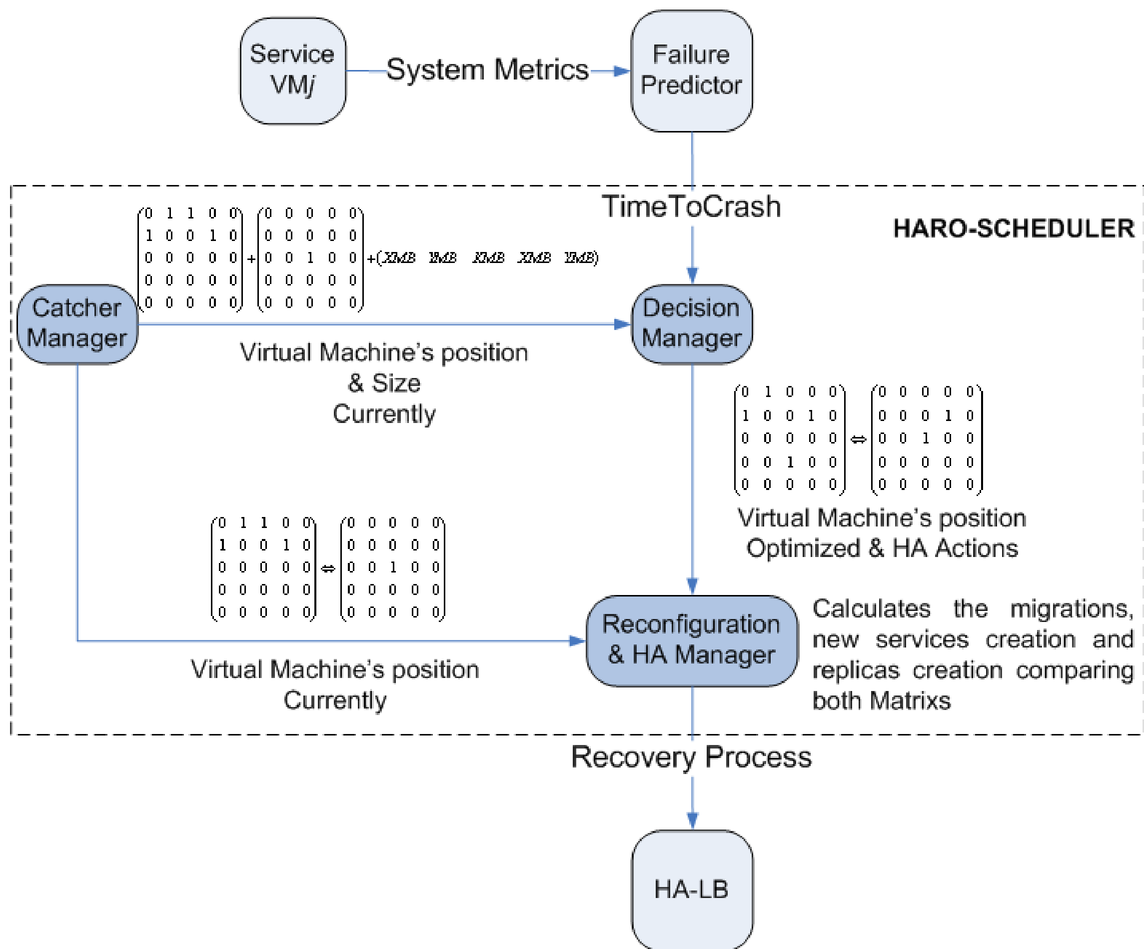


Figure 4.3 Communication process between framework components

the VM *size* and the physical resources available.

Note that the size of the waiting services to be accepted is the sum of the customer requirements (*Service VM*) and the *HA VM* requirements because the framework creates a *HA VM* associated to every *Service VM*, transparently to the customer in the same node. Later, once they are created, they could be decoupled. The *HA VM* are fixed by the administrators, however, the current version of this VM is quite small as we present later in Section 4.4.

All this information is represented by *CM* using two matrices and one vector, as it is presented below:

$$\widetilde{X} = \begin{pmatrix} \widetilde{X}_{0,0} & \widetilde{X}_{0,1} & \widetilde{X}_{0,2} \\ \widetilde{X}_{1,0} & \widetilde{X}_{1,1} & \widetilde{X}_{1,2} \\ \widetilde{X}_{2,0} & \widetilde{X}_{2,1} & \widetilde{X}_{2,2} \end{pmatrix} \widetilde{X}' = \begin{pmatrix} \widetilde{X}'_{0,0} & \widetilde{X}'_{0,1} & \widetilde{X}'_{0,2} \\ \widetilde{X}'_{1,0} & \widetilde{X}'_{1,1} & \widetilde{X}'_{1,2} \\ \widetilde{X}'_{2,0} & \widetilde{X}'_{2,1} & \widetilde{X}'_{2,2} \end{pmatrix}$$

$$W = \begin{pmatrix} W_0 & W_1 & W_2 & W'_0 & W'_1 & W'_2 \end{pmatrix}$$

where:

$$\begin{cases} \widetilde{X}_{i,j} = 1 & \text{if the VM j is in node i} \\ \widetilde{X}_{i,j} = 0 & \text{otherwise} \end{cases} \quad \begin{cases} \widetilde{X}'_{i,j} = 1 & \text{if a replica of VM j is in node i} \\ \widetilde{X}'_{i,j} = 0 & \text{otherwise} \end{cases}$$

$$W_j (= W'_j) \text{ represents the size of VM } j \text{ (or replica)}$$

The matrices $\widetilde{X}$ and $\widetilde{X}'$ represent the position of running/waiting VMs (both *Service VM* and *HA VM*) and *Replica VMs* respectively. Where the rows represent the physical nodes and columns the virtual machines. A waiting VM is represented by all-zeros column. But a 1 in one position of the column represents the physical node where the VM is running. On the other side, W contains the *size* of all (Service, Replica and *HA VMs*) running/waiting VMs. We note that the size of waiting *Service VM* is the sum of the real *Service VM* and its *HA VM*.

All this information is sent to *Decision Manager* as presented in Figure 4.3. In our experimental evaluation we have used as *size* the memory required by the virtual machine. However, others parameters to calculate the *size* such as CPU or other resources or metrics are possible.

4.2.2.2 Decision Manager

It receives the two matrices and the vector composed by the *CM* and its main task is to calculate the maximum number of services that could be accepted by the infrastructure. We understand that the resource usage optimization is achieved if we accept as many

services as possible on the platform without performance degradation. As we guarantee that every deployed service has the resources requested by the customer and never violate the virtual resources threshold defined by the physical resources available, the performance is guaranteed.

The *DM* decides if a waiting service could be accepted or not. The decision is taken based on the size of the waiting VM (a waiting VM size is the sum of the *Service VM* and the *HA VM*) and the current available resources on the platform. Even more, if it is needed to migrate running VMs to free space for new services, the *DM* indicates it.

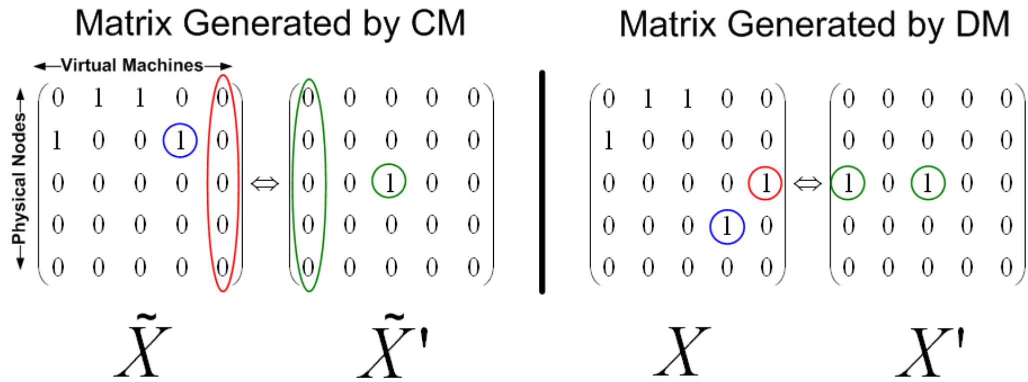
On the other hand, the *DM* also decides if a *Service VM* has to be replaced by its *Replica VM* or not. This decision is based on the time to crash (TTC) calculated by its *FPrd* and the threshold (time limit (TL)) defined by the system administrators. Moreover, the *DM* decides where (the physical node) the replica must be created.

The result of the *DM* process is two matrices (X and X') which represent the optimal position of all (*Service*, *Replica* and the *HA VM*) VMs. In Section 4.3, we will describe the mathematical model and reasoning used to calculate the optimal position of the VMs and accept the maximum number of services while offering high availability to the deployed services.

4.2.2.3 Reconfiguration and HA Manager

The matrices generated by *CM* and *DM* are sent to the *Reconfiguration & HA Manager* (RHAM) like shows Figure 4.3. The *RHAM* uses all (four) matrices to determine what actions have to be taken over the virtual machines to optimize the resource usage and guarantee the availability of the services deployed on the infrastructure. RHAM basically conducts a comparison between the current state of the environment (the matrices generated by CM, $\widetilde{X}$ and $\widetilde{X}'$) and the optimal state of the environment according to the solution generated by the *DM* (the matrices generated by the DM, X and X'). The process is simple and the framework could take the decisions quickly and adapt the environment to the optimal situation in a short period of time. Figure 4.4 presents the three main processes RHAM could conduct: Service Creation, Service Migration and Service Recovery.

– *Service Creation:* when the *RHAM* decides the creation of a new service is because at least one service is waiting to be deployed and there is enough space to allocate it. When a new service is deployed, transparently to the customer service, a *HA VM* is also created; creating two VMs. Figure 4.4 presents an example of how *RHAM* decides the creation of a new service. The all-zeros fifth column (red oval) at $\widetilde{X}$ indicates service 4 is waiting to be deployed. *RHAM* compares the fourth column from $\widetilde{X}$ generated by *CM* with the same column of X generated by *DM*. The fourth column of X is non-all-zeros



Actions Generated by RHAM

A) 0 → 1 Create the new service represented by column 4.
B) 1 → 1 Different row, represents a live migration of service 3 from node 1 to node 3.
C) 0 → 1 In second matrix, represents that service 0 needs a recovery action, creating a replica in node 2.

Figure 4.4 Matrix evaluation example conducted by RHAM

($X_{2,4} = 1$, red circle). This indicates that *DM* concluded that the best position to deploy service (*Service VM* and *HA VM*) 4 is node 2.

– *Service Migration:* it is conducted using the well-known technology offered by almost all virtualization manufacturers: live-migration [68]. This technology allows us to migrate from one physical node to another a running VM without any or minimal outage. In fact, in a very coupled data centers where all nodes are in a Gigabit Ethernet the live-migration avoids any outage during the process. This process could lead to decouple the *HA VM* and the *Service VM*. The *RHAM* decides a *Service Migration* comparing matrices $\widetilde{X}$ and $\widetilde{X}'$ with matrices X and X'. An example is presented in Figure 4.4, marked with blue circles. VM 3 is currently at node 1 ($\widetilde{X}_{1,3} = 1$), however, the *DM* process marks that the optimal position of the VM is at node 3 ($X_{3,3} = 1$). The *RHAM* has to order the migration of VM 3 from physical node 1 to 3. The same process can be conducted with *Replica VMs* or even *HA VMs*.

– *Service Recovery:* it is triggered due to the TTC of one or a set of VMs have violated the threshold (Time Limit) defined by the system administrators. However, the *DM* has to decide what the best place is to create the replica to optimize the resources at the same time. The best position of a replica is presented by matrix X'. Figure 4.4 presents an example (green circles). *DM* has detected that VM 0 gets into trouble and does not has a replica yet (all-zeros column of matrix $\widetilde{X}'$). *DM* has decided to create a replica at node 2 ($X'_{2,0} = 1$). The recovery process can be only conducted on *Service VMs*.

4.2.3 High Availability Load Balancer

The recovery process is conducted by the *HA-LB*. This process is a proactive process to avoid the crash due to the software aging in this case, becoming on a software rejuvenation solution [63]. It is composed by three phases presented in Figure 4.5.

1) Aging failure detection: the *HARO-Scheduler*, via the *FPdr* associated to the *Service VM*, detects that the TTC of the service is higher than the threshold defined by the administrators (TL).

2) Workload migration process: the *RHAM* creates a *Replica VM* in the physical node indicated by matrices generated by *DM*. At the same time, it warns to the *HA-LB* that a recovery action over its monitored service has been triggered. *HA-LB* starts to migrate the new requests to the replica, monitoring its primary service to know if there are on-going requests.

3) Faulty Service VM *elimination:* when the *HA-LB* detects that its monitored service is empty, *HA-LB* sends the order to *RHAM* to remove the previous *Service VM*. Then the *Replica VM* becomes the new *Service VM*.

The recovery process is based partially on the recovery technique presented in Chapter 2. It could be used to check the details of the process and its effectiveness to avoid the crash or unavailability of the service during the process. Although, in the previous approach we used a primary service and a hot standby service waiting. This approach consumes more resources to achieve the same result. This is possible because the prediction algorithm gives us an approximate time to crash, making possible to advance the creation of the replica with enough time and security.

4.3 Problem Formulation and Discussion

In previous section we have presented the framework proposed to manage a virtualized platform autonomously (or with minimum human intervention). We have presented the Decision Manager module which has the responsibility of looking for the optimal position of VMs to accept as much VMs as possible while guaranteeing the availability of the services deployed against software aging phenomena.

4.3.1 Problem Statement

The Decision Manager task could be reduced to well-known resource assignment problem (RAP) as follows: *For a given set of physical nodes, and for a given set of virtual machines with a size and availability requirements; decide to which physical node each*

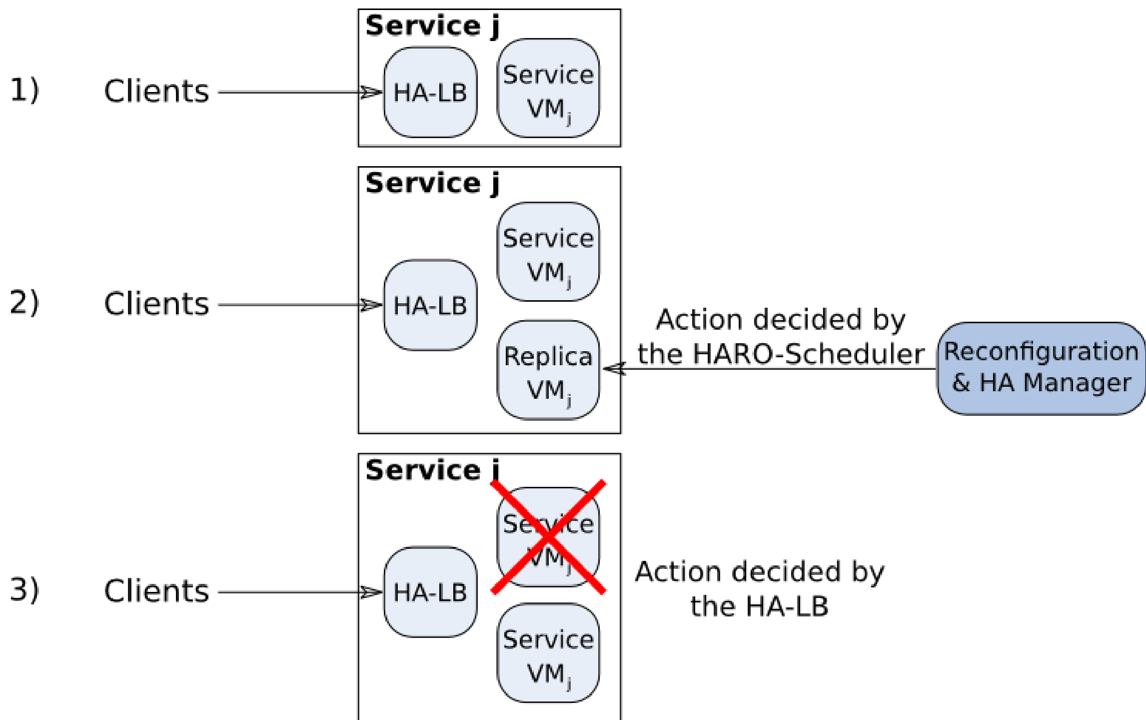


Figure 4.5 Recovery architecture

virtual machine must be allocated, such that number of virtual machines is maximized and virtual machine requirements are satisfied without exceeding physical nodes' capacity and whole platform capacity limits.

This type of problems could be formulated as a typical Constraint Satisfaction Problem (CSP). There are potentially many algorithms for tackling this kind of problems: greedy algorithms, Tabu search, genetic algorithms, and simulated annealing [74]. We chose *mathematical programming* (MP) [125] because it is a common and flexible technique for modeling a large class of optimization problems. A second reason to use MP is that there are commercially and free available MP solvers, extensively tested such as IBM ILOG CPLEX [65] or GLPK [54]. We have picked the last free and open source solver in our experiments. The third reason is that MP allows us to model a large number of hard constraints, a characteristic of RAP problems, which could be more complicated using other approaches such as genetic algorithms. However, MP requires good definition models to become useful.

4.3.2 Decision Manager Model Definition

In this section, we present the modeling of the previous described RAP problem and its reduction to a *binary integer programming*, a subset of MP, which allows us to find the

optimal solution.

We note and recall that from the DM perspective HA-LB and a service VM are the same: virtual machines. So, DM only observes virtual machines.

4.3.2.1 The Objective Functions

The problem input is the matrices and vector generated by Catcher Manager: $\widetilde{X}$, $\widetilde{X}'$ and W. On the other hand, the output of DM are other two matrices: X and X'. Matrix $\widetilde{X}$) represents where (which physical node) the virtual machines are currently deployed (if they are in) and $\widetilde{X}'$) presents where the *replica VM* are currently deployed. While X and X' represent the optimal position of all VMs (*service VMs*, *HA VMs* and *Replica VMs*). Following this definition, $X_{i,j} = 1$ indicates that VM j has to be in physical node i. Assuming that we have N physical nodes and M virtual machines to be allocated in our virtualized platform, we can define a first approach maximization function as:

$$Maximize \sum_{i=1}^{N} \sum_{j=1}^{M} X_{i,j} \tag{4.1}$$

However, Equation 4.1 assumes that all VMs have the same value for the provider. Then, the model could be generalized easily:

$$Maximize \sum_{i=1}^{N} \sum_{j=1}^{M} V_j \cdot X_{i,j} \tag{4.2}$$

where V_j represents the value we assign to the VM j. If a provider wants to assign more value to one type of VM in front of others, the provider only has to assign properly the V_j to the VM. In our case, we assign the same value to all VMs, so we have fixed $V_j = 1$.

This first maximization function (Equation 4.2) defines perfectly the maximization function we wanted to achieve: Maximize the number of services. However, there are two important actions in a virtualized platform to take into account: virtual machine creation and live-migration. The creation has an impact on the rest of virtual machines running on the same physical node where the new virtual machine would be created and live-migration could cause an outage (minimum or not) on the migrated virtual machine. We want to penalize those actions because they can impact on the availability and performance of the deployed services.

For this reason, we want to evaluate two different policies: Restrictive and Mixed.

The Restrictive model (RM) penalizes both actions. Modeling this penalization in the maximization function is presented in Equation 4.3

$$Maximize \sum_{i=1}^{N} \sum_{j=1}^{M} V_j \cdot X_{i,j} - C \sum_{i=1}^{N} \sum_{j=1}^{M} f(i,j) \tag{4.3}$$

where:

$$f(i,j) = \begin{cases} X_{i,j} & if \widetilde{X}_{i,j} = 0 \\ 0 & if \widetilde{X}_{i,j} = 1 \end{cases}$$

We defined the penalty factor as C. This factor has been defined to avoid that the penalization ($\sum_{i=1}^{N} \sum_{j=1}^{M} f(i,j)$) dominates the maximization function. In our experiments, $C = \frac{1}{2 \cdot M}$. This approach allows the model to migrate or adding more VMs if it is really needed. Function $f(i,j)$ penalizes creations and live-migrations. If $\widetilde{X}_{i,j}$ is 0, the VM j is not currently in node i. However, $\widetilde{X}_{i,j} = 1$ represents that in the current state, the VM j is in node i. So, if the optimal state indicates that $X_{i,j} = 1$, then any penalization has to be applied, the VM is in the same place. Otherwise, if the current position and the optimal position are different (a creation or migration), a penalization has to be applied. Note that a waiting VM column is all-zeros. If the VM is accepted, the optimal matrix will have one unique 1 in one row of its column. We must also penalize it.

Finally, the Mixed model (MM) sits on the fence between Restrictive model and no penalties at all. The Restrictive model penalizes the creation in the same way as the live-migrations. However, from the perspective of the platform provider, the creation impact could be accepted. If we penalize the creation of new virtual machines, the model could become in an underutilized infrastructure. Due to this reasoning, we decided to modify the Equation 4.3 to only penalize the live migrations, allowing as creations in the system as possible, thus avoiding to reject costumer requests when it is possible. The result is presented in next Equation:

$$Maximize \sum_{i=1}^{N} \sum_{j=1}^{M} V_j \cdot X_{i,j} - C \sum_{i=1}^{N} \sum_{j=1}^{M} f(i,j) \tag{4.4}$$

where:

$$f(i,j) = \begin{cases} 0 & \text{if } \sum_{k=1}^{N} \widetilde{X}_{k,j} = 0 \\ X_{i,j} & \text{if } \sum_{k=1}^{N} \widetilde{X}_{k,j} = 1 \end{cases}$$

We can observe that in the approach defined by Equation 4.4, only the live migrations are penalized. The penalization is performed according to where the virtual machine would be migrated. If in the matrix $\widetilde{X}$ generated by Catcher Manager, the column j which represents VM j is all zeros, then the VM j is waiting to be deployed. But, if there

is an 1 in the column j, this represents that VM was deployed and could be migrated or not. So, we only penalize the migration of the VMs deployed. In fact, we only penalize the current live-migrations.

4.3.2.2 Constraints

The three maximization functions need to carry out a set of constraints limited by the scenario and the goals we want to achieve.

- Every physical node has a limit on the number of virtual machines allowed and its limit cannot be exceeded.
- A virtual machine cannot be removed from the platform after it was deployed and a virtual machine is 1 or 0 in the platform.
- If a virtual machine crash is imminent, a replica virtual machine has to be deployed automatically, to guarantee the availability of the service.

First Constraint
The first constraint is fixed by the current virtualization technology used in the virtualized platforms. A VM cannot use resources from two physical nodes at the same time and cannot be divided between two physical nodes. In the same way, the number of VMs in a node is limited by the size of VMs deployed. The constraint has to guarantee that the current virtualization limitations are not violated. We note that during a short period of time the service and replica are running simultaneously. This fact has to be taken into account to count the running VMs. The constraint is defined by Equation 4.5:

$$\forall i, 1 \leq i \leq N, \sum_{j=1}^{M} W_j \cdot X_{i,j} + \sum_{j=1}^{M} W_j^{'} \cdot X_{i,j}^{'} \leq C_i \tag{4.5}$$

Where C_i is the maximum capacity of node i and W_j is the size of VM j. $X_{i,j}^{'}$ represents a replica of VM j running in node i and $W_j^{'}$ is its size. This constraint guarantees the model never adds a new virtual machine to the virtualized platform if there are not enough resources to run it.

However, in an ideal situation without any replica needed by the platform to guarantee the availability ($\forall i, 1 \leq i \leq N, \forall j, 1 \leq j \leq M, X_{i,j}^{'} = 0$), this constraint allows to consume all available resources in any node with new services ($X_{i,j}$). If this ideal situation becomes during enough time, all resources could be *saturated* ($\sum_{j=1}^{M} W_j \cdot X_{i,j} = C_i$). Then, if a replica is needed, the framework does not have enough space to conduct the recovery action process.

The models have to avoid this *saturation* situation, because it has an important impact on the availability of the services deployed. Our models reserve a part of the resources of the platform, to guarantee that there will be enough space to create replicas at any time, if it is needed. Or at least, we have enough space to create replicas in some order to avoid unplanned downtimes. We define the space reserved in the platform to offer high availability as β. The size of β defines the maximum number of simultaneous replicas on the infrastructure as follows:

$$\text{Number of simultaneous replicas} = \frac{\beta}{W_j}$$

assuming all replicas have the same size.

But we have to take into account the fact that it is impossible for a virtual machine use simultaneously resources from two different nodes. Following this idea, we have to allocate β_i space in every node to be used when we need to create a replica or a set of them. The β_i and the size of VMs defines the maximum number of simultaneous recovery process in the platform as follows:

$$\text{Total num. of simultaneous replicas at node } i = \frac{\beta_i}{W_j} \tag{4.6}$$

$$\text{Total num. of simultaneous replicas} = NumNodes \cdot \frac{\beta_i}{W_j} \tag{4.7}$$

(assuming all replicas have the same size W_j)

Following this first idea, we have to allocate/reserve β_i space in every node to be used when we need to create a replica or a set of them. Then, in an ideal world without any replica needed ($\forall j, 1 \leq j \leq M, TTC_j > TL$, where TL is Time Limit TL defined by the system administrator to trigger the replica creation) by any service the constraint would be as follows:

$$\textit{if } \forall j, 1 \leq j \leq M, TTC_j > TL$$

$$\textit{then } \forall i, 1 \leq i \leq N, \sum_{j=1}^{M} W_j \cdot X_{i,j} + \beta_i \leq C_i$$

Where M represents the number of total virtual machines.

Now, we analyze the situation where there is (at least) one service (already deployed, between $1 \leq j \leq M^{'}$) which needs a replica ($\exists j, 1 \leq j \leq M^{'}, TTC_j \leq TL$). The framework has to be ready to free the reserved space at some node (β_i) to create the replica. At the same time, the framework has to guarantee that the space released β_i is used solely to create the replica. The framework cannot use this space to create new services.

The first approach to guarantee the availability of the resources for replicas, it is to reject new services on the platform during the rejuvenation process. The following equation formalizes this approach:

$$\textit{if } \exists j, 1 \leq j \leq M^{'}, TTC_j \leq TL$$

$$\textit{then } \forall i, 1 \leq i \leq N, \sum_{j=1}^{M^{'}} W_j \cdot X_{i,j} + \sum_{j=1}^{M^{'}} W_J^{'} \cdot X_{i,j}^{'} \leq C_i$$

$$\textit{and } \forall j, M^{'} \leq j \leq M, \ \wedge \ \forall i, 1 \leq i \leq N, X_{i,j} = 0.$$

However, this approach is not the best option. Under these circumstances, we can not accept new services, even when the infrastructure has enough space to accept them during the rejuvenation process. Moreover, these rejuvenations processes could be triggered frequently avoiding to accept new services for long periods of time.

Our second approach of the constraint is focused on the idea of use the β_i of the same node where faulty VM (VM_j, where $TTC_j \leq TL$) is deployed. Meanwhile, the rest of nodes could accept new services. We define $M^{'}$ as the total number of running VMs and M as the total number of VMs, where $M^{'} \leq M$. The second part the constraint becomes as follows:

$$\forall j, 1 \leq j \leq M^{'}, TTC_j \leq TL$$

$$\textit{then, for } i, \textit{ where } X_{i,j} = 1, \ \sum_{j=1}^{M^{'}} W_j \cdot X_{i,j} + \sum_{j=1}^{M^{'}} W_j^{'} \cdot X_{i,j}^{'} \leq C_i$$

$$\wedge \forall j, M^{'} \leq j \leq M, X_{i,j} = 0.$$

$$\textit{While the rest of i's } \sum_{j=1}^{M} W_j \cdot X_{i,j} + \beta_i \leq C_i$$

We do not accept new services in the node i where we are applying a recovery action. This approach reduces the possibility to not accept new services at all, in front of the previous approach. Then, the final first constraint to replace constraint 4.5 to avoid the

saturation state of the platform is:

If $\forall j, 1 \leq j \leq M, TTC_j > TL$ then

$$\forall i, 1 \leq i \leq N, \sum_{j=1}^{M} W_j \cdot X_{i,j} + \beta_i \leq C_i$$

Otherwise $\forall j, 1 \leq j \leq M^{'}$, with $TTC_j \leq TL$.
For the unique i where $X_{i,j} = 1$:

$$\sum_{j=1}^{M^{'}} W_j \cdot X_{i,j} + \sum_{j=1}^{M^{'}} W_j^{'} \cdot X_{i,j}^{'} \leq C_i$$
$$\wedge \forall j, M^{'} \leq j \leq M, X_{i,j} = 0.$$

While the rest of i's:

$$\sum_{j=1}^{M} W_j \cdot X_{i,j} + \beta_i \leq C_i \tag{4.8}$$

Second Constraint
Due to the characteristics of MP modeling approach, we have to avoid the ubiquity of a VM. We accept that a customer deploys as many VM as he wants on the platform. So, a VM from one customer has to be deployed once. Moreover, we have to guarantee that the VMs already deployed on the virtualized platform have to be maintained on it. We cannot drive out any running service. This action will become in a crash of a service. We model this constraint in Equation 4.9:

$$\forall j, 1 \leq j \leq M^{'}, \sum_{i=1}^{N} X_{i,j} = 1$$

$$\wedge$$

$$\forall j, M^{'} \leq j \leq M, \sum_{i=1}^{N} X_{i,j} \leq 1 \tag{4.9}$$

Where $M^{'}$ represents the VMs deployed on the infrastructure and M the total number of VMs: the running VMs and waiting VMs. The first part of the constraint guarantees the running VMs ($\forall j, 1 \leq j \leq M^{'}$) have to be maintained on the environment. On the

other hand, the second part guarantees that the waiting VMs could be accepted only once.

Third Constraint

Finally, we need to guarantee that when a virtual machine crash is imminent, a replica is created: an availability constraint. We want that our model takes the responsibility to decide when a replica creation is needed and where is the best place to do it. As we have mentioned, we are able to know the Time To Crash of a VM, so we can model this constraint following Equation 4.10.

$$\forall j, 1 \leq j \leq M, f(X'_{i,j}) = \begin{cases} \sum_{i=1}^{N} X'_{i,j} = 1 & \text{if } TTC_j \leq TL \\ \sum_{i=1}^{N} X'_{i,j} = 0 & otherwise. \end{cases} \tag{4.10}$$

Where TTC_j is the time to crash of the VM j, and the TL (Time Limit) is a constant defined by the system administrators. So, if $TTC_j \leq TL$ a *Replica VM* of j has to be created, otherwise we don't have to do anything. The value of TL used is very important to guarantee the availability of the services in front of software failures. We suggest to the administrators that the best value of TL has to be a sum of the time to create and start a replica, the error introduced by the predictor, and a security margin. The security margin is added only to improve the robustness of the framework in front of the prediction system.

On the other hand, if $TTC_j > TL$, the framework has to guarantee that any replica has to be created. The replicas are only used when they are needed by the platform to guarantee the availability of the deployed services.

4.3.2.3 Restrictive Model (RM)

Maximization function:

$$Maximize \sum_{i=1}^{N} \sum_{j=1}^{M} V_j \cdot X_{i,j} - \frac{1}{2 \cdot M} \sum_{i=1}^{N} \sum_{j=1}^{M} f(i,j)$$

where:

$$f(i,j) = \begin{cases} X_{i,j} & if \widetilde{X}_{i,j} = 0 \\ 0 & if \widetilde{X}_{i,j} = 1 \end{cases}$$

Subject to:

1) If $\forall j, 1 \leq j \leq M, TTC_j > TL$ then

$$\forall i, 1 \leq i \leq N, \sum_{j=1}^{M} W_j \cdot X_{i,j} + \beta_i \leq C_i$$

Otherwise $\forall j, 1 \leq j \leq M^{'}$, with $TTC_j \leq TL$.
For the unique i where $X_{i,j} = 1$:

$$\sum_{j=1}^{M^{'}} W_j \cdot X_{i,j} + \sum_{j=1}^{M^{'}} W_j^{'} \cdot X_{i,j}^{'} \leq C_i$$
$$\wedge \, \forall j, M^{'} \leq j \leq M, X_{i,j} = 0.$$

While the rest of i's:

$$\sum_{j=1}^{M} W_j \cdot X_{i,j} + \beta_i \leq C_i$$

2)

$$\forall j, 1 \leq j \leq M^{'}, \sum_{i=1}^{N} X_{i,j} = 1$$

$$\forall j, M^{'} \leq j \leq M, \sum_{i=1}^{N} X_{i,j} \leq 1$$

3)

$$\forall j, 1 \leq j \leq M, f(X_{i,j}^{'}) = \begin{cases} \sum_{i=1}^{N} X_{i,j}^{'} = 1 & \text{if } TTC_j \leq TL \\ \sum_{i=1}^{N} X_{i,j}^{'} = 0 & otherwise. \end{cases}$$

4.3.2.4 Mixed Model (MM)

Maximization function:

$$Maximize \sum_{i=1}^{N} \sum_{j=1}^{M} V_j \cdot X_{i,j} - \frac{1}{2 \cdot M} \sum_{i=1}^{N} \sum_{j=1}^{M} f(i,j)$$

where:

$$f(i,j) = \begin{cases} 0 & \text{if } \sum_{k=1}^{N}, \widetilde{X}_{k,j} = 0 \\ X_{i,j} & \text{if } \sum_{k=1}^{N}, \widetilde{X}_{k,j} = 1 \end{cases}$$

Subject to:

1) If $\forall j, 1 \leq j \leq M, TTC_j > TL$ then

$$\forall i, 1 \leq i \leq N, \sum_{j=1}^{M} W_j \cdot X_{i,j} + \beta_i \leq C_i$$

Otherwise $\forall j, 1 \leq j \leq M^{'}$, with $TTC_j \leq TL$.
For the unique i where $X_{i,j} = 1$:

$$\sum_{j=1}^{M^{'}} W_j \cdot X_{i,j} + \sum_{j=1}^{M^{'}} W_j^{'} \cdot X_{i,j}^{'} \leq C_i$$
$$\wedge \forall j, M^{'} \leq j \leq M, X_{i,j} = 0.$$

While the rest of i's:

$$\sum_{j=1}^{M} W_j \cdot X_{i,j} + \beta_i \leq C_i$$

2)

$$\forall j, 1 \leq j \leq M^{'}, \sum_{i=1}^{N} X_{i,j} = 1$$

$$\forall j, M^{'} \leq j \leq M, \sum_{i=1}^{N} X_{i,j} \leq 1$$

3)

$$\forall j, 1 \leq j \leq M, f(X_{i,j}^{'}) = \begin{cases} \sum_{i=1}^{N} X_{i,j}^{'} = 1 & \text{if } TTC_j \leq TL \\ \sum_{i=1}^{N} X_{i,j}^{'} = 0 & otherwise. \end{cases}$$

4.3.3 Framework Availability Discussion

In this section we analyze the level of the availability offered by our architecture. Our approach to guarantee the availability of the services (a part of the Failure predictor module) is based on the resources reserved in every node to guarantee we have space (β and β_i) to create *replica VMs* in any moment. What happens if we have more services to rejuvenate than the infrastructure could conduct simultaneously: At one node (equation 4.6) or at whole infrastructure (equation 4.7). In other words, what happens if β or β_i are not enough space to create all replicas needed at an instant time.

To avoid this fact, the platform, specifically the Decision Manager, has to manage that situation. It maintains a queue of dangerous services sorted by TTC. At the moment of receiving the matrices from the Catcher Manager, Decision Manager only informs to the mathematical model the services with the smaller TTC. However, the number of replicas that the platform could create simultaneously is limited by β, β_i and the size of the VMs (W_j or W_j'). Based on these values, the Decision Manager only informs as many services as the platform could manage simultaneously. For example, if we have two VMs to rejuvenate in the same node, but the platform only could rejuvenate one of them according to the Equation 4.6, the Decision Manager only informs which virtual machine that has the smaller TTC to the model. The Decision Manager masks the real TTC of the other faulty virtual machine with a false TTC over the TL. The same behavior is conducted with the total number of simultaneous rejuvenations in the whole platform.

This approach avoids (or at least reduces the possibility) that the infrastructure could become saturated and another important factor: the possibility that the mathematical programming solver does not find the optimal solution due to the fact that the maximization problems are NP-Hard, making possible that we cannot find always the optimal solution. However, we have to be careful with the queue because the TTC could change and the queue has to be resorted using the new values of the TTC. For this reason, the Decision Manager has to monitor this queue to avoid a service crash due to *rejuvenation starvation* (a service was waiting too much in the replica queue).

According to this behavior, the value of β and β_i, defined by the administrators, has an important impact on the availability of the services. Furthermore, these values are used to calculate the number of replicas we can create simultaneously. This fact define the average time that a replica creation request could wait on the *replica queue*. If we want to prioritize the availability over resource usage optimization, β and β_i has to be bigger. If we want to prioritize the optimization resource usage, they could even be zero. Here, the system administrators have to achieve a trade-off between two goals according to the targets of the provider.

The administrators can take into account for example, statistical metrics to know how many replicas from different services are needed in the platform at the same time. This data can help to fix the size of β, and β_i. Another approach to fix the value of β_i can be to define different values in every node. For example, we can add some type of reasoning in the *Decision Manager* and *RHAM* processes. We queue the replica creation request and use only one fraction of one unique node to create the replicas in order to reduce the space reserved. However, this approach has to be conservative and prioritize the queue according to the time to crash of the VMs and the trend of this time, trying to reduce the waiting queue time of the replica requests as much as possible. Otherwise, if the waiting replica queue time is too long, it could happen that a waiting VM crashes before it was replaced.

4.4 Experimental Evaluation

4.4.1 Experimental Setup

The experiments conducted to test the functionality of our proposal has been performed in the testbed machines presented in Table 4.1. All machines use a Debian GNU/Linux 5.0 as Operating System and a XEN 3.3.2 hypervisor [26] provides paravirtualized environment and all the working nodes support NFSv4 as File System in order to apply live migration between nodes. The virtual machine creation over XEN is conducted using EMOTIVE [44].

	Experiment Role	CPU	Memory Size
Client	Clients	8 core at 2.83 GHz	16384MB
node0	Scheduler/Node	8 core at 2.83 GHz	16384MB
node1	Node	8 core at 2.60 GHz	16384MB
node2	Node	4 core at 3.0 GHz	16384MB
node3	Node	4 core at 2.66 GHz	16384MB
db0	DB	4 core at 2.66 GHz	16384MB
db1	DB	4 core at 3.16 GHz	16384MB
db2	DB	2 core at 2.40 GHz	16384MB

Table 4.1 Description of Testbed machines and their roles

All machines use XEN 3.3.2 to simulate a *virtualized data center*. Mainly, we use Nodes 0 to 3 to simulate it. On the other hand, the kind of virtual machines used to our three models are described in Table 4.2.

	Memory Size	Disk
Domain-0	4096MB	–
webserver	2048MB	3703MB
HA VM	256MB	801MB

Table 4.2 Description of Virtual Machines Type used

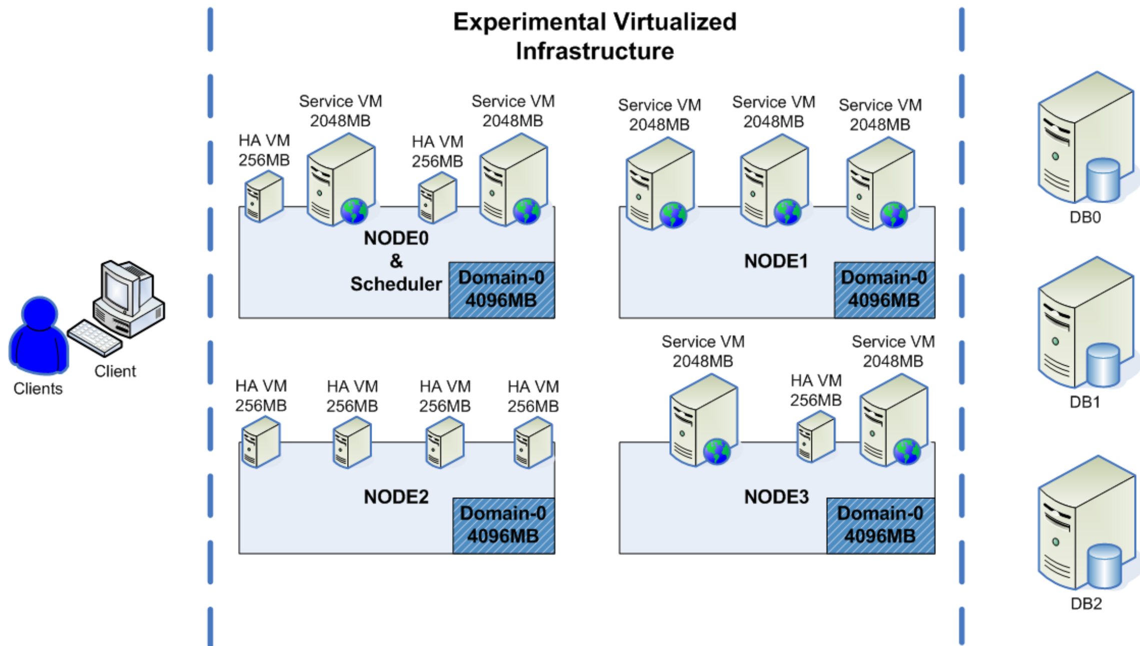


Figure 4.6 Experimental Scenario Proposed

Domain-0 is required and mandatory by the host operating system. So, the available memory in every physical node is the substraction Domain-0 memory required from total physical memory available. It is important to remark that there is a *Domain-0* VM in every node where we want to apply virtualization technology.

Figure 4.6 presents a diagram of the experimental scenario proposed to test the architecture and models previously presented.

We have used TPC-W benchmark [112] to emulate the services to be deployed on the virtualized infrastructure. To simulate the resource consumption which causes the software aging, we have modified a servlet of the TPC-W implementation. In our case, we have modified the *TPCW_search_request_servlet* class. This class calculates a random number between $0..N$ (in our experiments $N = 30$). This number determines how many clients have to request the Servlet before to inject a memory leak. This behavior makes that the variation of memory consumption depends of the number of clients and the frequency of servlet visits. According to the TPC-W specification, this frequency

depends on the workload: type and number of clients. TPC-W has three types of workload: Browsing, Shopping and Ordering. In our case, we have conducted all of our experiments using Shopping distribution. This distribution causes that the EBs visit the Servlet modified around 20% of times. This makes that with high workload our servlet injects quickly memory leaks, however with low workload, the consumption is lower too. The memory leak injected in our experiments is fixed to 1MB. As we can observe, using this experimental setup to simulate the software aging phenomena, it will be deterministic and constant. We have used this scenario to reduce the complexity. However, as we presented in Chapter 3, the M5P Machine Learning algorithm is able to predict the Time To Crash (TTC) with an acceptable accuracy in more complex scenarios.

We have defined a service as a web application server with TPC-W installed on a virtual machine. In our experiments, the databases used by the services are externalized to simplify the experimental environment because we had a limitation about physical nodes available. However, they could be managed by the infrastructure without problems. The software aging phenomena would be present only in the webservers. The HA-LB installed in every HA VM associated to every service, contains a Linux Virtual Server [79] which is able to balance load among different servers as we presented in detail in Chapter 2.

4.4.1.1 Analyzing the maximum capacity of our experimental environment

Taking into account the virtual machine and the physical node features, we can calculate the amount of VMs that fit in our experimental environment. We have used as a *size* of every VM only the memory required. But, other resources could be discussed. So, the *size* of the HA VM is 256MB and the VM *size* with TPC-W deployed is 2048MB as is showed in Table 4.2. Having just a web server per service, we would be able to get 24VMs. We have 16384MB of memory in every physical node as it was presented in Table 4.1, but every *Domain-0* needs 4096MB, so we have 12288MB available. We have 4 nodes in our experimental setup (Table 4.1 and Figure 4.6), so, we have $12288 \cdot 4 = 49152$MB in total in our scenario. If every VM needs 2048MB, then $\frac{49152}{2048}$=24 VMs are available to be deployed simultaneously on the platform.

If we want to deploy our services (with a HA VM associated to every *Service VM*), the maximum number of services (understood as Service VM + HA VM) allowed by the infrastructure would be 20 or 21, depending if we force to be together or not the HA VM and its service VM. If we do not allow to decouple HA VM and Service VM, $\lfloor\frac{12288}{2304}\rfloor = 5$ for every node. If we have 4 nodes, $4 \cdot 5 = 20$VMs. On the other hand, if we allow to decouple HA VM and its Service VM, $\lfloor\frac{49152}{2304}\rfloor = 21$ VMs. Our infrastructure is losing around of 17% or 12.5% of the resources to guarantee basically the availability of the

deployed services. This penalty is due directly to the architecture of our solution. Of course, these numbers are calculated without reserve any β or β_i to guarantee enough space to create replicas.

On the other hand, if we decide to allocate a part of every node to guarantee the replica space as it was described in Section 4.3.2.2, the maximum number of VMs depends on the value of β and β_i. In our experiments, we reserve $\beta_i = 2048$MB in every node. This value was chosen because a replica VM has that size. So, the maximum number of VMs would be 16 or 17, coupled or decoupled HA VM, respectively, becoming in a penalty of 33.3% or 29,1%.

However, if we decide to use a typical load balanced cluster solution with a load balancer and two service VMs (or even our previous solution presented in Chapter 2 composed by an intelligent load balancer and two service VMs, one active and one in stand by), only 8 or 11 services would be allowed. Depending on if the Load Balancer and the services are together or not. Then, these configurations pay around of 66.6% or 54.1% of the infrastructure resources to guarantee the availability. So, our architecture by itself reduces the resources dedicated to offer high availability against traditional approaches.

Table 4.3 resumes the maximum number of services described before, according to the different configurations.

	Maximum VMs allowed	Penalty associated
Basic Scenario	24	–
Our Scenario with HA VM coupled	20	17%
Our Scenario With HA VM decoupled	21	12.5%
Our Security Scenario with HA VM coupled ($\beta_i = 2048MB$)	16	33.3%
Our Security Scenario with HA VM decoupled ($\beta_i = 2048MB$)	17	29.1%
Load Balanced Scenario with LB coupled	8	66.6%
Load Balanced Scenario with LB decoupled	11	54.1%

Table 4.3 Maximum number of Services accepted by different solution scenarios

4.4.2 Experimental Results

In this section we present the results obtained after evaluating our infrastructure and the models proposed. First, we have conducted a set of experiments to evaluate the overhead introduced by our architecture, mainly, knowing the overhead introduced by

the creation of the *HA VM* associated with the service deployed by the customer.

Second, the live migrations could cause potential micro-outages [68]. However we want to know the real impact on performance of that technology in our scenario.

Finally, in Section 4.3.2 we have presented two main strategies gathered in two models: *Restrictive (RM)* and *Mixed (MM)*. To evaluate these strategies' effectiveness (to achieve the two goals: availability and resource optimization), we compare the two models using the following metrics: maximum services accepted by the models, number of migrations conducted, capacity to avoid the saturation process, and time needed to accept the maximum number of services.

The first metric evaluates if the model achieves the theoretical value obtained analytically, so if the maximization function works fine. The second metric evaluates the effectiveness of the penalizations and the potential risk of suffering micro outages during the process. The third metric evaluates the availability constraint (the third constraint of the models). It evaluates if the models are able to avoid the crash of a deployed service. The last metric is relevant in order to evaluate the capacity of the framework to accept new clients. If the framework causes that customers have to wait too much time to run their services, it is possible that customers move on to other providers. It is important to remark that this time could be affected by the number of rejuvenation actions conducted. The rejuvenation action in one node avoids the creation of new services in this node. Furthermore, this metric evaluates the impact of the creation penalization defined in the *Restrictive Model*.

4.4.2.1 Impact of the Framework Infrastructure

This experiment analyzes the overhead of our creation approach. The creation of the *HA VM* is automatic, transparent and simultaneous to the *Service VM* creation. The *HA VM* and *Service VM* are always created in the same physical node at first time. This conduct can be discussed because if we create the *HA VM* and the *Service VM* in different physical machines, we can reduce the overhead introduced by the creation process. But that approach has only sense in a cluster within the same network, because if the scenario was a Cloud (involving WANs) our approach has more sense in order to reduce the response time between the *HA VM* and its associated *Service VM*.

We have used EMOTIVE virtual machine creation process, which is described in [**?**]. We used the *node0* as the physical node to conduct the experiment. We define the creation process from the moment where the customer requests the creation of the VM until the moment of service is ready to attend the first task/request. We calculated how much time we need to create/use the *HA VM* alone, the *Service VM* alone and both created

simultaneously (Full Service creation).

	Average Time	Standard Deviation
HA VM Creation	34.6 seconds	± 11.1 seconds
Service VM creation	64.5 seconds	± 21.7 seconds
Full service creation	128.1 seconds	± 5.1 seconds

Table 4.4 Virtual Machines and service creation

Table 4.4 shows the creation time of *HA VM*, the *Service VM* and the both simultaneous creations. The creation time of the Full Service is around twice (128s) the time to run the same *Service VM* without creating the *HA VM* associated (64s). It shows clearly the important impact of creating simultaneously two virtual machines in the same node. However, this penalty could be acceptable if we really achieve an important level of availability. Furthermore, the full creation process is only conducted when the service is deployed the first time. According to these two facts, the penalty seems a good fair trade.

4.4.2.2 Impact of the Live-Migrations

Our second experiment evaluates the real impact of the live-migrations in our scenario. The live-migration has huge importance for our framework. Obtaining the optimal allocation of the VMs requires several live-migrations to reallocate deployed VMs to free space for new ones.

For this reason, we need to measure the potential outage on the service when the VM is live-migrated from one node to other. We have setup a web server that runs on top of the distributed shared file system described in [55], where every machine is able to access to the disks of the others. We stressed the webserver (with TPC-W installed) with a burst workload. TPC-W is configured to calculate a *thinking time* between requests of the same client. We modified TPC-W code to fix this *thinking time* to zero. Our first experiment was executed during 10 minutes with the burst workload and after 3 minutes of the execution, we triggered a live-migration. Figure 4.7 presents the throughput obtained during this experiment. We can observe how during the live-migration moment (red line) we do not observe a throughput degradation or sudden zero and any request was rejected by the server during the process.

After that, we decided to apply two live-migrations over the same service to validate the previous results. The execution was 15 minutes: 3 minutes of warm up, after that, trigger the first live-migration process, and 5 minutes later, a new live-migration process was triggered, and finished the experiment. Figure 4.8 shows clearly how the live-

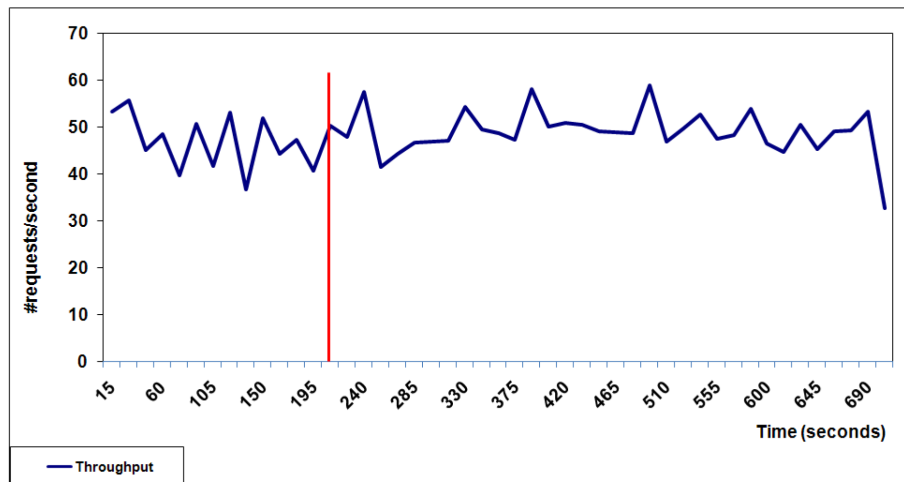


Figure 4.7 TPC-W Throughput with one live-migration

migration process has no impact on the performance of the service. This shows clearly the effectiveness of live-migration as a tool to increase the availability of the services in a coupled data center such as our physical nodes.

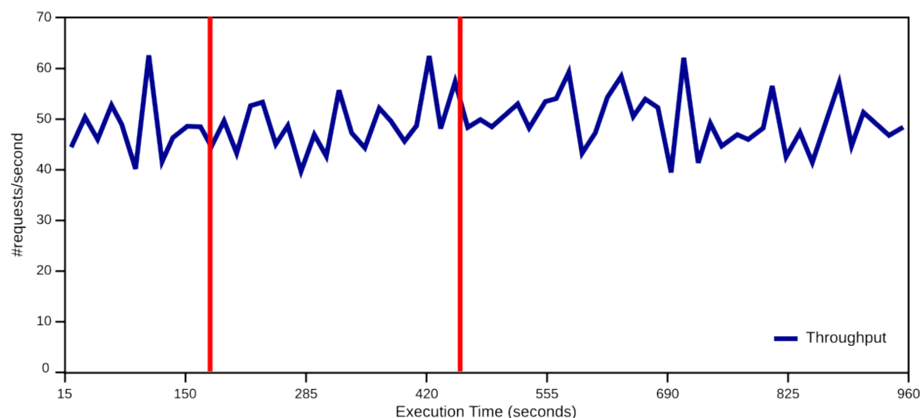


Figure 4.8 TPC-W Throughput with two live-migrations

4.4.2.3 Model Comparison

Finally, we evaluate the two presented strategies: *RM* and *MM*. *HARO-Scheduler* manages the four working nodes which will host the VMs. We also have three other machines (out of the *HARO-Scheduler* control) with the databases required by the web application.

The experiment consists of the submission of a new service every five minutes. We create 50% of services without software aging and 50% with it. The new creation requests are generated until the system reaches its maximum capacity, a *Service VM* crashes or a

Replica VM could not be created. The last reason will cause a future outage in one service due to the software aging. We defined this fact as *Saturation of the platform*, but if our models are well modeled, it will be avoided.

We use different workloads sizes in every service to vary the software aging trend in every faulty service, so the crash of every service will happen at different instants.

In terms of scheduling, every 60 seconds, every *FPdr* sends its TTC_j to the *HARO-Scheduler*. This via its *Decision Manager* runs the model used (*RM* or *MM*) to obtain the optimal allocation of VMs according to the current state.

We configured the parameters of our strategies as follows: $\beta_i = 2048MB, \forall i, 1 \leq i \leq N$, where $N = 4$ and $TL = 1000$ seconds. TL is calculated summing 900s from approximated error of the *FPdr* prediction and 100s as a security margin.

Once the system has reached its maximum capacity, the experiment lasts another 4000s in order to test the stability of the models and their behaviors.

We have also defined three base cases a part of the models defined previously (*RM* and *MM*): *Base case*, $C = 0$ model and $\beta = 0$ model. The base case represents a *RM* model without third constraint (availability Constraint) and no penalization ($C = 0$). So, the *Base case* only optimizes the resources available. $C = 0$ Model is the *RM* model without penalizations at all. The $\beta = 0$ model is the *RM* model without reserving any part of the resources to guarantee the service availability: β and β_i equal to zero: no-space for replicas was reserved.

These three models are used to evaluate the real impact of the penalizations introduced and the availability constraint defined in the models.

Maximum Services Accepted:

In Table 4.5 and Figure 4.9, we can observe how the *Base case* only accepts 12 services before one of them crashes due to the software aging introduced in the service. This model does not conduct any migration or replication because it was unnecessary. The $\beta = 0$ model shows how the platform can accept 21 services as we analyzed in the previous analytic evaluation of the maximum number of services acceptable by our experimental environment. At the same time, models $C = 0$, *RM* and *MM* achieve the maximum number of services: 17.

The number of replicas (Table 4.5) is more informative than quantifiable because, the number of replicas depends manly to the workload and software aging aggressiveness. However, this value demonstrates that the models with our framework are able to apply the necessary recovery actions.

Although, the number of migrations (Table 4.5) shows clearly how the penalty introduced in *Restrictive Model*, *Mixed Model* and $\beta = 0$ Model works perfectly

compared with $C = 0$ Model: 52 migrations against 11 (*RM*), 3 (*MM*) and 4 ($\beta = 0$ Model).

Model	Services	Replicas	Migrations
Base case	12	0	0
$\beta = 0$ model	21	6	4
$C = 0$ model	17	1	52
RM	17	4	11
MM	17	3	3

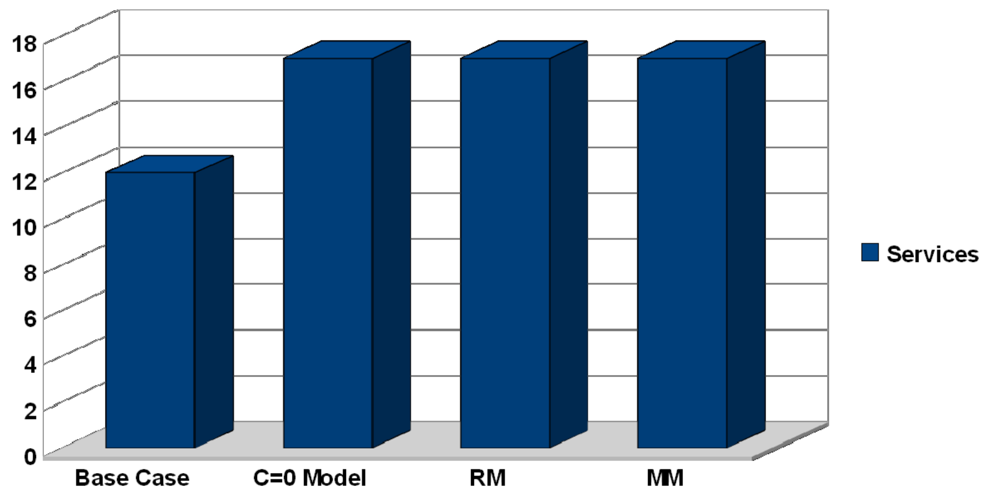
Table 4.5 Maximum services accepted by the models

Availability Achieved:

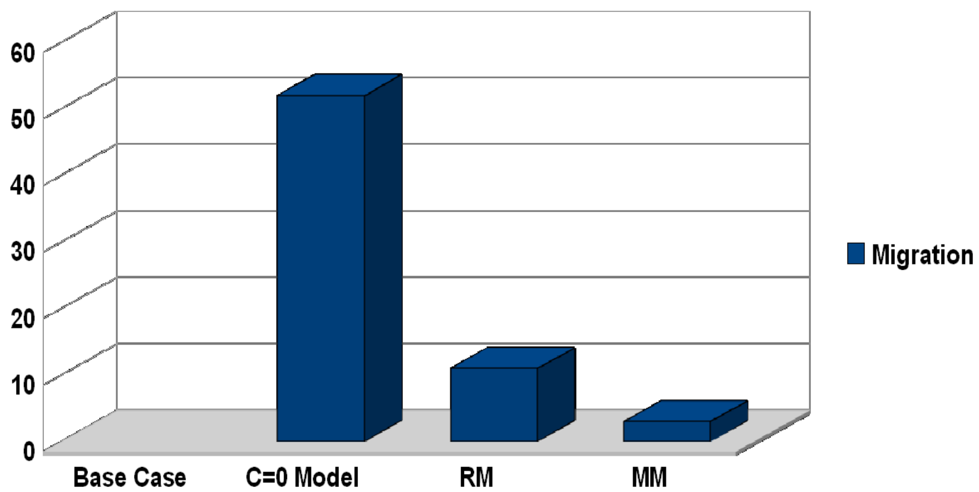
After confirming that models were valid to achieve the maximum number of services possible by the platform, we decided to evaluate how the β and β_i could influence on the availability of the services. We observed interesting results presented in Table 4.6. The *Base case* achieved the *saturation* after 3350s, while the $\beta = 0$ model saturates at 6958s. So, only using our model without preserve space explicitly for replicas, the continuous uptime is increased in more than 200%. The *saturation* time depends on the software aging trend defined in the services, but it is useful to use it as a baseline to compare the time alive of the rest of models, as well as, the improvement of our models to accept more services.

In terms of availability, if we assume in both cases (*Base Case* and $\beta = 0$ Model) the Mean Time To Recovery (MTTR) of the platform equal to 600s, and assuming the Mean Time To Failure (MTTF) equals to their *saturation times*, the availability of the *Base case* is 0.84 while the $\beta = 0$ Model achieves 0.92. This is an important improvement in terms of availability per year. However, if we assume that base case only has to restart 12 services and $\beta = 0$ model, 21 services the MTTR would be different. Assume that MTTR of Base case is 600s and the MTTR of $\beta = 0$ model is 1200s, then the availability would be 0.84 and 0.85 respectively. Even, in this case, our $\beta = 0$ model is better approach. In any case, our softare rejuvenation process was a MTTR equals to zero against software aging phenomena. This analysis is focused on the provider perspective, understanding that a failure of one VM becomes in unavailability of the platform.

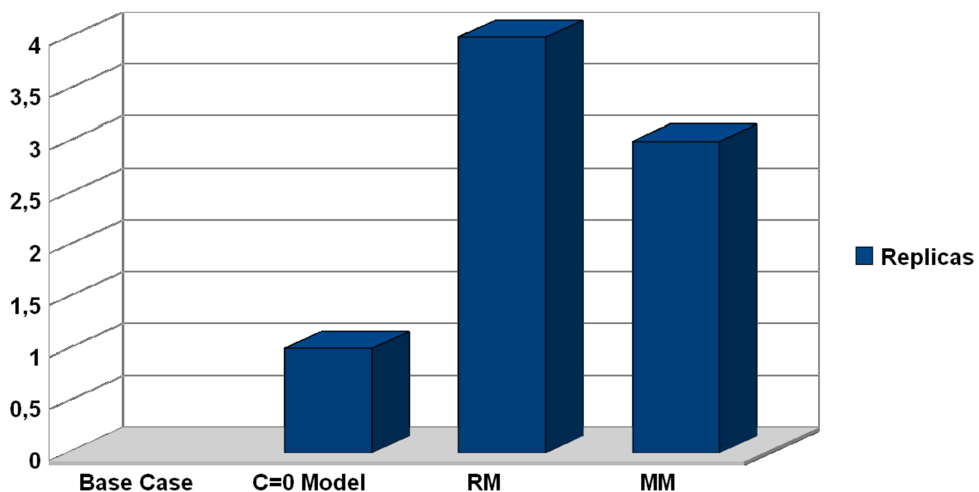
In the case of $C = 0$ model, *RM* and *MM*, they never saturated in our experiments because never happened that more than 2 replicas simultaneously were needed in the same node. So, in our experiments, they achieved 100% of availability against software aging phenomena.



(a) Services Accepted



(b) Live-Migrations Conducted



(c) Replicas Needed

Figure 4.9 Total number of a) Services, b) Migrations and c) Replicas by every model

Finally, we wanted to analyze the time needed by every model to create all services possible. We wanted to analyze the real impact of our penalization in the *Restrictive Model*. The *RM* model needed 8422s to accept the same services of $C = 0$ Model and *MM*. $C = 0$ Model needed only 5951s and *MM*, 5125s. These results show the important impact of our penalty, increasing in 2500s the time needed to achieve the maximum platform capacity. So, this means that the customers with *RM* have to wait too much.

According to results presented in Table 4.6 and Table 4.5, the best option is the *MM*: maximum services, maximum availability, minimum migrations.

Model	Max.Time	Saturation	Availability[a]
Base case	-	3351secs	0.84/0.84
$\beta = 0$ model	6958secs	6958secs	0.92/0.85
$C = 0$ model	5951secs	-	1
RM	8422secs	-	1
MM	5125secs	-	1

[a] Availability $= \frac{MTTF}{MTTF+MTTR}$

Table 4.6 Availability achieved by the models

Models Comparison:

In order to demonstrate the behavior of the Mixed Model, Figure 4.10 presents the evolution over time of the number of VMs and Services. We can observe that the number of services is always going up because any service crashed during the experiment. However, the number of VMs goes down around instants 3500, 4000 and 4100 because a faulty VM was replaced (3 replicas, Table 4.5). These instants show when a primary *Service VM* was replaced by a *Replica VM* after migrating the requests. This fact reduces the number of VMs on the platform, because during a short period of time the primary and replica are running simultaneously.

Figure 4.11a shows the number of VMs allocated in each node along the time. The figure shows the events (creating and destroying VMs) related to its instant time, this is the reason why the time scale is not homogeneous. It has a first stage, until second 3800, where the system is creating new services. We can observe how the model tries to allocate in a balanced way the new services between all nodes. As we penalize the migrations, the model is allocating the new services using every node until the node is full. For example, *node2* is not used until second 2800 approx. when the *node1* is full, assuming full that *HA VM* is coupled with its *Service VM*. When all nodes are full without apply any migration (coupling *HA VMs* with *Service VMs*), the model starts to migrate the VMs to accept more services (instant 4900s). We observe the three replica creation process (at 3500, 4000 and

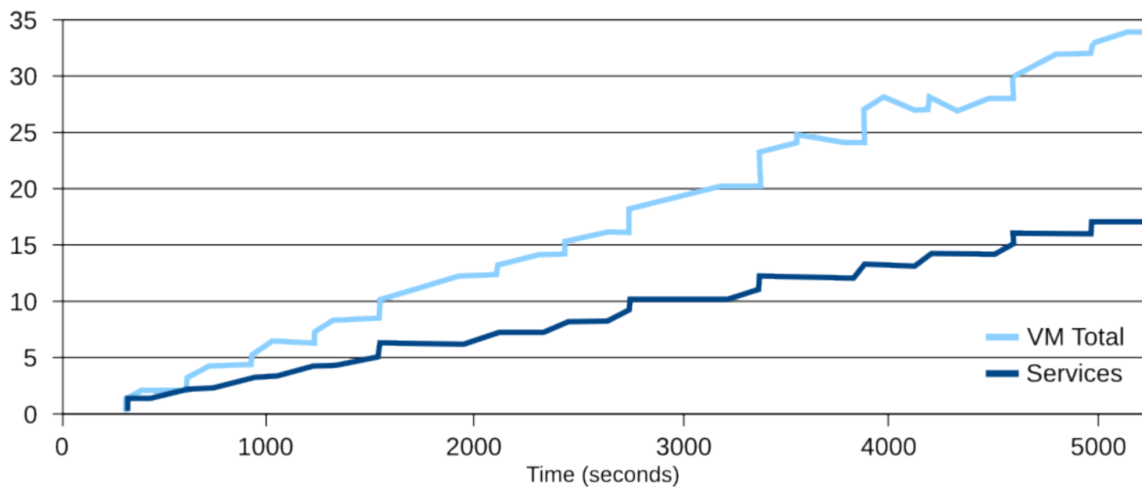


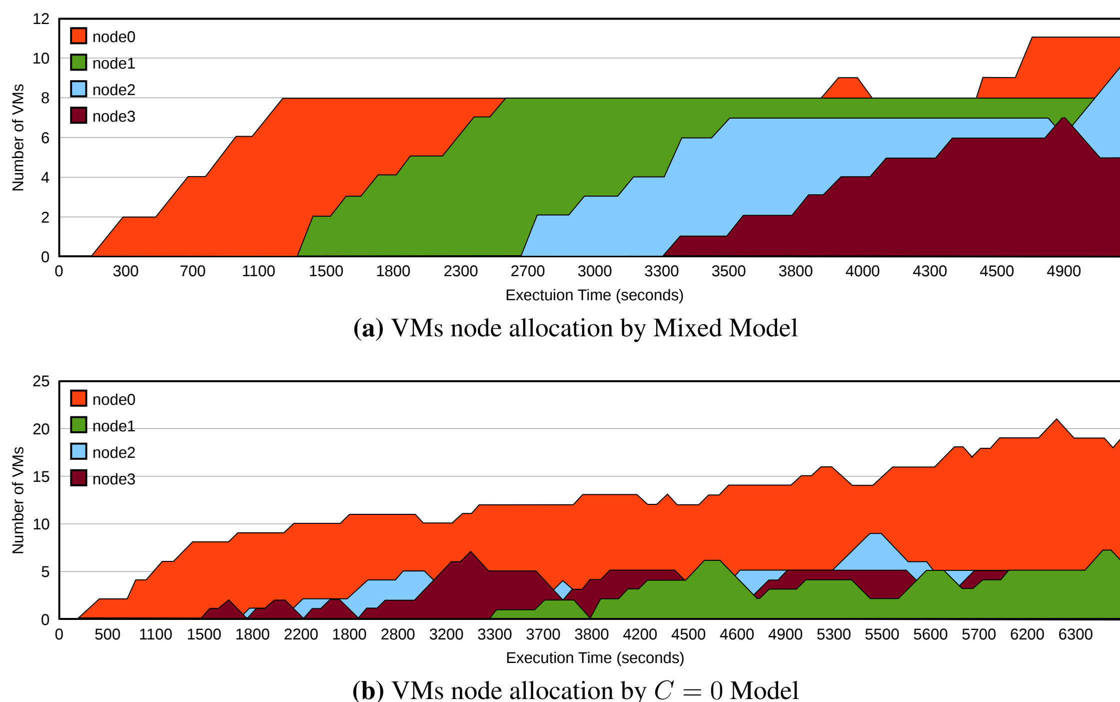
Figure 4.10 Experiment execution using MM: Amount of Services and VMs

4100 seconds).

For example, if we compare that behavior with $C = 0$ Model (no penalizations), the migration process starts too early when it is completely useless for the quality perceived by our customers as is showed in Figure 4.11b. This figure follows the same event-pattern as Figure 4.11a. We can observe how at instant 1700 *node3* creates 2 VMs (first peak) and 200s later, these two VMs are migrated to *node0*. This process is repeated twice more. This type of behavior causes an unnecessary number of migrations (52) because the model does not have memory of the platform state to know if it is a optimal state or not previously.

4.5 Related Work

The use of virtualization to consolidate services and improve the resource utilization is not a new idea and it has been evaluated largely in several studies [95, 119]. Furthermore, the usage of *mathematical programming* to optimize the service allocation is also not new. The MP approach has been used in several studies. Pmapper [116] uses the maximization function to improve the energy efficiency of a virtualized clusters. Furthermore, we can observe parallelism between their approach with our approach. They penalize the service migration because they assume that live-migration has an important energy cost. [109] presents a model to maximize the number of services to deploy in a virtualized data center, similar to our approach, but they do not take into account the availability as a factor on the model. [36] evaluates how to allocate every tier of a multi-tiered application to maximize the response time of those type of applications. It has also been used for minimizing the power consumption and maximizing the performance of a virtualized cluster [99].



(a) VMs node allocation by Mixed Model

(b) VMs node allocation by $C = 0$ Model

Figure 4.11 Comparison of VM allocation of *MM* (a) and $C = 0$ Model (b)

On the other hand, we can find several studies where the virtualization is used as an elegant way to improve the dependability and availability of the Software and Hardware systems [66]. Remus [42] uses asynchronous virtual machine replication to provide high availability to server in the face of hardware failures. In [37], authors present Mercury, an on-demand virtualization for HPC clusters. Mercury offers the possibility to switch from virtualized to native mode without important penalty. This approach is quite interesting because, they achieve to maintain the important capabilities of the virtualization like live-migration or checkpointing, reducing the overhead introduced by the virtualization layer. This solution could be integrated with our solution without miss any advantage of our approach. [75] presents the use of virtualization technologies to offer dependability in Avionics. Mainly, the work presented in [75] uses the virtualization technology to ensure dependability of critical avionic applications. The avionic applications have to communicate with off-board systems without the same level of validation that suffers on-board systems. To avoid the communication with off-board systems could be dangerous, the virtualization is used.

In [50] is presented a work similar to us. They presented a model to offer a self-reconfiguration system for HPC virtualized environments and propose a set of strategies to select the nodes to deploy the tasks to run. They take into account the performance and the reliability status of the node to be selected. They use a proactive mechanism

to calculate the probability that a node could fail while the task is running. They also evaluate the current workload on the node and both data, the best node is chosen.

[73] presents a very interesting approach to offer fast rejuvenation mechanism to avoid the consequences of software aging on the virtualization middleware more than virtual machines running over it. This solution could be integrated with our proposal to offer a full software rejuvenation mechanism on all layers of the virtualization stack.

4.6 Summary

In this chapter we have presented a self-reconfigurable framework for high availability and resource usage optimization in virtualized environments. The approach is based on a Resource Allocation Problem (RAP) definition. The problem formulation has been modeled using *mathematical programming* (MP). The model optimizes the resource usage while maintaining the availability of the services deployed. The availability is guaranteed by our prediction mechanism, the High Availability Virtual Machine (HA VM) and the constraint defined and added to the models.

We have presented two different approaches of the same target. Two actions have been under consideration: Creation and Live-migration. They can cause some impact on the performance and availability of the created/migrated service VM or the rest of services. Based on this fact, we have defined two approaches: a restrictive model (RM) which penalizes both actions and a mixed model (MM) which only penalizes the live migration. Moreover, we have conducted two other variations of these models as base cases. $C = 0$ Model represents the RM with no penalties and the Base case represents the RM without the availability constraint.

Our experiments show clearly how the Mixed Model is the best option, reducing drastically the number of live-migrations to achieve the same results of $C = 0$ Model and accepting the maximum allowed services by the platform in less time that RM and even $C = 0$ Model. Moreover, we have showed clearly that our framework, even in the worst case, increases the availability of the services of the platform from a model without the *availability constraint* added to the model. A Base case achieved a 0.84 of availability while the $\beta = 0$ model (a model with the availability constraint) achieves 0.92 or in the worst case 0.85. The final conclusion is that our framework achieves both goals proposed in the first lines of this chapter: Maximizing the number of services, guaranteeing the availability of the services deployed on the virtualized platform.

The work described in this chapter has resulted in submitted publication presented in Section 1.5.3.

Chapter 5

Conclusions and Future work

5.1 Conclusions

5.1.1 Proactive Software rejuvenation framework

In the area of proactive software rejuvenation we have presented a non-intrusive mechanism to avoid the unplanned downtimes caused by resource exhaustion, one of the most common causes of software aging. We have evaluated how the usage of virtualization can help us to offer a non-intrusive mechanism for even legacy systems and at the same time reduce the complexity of deploying the solution. Even, in this way the framework has no-impact on the IT budget because our approach avoids to buy new hardware.

Furthermore, our *virtualized clustering rejuvenation* mechanism avoids missing any (new or on-going) request during the rejuvenation action. Our solution triggers proactive rejuvenation actions when a threshold fixed by the system administrators is violated. During these rejuvenation actions, the availability perception of final users is 100%. We have evaluated the effectiveness of our solution in front a variety of scenarios (webservice, Grid, web) and configurations, showing its adaptability. Our experiments also show how the overhead penalty introduced by our solution according to the throughput or response time is acceptable, making effective our solution in production stage.

On the other hand, we have extended our approach, presenting a dissertation about a mechanism for legacy web applications based on the crash only concept. We have presented how to move on the crash only component concept to a whole application level. The idea is to create a wrapper to contain the legacy application and capture the communications of the container with another external components.

5.1.2 A framework for software aging prediction based on Machine Learning

Our previous threshold-based proactive solution had several weaknesses to become completely useful in production stage. First, we need to know *a priori* the resource which causes the software aging. Second, we need an expert to fix the optimal threshold value and third, a threshold-based solution could predict false positives, because we do not know the rate of the resource exhaustion, which also may vary overtime.

For these reasons, we decided to evaluate the effectiveness of Machine learning to overcome those weaknesses of our proactive solution, resulting in a predictive solution. More in detail, we have evaluated Machine Learning techniques to predict the time to crash due to resource exhaustion. Machine Learning helps us to determine which metrics

are relevant or not, without human intervention.

Our prediction assumption of software aging was based on the fact that software aging could seem non-linear, but it may be composed (or approximated) by a reasonable number of linear patches. Following this idea, we have evaluated a model tree called M5P. It is a Machine Learning algorithm that follows this idea, showing clearly that it is better than linear Regression.

On the other side, predicting time to crash numerically could be too hard even as a baseline. For this reason, we decided to divide the state of the system: Green (all works perfectly), Orange (warning), Red (imminent crash). Using this division, we have conducted an evaluation of three well-known classifiers to predict the system state. Although our results are not conclusive, they are interesting. In our experiments none of them showed clearly better capabilities to predict the state of the system. But it could be interesting to evaluate a prediction board composed by all of them (with different weights) because each of them is good for different system states. Another important point to remark it is the importance of the training phase. We have showed how the training phase could influence on the accuracy of the predictions.

Finally, during our experiments we saw the importance of the monitoring tools. We need more fine-grain and adaptable monitoring tools to detect the root cause of the resource exhaustion causing the software aging. Moreover, these fine-grain monitoring tools could help Machine Learning algorithms to predict with better accuracy. We have proposed a monitoring tool based on Aspect Oriented Programming paradigm to evaluate the resources consumed by every application component. Using this information our monitoring tool creates a map of suspicious components of the software aging. Furthermore, our approach uses a set of technologies that allow us to use it even in third party pieces of software.

5.1.3 Autonomic High Availability and Resource Usage Optimization

Finally, the last contribution of the thesis is a framework to manage virtualized environments to guarantee the availability of the services deployed and optimize the resources available in the infrastructure. We have proposed *Mathematical Programming* in order to model the resource assignment problem (RAP) proposed. The RAP problems are defined by a maximization or minimization function (in our case a maximization function) and a set of constraints. Our approach follows the idea of maximizing the number services according to the resources in order to achieve resource optimization. The availability warranty is achieved via constraints and architecture.

We have discussed in detail the mathematical model proposed to solve the RAP

problem and presented two different approaches: Restrictive Model and Mixed Model. The seed of these two models is the fact that there are two actions (in virtualized platforms) that could affect the availability and performance of the services deployed: creation and live-migration. The Restrictive Model applies a penalty to both actions. The Mixed Model only penalizes the live-migrations because it is the unique action that could affect the availability of the services.

The experiments conducted showed that the last one (MM) becomes the best option to manage the infrastructure. We have evaluated two models (and some concrete cases of these models) according to a set of metrics: number of services accepted, availability achieved, number of live-migrations conducted and the time needed to achieve the maximum number of services accepted. The *Mixed Model* offers the best trade off between number of services accepted and the availability offered as our experimental results showed.

5.2 Future Work

The work performed in this thesis opens several interesting paths that can be explored as a future work:

- The Rejuvenation framework presented in Chapters 2 and 3 is designed to offer high availability to Internet Services. However, currently is common to find clustered Internet Services. We can find several machines in cluster and one or several load balancers as front ends. All machines have the same application installed. This approach allows to attend more users (increasing the performance) and of course the availability. We have evaluated our approach in front of these scenarios. However, what happens if we have to apply the rejuvenation action on several machines at the same time? Some type of scheduling is necessary to avoid rejuvenating all machines at the same time. We want to evaluate different approaches from simple and well-known schedulers (such as First-In-First-Out (FIFO), Round-Robin or Shortest Job First (SJF)) until more sophisticated techniques such as Automated planning and scheduling [90], a branch of artificial Intelligence, in order to maintain the availability of the whole clustered application. On the other side, we want to extend our approach to all tiers of today common multi-tier applications: front-ends, business logical and databases.

- In Chapter 3, we have evaluated a set of Machine learning predictors and classifiers in order to predict the time to crash due to resource exhaustion. We will continue

evaluating other Machine learning algorithms and even try to build an ad hoc algorithm for this kind of phenomena. This is because it is possible that an ad hoc solution increases the accuracy compared to general purpose algorithms. We want to evaluate how the Machine Learning techniques could be used in a complementary way with other modeling approaches like Markov Chains or other modeling approaches. The comparison area has a quite importance to find the best option to predict the software aging phenomena.

- In our thesis, we have focused on offering software rejuvenation seeing the web applications as *black boxes*. Our unique approach to zoom into the applications was focused on monitoring tools. This first approach opens us a new architecture of web application servers: micro-rejuvenations using a hot standby component. SEDA architecture [122] proposes a new paradigm to design a web application container based on queues before every component. This architecture offers us a perfect scenario to apply a rejuvenation action at component level, using the queues as our current load balancer. We may be determine the component suspicious of the resource exhaustion and apply the micro rejuvenation only over this component, replacing it by a new or stand by component, reducing the MTTR.

- Chapter 4 has some still open questions. Our way to guarantee the availability of the services is giving up resources in every node (defined by β and β_i in our models). However, we believe that we can reduce that space reserved, reserving only space in a subset of nodes, instead of all of them. Then, at the moment to apply the recovery action, we could apply some intelligent strategy or scheduler (with ordered queues) to stagger the recovery actions and use the same space for all replicas. This is possible because the both virtual machines involved in the rejuvenation action (primary and replica) are running together only for a few seconds. Furthermore, we want to evaluate different values for TL and β or β_i to find the best values and help administrators to find the accurate configuration according to their environments and goals. Finally, we want to introduce the concept of memory on the system. If the current situation is optimal or anything has changed from the last model usage, we can avoid triggering the model again, reducing the number of times the model is used and the time consumed to its calculation. This would mean modifying or using new solver that takes the previous solution.

- There is another important point to take into account: the relationship between physical and virtual resources and in particular: The elasticity of the physical resources and its relationship with virtual ones. We can assign to a physical node

more vCPUs (virtual CPUs) than physical CPUs it has. However, the sum of RAM memory allocated to every virtual machine could never exceed the maximum physical RAM available on the node. We can see the CPU as an elastic resource and the RAM as an non-elastic resource. Moreover, in a fine-grain approach we can overload the CPUs, working over the 100%, paying a penalty on the performance (usually reflected on the response time metric). This penalty could be acceptable (from the perspective of the users and platform provider) if we can accept more services in our platform without reduce too much the QoS of the services deployed. The main future goal woulb be to extend the Mathematical programming formulation to deal with several resources at a time or all together when in addition some can be elastic.

- Finally, we want to introduce the concept of Green computing on our Mathematical Programming formulation. Today, the resource optimization is not the unique goal of the virtualized platform providers, they want to reduce the energy consumed because it is becoming an important part of the budget. We need to merge the resource optimization with an efficient and sustainable usage of resources to reduce as much as possible the energy consumption. We want to measure the real impact (in relation with energy consumption) of live migration and creation. It is interesting to evaluate how the energy consumed by a physical node varies according to the number of VMs deployed on it. Because, this must be taken into account by the model to optimize the resources in an economic way. Even more, it is quite interesting to evaluate the real cost (in energy) of every request. Because it could be interesting to advance the recovery action to avoid the performance degradation in order to reduce the watt per request. Because during the degradation process we are consuming (assumption) the same energy to achieve lower levels of performance.

List of Acronyms

AC Aspect Component. 115

AOP Aspect Oriented Programming. 114

CM Catcher Manager. 135

DM Decision Manager. 138

EB Emulated Browser. 29

FPdr Failure Predictor. 134

HA-LB High Availability Load Balancer. 134

HARO-Scheduler High Availability and Resources Optimization Scheduler. 134

IT Internet Services. 3

JMX Java Management Extensions. 114

LB Load Balancer. 53

LVS Linux Virtual Server. 27

MAE Mean Absolute Error. 85

ML Machine Learning. 12

MM Mixed Model. 143

MP Mathematical Programming. 141

MTTF Mean Time To Failures. 17

MTTR Mean Time To Repair. 17

OLTP OnLine Transaction Processing. 9

OS Operating System. 74

POST-MAE POST Mean Absolute Error. 86

PRE-MAE PRE Mean Absolute Error. 86

QoS Quality of Service. 20

RAID Redundant Array of Independent Disks. 22

RAP Resource Assignment Problem. 140

RH Request Handler. 60

RHAM Reconfiguration and High Availability Manager. 138

RM Recovery Manager. 60

RM Restrictive Model. 142

SLA Service Level Agreement. 38

S-MAE Soft Mean Absolute Error. 86

SMP Storage Management Proxy. 60

SOAP Simple Object Access Protocol. 22

SRA Software Rejuvenation Action. 27

TCO total cost of ownership. 20

TL Time Limit. 145

TTC Time To Crash. 71

VM Virtual Machine. 4

VMM Virtual Machine Monitor. 4

WAS Web Application Server. 114

Appendix A

Prediction Framework: System and Derived Metrics used

A.1 Detailed description of variables used in the Training Process

In this appendix we list the system metrics and the derived metrics used to build our prediction approaches (ML algorithms). We add a short description of the variables, though they have self-described names. This list a complement of the table 3.2.

- **Throughput:** The average number of requests per second received by the application in the last X seconds.
- **Workload:** The average number of requests per second processed by the application in the last X seconds.
- **Response Time:** The average response time per request processed by the application in the last X seconds.
- **Disk Used:** The average percentage of system disk used during the last X seconds.
- **Swap Free:** The average percentage of swap free during the last X seconds.
- **Number Processes:** The average number of processes running on the system during the last X seconds.
- **System Memory Used:** The average memory available on the system during the last X seconds.

- **Tomcat Memory Used:** The average percentage of memory used by the tomcat respect to the maximum available during the las X seconds. In our experiments the memory available for Tomcat was 1GB.

- **Number of Threads:** The average number of threads running on the system during the last X seconds.

- **Number of HTTP Connections:** The average number of in-HTTP connections to the application server during the last X seconds.

- **Number of Mysql Connections:** The average number of out-Mysql connections from the application server to the Database server during the last 15 seconds.

- **Maximum MB Young:** The maximum memory available in the Young Heap Zone.

- **Maximum MB Old:** The maximum memory available in the Old Heap Zone.

- **MB Young Used:** The MB used in the Young Heap Zone.

- **MB Old Used:** The MB used in the Old Heap Zone.

- **% Used Young:** The percentage used of the Young Heap Zone when the observation is collected.

- **% Used Old:** The percentage used of the Old Heap Zone when the observation is collected.

- **Sliding Window Average (SWA) Young Variation:** The sliding window average of the Young Heap Zone variation. Every observation used to calculate the SWA is the difference of the current Young Heap Zone used and the previous Young Heap Zone used metric collected.

- **Sliding Window Average (SWA) Old Variation:** The sliding window average of the Old Heap Zone variation. Every observation used to calculate the SWA is the difference of the current Old Heap Zone used and the previous Old Heap Zone used metric collected.

- **SWA Number of Threads variation:** The sliding window average of the Number of threads variation. Every observation used to calculate the SWA is the difference of the current number of threads and the previous value collected.

- **SWA Tomcat memory used variation:** The sliding window average of the Tomcat memory used variation. Every observation used to calculate the SWA is the difference of the current Tomcat memory used and the previous value collected.
- **SWA System memory used variation:** The sliding window average of the System memory used variation. Every observation used to calculate the SWA is the difference of the current System memory used and the previous value collected.
- **SWA Tomcat memory used variation/throughput:** The normalized sliding window average of the Tomcat memory used variation. Every observation used to calculate the SWA is the difference of the current Tomcat memory used and the previous value collected. The SWA calculated is divided by the current throughput.
- **SWA System memory used variation/throughput:** The normalized sliding window average of the System memory used variation. Every observation used to calculate the SWA is the difference of the current System memory used and the previous value collected. The SWA calculated is divided by the current throughput.
- **Sliding Window Average (SWA) Young Variation/Throughput:** The normalized sliding window average of the Young Heap Zone variation. Every observation used to calculate the SWA is the difference of the current Young Heap Zone used and the previous Young Heap Zone used metric collected. The SWA calculated is divided by the current throughput.
- **Sliding Window Average (SWA) Old Variation/Throughput:** The normalized sliding window average of the Old Heap Zone variation. Every observation used to calculate the SWA is the difference of the current Old Heap Zone used and the previous Young Heap Zone used metric collected. The SWA calculated is divided by the current throughput.
- **1/SWA Number of Threads variation:** The inverse of sliding window average of the Number of threads variation. Every observation used to calculate the SWA is the difference of the current number of threads and the previous value collected.
- **1/SWA Tomcat memory used variation:** The inverse of sliding window average of the Tomcat memory used variation. Every observation used to calculate the SWA is the difference of the current Tomcat memory used and the previous value collected.
- **1/SWA System memory used variation:** The inverse of sliding window average of the System memory used variation. Every observation used to calculate the SWA is the difference of the current System memory used and the previous value collected.

- **1/SWA Young Variation:** The inverse of sliding window average of the Young Heap Zone variation. Every observation used to calculate the SWA is the difference of the current Young Heap Zone used and the previous Young Heap Zone used metric collected.

- **1/SWA Old Variation:** The inverse of sliding window average of the Old Heap Zone variation. Every observation used to calculate the SWA is the difference of the current Old Heap Zone used and the previous Old Heap Zone used metric collected.

- **Young used / SWA Young Variation:** Division of current Young Heap zone used by the sliding window average of the Young Heap Zone variation. Every observation used to calculate the SWA is the difference of the current Young Heap Zone used and the previous Young Heap Zone used metric collected.

- **Old Used / SWA Old Variation:** Division of current Old Heap Zone used by the the sliding window average of the Old Heap Zone variation. Every observation used to calculate the SWA is the difference of the current Old Heap Zone used and the previous Old Heap Zone used metric collected.

- **Number of Threads/SWA Number of Threads variation:** Division of the number of threads by the sliding window average of the Number of threads variation. Every observation used to calculate the SWA is the difference of the current number of threads and the previous value collected.

- **Tomcat Memory Used/SWA Tomcat memory used variation:** Division of the average tomcat memory used in the last X seconds by the sliding window average of the Tomcat memory used variation. Every observation used to calculate the SWA is the difference of the current Tomcat memory used and the previous value collected.

- **System Memory Used/ SWA System memory used variation:** Division of the average System memory available in the last X seconds by the sliding window average of the System memory used variation. Every observation used to calculate the SWA is the difference of the current System memory used and the previous value collected.

- **1/SWA Tomcat memory used variation/TH:** The normalized inverse of sliding window average of the Tomcat memory used variation. Every observation used to calculate the SWA is the difference of the current Tomcat memory used and the previous value collected.

- **1/SWA System memory used variation/TH:** The normalized inverse of sliding window average of the System memory used variation. Every observation used to calculate the SWA is the difference of the current System memory used and the previous value collected.
- **1/SWA Young Variation/TH:** The normalized inverse of sliding window average of the Young Heap Zone variation. Every observation used to calculate the SWA is the difference of the current Young Heap Zone used and the previous Young Heap Zone used metric collected.
- **1/SWA Old Variation/TH:** The normalized inverse of sliding window average of the Old Heap Zone variation. Every observation used to calculate the SWA is the difference of the current Old Heap Zone used and the previous Old Heap Zone used metric collected.
- **Tomcat Memory Used/SWA Tomcat memory used variation/TH:** Normalized division of the average tomcat memory used in the last X seconds by the sliding window average of the Tomcat memory used variation. Every observation used to calculate the SWA is the difference of the current Tomcat memory used and the previous value collected.
- **System Memory Used/ SWA System memory used variation/TH:** Normalized division of the average System memory available in the last X seconds by the sliding window average of the System memory used variation. Every observation used to calculate the SWA is the difference of the current System memory used and the previous value collected.
- **Young used / SWA Young Variation:** Normalized division of current Young Heap zone used by the sliding window average of the Young Heap Zone variation. Every observation used to calculate the SWA is the difference of the current Young Heap Zone used and the previous Young Heap Zone used metric collected.
- **Old Used / SWA Old Variation:** Normalized division of current Old Heap Zone used by the the sliding window average of the Old Heap Zone variation. Every observation used to calculate the SWA is the difference of the current Old Heap Zone used and the previous Old Heap Zone used metric collected.
- **SWA of Response Time:** Sliding window average of Response Time values.
- **SWA of Throughput:** Sliding window average of Throughput values.

- **SWA of System memory:** Sliding window average of System memory used values.
- **SWA of Tomcat memory:** Sliding window average of Tomcat memory used values.
- **Time to Failure:** The time to failure.

Note: In our experiments $X = 15$ (seconds).

Appendix B

Extended Prediction Algorithms comparison

B.1 Introduction

In this Appendix, we present the results obtained by other more sophisticated (and higher computational cost) machine learning algorithms using the same training data set and testing data set used with M5P. These results show the good trade off obtained by M5P between accuracy and training data set size.

We have divided the results following the same order of the experiments presented in section 3.4. The Machine Learning used are: Linear Regression (Lin. Reg), Support Vector Machines (SVM), Bagging M5P (Bag. M5P) and Bagging Linear Regression (Bag. Lin. Reg.). It is important to remark that all algorithms are included in WEKA distribution [121] and default values.

B.2 Deterministic Software aging

We observe in Table B.1 how only the Bagging M5P (an improvement of M5P) performs better. The rest of more sophisticated algorithms obtain poor results.

B.3 Dynamic and Variable Software aging

In this comparison (Table B.2) we observe an interesting facts. In average all algorithms, except M5P, obtain complete unacceptable results. However, if we focus on the last 10 minutes of the experiment, we observe how SVM or Linear Regression obtains

	Lin. Reg	M5P	SVM	Bag. M5P	Bag. Lin. Reg.
75EBs MAE	19min 35secs	15min 14secs	25min 48secs	14min 42secs	20min 44secs
75EBs S-MAE	14min 17secs	9min 34secs	20min 21secs	8min 57secs	19min 36secs
150EBs MAE	20min 24secs	5min 46secs	21min 33secs	4min 54secs	19min 36secs
150EBs S-MAE	17min 24secs	2min 52secs	18min 34secs	2min 12secs	16min 41secs
75EBs PRE-MAE	21min 13secs	16min 22secs	27min 37secs	15min 48secs	22min 7secs
75EBs POST-MAE	5min 11secs	2min 20secs	5min 11secs	2min 20secs	5min 11secs
150EBs PRE-MAE	19min 40secs	6min 18secs	21min 49secs	5min 18sec	19min 4secs
150EBs POST-MAE	24min 14secs	2min 57secs	20min 10secs	2min 51secs	22min 20secs

Table B.1 MAE S-MAE PRE and POST MAE obtained under constant software aging experiment by a diverse set of ML algorithms

	Lin. Reg.	M5P	SVM	Bag. M5P	Bag. Lin. Reg.
MAE	$3.30x10^{14}$secs	16min 26secs	$1.2x10^{15}$secs	$2.4x10^{12}$secs	$2.69x10^{14}$secs
S-MAE	$3.30x10^{14}$secs	13min 3secs	$1.29x10^{15}$secs	$2.4x10^{12}$secs	$2.69x10^{14}$secs
PRE-MAE	$3.60x10^{14}$secs	17min 15secs	$1.43x10^{15}$secs	$2.7x10^{12}$secs	$2.9x10^{14}$secs
POST-MAE	**5min 14secs**	8min 14secs	**2min 20secs**	8min 32secs	5min 10secs

Table B.2 MAE S-MAE PRE and POST MAE obtained under dynamic and variable software aging experiment by a diverse set of ML algorithms

very good results. However, if analyze the results we observed how the large errors are due mainly to outliers predictions. And the good results in the last 10 minutes were because these algorithms start to predict 0 seconds to crash 7 minutes before the crash. So, this behavior could be if we are able to fix it as an alert of imminent crash.

B.4 Dynamic and Variable Software aging due to Two Resources

	Lin. Reg.	M5P	SVM	Bag. M5P	Bag. Lin. Reg.
MAE	1h 4min 15secs	16min 52secs	53min 38secs	17min 53secs	1h 0min 55secs
S-MAE	57min 20secs	13min 22secs	48min 57secs	14min 58secs	55min 26secs
PRE-MAE	1h 7min 36secs	18min 16secs	58min 3secs	19min 6secs	1h 4min 50secs
POST-MAE	28min 42secs	2min 5secs	6min 48secs	4min 55secs	19min 21secs

Table B.3 MAE S-MAE PRE and POST MAE obtained under dynamic and variable software aging due to two resources experiment by a diverse set of ML algorithms

In Table B.3 we can observe how SVM obtains good results in the last 10 minutes of the experiment. However, this result is not completely real because again SVM is too pessimistic. SVM starts to predict "0 seconds to crash" too early, becoming useless to start the rejuvenation.

Bibliography

[1] J. Alonso, J. L. Berral, R. Gavaldà, and J. Torres. Adaptive on-line software aging prediction based on machine learning. In *Proceedings of The 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2010), Chicago, IL, 2010.*, pages –, 2010.

[2] J. Alonso, J. L. Berral, R. Gavaldà, and J. Torres. J2ee instrumentation for software aging root cause application component determination with aspectj. In *(IEEE DPDNS 2010), In Proceedings of the 15th IEEE Workshop on Dependable Parallel, Distributed and Network-Centric Systems, in conjunction with the 24rd International Parallel and Distributed Processing Symposium IPDPS 2010*, Washington, DC, USA, 2010. IEEE Computer Society.

[3] J. Alonso, I. Goiri, J. Guirtart, R. Gavaldà, and J. Torres. High availability on virtualized platforms with minimal physical resource impact. In *Submitted to Revision*, 2010.

[4] J. Alonso, L. Silva, A. Andrzejak, P. Silva, and J. Torres. High-available grid services through the use of virtualized clustering. In *GRID '07: Proceedings of the 8th IEEE/ACM International Conference on Grid Computing*, pages 34–41, Washington, DC, USA, 2007. IEEE Computer Society.

[5] J. Alonso, J. Torres, and R. Gavaldà. Predicting web server crashes: A case study in comparing prediction algorithms. In *ICAS '09: Proceedings of the 2009 Fifth International Conference on Autonomic and Autonomous Systems*, pages 264–269, Washington, DC, USA, 2009. IEEE Computer Society.

[6] J. Alonso, J. Torres, R. Griffith, and G. K. L. M. Silva. Towards self-adaptable monitoring framework for self-healing. In *Proceedings of the 3rd CoreGrid Workshop on Middleware 2008, Spain*, 2008.

[7] J. Alonso, J. Torres, R. Griffith, and G. K. L. M. Silva. *Grid and Services Evolution*, chapter Towards Self-adaptable Monitoring Framework for Self-healing, pages 1–9. Springer US, 2009.

[8] J. Alonso, J. Torres, and L. Moura Silva. Carrying the crash-only software concept to the legacy application servers. In *CoreGRID Workshop on Grid Programming Model, Grid and P2P Systems Architecture, Grid Systems, Tools and Environments*, Heraklion, Crete (Greece), June 2007.

[9] J. Alonso, J. Torres, and L. Silva. *Making Grids Work*, chapter Carrying the Crash-Only Software Concept to the Legacy Application Servers, pages 165–174. Springer US, 2008.

[10] J. Alonso, J. Torres, L. M. Silva, and P. Silva. Dependable grid services: A case study with ogsa-dai. In *Proceedings of the CoreGRID Symposium 2007, August 27-28, Rennes, France*, pages 291–300, 2007.

[11] J. Alonso, J. Torres, L. M. Silva, and P. Silva. *Towards Next Generation Grids*, chapter Dependable Grid Services: A Case Study with OGSA-DAI, pages 291–300. Springer US, 2007.

[12] A. Andrzejak and L. Silva. Using machine learning for non-intrusive modeling and prediction of software aging. In *In IEEE/IFIP Network Operations & Management Symposium (NOMS 2008*, pages 7–11, 2008.

[13] A. Andrzejak, L. Silva, L. Silva, and C. Tr. Deterministic models of software aging and optimal rejuvenation schedules. In *In 10th IFIP/IEEE Symposium on Integrated Management (IM 2007*, 2007.

[14] Apache Axis. *Axis*
http://ws.apache.org/axis/.

[15] Apache Software Foundation. *Apache Server*
http://httpd.apache.org/docs/.

[16] Apache Tomcat. *Apache Tomcat*
http://tomcat.apache.org/.

[17] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. Fail-stutter fault tolerance. In *HOTOS '01: Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, page 33, Washington, DC, USA, 2001. IEEE Computer Society.

[18] AspectC. *AspectC* *http://www.aspectc.org*.

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

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

[21] R. Barga and D. Lomet. Phoenix project: fault-tolerant applications. *SIGMOD Rec.*, 31(2):94–100, 2002.

[22] R. Barga, D. Lomet, S. Paparizos, H. Yu, and S. Chandrasekaran. Persistent applications via automatic recovery. *Database Engineering and Applications Symposium, International*, 0:258, 2003.

[23] R. Barga, D. Lomet, G. Shegalov, and G. Weikum. Recovery guarantees for internet applications. *ACM Trans. Internet Technol.*, 4(3):289–328, 2004.

[24] R. S. Barga and D. B. Lomet. Measuring and optimizing a system for persistent database sessions. In *Proceedings of the 17th International Conference on Data Engineering*, pages 21–30, Washington, DC, USA, 2001. IEEE Computer Society.

[25] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using magpie for request extraction and workload modelling. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 18–18, Berkeley, CA, USA, 2004. USENIX Association.

[26] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM.

[27] L. Bernstein, Y. D. Yao, and K. Yao. Software avoiding failures even when there are faults. *The DoD SoftwareTech News*, 6(2):8–11, Oct 2003.

[28] C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[29] P. Bohrer, E. N. Elnozahy, T. Keller, M. Kistler, C. Lefurgy, C. McDowell, and R. Rajamony. The case for power management in web servers. pages 261–289, 2002.

[30] O. Bousquet, S. Boucheron, and G. Lugosi. Introduction to statistical learning theory. In *In , O. Bousquet, U.v. Luxburg, and G. Rsch (Editors*, pages 169–207. Springer, 2004.

[31] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen. *Classification and Regression Trees*. Chapman and Hall/CRC, 1 edition, January 1984.

[32] G. Candea and A. Fox. Crash-only software. In *HOTOS'03: Proceedings of the 9th conference on Hot Topics in Operating Systems*, pages 12–12, Berkeley, CA, USA, 2003. USENIX Association.

[33] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot — a technique for cheap recovery. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 3–3, Berkeley, CA, USA, 2004. USENIX Association.

[34] K. J. Cassidy, K. C. Gross, and A. Malekpour. Advanced pattern recognition for detection of complex software aging phenomena in online transaction processing servers. In *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 478–482, Washington, DC, USA, 2002. IEEE Computer Society.

[35] V. Castelli, R. E. Harper, P. Heidelberger, S. W. Hunter, K. S. Trivedi, K. Vaidyanathan, and W. P. Zeggert. Proactive management of software aging. *IBM J. Res. Dev.*, 45(2):311–332, 2001.

[36] A. S. R. T. R. Z. A. Z. Y. Chaudhuri, Kamalika; Kothari. Server allocation problem for multi-tiered applications. Technical Report HPL-2004-151, HP Labs, 2004.

[37] H. Chen, R. Chen, F. Zhang, B. Zang, and P.-C. Yew. Mercury: Combining performance with dependability using self-virtualization. In *ICPP '07: Proceedings of the 2007 International Conference on Parallel Processing*, page 9, Washington, DC, USA, 2007. IEEE Computer Society.

[38] M. Y. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer. Path-based faliure and evolution management. In *NSDI'04: Proceedings of the*

1st conference on Symposium on Networked Systems Design and Implementation, pages 23–23, Berkeley, CA, USA, 2004. USENIX Association.

[39] L. Cherkasova, K. M. Ozonat, N. Mi, J. Symons, and E. Smirni. Anomaly? application change? or workload change? towards automated detection of application performance anomaly and change. In *The 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2008, June 24-27, 2008, Anchorage, Alaska, USA, Proceedings*, pages 452–461. IEEE Computer Society, 2008.

[40] I. Cohen, M. Goldszmidt, T. Kelly, J. Symons, and J. S. Chase. Correlating instrumentation data to system states: a building block for automated diagnosis and control. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 16–16, Berkeley, CA, USA, 2004. USENIX Association.

[41] F. Cucker and S. Smale. On the mathematical foundations of learning. *Bulletin of the American Mathematical Society*, 39:1–49, 2002.

[42] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: high availability via asynchronous virtual machine replication. In *NSDI'08: Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 161–174, Berkeley, CA, USA, 2008. USENIX Association.

[43] T. Dohi, K. Goseva-popstojanova, and K. S. Trivedi. Analysis of software cost models with rejuvenation. In *Proc. of the IEEE Intl. Symp. on High Assurance Systems Engineering, HASE-2000, November 2000. ? Statistical Non-Parametric Algorithms to Estimate the Optimal Software Rejuvenation*, pages 25–34, 2000.

[44] EMOTIVE Cloud Barcelona. *EMOTIVE http://www.emotivecloud.net*.

[45] M. Engel and B. Freisleben. Supporting autonomic computing functionality via dynamic operating system kernel aspects. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 51–62, New York, NY, USA, 2005. ACM.

[46] Essential about Java Servlet Filters. *Java Servlet Filters http://java.sun.com/products/servlet/Filters.html*.

[47] R. Figueiredo, P. A. Dinda, and J. Fortes. Guest editors' introduction: Resource virtualization renaissance. *Computer*, 38(5):28–31, 2005.

[48] R. J. Figueiredo, P. A. Dinda, and J. A. B. Fortes. A case for grid computing on virtual machines. In *ICDCS '03: Proceedings of the 23rd International Conference on Distributed Computing Systems*, page 550, Washington, DC, USA, 2003. IEEE Computer Society.

[49] A. Fox and D. Patterson. When does fast recovery trump high reliability? In *in Proc. 2nd Workshop on Evaluating and Architecting System Dependability*, 2002.

[50] S. Fu. Failure-aware construction and reconfiguration of distributed virtual machines for high availability computing. In *CCGRID '09: Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 372–379, Washington, DC, USA, 2009. IEEE Computer Society.

[51] Ganglia Monitoring System. *Ganglia*
http://ganglia.sourceforge.net.

[52] S. Garg, A. Van Moorsel, K. Vaidyanathan, and K. S. Trivedi. A methodology for detection and estimation of software aging. In *ISSRE '98: Proceedings of the The Ninth International Symposium on Software Reliability Engineering*, page 283, Washington, DC, USA, 1998. IEEE Computer Society.

[53] Glassbox. *Glassbox*
http://www.glassbox.com.

[54] GNU Linear Programming Kit (GLPK). *GLPK*
http://www.gnu.org/software/glpk.

[55] I. Goiri, F. Julia, J. Ejarque, M. d. Palol, R. M. Badia, J. Guitart, and J. Torres. Introducing virtual execution environments for application lifecycle management and sla-driven resource distribution within service providers. In *NCA '09: Proceedings of the 2009 Eighth IEEE International Symposium on Network Computing and Applications*, pages 211–218, Washington, DC, USA, 2009. IEEE Computer Society.

[56] J. Gray. Why do computers stop and what can be done about it? In *SRDS'86: Proceedings of the 5th Symposium on Reliability in Distributed Software and Database Systems*, pages 1–1, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press.

[57] J. Gray. A census of tandem system availability between 1985 and 1990. *Reliability, IEEE Transactions on*, 39(4):409 – 418, 1990.

[58] K. C. Gross, V. Bhardwaj, and R. Bickford. Proactive detection of software aging mechanisms in performance critical computers. In *SEW '02: Proceedings of the 27th Annual NASA Goddard Software Engineering Workshop (SEW-27'02)*, page 17, Washington, DC, USA, 2002. IEEE Computer Society.

[59] M. Grottke, L. Li, K. Vaidyanathan, and K. S. Trivedi. Analysis of software aging in a web server. *IEEE Transactions on Reliability*, 55:411–420, 2006.

[60] M. J. R. Grottke, M. and K. S. Trivedi. The fundamentals of software aging. In *Proc. of the 1st International Workshop on Software Aging and Rejuvenation/19th IEEE International Symposium on Software Reliability Engineering*, 2008.

[61] G. A. Hoffmann, K. S. Trivedi, and M. Malek. A best practice guide to resources forecasting for the apache webserver. *Pacific Rim International Symposium on Dependable Computing, IEEE*, 0:183–193, 2006.

[62] G. J. Holzmann. Conquering complexity. *Computer*, 40(12):111–113, 2007.

[63] K. C. K. N. Huang, Yennun and N. D. Fulton. Software rejuvenation: Analysis, module and applications. In *FTCS '95: Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing*, page 381, Washington, DC, USA, 1995. IEEE Computer Society.

[64] J. Humphreys and V. Turner. On-demand enterprises and utility computing: A current market assessment and outlook. Technical Report Technical Report 31513, IDC, 2004.

[65] IBM ILOG CPLEX Optimizer. *CPLEX http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/*.

[66] B. Jansen, H. V. Ramasamy, M. Schunter, and A. Tanner. Architecting dependable and secure systems using virtualization. pages 124–149, 2008.

[67] JMX: Java Management Extensions. *JMX http://java.sun.com/javase/technologies/core/mntr-mgmt/javamanagement/*.

[68] C. C. Keir, C. Clark, K. Fraser, S. H, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *In Proceedings of the*

2nd ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI, pages 273–286, 2005.

[69] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[70] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag.

[71] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. marc Loingtier, and J. Irwin. Aspect-oriented programming. In *Lecture Notes in Computer Science 1357*, pages 48–3. Springer Verlag, 1998.

[72] H. Kim. Aspectc#: An aosd implementation for c#.

[73] K. Kourai and S. Chiba. A fast rejuvenation technique for server consolidation with virtual machines. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '07, pages 245–255, Washington, DC, USA, 2007. IEEE Computer Society.

[74] D. L. Kreher and D. R. Stinson. Combinatorial algorithms: generation, enumeration, and search. *SIGACT News*, 30(1):33–35, 1999.

[75] Y. Laarouchi, Y. Deswarte, D. Powell, J. Arlat, and E. De Nadai. Enhancing dependability in avionics using virtualization. In *VDTS '09: Proceedings of the 1st EuroSys Workshop on Virtualization Technology for Dependable Systems*, pages 13–17, New York, NY, USA, 2009. ACM.

[76] Ldirectord. *Ldirectord http://www.vergenet.net/linux/ldirectord/*.

[77] L. Li, K. Vaidyanathan, and K. S. Trivedi. An approach for estimation of software aging in a web server. In *ISESE '02: Proceedings of the 2002 International Symposium on Empirical Software Engineering*, page 91, Washington, DC, USA, 2002. IEEE Computer Society.

[78] B. C. Ling, E. Kiciman, and A. Fox. Session state: beyond soft state. In *NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, pages 22–22, Berkeley, CA, USA, 2004. USENIX Association.

[79] LVS - Linux Virtual Server. *LVS*
http://www.linuxvirtualserver.org.

[80] D. Mahrenholz, O. Spinczyk, and W. Schröder-Preikschat. Program instrumentation for debugging and monitoring with aspectc++. In *IN INTERNATIONAL SYMPOSIUM ON OBJECT-ORIENTED REAL-TIME DISTRIBUTED COMPUTING (ISORC*, pages 249–256. IEEE Computer Society, 2002.

[81] E. Marcus and H. Stern. *Blueprints for high availability*. John Wiley & Sons, Inc., New York, NY, USA, 2003.

[82] E. marshall. Fatal error: How patriot overlooked a scud. *Science 13*, 255(5050):1347, 1992.

[83] MemProfiler. *MemProfiler*
http://memprofiler.com.

[84] D. A. Menascé. Qos issues in web services. *IEEE Internet Computing*, 6(6):72–75, 2002.

[85] D. A. Menasce. Virtualization: Concepts, applications, and performance modeling. In *in Proc. Int. CMG Conference*, 2005.

[86] Microsoft IIS. *IIS*
http://www.microsoft.com.

[87] R. Miles. *AspectJ Cookbook*. O'Reilly Media, Inc., 2004.

[88] MySQL Data Base Server. *MySQL*
http://www.mysql.com.

[89] Nagios - The Industry Standard in IT Infrastructure Monitoring. *Nagios*
http://www.nagios.org/.

[90] D. Nau, M. Ghallab, and P. Traverso. *Automated Planning: Theory & Practice*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[91] Netcraft. *Netcraft*
http://news.netcraft.com/archives/2010/01/.

[92] OGSA-DAI. *OGSA-DAI*
http://www.ogsadai.org.uk/.

[93] OGSA-DAI projects. *OGSA-DAI projects http://www.ogsadai.org.uk/about/projects.php*.

[94] D. Oppenheimer, A. Ganapathi, and D. A. Patterson. Why do internet services fail, and what can be done about it? In *USITS'03: Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems*, pages 1–1, Berkeley, CA, USA, 2003. USENIX Association.

[95] P. Padala, X. Zhu, Z. Wang, S. Singhal, K. G. Shin, P. Padala, X. Zhu, Z. Wang, S. Singhal, and K. G. Shin. Performance evaluation of virtualization technologies for server consolidation. Technical Report HPL-2007-59, HP Labs.

[96] ParaSoft Insure++. *ParaSoft http://www.parasoft.com*.

[97] D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft. Recovery oriented computing (roc): Motivation, definition, techniques,. Technical report, Berkeley, CA, USA, 2002.

[98] S. Peret and P. Narasimham. Causes of failure in web applications. Technical Report Technical Report CMU-PDL-05-109, Carnegie Mellon University, 2005.

[99] V. Petrucci, O. Loques, and D. Mossé. Dynamic optimization of power and performance for virtualized server clusters. In *SAC '10: Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 263–264, New York, NY, USA, 2010. ACM.

[100] T. Poggio and S. Smale. The mathematics of learning: Dealing with data. *Notices of the American Mathematical Society*, 50:2003, 2003.

[101] J. R. Quinlan. Learning with continuous classes. pages 343–348. World Scientific, 1992.

[102] J. R. Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[103] ROC Project Website. *ROC Project http://roc.cs.berkeley.edu*.

[104] M. Rosenblum and T. Garfinkel. Virtual machine monitors: Current technology and future trends. *Computer*, 38:39–47, 2005.

[105] R. K. Sahoo, A. J. Oliner, I. Rish, M. Gupta, J. E. Moreira, S. Ma, R. Vilalta, and A. Sivasubramaniam. Critical event prediction for proactive management in large-scale computer clusters. In *KDD '03: Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 426–435, New York, NY, USA, 2003. ACM.

[106] L. Silva, H. Madeira, and J. G. Silva. Software aging and rejuvenation in a soap-based server. In *NCA '06: Proceedings of the Fifth IEEE International Symposium on Network Computing and Applications*, pages 56–65, Washington, DC, USA, 2006. IEEE Computer Society.

[107] L. M. Silva, J. Alonso, P. Silva, J. Torres, and A. Andrzejak. Using virtualization to improve software rejuvenation. In *Sixth IEEE International Symposium on Network Computing and Applications (NCA 2007), 12 - 14 July 2007, Cambridge, MA, USA*, pages 33–44. IEEE Computer Society, 2007.

[108] L. M. Silva, J. Alonso, and J. Torres. Using virtualization to improve software rejuvenation. *IEEE Trans. Comput.*, 58(11):1525–1538, 2009.

[109] B. Speitkamp and M. Bichler. A mathematical programming approach for server consolidation problems in virtualized data centers. *IEEE Transactions on Services Computing*, 99(RapidPosts), 2010.

[110] M. Stonebraker. The design of the postgres storage system. In *VLDB '87: Proceedings of the 13th International Conference on Very Large Data Bases*, pages 289–300, San Francisco, CA, USA, 1987. Morgan Kaufmann Publishers Inc.

[111] A. T. Tai, H. Hecht, S. N. Chau, and L. Alkalaj. On-board preventive maintenance: Analysis of effectiveness and optimal duty period. In *WORDS '97: Proceedings of the 3rd Workshop on Object-Oriented Real-Time Dependable Systems - (WORDS '97)*, page 40, Washington, DC, USA, 1997. IEEE Computer Society.

[112] TPC-W Benchmark Java Version. *TPC-W http://www.ece.wisc.edu/~pharm/.*

[113] K. S. Trivedi, K. Vaidyanathan, and K. Goseva-Popstojanova. Modeling and analysis of software aging and rejuvenation. In *SS '00: Proceedings of the 33rd Annual Simulation Symposium*, page 270, Washington, DC, USA, 2000. IEEE Computer Society.

[114] K. Vaidyanathan and K. S. Trivedi. A measurement-based model for estimation of resource exhaustion in operational software systems. In *ISSRE '99: Proceedings of the 10th International Symposium on Software Reliability Engineering*, page 84, Washington, DC, USA, 1999. IEEE Computer Society.

[115] K. Vaidyanathan and K. S. Trivedi. A comprehensive model for software rejuvenation. *IEEE Trans. Dependable Secur. Comput.*, 2(2):124–137, 2005.

[116] A. Verma, P. Ahuja, and A. Neogi. pmapper: power and migration cost aware application placement in virtualized systems. In *Middleware '08: Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, pages 243–264, New York, NY, USA, 2008. Springer-Verlag New York, Inc.

[117] Virtuoso website. *Virtuoso*
http://www.virtuoso.com.

[118] VMWare website. *VMWare*
http://www.vmware.com.

[119] W. Vogels. Beyond server consolidation. *Queue*, 6(1):20–26, 2008.

[120] Y. Wang and I. H. Witten. Inducing model trees for continuous classes. In *In Proc. of the 9th European Conf. on Machine Learning Poster Papers*, pages 128–137, 1997.

[121] WEKA 3.5.8. *WEKA*
http://www.cs.waikato.ac.nz/ml/weka/.

[122] M. Welsh, D. Culler, and E. Brewer. Seda: an architecture for well-conditioned, scalable internet services. *SIGOPS Oper. Syst. Rev.*, 35(5):230–243, 2001.

[123] T. Wilfredo. Software fault tolerance: A tutorial. Technical report, 2000.

[124] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann, June 2005.

[125] L. A. Wolsey and G. L. Nemhauser. *Integer and Combinatorial Optimization*. Wiley-Interscience, 1 edition, November 1999.

[126] Xen Source website. *Xen*
http://www.xensource.com.

[127] N. Ye. *The Handbook of Data Mining (Human Factors and Ergonomics Series)*. L. Erlbaum Associates Inc., Hillsdale, NJ, USA, 2004.

[128] G. Yoo, J. Park, and E. Lee. Hybrid prediction model for improving reliability in self-healing system. In *SERA '06: Proceedings of the Fourth International Conference on Software Engineering Research, Management and Applications*, pages 108–116, Washington, DC, USA, 2006. IEEE Computer Society.

[129] Q. Zhang, L. Cherkasova, G. Mathews, W. Greene, and E. Smirni. R-capriccio: a capacity planning and anomaly detection tool for enterprise services with live workloads. In *Middleware '07: Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware*, pages 244–265, New York, NY, USA, 2007. Springer-Verlag New York, Inc.

[130] Q. Zhang, L. Cherkasova, N. Mi, and E. Smirni. A regression-based analytic model for capacity planning of multi-tier applications. *Cluster Computing*, 11(3):197–211, 2008.