João Paulo Ferreira de Magalhães

Self-healing Techniques for Web-based Applications

PhD Thesis: Doctoral Program in Information Science and Technology
Thesis advisor: Professor Luís Alexandre Serras Moura e Silva
Department of Informatics Engineering
Faculty of Sciences and Technology
University of Coimbra

September 2013

· U C ·

Universidade de Coimbra

This investigation was partially supported by the *Programa de Formação Avançada de Docentes do Ensino Superior Politécnico* (PROTEC/PFAD) through a scholarship (SFRH/PROTEC/49504/2009), by ESTGF-Porto Polytechnic Institute and by Centre for Informatics and Systems of the University of Coimbra (CISUC).

Acknowledgments

The conclusion of this work is undoubtedly an important milestone in my life. It is the result of a path, marked by many moments and people who over the years have been with me and supported me to go on it.

I would like to start by thanking my advisor, Professor Luís Silva, without whom I would not be concluding this work. He is responsible for the proposed theme and for opening my mind to this very interesting area of research. It is certainly a privilege to work with someone who never sees the work as done and finds always room for improvement.

I also wish to thank Professor Paulo Marques for their support and pragmatism. There was a conversation between me and Professor Paulo Marques that I will never forget. It was this conversation that led me to reflect and define a work strategy that was essential to combine my professional activity with the PhD work.

I am also truly grateful to Carlos Cunha, Carlos Vaz and Hugo Fonseca for their contribution to this dissertation. Carlos Cunha and Carlos Vaz were, during this time, a good traveling companions. Hugo Fonseca has performed some of the preparatory work essential to the implementation of the experimental studies conducted in this work.

I am grateful to the School of Technology and Management of Felgueiras and to the Porto Polytechnic Institute for the conditions that allowed me to focus on research. Likewise, I would like to thank the Engineering Informatics Department, CISUC - Software and Systems Engineering Group and to CIICESI - Center for Research and Innovation in Business Sciences and Information Systems, by the financial and material support granted.

Finally, I want to thank to my lovely wife Maria João, to my lovely daughter Sofia and to my lovely son João Pedro for making me happy. I am also deeply thankful to my parents for their amazing support and care. I also want to thank my friends for the support that each, in his own way, gave me.

Abstract

Web-based applications play an important role for the business of many enterprises. Numbers recently published, in the *Internet Retailer Magazine*, reveal the vibrancy of these businesses: in 2012 the volume of online sales grew about 19% in Latin America reaching \$15.42 billion, grew 16% in US reaching \$225.54 billion, grew 16% in Europe reaching \$302.20 billion and grew 32% in Asia reaching \$256.50 billion. Supporting this growth, there exist applications that are becoming increasingly complex in terms of functionalities and management. Such complexity can lead to scenarios of service unavailability or performance loss, damaging the revenue and the company's reputation. According to Joshua Bixby, the president of Strangeloop Corporation, *"one second of performance loss could cost \$2.5 million in sales per year for a site that typically earns \$100,000 a day"*. Likely, *"eight out of 10 visitors who have an unsatisfactory experience with a website are unlikely to return. Of these, three will go on to tell others about their poor impression of the site"*.

In 2001, Paul Horn from IBM, has referred the systems complexity as the major blocking factor affecting the IT industry sectors. According to him, the complexity of systems tends to be so high that they become difficult to maintain and manage. To tackle this problem he proposed the concept of Autonomic Computing (AC). AC is an holistic vision in which systems have autonomous management capabilities. It defines four self-managing attributes: self-configure, self-heal, self-optimize and self-protect.

In this Thesis we present a self-healing system for Web applications. Its adoption in Web applications, aims the detection and recovery from anomalies that occur at runtime, in order to maximize the system availability and performance. In this context, we propose a framework called *SHõWA*. *SHõWA* is composed by several modules that do the monitoring of the application, analyze the data to detect and pinpoint anomalies and execute the necessary recovery actions. Monitoring is done through a small software agent, implemented by using the Aspect-Oriented Programming (AOP) paradigm. This agent does not require prior knowledge about the application under monitoring or changes to the application source code. To minimize the impact on performance induced by the monitoring process, the agent includes mechanisms that allow it to self-adapt, at runtime, the monitoring frequency and the amount of application calls to intercept. The data analysis is performed by using statistical correlation. It detects changes in the server response time and analyzes if those changes are associated with changes in the workload or are due to a performance anomaly. In presence of performance anomalies, the data analysis pinpoints the causes behind the anomaly. In particular, it observes changes in the system, changes on the application server or changes in the application, and establishes causal relationships between the response time variations and these changes. Upon the detection and anomaly pinpointing, the *SHõWA* framework selects and executes a recovery procedure (e.g., dynamic provisioning of system resources, downgrade the application, restart services or servers).

In the final part of this Thesis, we present a *SHõWA* extension, targeted for predicting in advance the occurrence of performance anomalies. The extension combines machine learning and time series analysis to estimate the parameters up to N epochs ahead and classifies the estimates to assess the system state. The prediction of anomaly aims to recover the system before the users are affected by the anomaly.

The *SHõWA* framework was implemented and tested in this work. We highlight the following results:

- Using a set of frequent anomalies, we did a comparison between different monitoring systems commonly used in Web-based applications. *SHõWA* was able to detect 100% of the anomalies we injected and it was the fastest detection technique for 82% of the injected anomalies;

- Due to the adoption of self-adaptive monitoring, the *SHõWA* framework allows the interception of all the application calls without causing a significant performance overhead. According to the results, the server throughput is affected in less than 1% and the response time penalty is inferior to 2 milliseconds per request;

- The *SHõWA* framework is able to distinguish between workload variations and performance anomalies;

- The framework did not detect any anomalies when no anomalies were injected into the system. It detected all the performance anomalies we injected and, for all of them, there were users being affected by performance issues;

- The performance anomalies were detected and recovered automatically. The recovery process mitigated the impact of the anomalies and without causing any visible error or loss of the work-in-progress;

- Considering a Web-based application hosted in a cloud platform, the *SHõWA* framework has proved to be useful to self-provision the necessary resources (e.g., CPU, number of instances). It has detected workload and resource contention scenarios and has allowed to adjust the computational resources very quickly, reducing the impact of the anomalies on the end-users;

- The monitoring, detection, localization and recovery of anomalies is done automatically, reducing to a minimum the need for human intervention.

These results, argue in favor of the adoption of *SHõWA* for production environments. *SHõWA* contributes to ensure the performance and availability of the service, maximizing the financial returns of the companies that rely on Web-based applications to perform business operations.

Resumo

As aplicações Web desempenham um papel importante para o negócio de muitas empresas. Dados recentemente publicados na *Internet Retailer Magazine*, revelam que em 2012 as vendas *online* cresceram: 19% na América Latina atingindo 15,42 biliões de dólares; 16% nos EUA atingindo 225,54 biliões de dólares; 16% na Europa atingindo 302,20 biliões de dólares; 32% na Asia atingindo 256,50 biliões de dólares. Por detrás deste crescimento, existem aplicações que são cada vez mais complexas e difíceis de manter. Estes aspetos, se não forem devidamente acautelados, podem levar a situações de indisponibilidade ou a anomalias de *performance*, afetando o retorno financeiro e a imagem das empresas. De acordo com Joshua Bixby, presidente da Strangeloop Corporation, *"um atraso de um segundo no tempo de resposta, pode representar uma perda de 2,5 milhões de dólares por ano, considerando um site que fatura 100 mil dólares por dia"*. O próprio afirma que, *"oito em cada dez visitantes que tenham uma experiência negativa não voltam e três vão partilhar a sua experiência com outros"*.

Em 2001, Paul Horn da IBM, referiu-se à complexidade dos sistemas como um dos maiores entraves ao desenvolvimento tecnológico. Segundo o próprio, a complexidade dos sistemas tende a ser tão elevada que estes se tornam difíceis de gerir e manter. Para fazer face a este problema é proposto o conceito de *Autonomic Computing* (AC). Este conceito é uma visão holística, na qual os sistemas dispõem de capacidades de gestão autónoma. Foram identificadas quatro atributos fundamentais, nas quais o AC deve assentar: *self-configure*, *self-heal*, *self-optimize* e *self-protect*.

Nesta Tese apresentamos um sistema de *self-healing* para aplicações Web. O *self-healing* refere-se à capacidade de auto detetar e recuperar de anomalias. A sua adoção em aplicações Web visa a deteção de anomalias que possam ocorrer em tempo de execução e na recuperação automática dessas anomalias. Neste âmbito, propomos um *framework* denominado de *SHõWA*. O *SHõWA* é composto por vários módulos que cooperam entre si, realizando as tarefas de monitorização da aplicação, a análise de dados para a deteção e localização de anomalias e a execução de procedimentos de recuperação. A monitorização é feita através de um agente, implementado segundo o paradigma de Aspect-Oriented Programming (AOP). Este agente não requer conhecimento prévio sobre a aplicação a monitorizar nem alterações no seu código fonte. Ele dispõe de algoritmos que permitem, de forma autónoma, ajustar a frequência de monitorização e o nível de *profiling*, reduzindo assim o impacto de *performance* causado pela própria monitorização. A análise de dados é feita através de correlação estatística, e tem como objetivo detetar variações de *performance*, analisando para o efeito se uma variação é motivada por alterações de *workload* ou devido a algum tipo de anomalia. Perante uma anomalia de *performance*, a análise de dados é feita com o intuito de localizar a anomalia, nomeadamente se na sua origem estão alterações ao nível do sistema, do servidor aplicacional ou da própria aplicação, identificando neste caso, os componentes envolvidos na anomalia de *performance*. Mediante a deteção e localização da anomalia, o *framework* seleciona e executa um plano de recuperação (p.ex.: redimensionar os recursos do sistema, fazer *downgrade* da aplicação, fazer *restart* sos serviços). Na parte final da Tese, apresentamos uma extensão do *framework*, destinado a prever a ocorrência de anomalias de *performance*. Esta extensão faz uso de modelos de análise de séries temporais para estimar valores e predizer, através de algoritmos de classificação (*machine learning*), o estado do sistema em função dessas estimativas.

O *framework SHõWA* foi implementado e testado neste trabalho. Destacam-se os seguintes resultados:

- Através de um conjunto de anomalias frequentemente observadas em aplicações Web, fizemos um estudo comparativo utilizando sistemas de monitorização diferentes. O *framework SHõWA* detetou 100% das anomalias injetadas e em 82% dos casos foi a ferramenta mais rápida a detetar as anomalias;

- Com a adoção de mecanismos de monitorização adaptativa e seletiva, o *framework SHõWA* permite intercetar todos os métodos da aplicação sem afetar severamente a *performance* do sistema. Os resultados revelam que o *throughput* é afetado em menos de 1% e o atraso no tempo de resposta é inferior a 2 milissegundos;

- O *framework SHõWA* é capaz de distinguir entre variações de *workload* e anomalias de *performance*;

- O *framework SHõWA* não detetou anomalias quando estas não foram injetadas. Ele detetou todas as anomalias de *performance* injetadas e todas as vezes que detetou existiam utilizadores afetados por atrasos no tempo de resposta;

- As anomalias detetadas foram recuperadas de forma automática. O processo de recuperação mitigou o impacto das anomalias sem causar erros ou perdas de trabalho em progresso;

- Considerando uma aplicação Web a correr numa infraestrutura em *cloud*, o *framework SHõWA* revelou-se útil para redimensionar a infraestrutura e manter um bom nível de serviço. Ele detetou cenários de contenção no *workload* e de contenção nos recursos do sistema, e permitiu ajustar a infraestutura de forma rápida, reduzindo o impacto das anomalias junto dos utilizadores;

- A monitorização, deteção, localização e recuperação de anomalias é feita automaticamente, reduzindo assim a necessidade de intervenção humana.

Os resultados obtidos favorecem a adoção do *framework* em aplicações Web de produção. O *framework SHõWA* contribui para assegurar a *performance* e disponibilidade das aplicações, maximizando o retorno financeiro para as empresas que utilizam estas aplicações no âmbito do seu negócio.

Chapter 1 Introduction

In this opening chapter, we lay out the motivation for the theme of this Thesis - Self-healing Techniques for Web-based Applications.

The self-healing is one of the four attributes of Autonomic Computing. Autonomic Computing refers to the self-managing characteristics of a system, adapting to unpredictable changes, while hiding, the intrinsic complexity to operators and users. The self-healing attribute aims to cope with the growing complexity by providing systems with mechanisms for automatic detection and recovery of anomalies.

Web applications are widely used and are a visible face of business for many companies. Despite the efforts in the design of the infrastructure and the good programming practices, these applications are still affected by problems that compromise their performance and availability.

Mitigate the occurrence and impact of these problems through the development of self-healing techniques is the basis of this work. In this work we present the implementation of a self-healing autonomic element targeted for the detection, localization and recovery of performance anomalies in Web-based applications. In this chapter we deepen our motivation and we present the list of fundamental decisions that guided our work, as well as the list of contributions obtained. The chapter ends with the Thesis outline and organization.

1.1 Motivation

The turnover conducted through Web applications continues to grow. Numbers referring to 2012, published on the Internet Retailer Magazine, reveal that the turnover from online sales grew: 19% in Latin America reaching \$15.42 billion; 16% in US reaching \$225.54 billion; 16% in Europe reaching \$302.20 billion; 32% in Asia reaching \$256.50 billion. These numbers are representative of how much the businesses rely on Web applications to achieve their business goals. Unfortunately, and despite all the efforts that have been made, these systems remain vulnerable to numerous factors that may affect their availability or performance. These factors may lead to losses on the company's revenue, leaving customers dissatisfied, damaging the company's reputation, causing impact on the company's stock price and on the employee productivity.

Naturally, the impact varies according to the type of business and type of failure observed. According to a study presented in [Patterson 2002b], one hour of downtime represents a loss of income between \$14,000 and \$6,450,000, for companies that sell products online. These numbers ignore the loss due to the wasting of time of employees and the impact on company reputation. More recently Bojan Simic, the president and principal analyst at TRAC Research, presented a study [Simic 2010], which reports that website slowdowns can have twice of the revenue impact on an organization as an outage. According to that study, the average revenue loss for one hour of website downtime is $21,000 while the average revenue loss of website slowdown is estimated in $4,100 per hour. The study refers that website slowdowns may occur 10 times more frequently than outages. Likely, according to a recent report provided by the Aberdeen Group [Aberdeen Group], a delay of just one second in the page load time can represent a loss of $2.5 million in sales per year, for a site that typically earns $100,000 a day.

User dissatisfaction and its impact on the business is something hard to be estimated. Although, a study from PhoCusWright and Akamai [Rheem 2010], refers that 8 out of 10 people will not return to a site after a disappointing experience and of these, 3 will go on to tell others about their experience. Likely, Sean Power in [Power 2010] said that 37% to 49% of users who experience performance issues when completing a transaction will either abandon the site or switch to a competitor. Of these, 77% will share their experiences with others. A study about underperforming Web applications, presented in [Forrester Consulting], reveals that 79% of online shoppers who experience a dissatisfying visit are likely to no longer buy from that site. Also, 46% of dissatisfied online shoppers are more likely to develop a negative perception of the company, and 44% will tell their friends and family about the experience.

When analyzing the cause of failures, is observed that failures are mainly motivated by three type of problems: hardware, software and human-errors. While hardware and network problems have been controlled over the last years the software failures and human errors stills representing a big challenge. A report focusing the cause of failures in Web applications [Pertet 2005], indicates that 40% of the failures are motivated by operator errors, and the other 40%, are due to application failures. It is unclear whether these failures are mainly due to system complexity, inadequate testing or poor understanding of system dependencies. Although, is simply stated that, the management of complex systems has grown too costly and prone to error: administering a large number of system details is too labor-intensive; people under pressure make mistakes; and complexity makes the recovery routines more complex and time consuming.

Aware of these challenges, IBM has launched in 2001 the concept of Autonomic Computing (AC) [Ganek 2003]. The ultimate aim of AC is to automate the management of computing resources to address the increasingly complex and distributed computing environments of today. It defines four important attributes:

- Self-configuration: automatic configuration of components;
- Self-healing: automatic detect, diagnose and repair of anomalies;
- Self-optimization: automatic monitoring and control of resources to ensure the optimal functioning with respect to the defined requirements;

- Self-protection: proactive identification and protection from arbitrary attacks.

With the adoption of self-* attributes, systems will be able to self-managing, adapting automatically to the environment where they run and capable of reacting to scenarios of failure. Likewise, the involvement of the IT staff will be smaller, thus reducing the risk of potential human errors.

In this Thesis we focus the self-healing attribute with the aim of timely detect and recover from anomalies affecting Web-based applications. According to an analysis presented in [Kiciman 2005], 75% of the time to recover from application-level failures is spent just detecting them. The same analysis refers situations where the application-level failures took days until be detected. The fast detection of failures and the detection, localization and recovery of failures, in an autonomic manner, are key aspects to improve the availability and performance of Web-based applications. In this context, the development of self-healing systems is seen as extremely important. These systems should be able to monitor Web-based applications, detect and pinpoint anomalies and execute corrective actions automatically and whenever necessary. Optimize the detection and recovery processes is of utmost importance to reduce the MTTR (mean-time-to-recovery) and so minimize the impact of anomalies.

1.2 Proposal

In this Thesis we present a self-healing framework for Web-based Applications (*SHõWA*). This framework aims to provide Web-based applications with the ability to deal with anomalies that arise at runtime and affect its operation.

In the context of dependable systems, *SHõWA* can be seen as a fault tolerance and reliability system. In the area of fault tolerance there are three fault models: **fail-stop**, **byzantine** and **fail-stutter**. The **fail-stop** failure model [Schneider 1990] is characterized by being a model that upon the occurrence of a failure, all the operations are stopped. The **byzantine** failure model [Lamport 1982] considers that a process affected by a fault can continue its execution producing incorrect results. The **fail-stutter** failure model [Arpaci-Dusseau 2001], takes into account that the components of a system sometimes fail, and that they also sometimes perform erratically (e.g., low performance) and that it is not reflected in the final results. These unexpected behaviors are defined as performance faults.

SHõWA fits the **fail-stutter** failure model. It is targeted for the detection, localization and recovery of performance anomalies. The automatic detection and recovery from anomalies, contributes to the maintenance of the performance of the system, thereby increasing the availability, performance and maintainability of the service.

The *SHõWA* framework is composed by several modules. Each module has a well-defined function, and cooperate with the others to meet the requirements considered essential in an Autonomic Computing self-healing system. The *Sensor* module is responsible for collecting system, application server and application level data from the managed resource. There is a set of requirements that the *Sensor* module must meet:

- It must collect data at runtime to support the detection and pinpointing of anomalies that are affecting the behavior of the application;
- It should not require manual changes in the application source code or knowledge about its implementation details;
- The performance impact induced by the *Sensor* module should be minimal;
- It should be as autonomous as possible and operate across different Web applications and application containers.

As the data is collected by the *Sensor* module, it is sent to the *Data Preparation* module. The *Data Preparation* module is responsible for preparing the data for the analysis process. Data is grouped in time intervals. Per interval is: determined the mix of user-transactions processed; associated the state of resources; established the relationship between the application calls and the user-transactions that gave rise to it.

The data analysis is performed by *Performance Anomaly Analysis*, *Workload Variation Analysis*, *Anomaly Detector* and *Root-cause Failure Analysis* modules. The data analysis aims to detect and pinpoint performance anomalies. More specifically, the data analysis should:

- Distinguish performance anomalies from workload variations;
- Provide timely detection;
- Pinpoint anomalies at different levels: system, application server, application, local or remote;
- Have good detection coverage and accuracy;
- Be easily re-usable across different Web applications.

The ability to detect and localize anomalies is critical to the recovery process. Given a performance anomaly, the *Recovery Planner* module is responsible for selecting a recovery procedure. The recovery procedure corresponds to a set of actions that once performed should restore the system to its normal operation. Its implementation applies either to the recovery of performance anomalies, as well to the system dimensioning, for example, to meet the workload demands, ensuring an appropriate level of performance.

The recovery procedures may be created by means of engineering processes or machine learning methodologies. The engineering process corresponds to the creation of recovery procedures by an human operator. In the learning method, the system should be able to define, autonomously, which recovery actions should be performed to recover from the anomaly. Regardless of the approach, the recovery modules included in the *SHõWA* framework should:

- Cope with different types of anomalies;
- Restore the normal functioning of the system as soon as possible;
- Restore the system without causing service disruptions;
- Be easily adaptable to the execution environment (e.g., traditional cluster environments, cloud-hosted applications).

The recovery procedure is executed by a module called *Executor*. The *Executor* module interacts with the system affected through the *Effector* module.

In addition to the ability to autonomously detect and recover from anomalies in Web-based applications, in this Thesis we also present an extension of the *SHõWA* framework that aims to predict in advance the occurrence of anomalies. The extension includes four new modules, divided into two groups of key features. The modules *Build Dataset* and *Train Classifier* belong to the "classifier" group. The modules *Parameter Estimation* and *Classify* make part of the "prediction" group. The modules that belong to the "classifier" group work in background, and are responsible for preparing a dataset (using the information already processed by *SHõWA*) and train that dataset using different machine learning classification algorithms. The modules in the "prediction" group, work at runtime, and have as functions, estimate, by means of time series analysis, the state of the parameters collected by the *Sensor* module up to N time intervals ahead and classify the estimate using the classification algorithms trained by the *Train Classifier* module. To predict in advance the occurrence of performance anomalies, the *SHõWA* framework extension that we propose should:

- Model, from the historical data, the behavior of the application with the maximum possible accuracy;
- Estimate the future state of the system, taking into account its current status;
- Provide accurate information on the occurrence of a performance anomaly, given the parameter estimates incurred;
- Anticipate the occurrence of anomalies in useful time.

Each of the major topics included in this work - monitoring, anomaly detection and localization, root-cause analysis, recovery, system dimensioning and prediction of anomalies - is an open area of research. Throughout this Thesis, we detail each of these areas by presenting the state of the art, describing the implementation of the *SHõWA* framework and the approaches we adopted, formulating several research questions and conducting experimental analysis to find the answers.

1.2.1 Contributions

The development of a self-healing system, as *SHõWA*, encompasses several areas. Throughout this Thesis are carried out studies and proposed solutions, aimed at contributing to the state of the art in the different areas. More objectively, this work presents contributions about:

- Monitoring of Web-based applications: In this work we compare different monitoring systems commonly adopted in Web-based applications: system-level monitoring, end-to-end monitoring, log-analyzer and application level monitoring. The comparison is made on the detection coverage, the time-to-detect and the on the effectiveness. For this comparison, we injected a set of common problems in Web applications. Considering the anomalies injected, the results indicate which monitoring system detects more anomalies, which detects the anomalies more quickly and which has more potential to mitigate the effect of anomalies from the end-users perspective.

- Development of application-level monitoring: Based-on the Aspect-Oriented Programming (AOP) principles, *SHõWA* allows to monitor the application, without previous knowledge or manual source code changes to the application. The monitoring tool provides two operational modes: *user-transaction* mode and *profiling* mode. In the *user-transaction* mode it only intercepts and measures the user-transactions duration and collects different system and container parameter. In the *profiling* mode, it intercepts and records the execution time of the calls belonging to the user-transactions call-path. The current tool is implemented in a small file. This file may be placed under one of the directories belonging to the classpath of a J2EE application server and allows monitoring any Web application running on that server.

- Performance impact induced with application-level monitoring: An important aspect of a monitoring system is the performance overhead it introduces in the system. *SHõWA* profiles the application at runtime. Profiling the applications at runtime may impose a significant performance degradation. In this work we propose and evaluate three adaptive and selective algorithms. These algorithms self-adapt the monitoring frequency on-the-fly and select, dynamically, which parts of the application should be intercepted. We use these algorithms to reduce the impact caused by the *SHõWA* monitoring, but they can be applied to self-adapt the behavior of other systems.

- Detection of performance anomalies in Web applications: *SHõWA* make use of the Spearman's rank correlation coefficient to keep track of the association between the user-transactions server response time and the number of concurrent user-transactions in a given interval. A misalignment between the accumulated response time and the number of concurrent user-requests processed is denoted, by the correlation analysis, as a loss of correlation and interpreted as a symptom of performance anomaly. *SHõWA* distinguishes workload variations from response time variations. It detects performance anomalies under different workload conditions. The Spearman's rank correlation coefficient does not depend on the value of variables, which allows the adoption of *SHõWA* to detect performance anomalies in different Web applications and user-transactions.

- Pinpointing of performance anomalies: The ability to timely detect and pinpoint the anomalies is important to determine the cause of the anomaly and to execute the appropriate recovery actions. The analysis mechanism of the *SHõWA* framework includes the pinpointing of anomalies. The mechanism measures the correlation between the parameters of the system, the application server parameters, the response time of the application calls and the number of user-transactions processed. By the correlation degree variations it becomes possible to identify if the performance anomaly is due to a resource exhaustion (e.g. CPU load, low memory), due to a recent upgrade in the application or due to changes in the remote services (e.g., database, Web Service).

- Timely recovery from anomalies in Web-based applications: When a performance anomaly is detected a recovery plan should be executed as quickly as possible, to mitigate the impact of anomalies. In this Thesis we present, how with some simple recovery procedures, the *SHõWA* framework allows to recover, automatically, from different anomaly scenarios without generating errors or loss of the work-in-progress.

- Dynamic provisioning of computing resources: Considering the current trend of deliver applications from cloud-based physical infrastructure, we propose a feature that allows to self-adapt the cloud infrastructure from the cloud consumer perspective. For this purpose, *SHõWA* exchanges details about the anomalies that it detects with the cloud service provider, and decides, autonomously, which provisioning strategy should be executed to adapt the infrastructure, in order to mitigate the anomalies. The monitoring and detection of anomalies in cloud-hosted Web-based applications, from the consumer-perspective, allows to accelerate the processes of resources provisioning, contributing to the satisfaction of SLAs and consequently to the financial revenue of the parties involved (cloud consumer and cloud provider).

- Prediction of performance anomalies: In the final part of this Thesis, we present an extension of the *SHõWA* framework that aims to predict in advance the occurrence of performance anomalies. This extension combines time series models with machine learning classification algorithms. At runtime, as the parameters are collected by the *Sensor* module, the framework estimates its values up to N periods ahead. The estimations are then classified using the machine learning classification algorithms previously trained. The results are intended to predict how the incoming user-transitions and parameters state will affect the performance of the system, and anticipate some necessary recovery actions. Predict anomalies in advance, to avoid the impacts of failures is definitely an important contribution.

1.3 Structure of the Thesis

As schematized in Figure 1, this Thesis can be divided in six main parts:

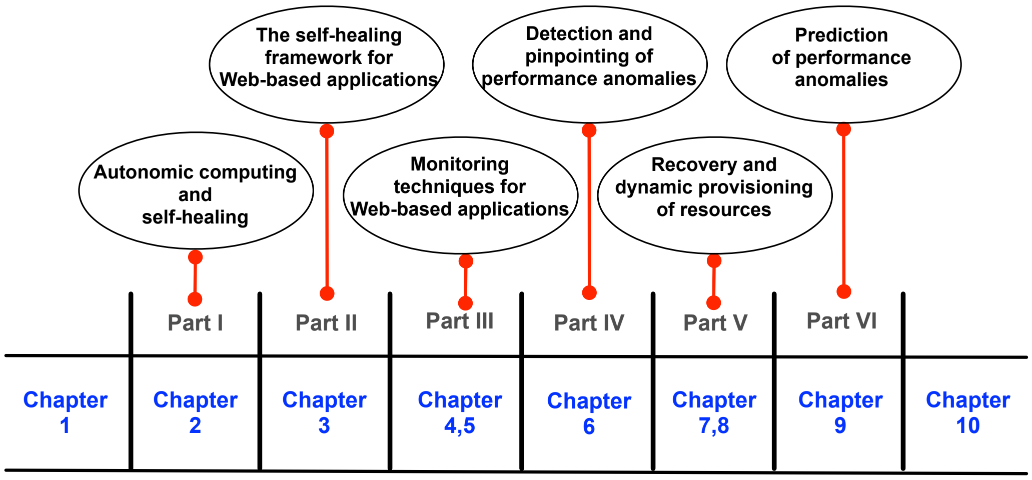

The self-healing framework for Web-based applications
Detection and pinpointing of performance anomalies
Prediction of performance anomalies
Autonomic computing and self-healing
Monitoring techniques for Web-based applications
Recovery and dynamic provisioning of resources
Part I
Part II
Part III
Part IV
Part V
Part VI
Chapter 1
Chapter 2
Chapter 3
Chapter 4,5
Chapter 6
Chapter 7,8
Chapter 9
Chapter 10

Figure 1: Division of chapters of the Thesis in six parts

In this first chapter we make the introduction of our work and in Chapter 10 we present the conclusions.

Part I corresponds to the Chapter 2. In this part, we present the Autonomic Computing paradigm and what is the motivation behind the development of a self-healing systems. Chapter 2 starts by describing the motivation behind the development of systems able to configure, heal, optimize and protect by themselves. We present a relationship between the different attributes of self-managing systems to attain the Autonomic Computing principles. We also present the most relevant projects in the field. The second part of Chapter 2 centers its attention on self-healing.

The self-healing framework we propose is presented in Chapter 3 (part II). This chapter addresses the motivation behind the development of a self-healing system for Web-based applications and presents a self-healing framework targeted for the detection, localization and recovery of performance anomalies in this type of applications - the *SHõWA* framework. In this part we present: the adaptive and selective algorithms that aim to reduce the performance penalty induced by application-level monitoring; the rationale of the data analysis included in the *SHõWA* framework, and that is used to detect and pinpoint performance anomalies; the recovery approach used to recover from the anomalies. This part plays a central role for the evaluation of the *SHõWA* framework presented in the next chapters.

In the third part we describe the monitoring systems that are commonly adopted for Web-based applications and the performance impact induced by these systems. This part includes chapters 4 and 5. In Chapter 4 we present a characterization about the monitoring systems commonly adopted for Web-based applications. In addition to describing the various types of monitoring, we also present a study on the detection latency, the number of end-users affected, the coverage and the performance impact induced by each monitoring system. In the study we consider a set of anomalies, that are common in Web applications. In Chapter 5 we describe the implementation of the *SHõWA Sensor* module, highlighting the role of the selective and adaptive algorithms included in it. In this chapter, we also present an experimental study about the performance impact induced with application-level monitoring and the importance of using selective and adaptive algorithms to reduce the performance impact induced by this type of monitoring.

In Chapter 6 (part IV) we present the data analysis mechanism included in the *SHõWA* framework. It includes an experimental evaluation that shows the ability of the *SHõWA* framework to distinguish workload variations from performance anomalies, pinpoint performance anomalies and detect if a performance anomaly is motivated by a change in the system, an application server change or an application or remote service change.

Part V covers chapters 7 and 8. These chapters focus on the recovery of performance anomalies and on the dynamic dimensioning of the system. In Chapter 7 we consider an execution environment equal to that found in production environments: a Web-based application running in a high-availability cluster with load balancing services. We inject different anomalies and verify how *SHõWA* is able to recover from them. The results presented, in this chapter refer to the advantages of using *SHõWA* to: detect, pinpoint and recover from different types of anomalies; the time-to-repair achieved by the *SHõWA* framework; the usefulness of doing a controlled recovery even knowing that there are high available mechanisms implemented at the infrastructure level. In Chapter 8 we present an experimental study on the dynamic adjustment of the infrastructure. In this chapter we present a system that allows *SHõWA* to interact, autonomously, with the cloud service provider. This interaction allows to report anomalies, from the cloud consumer point of view, and to trigger the provisioning of resources. We also present an experimental study that reveals the role of *SHõWA* to provision resources in the cloud, in order to mitigate resource and workload contention scenarios.

In Chapter 9 (part VI), we present an extension of the *SHõWA* framework that aims to predict the occurrence of performance anomalies. The extension relies on machine learning algorithms and historical knowledge to train classification algorithms. At runtime, the extension estimates the parameters status by means of time series analysis and classifies, using the machine learning algorithms previously trained, the expectable impact that such estimations may have on the application performance. In this chapter, in addition to describing the approach, we present an experimental study on the ability predicting in advance the occurrence of performance anomalies.

1.3.1 Inter-chapter dependencies

The reading of chapters 1, 2 and 10 does not involve consulting other chapters. In Chapter 3 we present the *SHõWA* framework, including the details of its operation. In chapters 4, 5, 6, 7, 8 and 9, we evaluate the different aspects included in the *SHõWA* framework. In an introductory section of these chapters we make a description of the experimental study performed and contextualize it with the *SHõWA* framework. Nevertheless, for a better understanding of the techniques presented in these chapters we recommend the prior reading of Chapter 3.

The cases of greater interdependence between chapters are identified in the diagram, illustrated in Figure 2. The capital letter is used to facilitate the presentation of inter-dependencies.

Inter-chapter dependency A: Part III, is focused on the monitoring of Web applications and the adoption of application-level monitoring (Chapter 4 and 5). In that chapter are evaluated the adaptive and selective monitoring algorithms presented in Chapter 3. Since the adaptive and selective algorithms depend on the analysis performed by the *SHõWA* framework, it is also important to realize how the anomalies are detected. In this context it is important to read the sections related with the *SHõWA* data analysis presented in Chapter 3;

Inter-chapter dependency B: In Part IV we evaluate the ability of the *SHõWA* framework to detect and pinpoint different types of anomaly. The anomaly detection and pinpointing depends on the monitoring and data analysis described

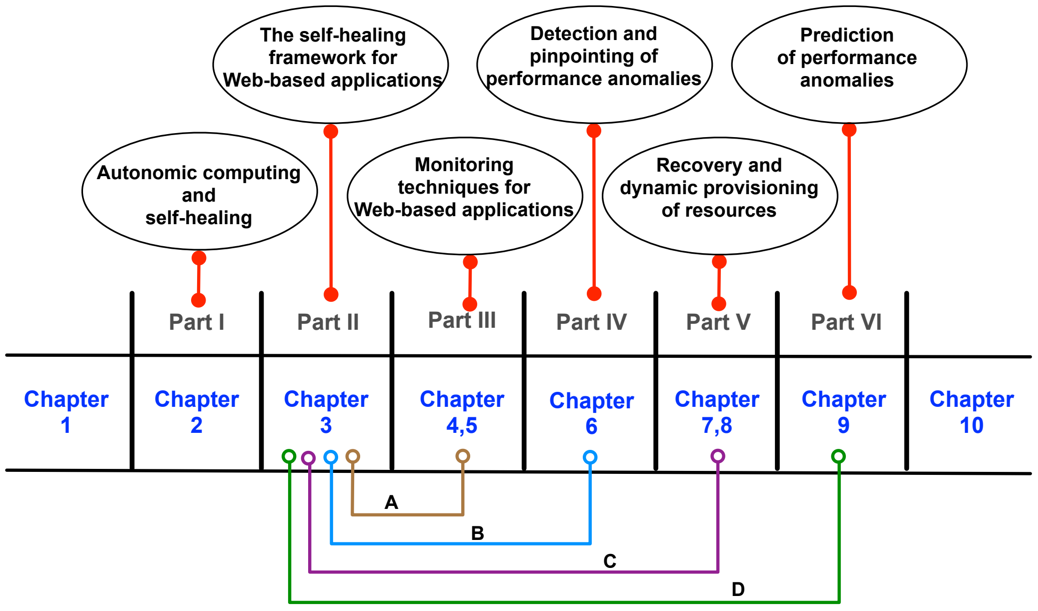

The self-healing framework for Web-based applications
Detection and pinpointing of performance anomalies
Prediction of performance anomalies
Autonomic computing and self-healing
Monitoring techniques for Web-based applications
Recovery and dynamic provisioning of resources
Part I
Part II
Part III
Part IV
Part V
Part VI
Chapter 1
Chapter 2
Chapter 3
Chapter 4,5
Chapter 6
Chapter 7,8
Chapter 9
Chapter 10
A
B
C
D

Figure 2: Inter-chapter dependencies

in Chapter 3. Thus it is suggested the reading of the sections related with the monitoring and data analysis modules;

Inter-chapter dependency C: In chapters 7 and 8 (part V), we present the recovery of anomalies either in legacy and cloud-hosted environments. The recovery of the anomalies depends on the monitoring and detection and localization of anomalies. Given this dependency, we recommend the reader to read Chapter 3 before reading these chapters.

Inter-chapter dependency D: The prediction of performance anomalies is presented in Chapter 9 (Part VI). The prediction of performance anomalies takes advantage of the data collected by the *Sensor* module and the data analysis, presented in Part II. Reading Chapter 3 is extremely important to understand the framework extension we propose in this chapter.

1.4 Publication record

The following publications are the result of the investigation undertaken during this Thesis.

Publications on conference proceedings

- Magalhães, J. P. and Silva, L. M., "A Framework for Self-healing and Self-adaptation of Cloud-hosted Web-based Applications", in Proc. of the 5th IEEE International Conference on Cloud Computing Technology and Science, Bristol, UK, December 2013 (in press) (*17.8% acceptance rate*)

- Magalhães, J. P. and Silva, L. M., "Self-healing Performance Anomalies in Web-based Applications", in Proc. of the 12th IEEE International Symposium on Network Computing and Applications, pp. 81-88, Cambridge, Massachusetts, USA, August 2013 (*27% acceptance rate, ERA 2010 ranking A*)

- Magalhães, J. P. and Silva, L. M., "Adaptive Monitoring of Web-based Applications: A Performance Study", in Proc. of the 28th Symposium On Applied Computing (DADS), pp. 471-478, Coimbra, Portugal, March 2013 (*24% acceptance rate, ERA 2010 ranking B*)

- Magalhães, J. P. and Silva, L. M., "Anomaly Detection Techniques for Web-Based Applications: An Experimental Study", in Proc. of the 11th IEEE International Symposium on Network Computing and Applications, pp. 181-190, Cambridge, Massachusetts, USA, August 2012 (*29% acceptance rate, ERA 2010 ranking A*)

- Magalhães, J. P. and Silva, L. M., "Root-cause Analysis of Performance Anomalies in Web-based Applications", 26th Symposium On Applied Computing (DADS), pp. 209-216, Taichung, Taiwan, March 2011 (*30% acceptance rate, ERA 2010 ranking B*)

- Magalhães, J. P. and Silva, L. M., "Adaptive Profiling for Root-Cause Analysis of Performance Anomalies in Web-Based Applications", in Proc. of the 10th IEEE International Symposium on Network Computing and Applications, pp. 171-178, Cambridge, Massachusetts, USA, August 2011 (*26% acceptance rate, ERA 2010 ranking A*)

- Magalhães, J. P. and Silva, L. M., "Prediction of Performance Anomalies in Web-Applications Based-on Software Aging Scenarios", Second International Workshop on Software Aging and Rejuvenation (WoSAR) at the 21st IEEE International Symposium on Software Reliability Engineering (ISSRE 2010), pp. 1-7, San Jose, California, USA, November 2010

- Magalhães, J. P. and Silva, L. M., "Detection of Performance Anomalies is Web-based Applications", in Proc. of the Ninth IEEE International Symposium on Network Computing and Applications, pp. 60-67, Cambridge, Massachusetts, USA, July 2010 (*ERA 2010 ranking A*)

Publications on journals

- Magalhães, J. P. and Silva, L. M., "*SHõWA*: A Self-healing Framework for Web-based Applications" (currently submitted to ACM Transactions on Autonomous and Adaptive Systems) (*Impact Factor 1.29*)

1.5 Citations known in the literature

A total of 12 citations referring publications authored by João Paulo Magalhães are known currently. These citations do not include the citations from authors that are directly or indirectly connect to the author and his research group.

- *Magalhães, J. P. and Silva, L. M., "Root-cause Analysis of Performance Anomalies in Web-based Applications", 26th Symposium On Applied Computing (DADS), pp. 209-216, Taichung, Taiwan, March 2011* (*30% acceptance rate, ERA 2010 ranking B*) - **Cited by (4):**

  1. Wang, T., Wei, J., Qin, F., Zhang, W., Zhong, H., and Huang, T. (2013). Detecting performance anomaly with correlation analysis for Internetware. Science China Information Sciences, 56(8), 1-15.
  2. Huang, F., Zhang, S., Yuan, C., and Zhong, Z. (2012). Memory performance prediction of web server applications based on grey system theory. In Web Technologies and Applications (pp. 660-668). Springer Berlin Heidelberg.
  3. Wang, T., Wei, J., Zhang, W., Zhong, H., and Huang, T. (2013). Workload-Aware Anomaly Detection For Web applications. Journal of Systems and Software.
  4. Tang Fei, Tang Haina, and Qiahan Hezier (2013). A Network Performance Assessment Method Based on Passive Measurement. Journal of Computer and Digital Engineering.

- *Magalhães, J. P. and Silva, L. M., "Prediction of Performance Anomalies in Web-Applications Based-on Software Aging Scenarios", Second International Workshop on Software Aging and Rejuvenation (WoSAR) at the 21st IEEE International Symposium on Software Reliability Engineering (ISSRE 2010), pp. 1-7, San Jose, California, USA, November 2010* **Cited by (3):**

  1. Cotroneo, D., Natella, R., Pietrantuono, R., and Russo, S. (2011, November). Software aging and rejuvenation: Where we are and where we are going. In Software Aging and Rejuvenation (WoSAR), 2011 IEEE Third International Workshop on (pp. 1-6). IEEE.

2. Cotroneo, D., Natella, R., and Pietrantuono, R. A Survey of Software Aging and Rejuvenation Studies.
3. Yang, H., Yi, S., Liang, Y., Fu, J., and Tan, C. (2012). Dendritic cell algorithm for web server aging detection. In Proceedings of the International Conference on Automatic Control and Artificial Intelligence (ACAI 2012), (pp. 760-763). IET.

- *Magalhães, J. P. and Silva, L. M., "Adaptive Profiling for Root-Cause Analysis of Performance Anomalies in Web-Based Applications", in Proc. of the 10th IEEE International Symposium on Network Computing and Applications, pp. 171-178, Cambridge, Massachusetts, USA, August 2011* (*26% acceptance rate, ERA 2010 ranking A*) **Cited by (3):**

  1. Sosnowski, J., Gawkowski, P., and Cabaj, K. (2013). Exploring the Space of System Monitoring. In Intelligent Tools for Building a Scientific Information Platform (pp. 501-517). Springer Berlin Heidelberg.
  2. Kubacki, M., and Sosnowski, J. Enhanced Instrumentation of System Monitoring. Information Systems in Management XVI, 29.
  3. Kubacki, M., Sosnowski, J., and Kubacki, M. (2013). Creating a knowledge database on system dependability and resilience. Control and Cybernetics, 42(1).

- *Magalhães, J. P. and Silva, L. M., "Detection of Performance Anomalies is Web-based Applications", in Proc. of the Ninth IEEE International Symposium on Network Computing and Applications, pp. 60-67, Cambridge, Massachusetts, USA, July 2010* (*ERA 2010 ranking A*) **Cited by (2):**

  1. Wang, T., Wei, J., Qin, F., Zhang, W., Zhong, H., and Huang, T. (2013). Detecting performance anomaly with correlation analysis for Internetware. Science China Information Sciences, 56(8), 1-15.
  2. Wang, T., Wei, J., Zhang, W., Zhong, H., and Huang, T. (2013). Workload-Aware Anomaly Detection For Web applications. Journal of Systems and Software.

Chapter 2

Autonomic Computing Self-healing Systems

KEY POINTS

◇ The exponential grow in the number and variety of systems, the growth and value of database systems, the interconnectivity between distributed and heterogenous systems and the workload and environment variability is making the systems managing task too complex, costly and error prone.

◇ Autonomic Computing is proposed as an holistic approach to cope with the complexity of integrating, managing and operating computing systems. It is inspired in the autonomous nervous system and aims the development of systems provided with self-configure, self-healing, self-optimize and self-protection attributes.

◇ Different projects are already using these properties to improve the systems reliability, efficiency, usability, functionality, portability and maintainability. Throughout this chapter we present the concepts and projects related with Autonomic Computing, giving emphasis to the self-healing attribute.

In this chapter we present the Autonomic Computing paradigm and what is the motivation behind the development of systems that should be able to configure, heal, optimize and protect in a more automatic manner. We also present the relationship between the pieces of self-managing systems to attain the Autonomic Computing principles and some of the most relevant projects in the field.

The second part of this chapter centers its attention on self-healing. Self-healing is one of the attributes of Autonomic Computing and it aims to devise systems able to heal themselves. Considering the common causes of failures and the increasingly complexity to manage and timely detect failures, the development of self-healing systems is assumed very important to mitigate the occurrence of failures. Some of the most relevant self-healing projects are presented in this chapter.

2.1 The Autonomic Computing paradigm

In 2001, Paul Horn from IBM introduced the concept of Autonomic Computing. During his presentation, [Ganek 2003], he referred the systems complexity as the major blocking factor affecting the IT industry. This complexity is justified by the exponential grow in the number and variety of systems, the growth and value of database systems, the interconnectivity between distributed and heterogenous systems and the workload and environment variability. Complexity is making the management of systems too complex, costly and prone to error:

- The administration tasks are too labor-intensive and require highly skilled IT professionals;
- The testing and tuning is complex;
- People under pressure make more mistakes and require more time to understand and make decisions.

In this context, Autonomic Computing emerges as an holistic approach to cope with the complexity of integrating, managing and operating computing systems. It is inspired in "sensing" and "effecting" actions taken by the Autonomic Nervous System [Sterritt 2005b]. Like the Autonomic Nervous System, a computer system should be able to regulate itself in accordance with high-level guidance from humans. Its application to computational context aims the design and development of systems able to adapt by themselves in order to meet requirements like performance, fault tolerance, reliability and security. To achieve this, an autonomic system should:

- Include automated mechanisms to collect and aggregate data in support of decisions by IT staff;
- Provide advice to humans suggesting possible lines of action;
- Be consciousness about its environment and the expected versus current state;
- Be capable to select and take regulate actions automatically.

These self-managing features are considered mandatory to handle the increasing system complexity, the IT management costs and to improve the systems dependability.

Figure 3 show the different levels at which a system can be found in terms of Autonomic Computing operations [Ganek 2003]. As we move through the levels, we go from a system that requires performing tasks manually to a system that is completely autonomous. As systems become more autonomous, the IT management becomes more aligned with the business needs.

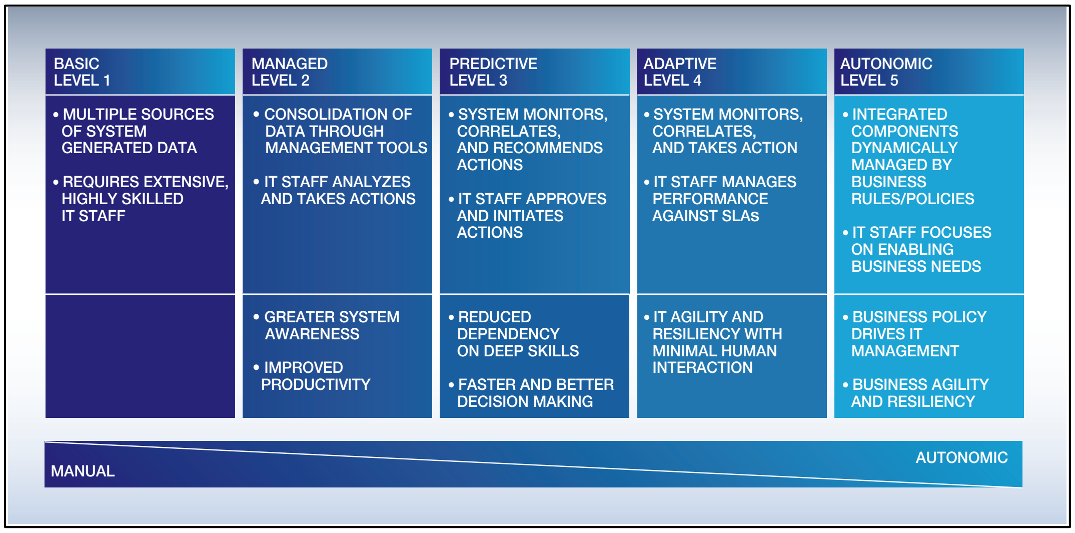

BASIC LEVEL 1
• MULTIPLE SOURCES OF SYSTEM GENERATED DATA
• REQUIRES EXTENSIVE, HIGHLY SKILLED IT STAFF
MANAGED LEVEL 2
• CONSOLIDATION OF DATA THROUGH MANAGEMENT TOOLS
• IT STAFF ANALYZES AND TAKES ACTIONS
• GREATER SYSTEM AWARENESS
• IMPROVED PRODUCTIVITY
PREDICTIVE LEVEL 3
• SYSTEM MONITORS, CORRELATES, AND RECOMMENDS ACTIONS
• IT STAFF APPROVES AND INITIATES ACTIONS
• REDUCED DEPENDENCY ON DEEP SKILLS
• FASTER AND BETTER DECISION MAKING
ADAPTIVE LEVEL 4
• SYSTEM MONITORS, CORRELATES, AND TAKES ACTION
• IT STAFF MANAGES PERFORMANCE AGAINST SLAs
• IT AGILITY AND RESILIENCY WITH MINIMAL HUMAN INTERACTION
AUTONOMIC LEVEL 5
• INTEGRATED COMPONENTS DYNAMICALLY MANAGED BY BUSINESS RULES/POLICIES
• IT STAFF FOCUSES ON ENABLING BUSINESS NEEDS
• BUSINESS POLICY DRIVES IT MANAGEMENT
• BUSINESS AGILITY AND RESILIENCY
MANUAL
AUTONOMIC

Figure 3: Evolving of Autonomic Computing operations

As shown in Figure 3, in the basic level we are mainly focused in the IT part. The aim is to know how much time is required to finish major tasks or fix major problems as they occur. As we advance in the direction of the managed level, we get focus on the availability of the managed resources or the time necessary to repair the system. In the predictive level we are able to project the future needs or potential problems and make recommendations to improve the systems dependability. The adaptive level complements the predictive level. It enables automatic provision of resources based-on business policies, business priorities and service-level agreements. Finally, at the autonomic level, the business and systems are seen as one. At this level there are IT tools that understand the business metrics and contain advanced modeling techniques to know how to contribute to the e-business performance and how to quickly adapt to achieve the business goals.

2.1.1 Key elements of Autonomic Computing

An Autonomic Computing system can be expressed in terms of properties that a system should have to provide autonomic behaviors.

In Figure 4 we present a tree, adopted from [Sterritt 2005a], that shows how the Autonomic Computing properties are organized. The vision of Autonomic Computing corresponds to the objective of creating systems capable of self-managing key functions, thereby mitigating the complexity required to manage the system. It includes four objectives (self-configuring, self-healing, self-optimizing and self-protecting) and four attributes (self-aware, environment aware, self-monitoring and self-adjusting).

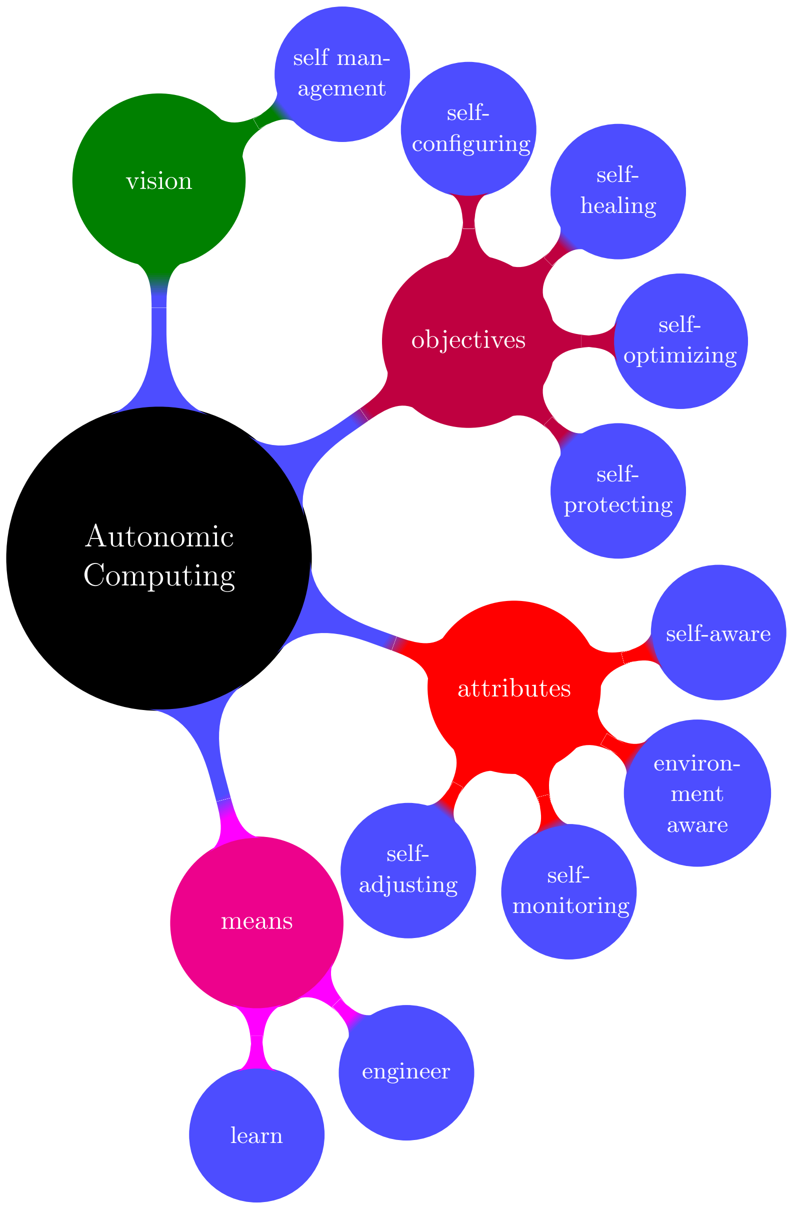


Figure 4: Autonomic Computing properties tree

The four objectives refer to the broad system requirements, i.e., to *what* an autonomous system should provide. These include:

- **Self-configuring**: A self-configuring system should be able to adapt itself when an environment change occurs. For example, add or remove servers from the IT infrastructure or adjust the configuration parameters on-the-fly, are some of the self-configuration actions that can be performed to meet the system/business needs.

- **Self-healing**: Systems should be able to discover, diagnose and react to anomalies. For a system to be self-healing it must detect and isolate the failed component, taking it offline, fixing or isolating the failed component, and reintroducing the fixed or a replacement component into service. All of this must occur without any apparent service disruption. The main objective is to maximize the availability, survivability, maintainability and reliability of the system, mitigating the impact of failures.

- **Self-optimizing**: Monitor and tune resources automatically is an important characteristic to maximize the resource utilization and meet end-users needs. Aspects like dynamic workload management and dynamic server clustering should be adopted to automatically redistribute the workload to systems that have the necessary resources to meet the workload requirements. Similarly, continuous system and application tuning must be done to enable efficient operations even in unpredictable environments.

- **Self-protecting**: Systems should be able to anticipate, detect, identify and protect themselves from attacks like unauthorized accesses or intrusion attempts. The systems should manage the user accesses to all computing resources, report and provide backups and recovery capabilities to deal with scenarios of attack, removing the burden of these tasks from system administrators.

The four attributes identify the basic implementation mechanisms for a self-managing system. These attributes are:

- **Self-aware**: An Autonomic Computing system needs to know itself. Since a system can exist at many levels, an autonomic system need to have detailed knowledge of its components, be aware of its current status, know its ultimate capacity, and know all the connections to other systems in order to govern itself;

- **Environment aware**: An Autonomic Computing system must know its environment and the context surrounding its activity, and act accordingly. It should find and generate rules for how best to interact with the environment;

- **Self-monitoring**: An Autonomic Computing system must be able to know its own state and behavior towards the system;

- **Self-adjusting**: An Autonomic Computing system must be able to autonomously adapt to the heterogeneous environment in accordance with its own monitoring.

According to Roy Sterritt, in [Sterritt 2002], an autonomic system can be implemented through engineering processes, or through processes of self-learning. The engineer processes corresponds to the explicit implementation of the functions that make system operations more autonomous. The learning processes relies on artificial intelligence, evolutionary computation and adaptive learning, which through machine learning processes and algorithms, provide autonomous functions to the system.

2.1.2 Autonomic Computing architecture

An autonomic system comprises one or more autonomic elements. An autonomic element, as illustrated in Figure 5, is composed by two modules. A functional module - Managed Element - that performs the required services and functionality. A management module - Autonomic Manager - that monitors the state and context of the element, analyzes its current requirements and provides the adaptation plans to satisfy the defined requirements (e.g.: performance, fault tolerance, security).

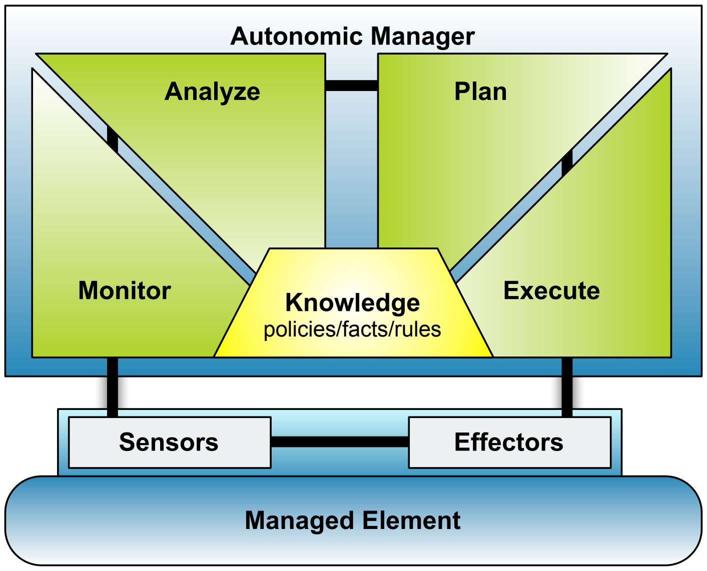


Figure 5: Autonomic element: The control loop in action

A managed resource could be a server, a storage unit, a database system, an application server, an application or other entity. The interaction between the managed element and the autonomic manager is taken through *sensor* and *effector* operations. The *sensor* provide mechanisms to collect information about the state, and state transition, of an element. The *effectors* are mechanisms that have the ability to change the state of a managed element.

The autonomic manager can perform various self-management tasks, so they em-body different intelligent control loops. These control loops iterate over four distinct areas of functionality. These iterations are know as the MAPE (Monitor, Analyze, Plan and Execute) loop. The Monitor and Analyze modules process the information collected by the sensors. The Plan and Execute modules decide on the necessary self-management behavior that will be executed through the effectors.

The MAPE components make use of a knowledge repository that defines policies (goals or objectives) to govern the behavior of intelligent control loops. These control loops can be used for example to: restart a component when a performance anomaly is detected; adjust the resources allocation to cope with workload changes; add rules to block intrusion attempts.

2.1.3 Autonomic Computing landscape

According to [Horn 2001], the implementation of Autonomic Computing constitutes a grand challenge. Part of the challenge lies in creating the open standards and new technologies to allow the systems to interact effectively and with a minimal dependence on IT support.

During the last decade, there has been a number of academia and industry efforts addressing Autonomic Computing concepts. As examples of industry-oriented projects, we have: IBM StorageTank [Menon 2003], IBM Océano [Appleby 2001], IBM Smart DB [Lohman 2002], Microsoft AutoAdmin [Chaudhuri 1998] and Sun N1 Grid System [N1 Grid Technology].

IBM StorageTank [Menon 2003] is a distributed storage system that includes self-optimization and self-healing characteristics. It allows clients to connect to a storage area network and then access directly to large volumes of data directly from storage devices, using high-speed, low-latency connections. It automatically provisions capacity according to pre-defined policies and has the ability to choose different classes of storage according to the quality of service requirements. Different types of backups/restore and security mechanisms are also used to improve the availability and performance of the system.

The IBM Océano project [Appleby 2001] is a management software for Web-based infrastructures. The management software is able to provide autonomic resource allocation in response to operator commands or workload conditions. It includes self-optimization and self-awareness to dynamically assign resources to accommodate planned and unplanned fluctuation of workloads and reduce the costs of setting up and operate hosting farms. The resources allocation is guided by a policy layer that includes predefined service level agreements.

The IBM Smart DB [Lohman 2002] and the Microsoft AutoAdmin [Chaudhuri 1998] are two examples of projects that incorporate Autonomic Computing mechanisms. They focus on the self-management of database systems by means of self-optimization and self-configuration capabilities. These capabilities are used to tune the configuration parameters dynamically without having to pause or alter the state of running applications and with low database administrators involvement.

The Sun N1 Grid System [N1 Grid Technology] aims to reduce the management complexity in the data center caused by the explosion of new applications and servers. It gives a unified vision of the underlying computing, networking and storage resources. It is helpful to reduce the system configuration and management burden on IT staff, help shrink costs, and increase the agility of enterprises that adopt it.

As examples of research projects, we have: OceanStore [Kubiatowicz 2000], Anthill [Montresor 2001], KX [Kaiser 2002], Autonomia [Dong 2003], JADE [Bouchenak 2011], AutoMate [Parashar 2006], the Recovery Oriented Computing [Patterson 2002] and the Software Aging and Rejuvenation [Trivedi 2000] initiatives.

OceanStore [Kubiatowicz 2000] is a distributed storage system enhanced with several self-management facilities. It includes self-healing, self-optimization, self-configuration and self-protection mechanisms to provide high-reliability, high performance and security on access to the data. These features result from the combination between event monitoring systems that are able to collect and analyze information (e.g, usage patterns, network activity and resources availability) and policy-based caching, routing adaptation, autonomic replication, autonomic repair and dynamic cryptographic techniques used to self-manage the system.

Anthill [Montresor 2001] is a framework to support the development of autonomic systems and applications. It is based-on the ant colony paradigm [Liang 2007] and abstracts programmers of peer-to-peer application from low-level details such as communication, security and scheduling.

The Kinesthetics eXtreme (KX) project [Kaiser 2002] enables autonomic properties in legacy systems. It allows to attach a standard feedback-loop infrastructure to existing distributed systems for the purposes of continuous monitoring and dynamic adaptation of their activities and performance. The running system is instrumented with probes and the gathered data is then collected, collated, filtered and aggregated into system level measurements. The measurements are analyzed and based-on the results are carried out runtime modifications to the system. These management functions are done with little or no human intervention.

Autonomia [Dong 2003] is an autonomic architecture to achieve automated control and management of networked applications and their infrastructure. It includes controls and management policies (e.g. performance, faults, security) useful for the implementation of applications provided with self-configuring and self-healing features. Fault handler mechanisms are used to recover from different types of faults. The migration of application components, changes on the resources allocation and load balancing techniques are some examples of features used in Autonomia to heal and optimize the running application. The AutoMate project [Parashar 2006] aims to provide grid applications with automatic formulation, composition and runtime management mechanisms. Information about the components behavior and resource usage, drive the mechanisms to setup or adapt the execution environment in runtime.

JADE [Bouchenak 2011] is an architecture-based autonomic system for loosely coupled distributed systems. It uses different autonomic managers to observe and monitor the managed system. Based-on the observations, it takes the self-repair, self-protection or self-optimizations steps required to maintain the managed system within the predefined goals.

The Recovery-Oriented Computing (ROC) project [Patterson 2002] is a research project that is investigating novel techniques for building highly-dependable services over the network, including both Internet services and enterprise services. ROC focus on different recovery approaches, all with the aim of reducing the Mean-Time-To-Repair (MTTR), thereby improving the systems availability. Another line of research that has several projects is Software Aging and Rejuvenation (SAR) [Trivedi 2000]. Like ROC, the SAR initiative seeks ways to detect deviations in the normal behavior of systems, and automatically adopt recovery actions to recover from them as soon as possible. Throughout this Thesis we will cite some of most relevant research projects included in the ROC and SAR initiatives.

According to Parashar and Hariri [Parashar 2005], the existing projects can be classified: as systems that incorporate autonomic mechanisms for problem determination, monitoring, analysis and management (IBM StorageTank, IBM Océano, IBM Smart DB, JADE, OceanStore, Microsoft AutoAdmin, ROC and SAR); or systems that investigate models, programming paradigms and development environments to support the development of autonomic systems and applications (Anthill, KX, Autonomia, AutoMate).

2.2 Self-healing

This Thesis focuses self-healing techniques for Web-based applications. In this section we emphasize the motivation for the development of systems able to detect and recover from anomalies by themselves. We also present the failure models, highlighting the most frequent causes of failure and its impact. The most relevant projects that we found in the area of self-healing are also presented in this section.

2.2.1 Motivation for Self-healing

The availability of the systems is a critical factor for business success of companies. According to the study "Data from IT Performance Engineering and Measurement Strategies: Quantifying Performance Loss", published by the Meta Group in October 2000, the impact of downtime can represent a revenue loss between 1 and 2.8 Millions dollars per hour. The impact of downtime is illustrated in Figure 6. It varies by industry sector. Besides the tangible costs is also noteworthy that downtime also harms the brand, the trust, the customers satisfaction and loyalty.

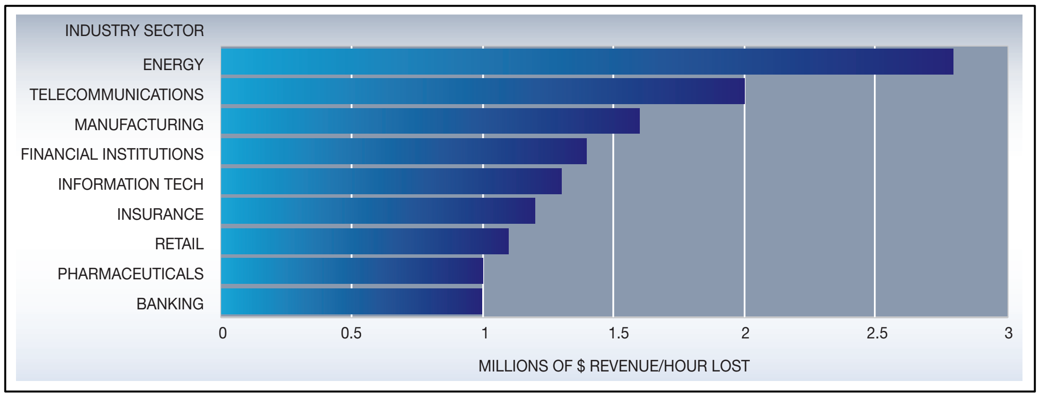


Figure 6: Impact of downtime, in million of dollars per hour, by industry sector

The impact caused by scenarios of unavailability or low performance are well known. These problems have been the subject of extensive research and are wrapped in engineering processes aimed at reducing their risk of occurrence. The scientific and technological advances over the last few years made the hardware and network the infrastructures more fault tolerant. With respect to software and human intervention is no longer possible to say the same. According to Soila Pertet and Priya Narasimhan in [Pertet 2005], 80% of the failures that occur are due to software and human error issues.

There may be several reasons for the occurrence of failures, but the complexity encapsulated by the software and the complexity in managing the systems turns out to be the main reason. The complexity makes the detection and recovery problems a tough and time consuming task. Consequently, the efficiency of many computer environments and departments, and the financial returns and reputation of the companies is severely affected. According to a study provided in [Patterson 2002], one third to one half of the total IT budget is spent preventing or recovering from crashes. According to the same study, the labor costs exceed equipment by a factor of three to eighteen. A report provided in [Kiciman 2005], refers that 75% of the time to recover from application-level failures is spent, by the IT staff, just to detect and localize them.

The major factors contributing to the complexity in problem determination are the various ways that different parts of the system report events, conditions, errors and alerts. Ganek and Corbi in [Ganek 2003], classify the common causes behind the outages according to the data center operations: systems, networks, database, and applications:

- For systems: operational error, user error, third-party software error, internally developed software problem, inadequate change control, lack of automated processes;
- For networks: performance overload, peak load problems, insufficient bandwidth;
- For database: out of disk space, log file full, performance overload;
- For applications: application error, inadequate change control, operational error, non automated application exceptions.

The development of self-healing systems is very important to deal with the diversity of situations that can occur at runtime and affect the availability and performance of the service. A self-healing system is a system that is able to detect and to recover from anomalies. To discover anomalies, it is important to know the expected system behavior and determine if the actual behavior is consistent and expected in relation of the environment. To do so, it must take into account the complex dependencies between the hardware and software failures and the variations motivated by workload changes. It should include tools to collect data from various types of monitors, perform data aggregation and data analysis to determine correlations between events and diagnose faults. All of this should be done autonomously. These steps fits in the MAPE processes [Laster 2007], as follows:

1. Monitoring: the system gathers data from the system to observe for unexpected behaviors;
2. Analyze - error detection and diagnosis: if the diagnosis reports that there is no fault/error in the system then it will loop back to the monitor for more observations. If a fault/error is detected it will be reported to the next stage;
3. Plan - analysis and selection of a repair plan: depending on the type of fault/error, a repair plan is selected to be passed onto the final stage;
4. Execution of repair actions (self-repair): any repair action that is needed its executed at this stage and the cycle begins all over again.

In [Psaier 2011] this closed-loop is reduced from four to three processes, but the essence keeps unchanged. The detecting process filters any suspicious status information received from samples and reports detected degradations to the diagnosis process. The diagnosing process includes root cause analysis and determines an appropriate recovery plan with the help of a policy database. The recovery process applies the planned adaptations to meet the constraints of the system capabilities and to avoid unpredictable side effects.

Regardless of being presented with more or fewer processes, it is well understood the importance of self-healing to improve the systems availability by detecting and recovering from anomalies very quickly and with minimal human intervention. By reducing the need for human intervention, the operational costs related to the management of the system will decrease and the risk of human errors is also reduced.

2.2.2 Failure characteristics

Understanding the failures characteristics is a fundamental aspect to devise dependable computer systems.

According to the dependable system taxonomy, presented by Avizienis et al. in [Avizienis 2004], *"the service delivered by a system is its behavior as it is perceived by its users"*. A service failure is an event that occurs when the delivered service is no longer consistent with the expected behavior.

Service failures may happen when the service fails, either because it does not comply with the functional specification, or because the specification did not adequately describe its function. The deviation from correct service may assume different forms that are called service failure modes and are ranked according to failure severities. The deviation from the correct service is called an error and errors may occur due to internal or external faults. In most cases, a fault first causes an error in the service state of a component that is part of the internal state of the system, and in this case the external state (visible by the system users) is not immediately affected. Stated in another way, a **fault** refers to an incorrect state of hardware and software resulting from a physical defect, design flaw or operator error. An **error** is a manifestation of a fault and a **failure** is the external visible deviation from the specification. A fault need not result in an error, nor an error in a failure.

Computer systems may experience permanent, intermittent, or transient faults. In face of a permanent fault, the system simply stops working until it is repaired. Burn out chips, software bugs and processor failures are some examples of permanent faults. An intermittent fault is characterized by appearing and disappearing at intervals, usually irregular, during the system operation and without interrupt its functioning. Because of their sporadic nature these faults are very difficult to identify and repair. An intermittent fault might occur in a software program. For example, if a variable which is required to be initially zero fails to be initialized then: the program will run normally in circumstances such that memory is almost always clear before it starts; a fault will occur on the rare occasions that the memory where the variable is stored happens to be non-zero beforehand. A transient fault occurs and then disappears without any apparent intervention.

A more specific classification about software faults is presented in [Gray 1986]. The author classifies the software faults as *Bohrbugs* and *Heisenbugs*. *Bohrbugs* are permanent design faults and hence almost deterministic in nature. They can be identified easily during the testing and debugging phase of the software life cycle. The *Heisenbugs* faults, belong to the class of intermittent or transient faults. They are faults whose conditions of activation occur rarely or are not easily reproducible. The author refers that these faults are extremely dependent on the operating environment, and they may not recur if the software is restarted.

The manifestation of an error, i.e. a failure, typically falls within the following classes of faulty behavior: **fail-stop**; **byzantine**; **fail-stutter**. Understand each of these classes is very important to correctly model the failures and to devise fault tolerance mechanisms.

The **fail-stop** failure model is presented in [Schneider 1990]. In this model, in response to a failure, the component changes to a state that allows the other components to detect that a failure has occurred and then it stops, rather than attempt to continue a flawed process. Thus, each component is either working or not, and when a component fails, all other components can immediately be made aware of it. This failure model is considered extremely simple, tractable and pragmatic: the components produce correct output or none; it can always detect that a component has stopped; assumes that there is no collusion across components. Fault tolerance systems built under the fail-stop model are prone to poor performance. They perform well when the components have well-defined failure-mode operating characteristics, but fails when just a single component does not behave as expected.

The **byzantine** failure model was proposed by Lamport et al. in [Lamport 1982]. As described by the author: "The component can exhibit arbitrary and malicious behavior, perhaps involving collusion with other faulty components". This model assumes that computers and networks may behave in unexpected ways due to hardware failures, network congestion and disconnection, as well as malicious attacks. These scenarios may lead to failures like: crash failures, fail to receive a request, fail to send a response, processing a request incorrectly, corrupt the local state, send an incorrect or inconsistent response to a request. In presence of a byzantine fault, unless the system is designed to have byzantine fault tolerance, the faulty component continues to run, but produces incorrect results. Dealing with byzantine faults is more troublesome than dealing with fail-stop faults.

While the **byzantine** failure model is considered general and difficult to apply, the **fail-stop** failure model is considered too simple, and inadequate to represent the behavior of modern systems or software components. Based-on these limitations, Remzi and Andrea Arpaci-Dusseau [Arpaci-Dusseau 2001], proposed the **fail-stutter** failure model. This model extends the fail-stop model by taking into account performance-faulty scenarios. A performance fault is an event in which a component provides unexpectedly low performance, but continues to function correctly with regard to its output.

The advantages of **fail-stutter** are clearly aligned with the aim of dependability. Dependability is an integrating concept that encompasses attributes like availability (readiness for correct service), reliability (continuity of correct service) and maintainability (easy of maintenance to correct, isolate or prevent anomalies and maximize the efficiency and reliability). According to [Arpaci-Dusseau 2001], the **fail-stutter** failure model improves the reliability because it allows to cope with the diversity of components and the potential performance anomalies that may arise from its operation. By detecting and recovering from performance anomalies, it becomes possible to prevent situations that may lead to the unavailability of the service.

Independently from the type and duration of failures, the means used to attain the dependability of systems can be grouped in four main categories:

- Fault removal: aims to reduce the number and severity of faults;
- Fault prevention: aims to prevent the occurrence of introduction of faults;
- Fault tolerance: means to avoid service failures in the presence of faults;
- Fault forecasting: means to estimate the present number, the future incidence and the consequences of faults.

Classic software verification and validation techniques, like defensive programming, formal verification, static and dynamic analysis, and testing are fundamental to handle the faults. However, these techniques are insufficient to remove or prevent all the faults, especially because there are many different configurations and situations that may occur at runtime and which cannot be foreseen at design time. Fault tolerance and forecasting is more focused on the continuous operation of a system in presence of faults. These techniques are considered more useful to mask hardware or software failures that may affect the production environments. The adoption of survivable techniques to sustain the unexpected behavior is also used to containing failing components and securing the essential services, representing a minimal but functioning system configuration.

In this Thesis we focus on production Web-based applications. Web-based applications are complex systems, that rely on a variety of hardware and software components, protocols, languages, interfaces and standards. In these systems the occurrence of a degradation is unpredictable and can lead to situations where the system continues to operate with a loss on performance. Upon the occurrence of a performance anomaly, there are actions that can be taken to restore the system operation, without causing a complete disruption. In this context the **fail-stutter** failure model is what best fits into our goals. By considering this failure model, it becomes possible to define fault tolerance mechanisms which take into account unexpected behaviors that may arise at runtime and define recovery strategies to avoid, or mitigate, the impact of failures. The inclusion of a fault tolerance mechanism in a self-management tool, as is the case of the self-healing framework we propose, allows to: automate the tasks of detecting and recovering from performance failures; reduce the dependence on human intervention; and increase the service dependability.

2.2.3 Causes of failure in Web-based applications

Web-based applications fit the class of business-critical applications. They are being used in many organizations as part of the business process, and scenarios of performance slowdown or unavailability have a negative impact in terms of revenue, customer satisfaction and productivity.

A study presented by Bojan Simic [Simic 2010] refers that one hour of website downtime represents a revenue loss of $21,000. According to the same study, the impact of a website slowdown is estimated in $4,100 per hour. The same study refers that website slowdowns are 10 times more frequent than downtimes. Another report, provided by the Aberdeen Group [Aberdeen Group], is also very clear about the impact of a performance slowdown: a delay of 1-second in page load time can represent a loss of $2.5 Million in sales per year for a website that typically earns $100,000 a day. Greg Linden, in [Linden 2006], refers that an increase of 500 milliseconds in the access latency to Google may result in a loss of 20% of its traffic, and an increase of just 100 milliseconds in the access latency to Amazon may result in a reduction of 1% of sales. In [Poggi 2011], authors present a study that considers the volume of sales lost for an online travel agency during performance degradation periods. Through the analysis performed by the authors, it was found that an increase in response time between 7 and 10 seconds leads to a loss of 5% in sales volume. A response time between 10 and 20 seconds, leads to the customers frustration, and a drop of 53% in sales volume.

The customer dissatisfaction is hard to be estimated, however, according to a study conducted by PhoCusWright and Akamai [Rheem 2010], 8 out of 10 people will not return to a site after a disappointing experience and of these, 3 will go on to tell others about their experience. Another report, provided by Sean Power in [Power 2010], says that 37% to 49% of users who experience performance issues will either abandon the site or switch to a competitor, and 77% of these will share their negative experience with others.

The causes of failures in Web-based applications are diverse. A technical report provided by Soila Pertet and Priya Narasimhan, in [Pertet 2005], provides a concrete vision about the different types of failures. From the analysis of 40 incidents of real-world site outages, the report outlines the different cause of failures and how they were detected. Considering four different causes of failures - software failures, operator error, hardware and environmental failures - the authors have concluded that 40% of the observed failures were due to software, 40% due to operator errors and 20% were due to hardware, operating system, network or power/disaster problems. According to the same study, a large number of software failures occurs due to routine maintenance, software upgrades and system integration. Situations of system overload, resource exhaustion (e.g., memory leaks, thread pool exhausted, fast-growing files) and complex recovery routines are also a significant cause for software failures. The operator errors occur mainly due to configuration errors, procedural mistakes and miscellaneous accidents.

Independently from the type and source of a failure, its occurrence is perceived by the end-users when a total or partial site becomes unavailable, exceptions are thrown, the execution process returns erroneous results (e.g., blank pages, wrong items), the access to data is compromised due to data losses or data corruptions or when a performance slowdown is experienced. The reasons behind these failures are not clearly defined, but is common sense that the software and the infrastructure complexity is a central piece of the problem. The number of errors involuntarily introduced by human-operators while they are managing the systems is also related with the system complexity. These statements, about the complexity in nowadays Web-based applications, is reinforced in [Kiciman 2005]. From conversations with Internet service operators, authors have estimated that 75% of the time to recover from application failures is spent with detection and localization. In some cases the failure detection can take days. This situation has a strong impact on the service availability with consequent repercussions on the business operations.

In this context, and to avoid the negative impacts of anomalies, it is important to work in offline optimization techniques for Web-based applications (like those ones presented by Steve Souders [Souders 2007]) and in online techniques able to detect and recover from performance anomalies that may arise at runtime. To handle with the complexity and reduce the dependence on human intervention, it is essential to automate the tasks of detecting, pinpointing and recover from anomalies. Thus, self-healing is assumed as an emerging need.

2.2.4 Related work

A system to be self-healing should be able to continuously gather information about itself and perform analysis to detect the occurrence of anomalies. When an anomaly is detected and diagnosed, the system should be able to automatically select and execute a recovery plan to avoid the harmful impact of failures.

In the literature we found numerous examples focusing and combining the tasks involved in a self-healing system: monitoring, anomaly detection, anomaly localization, anomaly diagnosing and recovery. All examples aim to maximize the system's dependability.

A survey on self-healing systems is provided in [Psaier 2011]. The authors begin by presenting the self-healing in the context of Autonomic Computing, adaptive systems, fault tolerance and survivable systems. Then, they present the vision, the objectives, the attributes and the means that support the existence of a self-healing system. The relations and properties between these aspects are illustrated in Figure 7. The survey also provides a summary about the self-healing approaches adopted in different research areas: operating systems, multi-agent based systems, legacy applications, Web-services and QoS based systems to name a few.

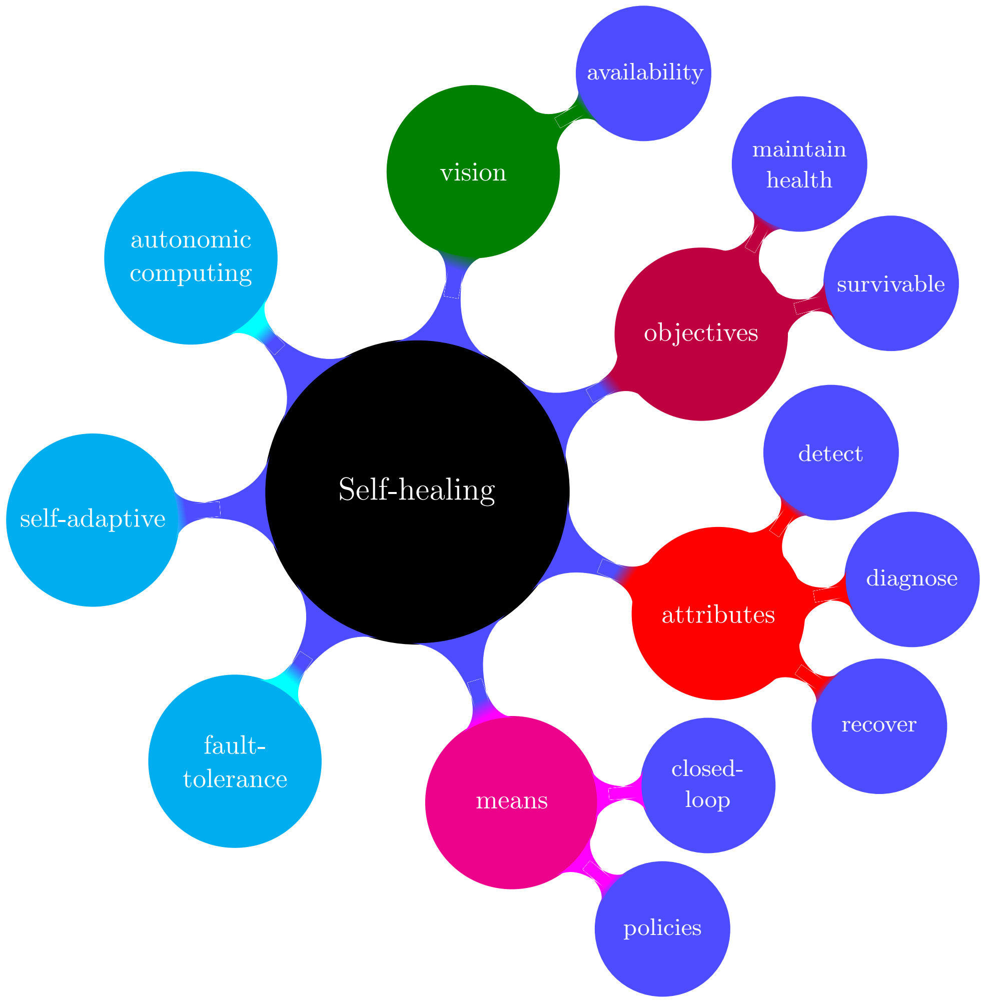


Figure 7: Relations and properties of self-healing

The self-healing approaches for operating systems encompass two categories of faults: soft faults and hard faults. Soft faults are considered application crashes that can be recovered. Hard faults require a complete system restart. In [Tanenbaum 2006] the authors refer that the main objective of self-healing in operating system is to avoid faults that require a system restart. According to the authors this might be achieved by releasing the supervising kernel from dependencies and by freeing the allocated resources and rerun the failing applications. In [Candea 2004a] authors present a recovery approach for soft faults, described as recursive *microreboots*. This approach starts by *microreboot* a minimal subset of components, and if that recovery is not enough, larger subsets of components are then rebooted. In [Herder 2006] authors present a system, based-on the implementation of microkernels. The microkernel is a small kernel that reduces the interdependencies between the operating system layers and allows to isolate faults more easily. All the server applications and drivers run in user mode. The recovery process is supported by a reincarnation server and a data server process. The reincarnation server is a parent process to all other processes and notices when a child thread fails. The data server holds configuration and status information. When an application fails, it is replaced with a fresh copy. In [Shapiro 2004] is presented a predictive self-healing mechanism. This mechanism is implemented in the Solaris-10 operating systems. It includes clever diagnosis engines that diagnose the messages provided by the system logging facilities. Each engine attempts to anticipate a possible degradation and triggers an automated recovery action when a degradation is observed. The recovery is supported by a service manager, which handles the dependencies between the application and daemons, defining the correct order of possibly multiple restarts.

The multi-agent based systems typically provide robustness by redundancy. They are composed by multiple cooperative agents which, by design, are self-aware about its environment and able to handle situations of unexpected behavior. In [Corsava 2003] the authors propose a multi-agent system to support the IT staff in diagnosing the behavior of different resources in a cluster environment. The set of agents were named as *intelliagents* and they are responsible for monitoring the system, detect performance, capacity and error problems and execute actions to repair the affected resources. The results achieved in a production environment shows that with the self-healing features provided by the *intelliagents*, the number of hours of downtime related with the resources has reduced from 276 hours to only one hour. The period of analysis covers the 16 months before its adoption until 16 months after of its adoption. As a complement to redundancy, in [Huhns 2003] and [Tesauro 2004] the authors consider *redundancy by diversity* and *redundancy by voting*. The *redundancy by diversity* is contemplated as a means of preventing that various agents, which run the same algorithm, fail due to the same error. The *redundancy by voting*, as well as allowing to validate the results, are used to decide which are the most suitable agents for an incoming task, promoting the system reliability.

The systems for self-healing legacy applications include areas related to the applications instrumentation, data analysis and fault-tolerance mechanisms. In [Griffith 2005] authors present a framework that allows to attach and detach a repair engine to a running .NET application to perform consistency checks and repairs over individual components and sub-systems. The framework instruments the application without requiring modifications on the target system's source code. When the classes are loaded for the first time, the just-in-time (JIT) compiler includes hooks into the generated bytecode. In case of failure, these hooks allow the execution to be redirected to the repair engine. In [Fuad 2006] a similar approach is used for Java based applications. In that work, the hooks provide checkpoint control at strategic points and each method is finalized with a try-catch block. In case of runtime failures (exceptions and errors), it is possible to interact with the managed code and the recovery procedure tries to reinitialize the code, so it can continue at the last checkpoint. When the recovery attempt fails, the system administrator is informed and supported with information of the application to diagnose such unsettled failures. Alonso et al. in [Alonso 2008] presents an Aspect-Oriented Programming (AOP) fine-grain monitoring framework, composed by multiple sensors and a monitor manager. The sensors intercept the application in runtime (e.g., before/after the methods are called, before the constructors are invoked or before one object is initialized). The monitoring manager includes data mining and forecasting methods to analyze the data in order to predict failures. In [Haydarlou 2005] the authors combine AOP, program analysis, artificial intelligence and machine learning techniques to heal failures in object-oriented applications. The AOP sensors are attached at the strategic objects of the application. The sensors are used to monitor event transitions, like method calls, state changes, branch selection and the occurrence of exceptions. When some runtime failure occurs, the sensor collects failure context information and triggers a recovery planner module. Depending on the type of failure, the recovery may be performed in different ways (e.g., reflexive, instinctive, cognitive). In [Chang 2008] authors present a technique to self-heal common application components integration faults. The authors consider faults that result from raised exceptions, particularly the faults motivated by the usage of invalid methods parameters, wrong methods invocation, faulty methods implementation and environment faults like the ones that occur due to IO exceptions or when a given class is not found. The type of faults was chosen based-on the most frequent faults observed in component-based applications and registered in the bug repositories. To cope with these faults, authors have used a predetermined set of healing strategies: intercept and change parameters followed by a retry operation, intercept and call some operations before attempt the original call, replace the calls, intercept the call to change some environmental aspects before retry the operations. From the preliminary experiments, authors have confirmed the usefulness of the approach to mask some application-faults before they result in application failures. The Pinpoint project [Candea 2006] and [Chen 2002] also includes self-healing services for Web applications. This project relies on a modified version of the application server [Candea 2003a] to monitor the application structure and pinpoint application-level failures without require prior knowledge about the application. The request execution is intercepted to model the interactions between components.

In runtime, differences between the expected and observed interactions are regarded as failures and recovery procedures are initiated. To reduce the functional disruption motivated by a server or process restart the components are *microrebooted*, according to the approach presented in [Candea 2004b].

A self-healing service for Web Services and Service-Oriented Architectures (SOA) is presented in [Baresi 2004]. Authors consider applications composed by multiple services discovered and attached in runtime (dynamic binding) along with faulty behaviors that are mostly unforeseeable at design/binding time. Services that fail to supply the service agreed during the service lookup phase and third-party changes are some situations that may lead to failures. To cope with these erroneous behaviors authors combine defensive process design with service runtime monitoring and recovery. The defensive design consists of designing the business process in such a way as to permit it to cope with erroneous behaviors, but exposing the running results. The runtime monitoring makes use of an external monitor service to check wether the functional and non-functional contracts are violated. Once a contract is violated, i.e., a fault is notified, different recovery techniques are used to circumvent the problem. Retry the invocation, dynamically bind to another service and the dynamic reorganization of the business process at runtime are some of the recovery strategies followed to address the faulty scenarios. Modafferi et al. [Modafferi 2006] extend a standard BPEL engine with a self-healing service. The resulting SH-BPEL engine extends the recovery mechanisms, specified by the composition designer, with fault handlers and extended recovery mechanisms that allow, for example, the substitution of services if they do not respond within an agreed time threshold.

In the previous examples we did not described the separation of concerns, the level of intrusiveness, the different techniques adopted to detect and report anomalies, the diagnosing approaches and the corresponding recovery techniques followed by the different self-healing projects. Since this Thesis addresses the self-healing techniques for Web-based applications we will, over the next chapters, detail the most influential projects in this area. The level of detail will be deepened as we present the various stages of the MAPE loop: Monitor, Analyze, Plan and Execute.

2.3 Summary

The increasing complexity of the systems is pointed as the main obstacle of the IT industry [Ganek 2003]. Autonomic Computing is presented as the alternative to cope with the growing complexity. It defines an holistic approach and aims to provide the self-managing of systems. For this purpose a system must be able to self-configure, self-heal, self-optimize and self-protect. These attributes are very important to improve the systems stability, availability, security, as well, to reduce the need for highly skilled staff to maintain the systems.

Web-based applications are a class of systems where the complexity is very high. The number of components involved, their interoperability and workload variations are factors that may lead to performance failures or unavailability scenarios. The occurrence of these scenarios causes impact on the revenue, image and productivity of business that rely on this type of applications to perform business operations. Fault removal, fault prevention and fault tolerance mechanisms are precious to cope with these problems, but despite being adopted, there are still situations in which applications become unavailable or are affected by performance degradations. Also, as indicated by some reports in the area, the detection, pinpointing and recovery of the anomalies in this type of applications is complex and time consuming.

The combination of all of these aspects, highlights the role of a self-healing mechanism. Self-healing enables enhanced detection, pinpointing and recovery from failures, freeing the human operators from these complex, time consuming and error prone tasks. Its adoption for Web-based applications can thus be seen as a very important contribution to detect and recover from problems that may arise at runtime, mitigating the losses that occur due to the availability and reliability issues. In the next chapter we describe a self-healing framework target for Web-based applications. The expected contribution of our work, the implementation details and an experimental evaluation is then presented in the following chapters.

Chapter 3
Self-healing of Web-based Applications

KEY POINTS

- ◇ Web-based applications are becoming increasingly complex. These applications are composed of multiple components developed by different entities, rely on services provided by third parties and operate in heterogeneous and very dynamic environments;
- ◇ Such complexity, turns the applications more difficult to configure and manage and makes them more exposed to the occurrences of anomalies;
- ◇ The occurrence of anomalies, like unavailability or performance failures, have a direct impact and a significant on the financial return and on the company's reputation;
- ◇ The development of self-healing systems is seen as extremely important for mitigating the problems of complexity, allowing to monitor the applications or systems at runtime, in order to, detect and recover from anomalies quickly and autonomously;
- ◇ In this chapter we present a self-healing framework targeted for the detection, localization and recovery of performance anomalies in Web-based applications.

The performance and availability of Web-based applications is a crucial aspect for the enterprises. These applications are business-critical and the occurrence of unavailability or performance failures can lead to revenue losses, lack of productivity and cause a negative impact to the image of the company.

To avoid such problems, it is mandatory to work either in offline optimization techniques for Web applications and in online techniques that allow these applications to be fault-tolerant and able to maintain the appropriate level of service. Adopt the best programming practices, like those ones presented by Steve Souders [Souders 2007], is fundamental. Perform software testing during the software development cycle to get a good application performance and to eliminate application errors, is a requirement. The use of redundancy and load balancing mechanisms is essential to cope with failures affecting the infrastructure and to improve the scalability of the service. Active monitoring systems are vital to detect and alert the IT staff for anomalous situations that occur at runtime.

In this work we propose a self-healing framework for Web-based applications. The framework is called *SHõWA* and it aims to detect the occurrence of runtime anomalies that are affecting the application behavior and mitigate its impact through the automatic execution of recovery procedures. The framework is illustrated in Figure 8.

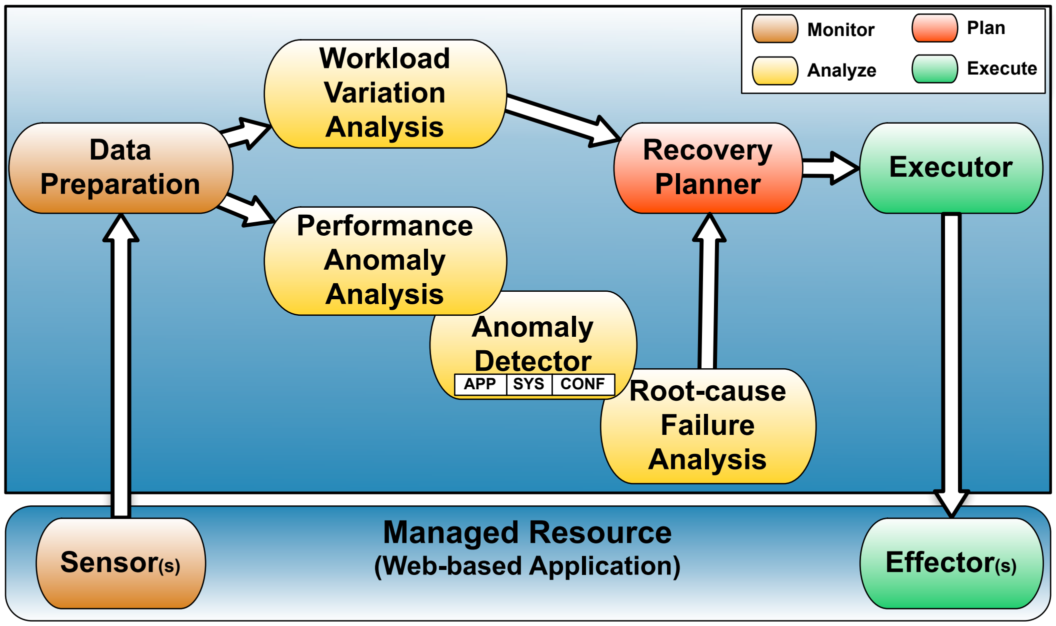

Monitor
Plan
Analyze
Execute
Workload Variation Analysis
Data Preparation
Recovery Planner
Executor
Performance Anomaly Analysis
Anomaly Detector
APP
SYS
CONF
Root-cause Failure Analysis
Sensor(s)
Managed Resource (Web-based Application)
Effector(s)

Figure 8: *SHõWA* framework - building blocks

The framework, illustrated in Figure 8, is composed by several building blocks that cooperate each other to achieve the self-healing property. It resembles an autonomic element. The managed resource corresponds to the Web-based application and its execution environment. The autonomic manager iterates over the four processes: Monitoring, Analyze, Plan, Execute. It gathers data from the managed resource, analyzes the data to detect and pinpoint anomalies, plans the recovery actions and executes recovery actions when necessary.

Generically, the *Sensor* module:

- Collects data from different system and application server parameters and measures the execution time of the user-transactions and application calls;
- Sends the data to a remote database.

The data is prepared by the *Data Preparation* module. The data preparation involves:

- The data aggregation in time intervals;

- The creation of a unique key at the end of each interval, which identifies the mix of transactions in a given interval;
- The creation of an association key to identify the list of calls belonging to a given user-transaction.

The *Workload Variation Analysis* and the *Performance Anomaly Analysis* modules:

- Carry out statistical analysis to detect response time variations;
- Verify if the response time variations are due to workload changes or are a symptom of performance anomaly.

If there is a variation in the response time, motivated by a performance anomaly, then:

- The *Anomaly Detector* module carries out statistical analysis to detect if there is any change in the system or application server parameters, correlated with the performance anomaly;
- The *Root-cause Failure Analysis* module carries out statistical analysis to analyze the response time of the application calls, in order to check if there are changes in the application, or changes in the invocation of remote services, correlated with the performance anomaly.

Upon the detection and localization of performance anomalies, it follows the recovery phase. The recovery phase involves:

- The *Recovery Planner* module which is responsible for selecting a recovery procedure;
- The *Executor* module which dispatches the recovery actions to be applied on managed resource by the *Effector* module.

The mode of operation and the methodologies we adopted in the implementation of the modules, are presented in more detail throughout this chapter.

3.1 Challenges to be addressed

Focusing the ability to self-heal production Web-based applications, the implementation of a such autonomic element involves some fundamental decisions and evaluations. The implementation of the *SHõWA* framework presented in this chapter addresses the following questions:

- *What type of failures should be considered?*

- *How to detect and pinpoint the anomalies?*
- *Which and how much data is relevant to pinpoint anomalies?*
- *How to collect data according to the open and self/context awareness characteristics defined in the Autonomic Computing principles?*
- *Which recovery techniques should/could be applied and how to apply them?*
- *How timely is the detection, pinpointing and recovery?*
- *How efficient are the proposed approaches?*
- *What is the penalty of introducing new layers in the system to achieve all of these features?*

The type of failures to be addressed is one of the most important considerations to be made. We did adopt the **fail-stutter** failure model in the *SHõWA* framework. This model takes into account performance-faulty scenarios, in which, a component provides unexpectedly low performance, but continues to function correctly with regard to its output.

System-level monitoring and end-to-end monitoring systems are commonly adopted to detect problems in Web-based applications. System-level monitoring observes the system parameters periodically and triggers alerts when the status of the parameters does not match the expected state. End-to-end monitoring is used to check the status of the service, as it is experienced by the end-users. The end-to-end monitoring systems typically execute, in a periodic manner, one transaction, or a set of transactions, in order to check the availability of the service, the occurrence of errors and the response time. Another monitoring system that is gaining expression within the Web applications, is application-level monitoring. This type of monitoring complements the previous ones. It provides specific data on the state of the application. For example, in addition to indicating if a transaction is slow, these systems can indicate whether the slowness is due to the state of the resources in the system (e.g., CPU load). The *SHõWA* framework performs application-level monitoring. It gathers the execution time of the application transactions and collects data from the system, in order to detect performance anomalies and pinpoint if the anomaly is motivated by a system change, a change in the application server or an application change.

Collect application-level data in production Web-based applications presents another challenge. Usually the access to the application source code is limited. Even if it is possible, nowadays enterprise applications are made up of multiple and heterogenous components that cross the enterprise boundaries, making the understanding of the source code a complex and daunting task. The *Sensor* module makes use of Aspect-Oriented Programming (AOP) [Kiczales 1997]. It allows to monitor the applications at runtime, without requiring manual changes in the application source code or previous knowledge about the implementation details.

The monitoring and data analysis performed by the *SHõWA* framework is intended to detect and pinpoint performance anomalies. Determine the location of the anomalies contributes to determine the cause of the same and to select the most appropriate recovery plan. To this end, the *SHõWA Recover Planner* module includes different recovery procedures and depending on the type and localization of the anomaly, it selects automatically what recovery procedure to apply.

As can be seen by the brief description given in this section, *SHõWA* comprises concepts from various fields: monitoring, data analysis, anomaly detection and localization, recovery. Over the next sections we describe the implementation details of *SHõWA*, focusing on the approaches used in each of these areas. Over the next chapters of the Thesis, we present the state of art of each area, and we evaluate, by means of experimental analysis, the behavior of the *SHõWA* framework.

3.2 Monitoring: *SHõWA Sensor* module

The *Sensor* module is a small program implemented according to the Aspect-Oriented Programming (AOP) paradigm [Kiczales 1997]. This program is installed with the application server and allows collecting information about the applications that run on the server, as well as, the state of the system parameters and application server parameters.

To activate the *Sensor*, we just need to put the program in a directory of the application server, so that when the application server is started the monitoring program is automatically loaded. Once loaded, the *Sensor* module intercepts the running application every time is verified a match between the method/call signature and the AOP pointcuts defined in the *Sensor* module source code.

The type of instrumentation, made by AOP, is known as bytecode instrumentation. It is very advantageous since it separates the monitoring code from the application code, thereby allowing an application to be monitored without the need for manually changing its source code. Also, with this separation, multiple applications can be monitored using the same monitoring code.

The *SHõWA Sensor* module is prepared to collect data with two different levels in terms of detail: *user-transaction* mode and *profiling* mode. As illustrated in Figure 9, in the *user-transaction* mode it intercepts the user-transactions at the moment they start and at the moment they finish, measures the transactions processing time and collects the state of the system and application server parameters. In the *profiling* mode it intercepts and measures all the internal calls (methods execution time) belonging to each one of the user-transactions.

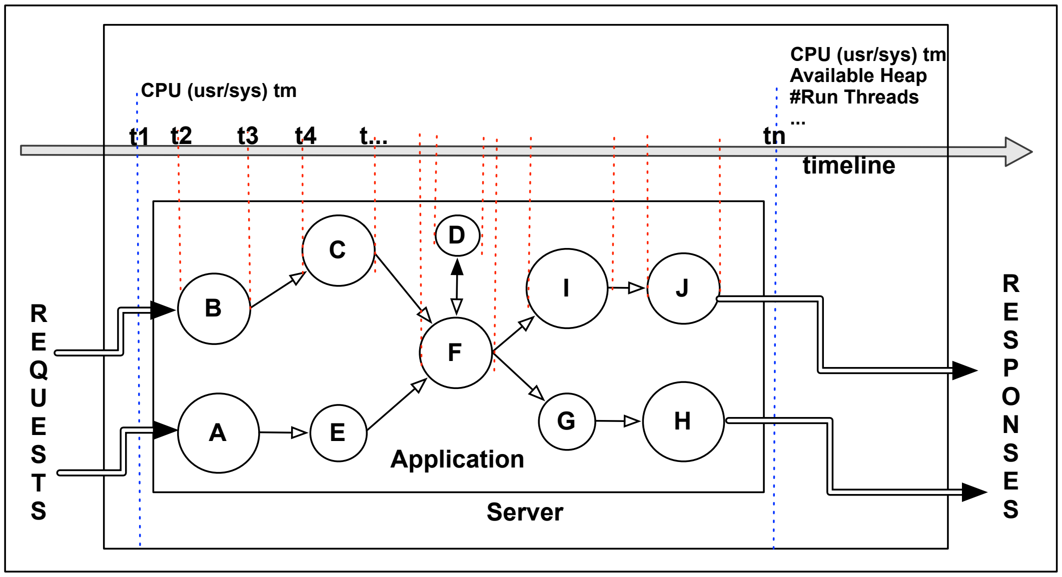


Figure 9: Application-level monitoring: user-transactions interception and measurement in a *user-transaction* mode (t_1 and t_n) and *profiling* mode (t_2 - t_{n-1})

In Table 1, we present the list of parameters collected by default. This list can be extended if more parameters are assumed as relevant for the analysis.

Table 1: Example of parameters collected by the *Sensor* module

Parameter	T	Description
User-Transactions	$t_n - t_1$	Duration per user-transaction (ms)
CPU (trans.)	$t_n - t_1$	CPU time per user-transaction
Memory	t_n	Amount of available memory
Threads	t_n	Number of running and created threads
Transaction	t_1	Signature/Transaction name
File Descriptors	t_n	Number of open and max number of files allowed
Classes	t_n	Number of running and created classes
CPU (OS)	t_n	Percentage of CPU USR, SYS and WIO usage
Net traffic	t_n	Amount of bytes received and sent
Disk traffic	t_n	Amount of bytes written and read
Profiling	$t_2 ... t_{n-1}$	Duration per user-transaction call (ms)

The data collected in the *user-transaction* mode is used by the *Workload Variation Analysis* and *Performance Anomaly Analysis* modules to detect slow user-transactions and identify if is motivated by a workload change or it is a symptom of performance anomaly. It is also used by the *Anomaly Detector* module to verify if there is any system or application server parameter change that is associated with the slowdown. The data collected in the *profiling* mode is used by the *Root-cause Failure Analysis* module to identify if there are calls/components associated with a given performance anomaly.

To minimize the performance impact related to the transmission of the monitoring data, we considered the use of a dedicated network segment. To facilitate the data analysis and to minimize performance problems we considered the use of a high-performance, in-memory relational database. The database, the data preparation and the data analysis processes should be done in a dedicated server, reducing the risk of affecting the performance of the system under monitoring.

To minimize the performance impact induced by application-level profiling, we consider the use of adaptive and selective monitoring algorithms. These algorithms adapt, dynamically, the behavior of the *Sensor* module, reducing or increasing the frequency of application profiling monitoring. Next we present the algorithms that are currently implemented.

3.2.1 Adaptive monitoring

The application *profiling* performed by the *Sensor* module, measures the processing time of all the application calls. In practice, this corresponds to take a timestamp before and after each line of code executed. With such low-level data, we can verify of there are application calls associated with a response time slowdown. Collect low-level data increases the risk of seriously affect the performance of the system under monitoring. To mitigate this problem, the *Sensor* module adopts an adaptive behavior. It includes adaptive and selective algorithms to self-adjust the frequency to which the *Sensor* module profiles the application calls, as well as, select dynamically the list of calls that should be intercepted.

The adaptive algorithms can be adopted for different type of adaptation (e.g. time-based adaptation, parameter-based adaption). Currently it is possible to choose between one of three algorithms: linear, exponential and polynomial. These algorithms use the correlation degree, computed by the *Performance Anomaly Analysis* module, to adjust the sampling frequency. The *Performance Anomaly Analysis* module is presented in Section 3.3.

3.2.1.1 Linear adaptation algorithm

In the linear adaptation algorithm the sampling frequency adjustment is proportional to the decrease or increase verified in the correlation degree computed by the *Performance Anomaly Analysis* module.

We use the notation M to denote the maximum sampling frequency and m to denote the minimal sampling frequency. The average correlation degree per user-transaction, $\bar{x}$, is derived from the historical correlation degrees and r refers to the last correlation degree computed. α corresponds to a correlation degree decrease, that is representative of a symptom of anomaly. The algorithm produces the value K, which is the amount of adjustment to be applied. It takes a value between M and m and corresponds to the sampling frequency used by the *Sensor* module. As higher the value of K is, lowest is the sampling frequency. As lowest the value of K is, higher is the sampling frequency.

The linear adjustment of the frequency of user-transactions profiling is given by Algorithm 1.

Algorithm 1 *SHõWA* user-transactions profiling: Linear adaptation of the sampling frequency

```
1: procedure LinearAdaptation(M,m,$\bar{x}$,r,$\alpha$)
2:     $K \leftarrow floor(M - ((\bar{x} - r) * (M/\alpha)))$
3:     if $(K > M)$ then
4:         $K \leftarrow M$
5:     end if
6:     if $(K < m)$ then
7:         $K \leftarrow m$
8:     end if
9:     Return $K$
10: end procedure
```

The $\bar{x} - r$ corresponds to the decrease observed in the correlation degree. The M/α corresponds to the adjustment to make to M, per unit of correlation degree above or below $\bar{x}$. Thus, the greater the decrease or increase in the observed correlation degree regarding to a reference correlation degree value $(\bar{x} - r)$, greater is the value to subtract $((\bar{x}-r)*(M/\alpha))$ to the maximum sampling frequency (M). The new sampling frequency is always between m and M.

3.2.1.2 Exponential adaptation algorithm

The exponential adaptation algorithm improves on the linear adaption algorithm by considering the adjustment of the sampling frequency K using a exponentiation factor (Eq. 3.2.1), that depends on M and α.

$$M = e^{F\alpha} \tag{3.2.1}$$

On solving the Eq. 3.2.1 we get the exponentiation factor F (Eq. 3.2.2),

$$F = \frac{\ln M}{\alpha} \tag{3.2.2}$$

By using an exponentiation factor for adjusting the frequency of user-transactions profiling we have: a moderate frequency adjustment when the variation in the correlation degree is small, a faster adjustment in the frequency when the change observed in the correlation degree approaches the defined correlation degree threshold (α).

The exponential adjustment of the frequency of user-transactions profiling is given by Algorithm 2.

Algorithm 2 *SHõWA* user-transactions profiling: Exponential adaptation of the sampling frequency

```
1:  procedure ExponentialAdaptation(M,m,$\bar{x}$,r,$\alpha$)
2:      $Q \leftarrow \bar{x} - r$;
3:      if ($Q \leq 0$) then
4:          $Q \leftarrow 1$
5:      end if
6:      if ($Q > M$) then
7:          $Q \leftarrow M$
8:      end if
9:      $K \leftarrow floor(e^{(F*Q)}) - M$                ▷ F is determined by Eq. 3.2.2
10:     if ($K < m$) then
11:         $K \leftarrow m$
12:     end if
13:     Return $K$
14: end procedure
```

With Algorithm 2, the sampling frequency remains large when r deviates only a few degrees from $\bar{x}$. This way the amount of data collected is reduced and the performance impact induced is minimized. For significant deviations, the sampling frequency increases, accelerating the application-level profiling frequency and providing more data, to support the data analysis process to pinpoint the anomaly.

3.2.1.3 Polynomial adaptation algorithm

The polynomial adaptation algorithm fits between the linear and exponential algorithms. It is composed by different exponential functions and it allows to keep the sampling K large until a given amount/percentage (identified by β) of correlation degree decrease is observed. After that point, the sampling frequency K can be adapted more frequently, to provide enough data for the pinpointing of anomalies.

For example, considering a polynomial function of order three ($f(x) = ax^3 + bx^2 + cx + d$), and a β value in percentage, the polynomial adaptation function can be achieved by solving the system of equations presented in Eq. 3.2.3.

$$\begin{cases} M = a + b + c + d \\ M\beta = (\alpha\beta)^3 a + (\alpha\beta)^2 b + \alpha\beta c + d \\ M\beta^2 = (\alpha 2\beta)^3 a + (\alpha 2\beta)^2 b + \alpha 2\beta c + d \\ m = \alpha^3 a + \alpha^2 b + \alpha c + d \end{cases} \tag{3.2.3}$$

Considering a system of linear equations with n equations and n variables, the independent terms a, b, c and d of Eq. 3.2.3, can be solved using the Cramer's rule. For each variable, the denominator is the determinant of the matrix of coefficients, while the numerator is the determinant of a matrix in which one column has been replaced by the vector of constant terms. The set of equations used to solve the a, b, c and d terms is presented in Eq. 3.2.4.

$$a = \frac{\begin{vmatrix} M & 1 & 1 & 1 \\ M\beta & (\alpha\beta)^2 & \alpha\beta & 1 \\ M\beta^2 & (\alpha 2\beta)^2 & \alpha 2\beta & 1 \\ m & \alpha^2 & \alpha & 1 \end{vmatrix}}{\begin{vmatrix} 1 & 1 & 1 & 1 \\ (\alpha\beta)^3 & (\alpha\beta)^2 & \alpha\beta & 1 \\ (\alpha 2\beta)^3 & (\alpha 2\beta)^2 & \alpha 2\beta & 1 \\ \alpha^3 & \alpha^2 & \alpha & 1 \end{vmatrix}} \quad b = \frac{\begin{vmatrix} 1 & M & 1 & 1 \\ (\alpha\beta)^3 & M\beta & \alpha\beta & 1 \\ (\alpha 2\beta)^3 & M\beta^2 & \alpha 2\beta & 1 \\ \alpha^3 & m & \alpha & 1 \end{vmatrix}}{\begin{vmatrix} 1 & 1 & 1 & 1 \\ (\alpha\beta)^3 & (\alpha\beta)^2 & \alpha\beta & 1 \\ (\alpha 2\beta)^3 & (\alpha 2\beta)^2 & \alpha 2\beta & 1 \\ \alpha^3 & \alpha^2 & \alpha & 1 \end{vmatrix}}$$

$$c = \frac{\begin{vmatrix} 1 & 1 & M & 1 \\ (\alpha\beta)^3 & (\alpha\beta)^2 & M\beta & 1 \\ (\alpha 2\beta)^3 & (\alpha 2\beta)^2 & M\beta^2 & 1 \\ \alpha^3 & \alpha^2 & m & 1 \end{vmatrix}}{\begin{vmatrix} 1 & 1 & 1 & 1 \\ (\alpha\beta)^3 & (\alpha\beta)^2 & \alpha\beta & 1 \\ (\alpha 2\beta)^3 & (\alpha 2\beta)^2 & \alpha 2\beta & 1 \\ \alpha^3 & \alpha^2 & \alpha & 1 \end{vmatrix}} \quad d = \frac{\begin{vmatrix} 1 & 1 & 1 & M \\ (\alpha\beta)^3 & (\alpha\beta)^2 & \alpha\beta & M\beta \\ (\alpha 2\beta)^3 & (\alpha 2\beta)^2 & \alpha 2\beta & M\beta^2 \\ \alpha^3 & \alpha^2 & \alpha & m \end{vmatrix}}{\begin{vmatrix} 1 & 1 & 1 & 1 \\ (\alpha\beta)^3 & (\alpha\beta)^2 & \alpha\beta & 1 \\ (\alpha 2\beta)^3 & (\alpha 2\beta)^2 & \alpha 2\beta & 1 \\ \alpha^3 & \alpha^2 & \alpha & 1 \end{vmatrix}} \tag{3.2.4}$$

The corresponding polynomial function is then used, within the polynomial adaptation algorithm, to define the sampling frequency K. The value of K is given by Algorithm 3.

Algorithm 3 *SHõWA* user-transactions profiling: Polynomial adaptation of the sampling frequency

```
1: procedure PolynomialAdaptation(M,m,$\bar{x}$,r,$\alpha$.$\beta$)
2:     $Q \leftarrow \bar{x} - r$;
3:     if $(Q \leq 0)$ then
4:         $Q \leftarrow 1$
5:     end if
6:     if $(Q > M)$ then
7:         $Q \leftarrow M$
8:     end if
9:     $K \leftarrow floor(aQ^3 + bQ^2 + cQ + d)$ ▷ a,b,c, and d: determined according to 3.2.3
10:    if $(m > 1$ and $K < m)$ then
11:        $K \leftarrow m$
12:    end if
13:    Return $K$
14: end procedure
```

By using Algorithm 3, the sampling frequency is adapted at three different rates. If the correlation degree decrease is below a given threshold (β), then the adaptation is very small, i.e., the sampling frequency will remain large. If the correlation degree degree is close to the α value, then the adaptation will occur more quickly, allowing to collect more low-level data to support the root-cause analysis.

3.2.2 Selective monitoring

While the adaptation algorithm redefines the sampling frequency as the correlation degree between user-transactions response time and the number of concurrent user-transactions changes, the selective monitoring technique adjusts the list of calls/components that are profiled by the *Sensor* module.

A user-transaction can hold a considerable number of calls, with different contribution to the total response time. Rather than intercepting all the calls, the *Sensor* module takes advantage of the AOP pointcut to decide, at runtime, if a call should be intercepted or not. To do this, the pointcut consults a data structure: If the call exists in the data structure then the body of the pointcut is executed and the call response time is measured; if the call do not exists in the data structure then the body of the pointcut is not executed.

The data structure contains all the calls/components which have a contribution over $\boldsymbol{X}\%$ to the total user-transaction response time. From $\boldsymbol{K} * \boldsymbol{R}$ times (being $\boldsymbol{K}$ the sampling frequency and $\boldsymbol{R}$ a refresh factor) all the calls are intercepted, and the calls with a processing time higher than $\boldsymbol{X}\%$ of the user-transaction response time are inserted in the data structure. The data structure is also updated in a reflexive manner, i.e., when the correlation degree decreases. A call is removed from the data structure when its response time is inferior to $\boldsymbol{X\%}$ of the user-transaction response time.

With this approach the number of calls to be intercepted is reduced. By reducing the number of calls the performance impact induced by the *Sensor* module will be lower. When compared to other monitoring systems, in which the parts of the application that is monitored is statically defined, the dynamic selection of calls being intercepted presents an advantage: if for some reason the response time of a component increases, it will be monitored allowing their detection by the data analysis process performed by the *SHõWA* framework.

3.2.3 Monitoring: *SHõWA Data Preparation* module

The *Data Preparation* module is responsible for preparing the data collected by the *Sensor* module.

The *Data Preparation* module includes two preparation functions: the data interval and the workload mix key. The data interval function, is used to determine the time interval at which the data collected belongs. Its operation is very simple but it is essential for the data analysis process. The lower limit of the first interval ($\boldsymbol{L_1}$), corresponds to the timestamp of the first transaction. The upper limit is achieved by summing $\boldsymbol{S}$ seconds to the lower limit ($\boldsymbol{U_1 = L_1 + S}$). Starting from the first, all the intervals are sequential: the lower limit of the $\boldsymbol{t^{th}}$ interval is given by $\boldsymbol{L_t = U_{t-1} + 1}$ and the upper limit is given by $\boldsymbol{U_t = L_t + S}$. The transactions response time, and the state of the system and application server parameters collected by the *Sensor* module, are assigned to an interval whenever its timestamp is between the interval limits determined by the data interval function.

The workload mix key function, is used to characterize the mix of user-transactions processed in an interval. This characterization is important because the resource demands may vary according to the mix of user-transactions processed. Doing this identification makes it possible to compare the response time between the intervals under analysis. The definition of the workload mix key, for a given interval, is given by Eq. 3.2.5.

$$\boldsymbol{key_t = hash(\%tm_1 \ldots \%tm_n)_t} \tag{3.2.5}$$

where,

$$\boldsymbol{\%tm_i = \frac{\sum_t UserTransaction_i}{\sum_t UserTransactions}}$$

Eq. 3.2.5 returns a $\boldsymbol{key}$ value based-on the percentage of work distribution in a given interval ($\boldsymbol{t}$). To reduce the key value space, each $\boldsymbol{\%tm_i}$ is rounded to the nearest percentage that is multiple of five percent (e.g., **0.17 => 0.15** and **0.18 => 0.20**).

3.3 Analyze: *SHõWA* detecting and pinpointing anomalies

The data analysis to detect and pinpoint performance anomalies and workload contention scenarios is performed by the *Workload Variation Analysis*, *Performance Anomaly Analysis*, *Anomaly Detector* and *Root-cause Failure Analysis* modules.

The *SHõWA* data analysis can be described as "application-centric" with a focus on how the end-users experience the service. It considers the state of the system parameters, application container parameters and the application response time to detect performance anomalies and identify whether the cause is related to:

- Workload changes;
- System or application server changes;
- Application changes.

By detecting application-level issues that occur in the server, it becomes possible to prevent the occurrence of performance failures before they reach the end-users. Network or end-user side anomalies are not captured by our analysis.

The analysis is supported by the Spearman's rank correlation coefficient, commonly represented by the Greek letter $\boldsymbol{\rho}$ (rho) [Zar 1972]. The correlation coefficient is given by Eq. 3.3.1, and it expresses how two variables ($\boldsymbol{X}$ and $\boldsymbol{Y}$) are associated. It works by calculating the Pearson's correlation coefficient on the ranked values of the data ($\boldsymbol{X}$ and $\boldsymbol{Y}$). Ranking is obtained by assigning a rank, from low to high, to the values of $\boldsymbol{X}$ and $\boldsymbol{Y}$ respectively. Unlike the Pearson's correlation, the Spearman's correlation do not requires the data to be normality distributed (it is a nonparametric statistic). The determination of Spearman's correlation coefficient and the subsequent significance testing requires the data to be measured in intervals or ratio level and the variables to be monotonically related. These assumptions are fulfilled in our analysis: the *Data Preparation* module aggregates the data in time intervals. Before being analyzed, the elements of $\boldsymbol{X}$ and $\boldsymbol{Y}$ form a pair.

$$\boldsymbol{\rho} = \frac{\sum_{i=1}^{n}(\boldsymbol{X_i} - \bar{\boldsymbol{X}})(\boldsymbol{Y_i} - \bar{\boldsymbol{Y}})}{\sqrt{\sum_{i=1}^{n}(\boldsymbol{X_i} - \bar{\boldsymbol{X}})^2}\sqrt{\sum_{i=1}^{n}(\boldsymbol{Y_i} - \bar{\boldsymbol{Y}})^2}} \tag{3.3.1}$$

From the analysis point of view, $\boldsymbol{\rho}$ can be interpreted as follows [Cohen 1988]: between $[\mathbf{-1.0, -0.5}]$ or $[\mathbf{0.5, 1.0}]$ stand for a large correlation; between $[\mathbf{-0.49, -0.3}]$ or $[\mathbf{0.3, 0.49}]$ for a medium correlation; between $[\mathbf{-0.29, -0.1}]$ or $[\mathbf{0.1, 0.29}]$ for a small correlation; between $[\mathbf{-0.09, 0.09}]$ there is no correlation.

Like in [Kelly 2005, Kelly 2006], the data analysis approach exploits typical properties of modern enterprise distributed applications:

- Workload consists of request-reply transactions;

- Transactions occur in a small number of types;
- Resource demands vary widely across but not within transaction types;
- Computational resources are adequately provisioned, so transaction times consist largely of service times, not queueing times;
- Crucial aspects of workload are statistically non-stationary, i.e., the frequency distributions of key workload characteristics (e.g., mix and load) vary dramatically over time.

Beside the properties presented above, commonly in Web-based applications:

- A single user-transaction with the system is relatively short-lived;
- The server response time vary widely across but not within user-transaction types.

3.3.1 Performance anomaly analysis

To detect if there is a performance anomaly affecting the Web-based application, vector $\boldsymbol{X}$ contains the sequence of the accumulated response server time per user-transaction and vector $\boldsymbol{Y}$ is the number of user-transactions processed in the same interval.

Since in a Web-based application, the transactions response time vary widely across but not within transactions types, $\boldsymbol{X}$ and $\boldsymbol{Y}$ are grouped by a workload mix key. This key defines the mix of user-transactions in an interval.

By using the accumulated response server time, rather than the average or mean values, it is established a relationship between the variables. This way, under normal behavior, when the number of user-transactions processed increases the accumulated response time will also increase, when it decreases the accumulated response time will also decrease.

We use the Spearman's rank correlation analysis to measure the degree of association between the variables ($\boldsymbol{\rho}$). The Spearman's rank correlation analysis consists of:

1. Sorting the vector $\boldsymbol{X}$ in a descending order;
2. Sorting the vector $\boldsymbol{Y}$ in a descending order;
3. Assigning a rank value to the data sorted in 1, so that the highest accumulated response is ranked as 1, the second highest as 2 and so on ($\bar{\boldsymbol{X}}$);
4. Assigning a rank value to the data sorted in 2, so that the highest number of user-transactions is ranked as 1, the second highest as 2 and so on ($\bar{\boldsymbol{Y}}$);
5. Using the Pearson's to measure the correlation between the ranked data ($\bar{\boldsymbol{X}}$, $\bar{\boldsymbol{Y}}$).

The Pearson's correlation [Ryan 2007] returns a measure of the strength of linear dependence between two variables. In our analysis, it is used to measure how much the ranked data ($\bar{\boldsymbol{X}}$, $\bar{\boldsymbol{Y}}$) changes together (covariance) relatively to its dispersion (standard deviation). In practice, if the relationship between the accumulated response time and the number of user-transactions processed holds all the time, then the $\boldsymbol{\rho}$ keeps a strong correlation. A decrease in the $\boldsymbol{\rho}$ indicates a change in the relationship, i.e., a dissociation between the response time and the number of transactions processed. Considering the typical properties of modern distributed applications, presented above, this dissociation is a symptom of performance anomaly.

One of the key mathematical properties of the Spearman's rank correlation analysis is that it is invariant to separate changes in the scale of the variables. So, $\boldsymbol{X}$ or $\boldsymbol{Y}$ can assume values of different magnitudes and that do not affect the correlation coefficient. This allows us to use the same type of analysis for all the user-transactions, and thus, detect when the response time of a transaction is no longer aligned with the number and mix of transactions processed.

The are some situations in the analysis that deserve some attention. In Figure 10, we illustrate the different variations that the variables under analysis may experience. The $\boldsymbol{\rho}$ decreases when one of the scenarios A2, A3, B1, B3, C1 or C2 occurs. The correlation will remain strong when one of the scenarios A1, B2 or C3 is verified. While for most of the changes illustrated, the value of $\boldsymbol{\rho}$ reveals the existence of deviations in the application behavior, the variation represented by the scenario A1 requires special attention. Assuming an increase in the accumulated response time accompanied by an increase in the number of transactions processed seems logical, but it obfuscates scenarios where the response time could be very high and causing a significant impact on the end-users. From the analysis it is also not possible to know if the dissociation is due to a change in the response time, or due to a change in the number of transactions. Likely, from the $\boldsymbol{\rho}$ degree variation it is not possible to quantify the variation observed in the response time.

To address these limitations (to control the variation scenario A1, shown in Figure 10, to know if the variation is due to a response time change and to quantify the response time variation), the *Performance Anomaly Analysis* module is provided with Algorithm 4. For a new interval of analysis $\boldsymbol{p}$, $\boldsymbol{n}$ a number of previous intervals, a maximum admissible response time $\boldsymbol{MART}$ and $\boldsymbol{K}$ a workload mix key, the algorithm measures the difference between the $\boldsymbol{\rho}$ degree observed in a given interval and the maximum value of $\boldsymbol{\rho}$ observed previously. The same is done with the response time. At the end it returns the product between the variance of $\boldsymbol{\rho}$ and the variance of response time.

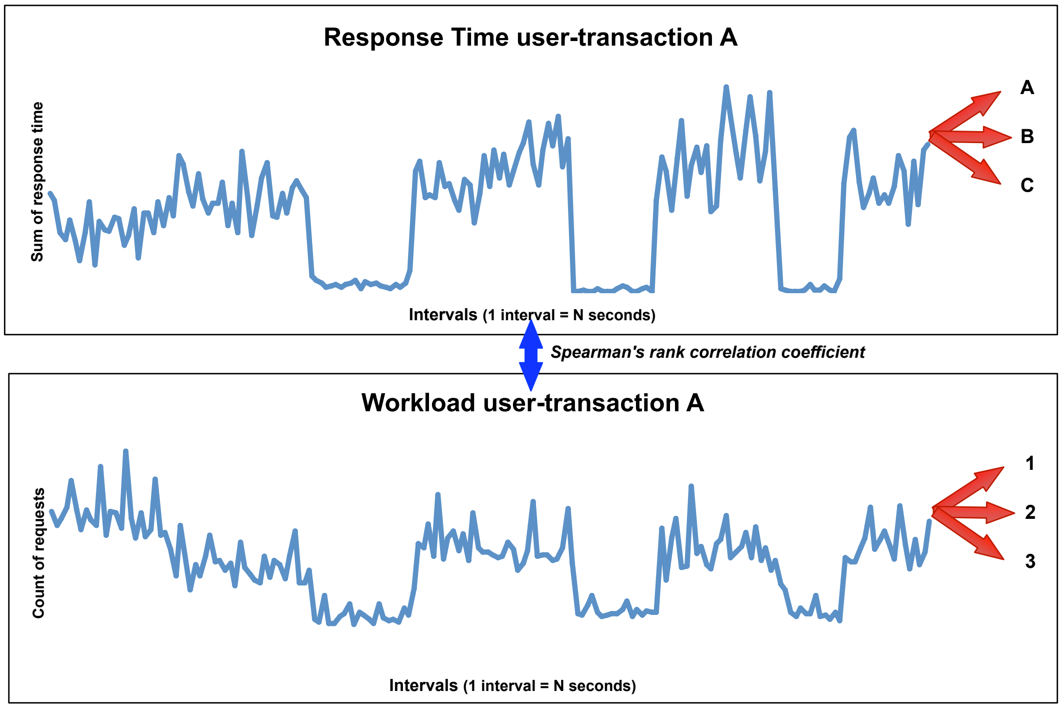


Figure 10: *Spearman's rank correlation - Performance Anomaly Analysis*: Set of possible variations affecting the variables under analysis

Algorithm 4 Performance Anomaly Analysis - degree of dissociation between the response time and the number of user-transactions processed

```
1: procedure PerformanceAnomalyAnalysis(dataset)
2:     for each $UserTransaction$ do
3:         $X[] \leftarrow$ Accumulated Response Time $UserTransaction$ per $p$ and $K$
4:         $Y[] \leftarrow$ Number of User-Transactions $UserTransaction$ per $p$ and $K$
5:         $\rho[p] \leftarrow$ Spearman Rank Correlation$(\bar{X}[], \bar{Y}[])$
6:         $V1 \leftarrow Avg\_RespTime[p] - Avg\_RespTime[p-n:(p-1)/2]$
7:         if $(V1 > V1 + MART)$ then
8:             $V1 \leftarrow MAXINT$
9:         end if
10:        $V2 \leftarrow \rho[p] - Max\_\rho[p-n:(p-1)/2]$
11:        $F[UserTransaction] \leftarrow \sqrt{(V1 * V2)^2}$
12:    end for
13: end procedure
```

The value of F, returned by the algorithm, quantifies the impact of dissociation between the response time and the number of user-transactions processed. It is obtained a value for each user-transaction. Since it is based-on the Spearman's rank correlation coefficient, the value of $\boldsymbol{F}$ can be generalized, allowing to define and control a single threshold value for all the user-transactions and, at the same time, quantify the impact of a performance anomaly per user-transaction. This approach is much more simple and less error prone that defining and controlling a threshold value per user-transaction.

In Table 2 we present a simulation considering the value $\boldsymbol{F}$ returned by the *Performance Anomaly Analysis* module. The simulation considers different response time delays and different variations on the number of user-transactions processed. For the simulation we used 200 pairs of $\boldsymbol{X}$ and $\boldsymbol{Y}$ that obey to relationship already described. Five new pairs of data were added to the end of $\boldsymbol{X}$ and $\boldsymbol{Y}$, representing the variations under test. Each pair contains the data aggregated in time intervals of five seconds.

Table 2: Determining the threshold value that quantifies the impact of a performance anomaly

#User-trans variation (%) → Resp_tm delay (ms) ↓	1	0.5	0	-0.5	-1
+2000	18.902	41.665	89.349	335.178	1794.432
+500	4.518	10.120	22.045	83.972	454.622
+100	0.682	1.708	4.097	16.983	97.340
+50	0.202	0.656	1.854	8.610	52.679
+25	0.006	0.082	0.497	3.346	23.867
0	0.006	0.007	0.006	0.022	0.52

From Table 2 is clear that the $\boldsymbol{F}$ value is higher when the response time increases and the number of user-transactions processed decreases. When the number of user-transactions processed varies but the response time remains stable, the value $\boldsymbol{F}$ is close to zero. For a response time delay equal or higher than 500 milliseconds, the value of $\boldsymbol{F}$ increases independently from the variations observed in the number of user-transactions processed. In the above test we consider only five new intervals of data. However, according to the analysis, a response time delay of just 50 milliseconds originates a value of $\boldsymbol{F}$ higher than 10 after 90 intervals, a deviation of 100 milliseconds originates a value of $\boldsymbol{F}$ higher than 10 after 60 intervals and a deviation of 500 milliseconds is detected at the second interval.

Based-on the results presented in Table 2, it is possible to define a global threshold value, that can be used to highlight the occurrence of a performance anomaly. For example, a value of $\boldsymbol{F}$ higher than 10 can be used to detect performance slowdowns that clearly deviates from the number of user-transactions processed.

3.3.2 Workload contention analysis

A workload variation might occur due to a change in the transaction mix or a change in the load (number of transactions). In a Web-based application, both the mix and the number of transactions are highly variable parameters. The number of users varies over time and the pages they visit are determined by their own interests. These features make the modeling of workload a challenging task.

The analysis provided by the *Performance Anomaly Analysis* module contemplates changes in the workload mix and load. In that module, the data is organized according to a key that corresponds to the mix of transactions. Per mix and number of transactions the data analysis detects delays in response time and verifies if that changes result from variations in response time. One cause of such variation may be a workload contention. In this context, the *Workload Variation Analysis* module, completes the data analysis. It measures the correlation between the observed response time and the number of requests waiting to be processed. To perform this analysis, the *Workload Variation Analysis* module defines $\boldsymbol{X}$ as the accumulated response time in a given time interval and $\boldsymbol{Y}$ as the total number of requests waiting in the application container queue in the same time interval. In Figure 11 we show a logical representation of the raw data ($\boldsymbol{X}$ and $\boldsymbol{Y}$) used by the analysis. The values are grouped using the workload mix key.

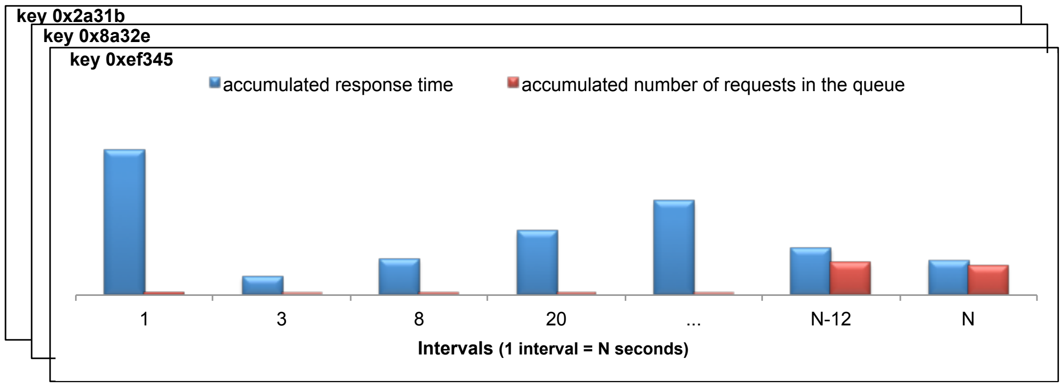


Figure 11: Workload variation analysis: Tracking the Spearman's rank correlation coefficient between the server response time and the number of requests waiting in the application container queue

The *Workload Variation Analysis* module makes use of the Spearman's rank correlation coefficient - $\boldsymbol{\rho}$. Assuming the existence of previous intervals already analyzed, and that the service was performing under normal conditions, the $\boldsymbol{\rho}$ degree is expected to be stable and low across the time intervals. The fact that the $\boldsymbol{\rho}$ degree is low and stable, it means two things: (1) the response time has not suffered delays and (2) there was no significant accumulation of requests in the application server queue waiting to be processed. If the $\boldsymbol{\rho}$ degree is medium or large, then it means there is a workload contention in progress.

In presence of performance anomalies that, according to the analysis provided by the *Workload Variation Analysis* module, are not associated with a workload contention, then the *Anomaly Detector* and *Root-cause Failure Analysis* modules continue the data analysis process. These modules search for system, application server or application changes, in order to pinpoint the cause behind the response time slowdown.

3.3.3 Anomaly detector

After detecting a performance anomaly the *Anomaly Detector* module aims to identify if there is any system or application server change associated with the anomaly.

The *Anomaly Detector* module takes $\boldsymbol{X}$ as the total number of user-transactions processed in an interval and $\boldsymbol{Y}$ as the accumulated value of the parameters collected by the *Sensor* module in the same interval. There is a vector $\boldsymbol{Y}$ for each parameter collected. In Figure 12 we show a logical representation of the raw data ($\boldsymbol{X}$ and $\boldsymbol{Y}$) used by the analysis. The data is grouped using the workload mix key, and it is continuously analyzed as new time intervals become available. As in previous analyzes, the module computes the Spearman's rank correlation coefficient between the variables.

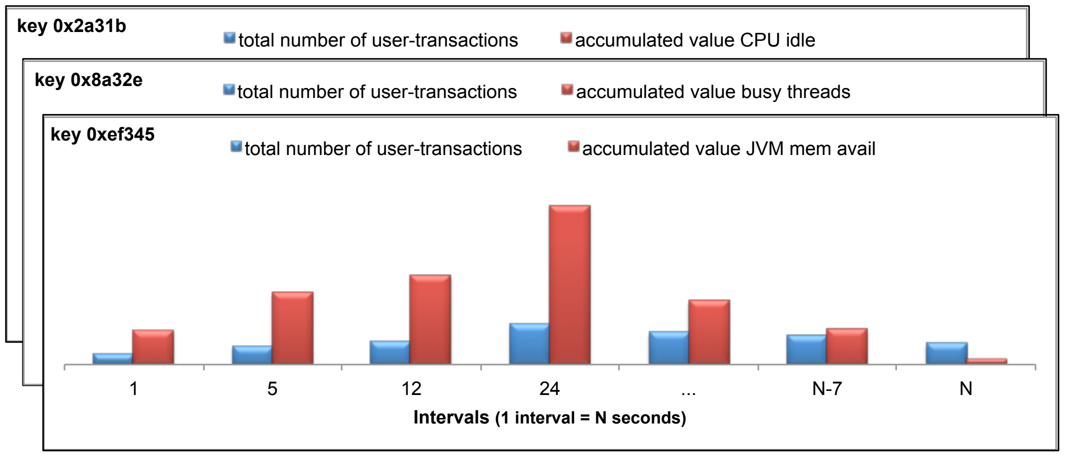


Figure 12: Anomaly detector: Tracking the Spearman's rank correlation coefficient between the total number of user-transactions and the accumulated value of the different system and application server parameters

For example, rather than say how much a parameter P changes according to the number and mix of user-transactions in a given time interval, the *Anomaly Detector* module tells us if the value of a given parameter increases or decreases according to the number of requests processed. From the way we prepare the data for analysis, if the number of requests increases then the accumulated value of each parameter should also increase. If the accumulated value of a given parameter increases, and that is not motivated by an increase in the number of requests, then the $\boldsymbol{\rho}$ degree will decrease, showing that a parameter (or set of parameters) is no longer aligned with the number of requests.

Given the sequence of results, we can tell that the application is facing a performance anomaly and that the anomaly is associated with changes in the parameters identified by the *Anomaly Detector* module analysis. Pinpoint the parameters associated with a performance anomaly is extremely important to select the most appropriate recovery strategy, in order to mitigate the effects of performance anomalies.

3.3.4 Root-cause failure analysis

To verify if a performance anomaly is associated with an application change or to a remote service change, the variable $\boldsymbol{X}$ is defined as the frequency distribution of the transactions response time and $\boldsymbol{Y}$ assumes the response time frequency distribution of the calls belonging to the transaction call-path. The correlation between the variables is determined using the Spearman's rank correlation coefficient. The *Root-cause Failure Analysis* module analyzes only the user-transactions that have reported a performance anomaly.

Due to the non-Gaussian nature of the data, we used the Doane's formula (Eq. 3.3.2) to determine the number of data bins to discretize the variables $\boldsymbol{X}$ and $\boldsymbol{Y}$. To determine the number of bins, it is necessary to calculate the kurtosis of the distribution $\bar{\boldsymbol{a}}$ (measure related to the shape "peakedness" of the data distribution) and provide the number of observations under analysis ($\boldsymbol{n}$).

$$\boldsymbol{Nbins = 1 + \log_e n + \log_e (1 + \bar{a}\sqrt{\frac{n}{6}})} \tag{3.3.2}$$

The number of bins returned by Eq. 3.3.2 is used to discretize the variables $\boldsymbol{X}$ and $\boldsymbol{Y}$. While one of the variables indicates the frequency of the response time of a user-transaction, the other contains the response time frequency for the calls belonging to the user-transaction under analysis. From the Spearman's rank correlation, it is expected that the association between the variables is stable, unless there are one, or more calls, which response time has changed together with a change in the response time of the user-transaction. In this situation, the $\boldsymbol{\rho}$ degree increases, highlighting the calls potentially associated with the performance slowdown.

The identification of the calls associated with a delay in response time allows, for example, to consult the change management database to see if the problem is related to any recent application change. It also allows to find patterns in the methods that are accusing problems. For example, several calls about queries in the database can be linked to problems in the database tier, calls regarding the interaction with Web Services can highlight problems with a Web Service.

3.4 Recovery: *SHõWA* planning and executing recovery

The self-healing actions should move the system towards a safe state. The recovery service included in the *SHõWA* framework is provided by three modules. When an anomaly is detected and localized, the *Recovery Planner* module selects a recovery procedure. The *Executor* module executes the recovery procedure, interacting with the *Effector* module.

3.4.1 Recovery procedures

The *Recovery Planner* module contains the recovery procedures, ready to be activated when an anomaly is detected. This module is activated by the *Workload Variation Analysis* after it detects a workload contention or by the *Anomaly Detector* and *Root-cause Failure Analysis* modules after a performance anomaly is detected and pinpointed.

In this module there are a set of actions, that can be grouped to create a recovery procedure. These actions indicate for example, what to do to inform a load balancer to stop sending requests to a given server or what command should be executed to stop the application server. These actions can be extended.

The recovery process that is currently implemented is procedure-based and defined by a human operator. The operator identifies the recovery actions to be included in the recovery-procedure and sets the rules in which the procedure is activated. The operator can define more than one recovery-procedure for the same rule. To that end it should identify the priority and the atomicity of the recovery actions. The priority corresponds to a numerical value which describes the expected effect after an action is performed. For example, if it is expectable that a server restart provides better recovery results than a simple application restart, then it must have a higher priority number. The atomicity is a boolean value which defines if the recovery action can be interrupted anytime or not. This value is important to prevent service inconsistencies motivated by the recovery process (e.g., force a restart while the application is being downgraded may lead to an inconsistent application version or compromise the normal application startup).

The recovery procedures remain under the *Recovery Planner* module and they are selected according to the type of anomaly detected. Once selected, the recovery procedures are executed and controlled automatically.

Although not currently implemented, the process is thought to evolve in order to become more autonomous. After the initial phase, the system can acquire features that allow it to learn and decide autonomously the set of recovery actions to execute.

For example, the historical recovery information, the system capabilities, the system state and the environment state can be used to compute utility functions. The recovery actions can then be selected based-on an utility value like: perform the recovery action that is the fastest; perform the recovery actions that cause fewer errors.

3.4.2 Executing recovery

The recovery procedure is selected by the *Recovery Planner* module and executed by the *Executor* module. The *Executor* module is responsible for the interaction with the *Effector* module, passing to it the sequence of recovery actions and controlling its execution. The communication between the nodes is made through a client-server program. The client sends the actions to be performed to the server and receives feedback about its implementation. The *Executor* module knows how many systems are being monitored and controls how many of these are in recovery. This control process is done to avoid service failures motivated by the execution of simultaneous recovery processes. This module also controls the execution of multiple recovery requests to a given node. By default, when a node is under recovery it does not accept new recovery actions. Exceptions are allowed when two conditions are simultaneously meet: (1) the new repair action has a higher recovery priority when compared to the recovery action in progress; (2) the repair action in execution can be interrupted anytime (atomic bit is set to false).

If for some reason a recovery procedure fails or it takes more than a given threshold to be executed, then a different recovery procedure can be taken. The hierarchy between recovery procedures is useful when we want to maximize the recovery process in terms of a metric. For example, start with a recovery procedure that reduces the recovery time (e.g. soft restart of the application). If it fails for some reason, then a recovery procedure with more guarantees of success (e.g. server restart) is initiated, sacrificing the recovery time.

3.5 Conclusions

In this chapter we presented the self-healing framework targeted for Web-based applications (*SHõWA*). It is composed by different building blocks that cooperate each other to monitor Web-based applications, analyze the data to detect anomalies, plan and execute the necessary recovery actions.

The *SHõWA* framework enables two modes of data gathering and makes use of statistical analysis to verify if a performance variation is due to a workload contention or if it is due to a performance anomaly. When a performance anomaly is detected, the analysis checks for changes in the system, application server or application that are correlated with the performance slowdown. Through this analysis we can more easily determine if the cause behind the performance anomaly is related with a resource consumption, with any recent application upgrade or with a remote service accessed by the application. When *SHõWA* detects a performance anomaly, or a workload contention, it automatically starts the recovery process. For this purpose, it selects a recovery procedure and performs the recovery actions, over the managed resource.

Considering that we are proposing an autonomous system to cope with performance anomalies, it is important to take care the performance impact introduced by the proposed system itself. The major concern, regarding the performance impact introduced by the *SHõWA* framework, is with the data gathering process. Profile the application at runtime may introduce a severe performance degradation on the system. Adaptive and selective algorithms are proposed to reduce the performance impact induced with application-level monitoring. These algorithms adapt the frequency at which the application calls are profiled and select, dynamically, the list of calls to be monitored.

The approach we used to analyze the data, in order to detect and pinpoint anomalies covered a substantial part of this chapter. We adopted the Spearman's rank correlation for all the modules that involve data analysis. This correlation analysis fits the way the data is prepared and presents several advantages in terms of analysis and applicability. *SHõWA* allows to detect performance anomalies without the need to specify application-specific thresholds. It can be adopted for any Web-based application with no need to set anything. The only thing that is required is to ensure that the *Sensor* module is loaded at startup of the application server. In addition to detecting anomalies, *SHõWA* allows to pinpoint the anomaly. The level of detail about the origin of the anomaly ranges from situations in which the application is affected by a resource contention, to situations where the response time is affected by an upgrade of the application or changes in the operation of remote services. The ability to detect problems at runtime, the ability to pinpoint and recover from the anomaly scenarios, the ability to profile Web-based applications and the ability to be used across Web-based applications, make the *SHõWA* a key tool to increase the availability, reliability, maintainability and performance of Web-based applications.

In the next chapters we present and evaluate, by means of experimental analysis, the operations performed by the *SHõWA* modules. We start by presenting the advantages of using application-level monitoring comparatively to the adoption of traditional monitoring system. Then we present a study about the performance penalty induced by the *Sensor* module, considering the adoption of selective and adaptive monitoring algorithms. The detection and pinpointing of performance anomalies analysis considers two different Web-based applications. Finally, we present results about the recovery and system redimensioning achieved with *SHõWA*. As we go through the different topics - monitoring, detection and location of anomalies, recovery and system redimensioning - we present the corresponding state of the art.

Chapter 4
Monitoring Techniques for Web-based Applications

KEY POINTS

- ◇ A monitoring system is defined as the systematic process of collecting data and analyzing it to produce useful information for decisions making.
- ◇ The combination of different monitoring systems (e.g, system-level monitoring, log analysis, end-to-end monitoring and application-level monitoring) is used to monitor Web-based applications at runtime and provide IT staff with a means of visualizing the service status and to drill down the problem analysis. While some monitoring systems keeps track of several system parameters (e.g., CPU usage, memory utilization, number of open files) or search actively for error statements in the log files, others attempt to capture the end-users perspective about the service.
- ◇ The coverage and the detection latency achieved with these systems are crucial aspects for the availability and performance of Web-based applications;
- ◇ In this chapter we study the detection latency and coverage of different monitoring systems commonly used to monitor Web-based applications.

There are numerous reports focusing severe business impacts motivated by performance slowdowns or unavailability scenarios. Even considering the efforts to optimize the applications, eliminate errors and the adoption of high-availability mechanisms, applications continue to be affected by problems that occur at runtime. These unexpected problems jeopardize the normal functioning of the applications. In this context, runtime monitoring systems are extremely important to identify and alert for situations that will probably evolve into a failure.

In Web-based applications is common the adoption of different monitoring techniques to provide IT staff with a means of visualizing the service status and to quickly analyze the problems. In this chapter we describe the different aspects and types of a monitoring system. An experimental study is conducted to evaluate the coverage, detection latency and the number of end-users affected before the anomaly detection. The study combines different monitoring systems and evaluates their ability to detect different types of anomalies. The monitoring service included in the *SHõWA* framework is considered in the analysis.

4.1 Monitoring systems

A monitoring system is defined as the systematic process of collecting and analyzing data to produce useful information for decisions making. It can be described as a hierarchy of three basic categories: technical, functional, and business process monitoring. A technical monitoring system focuses on the state of individual components, such as network components and critical servers, to find out what is wrong and how to fix it. A functional monitoring system looks at the functionality of the system. It is usually performed by tracking and modeling the operations on a system and report metrics of interest. It answers the question whether there is a problem in the application or system or not, but it does not provide information about how to solve problems and what is the impact it causes on the business. The business process monitoring systems improves the previous monitoring systems. It looks at workflow and related IT monitoring tools to ensure that the service meets the expectations of both the end-users and the business.

According to Schroeder, in [Schroeder 1995], a monitoring system can be classified according to its purpose: dependability, performance enhancement, correctness, security, debugging or performance evaluation. *Dependability* monitoring assesses the fault-tolerance and safety mechanisms. *Performance enhancement* includes monitoring techniques to enhance dynamic configuration and tuning. *Correctness checking* ensures application testing to detect errors. *Security* monitoring attempts to detect security violations. *Debugging and testing* employs monitoring techniques to extract data values from an application being tested. *Performance evaluation* consists on extracting data to assess the system performance.

In general a monitoring system gathers data from the systems under monitoring and compares it with baseline values, expected behaviors or error conditions. If a deviation is found then an event is triggered by the monitoring system. Some monitoring system allow to set predetermined actions that are activated in reply to an event or set of events. Each of the entities - monitoring, events, actions - is surrounded from complex implementation details. They may adopt techniques more or less intrusive, perform active or passive monitoring, observe different aspects of the system under monitoring and be implemented aside or apart of the application of system.

4.1.1 Intrusive versus nonintrusive monitoring

An important characteristic of a monitoring system is the level of interference it imposes in the application or system under monitoring. A system is said to be intrusive when it competes for the same computational resources used by the application under monitoring. If the computational resources used by the system under monitoring are not consumed by the monitoring system, then the monitoring is considered nonintrusive.

The probability of a monitoring system being intrusive is very high. The sensors used for the data gathering process are typically installed in the system under monitoring, sharing the computing resources with the system under monitoring. Even when this does not happen, there is always the need to inquire, on a regular basis, the application or the system under monitoring to gather its state.

Taking this into consideration, it is essential to measure the performance overhead induced by the monitoring system, otherwise it could be the monitoring service itself the reason for some of the problems affecting the application behavior.

4.1.2 Synthetic versus passive monitoring

Synthetic monitoring, also known as active monitoring, is done using an emulation tool that simulates the actions that a end-user would take while interacting with the system or application. The simulated actions are continuously monitored to assess the functionality, availability and the response time of the service. This is useful to identify problems and determine if the service is experiencing some type of unexpected behavior.

Synthetic testing is useful for measuring availability and response time but it does not monitor or capture actual end-user interactions. The monitoring coverage also tends to be a problem. Since the synthetic tests must be prepared in advance, it is not feasible to measure the performance for every navigational path that an end-user might take.

Passive monitoring consists on collecting and analyzing the system or application metrics while it is being used by the real end-users. This type of monitoring is also known as real user monitoring, and it is considered important to determine if users are being served quickly, error free and if not which part of a business process is failing. Contrary to the active monitoring, passive monitoring can only detect problems after they have already occurred and affected the end-users. For this reason it is mainly used to timely detect and quickly report issues or potential problems allowing IT staff to react promptly to prevent the aggravation of the anomaly impacts.

Is also frequent to combine active and passive monitoring. This combination provides visibility on application health during the peak hours as well during the hours when the application utilization is low.

4.1.3 Black-box, white-box or gray-box monitoring

Black-box, white-box and gray-box are terms used in the software testing field. They refer to the different forms how the applications can be tested with respect to the knowledge of the application source code or internal structure. These concepts are also applied to monitoring systems, particularly, to the way the sensor entity performs to gather the parameters state of the system under monitoring.

A black-box monitoring is a type of a system that do not rely on the application source code or on its internal structure. These systems are only aware of what the application is supposed to do, i.e., its functional aspects, without knowing how the result was produced. If the input or output is different from what is expected, then it is interpreted as an anomaly of the system and events are triggered reporting the anomaly. This makes black-box monitoring suitable for monitoring legacy systems or applications, which cannot be enhanced with monitoring support for various reasons. It can be used for example to verify if the server responds, if the application is working or if it returns the expected contents. To do so, it observes the log files reports, controls the levels of interaction between components, collects the output results or sends predefined requests to the service and checks the responses. From the functional deviations it is possible to infer the status of the service at some extent.

White-box monitoring is the opposite of black-box monitoring. It assumes the knowledge of the internal structure of the application being monitored and has mechanisms to gather data related with its execution. Sensors which adopt white-box monitoring are generally implemented as embedded code in the application data space, i.e., the application or system under monitoring is instrumented with hooks which collect metrics from the system being monitored.

Gray-box monitoring is a combination of black-box and white-box monitoring. The aim is to observe if the service is performing correctly according to the expected results while collecting the internal state of the application or system. This way, the vision about the system or application to pinpoint the components, that may be associated with a failure or potential failure, is improved.

4.1.4 Instrumentation

The white-box or gray-box monitoring requires the instrumentation of the system or application under monitoring. The instrumentation is the process of adding code instructions to monitor specific components in a system. It can be done by means of static or dynamic instrumentation.

The static instrumentation is previously defined by the programmers before the compilation and/or execution. The dynamic instrumentation can be redefined at runtime, allowing the monitoring system to make decisions about when and how to instrument a particular process or component, limiting the instrumentation overhead when suitable.

The problem of static instrumentation is that it is difficult to implement the instrumentation hooks without affecting the readability and complexity of the source code. The dynamic instrumentation simplifies the process, by allowing the separation of cross-cutting concerns.

The instrumentation is further divided into source code and binary instrumentation. Source-code instrumentation is usually addressed by creating some kind of instrumentation programming interface which the programmer may use within the application source code to capture entry/exit points of functions or data changes. Binary, or byte-code instrumentation, does not require manual/direct changes, or previously knowledge about the original application source code. In this case, the instrumentation hooks operate (statically or dynamically) at the virtual execution environment level.

4.2 Monitoring Web-based Applications: Common Approaches

Modern component-based applications are complex. They have tens or even hundreds of components that work together to achieve a coherent goal. It is becoming more and more vital that these applications are continuously monitored at runtime, being possible to detect changes in their behavior, localize the faulty components and repair the existing issues.

There are different monitoring techniques that are commonly adopted for Web-based applications. These techniques involve system-level, container-level, end-to-end, log analysis and application-level monitoring systems. They are being adopted (very often in combination), to ensure that if the service starts behaving differently from the expected, it will be promptly detected and reported.

In this section we present an overview about different monitoring techniques frequently used to detect anomalies in Web-based applications.

4.2.1 System-level monitoring

System-level monitoring keeps track of several system parameters, like CPU usage, memory utilization, number of open files or the availability of a service. It is commonly based-on a central system that collects the data (usually via SNMP or TCP services) from the systems under monitoring. For every piece of input data provided, the measured parameter value is compared against threshold values. The thresholds are set so that alert messages are triggered when the thresholds are reached, giving administrators time to correct a problem.

The HP Operations Manager [HP Operations Manager], IBM Tivoli Monitoring [IBM Tivoli Monitoring], Nagios [Nagios] and Zabbix [Zabbix] are some examples of tools widely used by the industry. These tools provide active monitoring of systems/services and provide live reports and alert messages to the system administrators.

4.2.2 Container-level monitoring

Application containers, like Web containers, specify a runtime environment for the applications. They usually include monitoring services, provide different levels of analysis and enable visual reporting and logging facilities. These monitoring services gather details of the application server parameters (e.g., JVM memory, thread pools, instanced objects, database connection pools, session details). This data is usually used by the IT staff to infer about the application server footprint and to become aware of adjustments to improve its performance.

There are some tools, like Applications Manager [Applications Manager] and AppInternals Xpert [AppInternals Xpert], that extract and consolidate the data from these monitoring facilities for detecting and diagnosing problems. These tools proactively monitors servers, application servers, database systems and Web Services to help IT administrators manage their resources effectively and ensure that the revenue achieved through business-critical applications meet the expectations.

4.2.3 End-to-end monitoring

End-to-end monitoring attempts to capture the users perspective about the service, i.e., know if users are having issues while accessing the application. It is an application monitoring solution that behaves like a synthetic user. It aims to collect data to help IT management staff and business managers to identify and resolve issues before they impact users and ongoing business operations.

Among many end-to-end monitoring examples, we found leading solutions like Gomez [Gomez], eXternalTest [eXternalTest] and Site24x7 [Site 24x7]. Most of these tools are based-on monitoring agents geographically dispersed around the globe that periodically polls the Web-applications by replaying predefined user-transactions. Each agent gathers details about the availability, response time, protocols status and content errors. The data collected by the agents is then consolidated and used to report anomalies and isolate its origin.

Other end-to-end monitoring approaches are browser instrumentation and embed instrumentation code with web pages to record access times and report statistics back to the server. These approaches rely on client-side testing and performance analysis. The former is based-on Web browsers specifically provided with measurement analysis tools. The e-Valid [e-Valid] and Page Detailer from IBM [Hellerstein 1999] are two examples of tools adopting this approach. Since 2010, a W3C Web Performance Working Group [WPWG] is actively working on standards to store and retrieve performance metrics from web pages considering the end-users perspective. A rich API and user agent features is already made available to be integrated within the Web browsers and collect timing information related to the navigation of the Web documents. The second approach consists in embedding HTML tags into the existing target web pages to collect the server processing time for a request.

4.2.4 Log-analysis

Log files contains data about the system and applications usage, warning and error messages. These records are quite important for anomaly inference. Popular log analysis tools, like Swatch [Swatch] and AWStats [AWStats], allows to efficiently scan ASCII-based log files and notify sysadmins when some predefined expressions are logged.

An interesting work based-on log file analysis is presented in [Bodíc 2005]. The authors have used the historical HTTP log files to model the access patterns of each web page. The model was integrated into a graphical visualization tool that compares the number of hits observed with the expected access pattern and alerts the sysadmins when a difference is observed.

4.2.5 Application-level monitoring

To cover a larger set of potential problems the techniques, described in this section, are commonly adopted in combination. The advantages of combining system-level, container-level, end-to-end and log analysis monitoring are indubitable, however it still is a bit of challenge to establish an association between the alert messages and the quality of service experienced by the end-users. End-to-end monitoring is useful to give some clues about the end-users experience, however, due to the timer polling-based monitoring, it does not guarantee that IT staff will be able to timely detect and recovery from anomalies. It also does not contribute to explain what is the problem and what needs to be fixed.

Application-level monitoring, may reduce the uncertainty about some of these issues. Application-level monitoring enables the monitoring of the functionality of the applications rather than just check whether the application is available or not. It also allows IT staff to focus the attention to the problem, reducing the uncertainty about some monitoring messages. For example, if there is an alert message provided by a system-level monitoring tool indicating high CPU consumption, is quite difficult to know if the application is being affected, what recovery action should be performed and which outcome is expected after the recovery. Otherwise, if application-level monitoring is used, then it is possible to detect degradations at the application level and then establish an association between the application metrics and the system parameters to pinpoint the cause of the observed behavior and to conclude, in general terms, what recovery actions should be performed.

A sound work that makes use of application-level monitoring is the Pinpoint project [Chen 2002]. This project relies on a modified version of the application server to record the request execution paths and build a model of the expected interactions between components. At runtime, differences between the expected and observed interactions are regarded as failures and recovery procedures are initiated. Regarding to the experimental results, authors have concluded that Pinpoint has a very high accuracy and precision for single-component faults, but not a so good accuracy when there are three or more component-faults. The authors have also noticed that Pinpoint introduces an overhead of about 8.4%. Another interesting work is proposed by Cherkasova et al., in [Cherkasova 2008] and [Mi 2008]. The authors isolate workload variations from performance anomalies by combining a regression-based transaction model and an application signature. This is used to detect which of the transactions is associated with an unexpected CPU consumption. The experimental study conducted by the authors has revealed the ability of the approach to detect anomalies and analyze performance changes in the application behavior. NEC researchers presented a work in [Jiang 2006], where they use commonly available monitoring data to model the flow intensity between the components, considering the volume of user requests. Flow intensities that hold during the time are regarded as invariants of the underlying system, and abrupt changes in those invariants are used to detect problems. Aguilera et al., in [Aguilera 2003], also adopted a black-box monitoring approach to capture and analyze the messages flow and debug performance bottlenecks between components.

The *Sensor* module included in the *SHõWA* framework enables application-level monitoring. It makes use of Aspect-Oriented Programming (AOP) [Kiczales 1997], and do not requires manual changes to the application source code. It collects data with two different levels of granularity: *user-transaction* level monitoring and *profiling* level monitoring. In the *user-transaction* level it intercepts and measures the server response time of the user-transactions and gathers different system and application server parameters (e.g., CPU load, JVM heap memory, number of open files, number of running threads). In the *profiling* level, it intercepts and records the execution time of the calls involved in the user-transactions call-path. The data collected at the *user-transaction* level is used to detect slow user-transactions and identify if there is a system or application server parameter change that is associated with the performance anomaly. The application *profiling* data is used to pinpoint the calls/components associated with a performance anomaly. The data gathered by the *Sensor* module is sent to the *Data Preparation* module. The *Data Preparation* module aggregates the data in intervals of analysis, and characterizes the mix of user-transactions in each interval. The data is then submitted to statistical analysis to check for workload variations or performance anomalies.

In the next section we show the results of an experimental study, carried out to verify the coverage, detection latency, and effectiveness of the different monitoring systems presented in this chapter.

4.3 Monitoring Web-based Applications: A coverage, detection latency and effectiveness study

In [Oppenheimer 2003] authors found that 65% of the reported user-visible failures in three major internet sites can be mitigated or avoided by earlier detection, and in [Kiciman 2005] authors estimate that 75% of the time to recover from application-level failures is spent just detecting and localizing them.

Taking the previous statements into account, we decided to conduct an experimental study considering different monitoring systems to evaluate their coverage, detection latency and effectiveness. The effectiveness analysis refers to the number of errors and requests affected by delays in the response time, upon detection of the anomaly. This analysis allow to quantify the importance of early detection of anomalies.

In detail, the experimental analysis conducted addresses the following questions:

- *Which sort of anomalies are more likely to escape the different monitoring techniques?;*
- *Which monitoring technique provides lower anomaly detection latency?;*
- *Which monitoring technique minimizes the number of visible errors and the number of slow user-transactions?;*
- *What is the performance impact that each monitoring and anomaly detection technique induces in the system?;*
- *What are the advantages/disadvantages of using application-level monitoring for anomaly detection?*

To answer these questions we have prepared a test environment composed by a Web-based application and different monitoring systems: system-level, container-level, end-to-end and application-level monitoring. Each monitoring was conveniently prepared to detect anomalies and provide alert notifications to the sysadmins. The experimental study consists on the injection of different types of anomalies and on the evaluation of the coverage and detection latency achieved by each monitoring system. A relationship between the detection latency, the number of visible errors and the number of users who have experienced slow response times is also provided.

4.3.1 Experimental setup

The experimental environment comprises a three-tier infrastructure running a benchmark application, different monitoring systems (system-level, end-to-end, log analyzer and application-level monitoring), dedicated servers to store the monitoring data, servers carrying out the analysis process for anomaly detection and multiple emulated clients performing requests. The setup of the monitoring parameters and the anomaly scenarios adopted in this study are described in this section.

4.3.1.1 Monitoring techniques adopted

In the category of system-level monitoring we decided to use Zabbix [Zabbix]. Zabbix is public-available and commercially supported. We adopted a client-server configuration. The clients configuration was extended to allow to capture some parameters provided by the application server (e.g., JVM memory used, number of busy threads, HTTP refusals, HTTP errors), and also some database server parameters (e.g, number of DB connections used). These parameters are collected by the Zabbix server according to a predefined polling interval. The alert messages are triggered in compliance with the rules defined in the Zabbix configuration. More details about the parameters setup might be seen in Table 3.

Container-level monitoring: the monitoring services available in the Oracle Glassfish Server are enabled by default but the monitoring levels are set to *"Off"*. To access useful monitoring data we set the monitoring levels to *"High"*. Then we prepared the Zabbix agent to request some parameters through the REST API provided by Glassfish. These parameters are collected by the Zabbix server according to a predefined polling interval and the alert messages are triggered when the predefined thresholds are violated (see Table 3).

For log-analysis we adopted Swatch [Swatch]. Swatch is a Perl script that was initially devised for actively monitoring log files, as they are being written, and take actions when the predefined expressions are found.

To avoid network effects, and let us to focus on the end-to-end monitoring results, we decided to implement a small agent that runs on the same network as the application server. The agent consists on a shell script program that periodically invokes a JMeter [Apache JMeter] script. The JMeter script replays two different user-sessions, that were previously recorded while a user was accessing the Web application. The sessions are based-on the TPC-W [Smith 2001] browsing and shopping traffic-mixes.

As application-level monitoring we used the *SHõWA* framework. The framework makes use of a *Sensor* module that is installed in the application container. The *Sensor* module is implemented in aspectJ [Laddad 2009] and attached to the application server using the Load-Time Weaving features. The parameters collected by the *Sensor* module are stored in a remote database and prepared by the *Data Preparation* module. This module aggregates the data in time intervals and determines the mix of user-transactions contained in each interval. As the monitoring data becomes available, the *SHõWA Workload Variation Analysis* and the *Performance Anomaly Analysis* modules, measure the Spearman's rank correlation correlation degree.

The *Performance Anomaly Analysis* module measures the association between the response time and the number and mix of user-transactions. A decrease in the correlation degree means that the response time is not associated with the number of requests. The decrease in the degree of correlation is used to, according to the rationale of the data analysis presented in Chapter 3, quantify the impact of dissociation in terms of response time latency and number of user-transactions processed per interval. The impact of dissociation is general to all transactions and remains valid across Web applications and user-transactions. An anomaly is detected when the impact of dissociation exceeds a certain value.

The *Workload Variation Analysis* module measures the association between the accumulated response time in a given interval and the total number of requests waiting in the application container queue in the same time period. When there is a strong association between these two parameters, it means that the response time is being influenced by the time that the requests are waiting in the server queue.

4.3.1.2 Testbed

The testbed is illustrated in Figure 13. The System Under Test consists of an application server (Oracle Glassfish Server v3.1 [Oracle Glassfish Server]) running a benchmark application (TPC-W). The user requests are simulated using several remote browser emulators (RBEs).

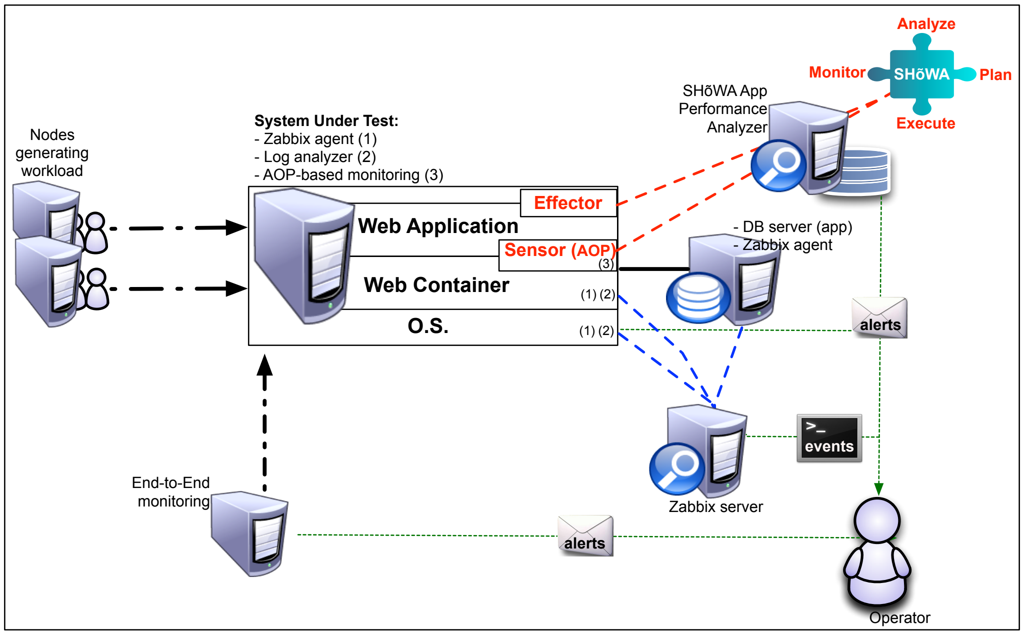


Figure 13: Test environment

The TPC-W benchmark [Smith 2001] simulates the activities of a retail store website. It defines 14 user-transactions, which are classified as browsing, shopping and ordering types. The TPC-W uses a MySQL database as a storage backend. The benchmark allows the execution with different concurrent users, three different traffic mixes (browsing, shopping and ordering), different session length, user think-times and ramp-up/ramp-down periods.

As stated by the specification, the number of emulated browsers is kept constant during the experiment. However, since real e-commerce applications are characterized by dynamic workloads we created a workload that runs for 5400 seconds. In this workload the type of traffic mix, the number of emulated-users and the duration of each period changes periodically. For the purpose of this study, and to avoid deterministic results, we have generated a new workload per group of anomaly scenario tested.

The application-level monitoring technique combines the *SHõWA Sensor* module, installed on the application server, and the *Data Preparation*, *Workload Variation Analysis*, *Performance Anomaly Analysis*, *Anomaly Detector* and *Root-cause Failure Analysis* modules, installed on the "*SHõWA* Application Performance Analyzer" server. The AOP *Sensor* module measures the response time of each one of the 14 user-transactions, and gathers different system and application servers parameters.

The data is stored in the "*SHõWA* Application Performance Analyzer" server and aggregated in time intervals/epochs of five seconds. The data analysis is performed on this server by the R [R Development Core Team 2010] tool, provided with the Spearman's rank correlation packages. In this study, when a workload variation or performance anomaly is detected, it is immediately triggered to the system operator.

The Zabbix agent and the Swatch log analyzer tools are installed and configured in the application server. While the Zabbix agent interacts with the Zabbix server, Swatch triggers the alert messages directly to the system operator. The end-to-end monitoring node contains a JMeter 2.4 script. This script replays two predefined user-transactions (browsing and ordering). It is executed periodically to analyze the response time, protocols errors and content returned by the application server. In compliance with the predefined rules, if an anomaly is detected then an alert message is immediately sent to the system operator.

The test environment is composed by several virtual machines and two dedicated server machines. Each virtual machine instance is running on a different physical server. During the tests phase only the necessary virtual machines were available. The four nodes responsible for generating workload include a 3GHz Intel Pentium CPU and 1GB of RAM. The virtual machine "*SHõWA* Application Performance Analyzer" includes two 2.66GHz Intel Xeon CPUs and 2GB of RAM. The virtual machine where the Zabbix is installed includes two 3.4GHz Intel i7 CPU cores and 2GB of RAM. The end-to-end application monitoring node includes two 3.4GHz Intel i7 CPU cores and 1GB of RAM. We used the XEN virtualization technology.

The application server and the database server runs on two dedicated server machines. The application server machine includes an i7 2600 3.4 GHz CPU, with four cores and 16GB of RAM. The database server machine includes four 2.66GHz Intel Xeon CPUs and 2GB of RAM. All the nodes run Linux with 2.6 kernel version and are interconnect through a 100 Mbps Ethernet LAN. The operating system, the TCP-IP and the application server parameters were adjusted to the recommended settings.

4.3.1.3 Parameters setup

In this subsection we describe how we configured each one of the parameters to be monitored. Table 3 summarizes the set of items and triggers we have chosen.

Table 3: Zabbix parameters: items and trigger rules

Item	Rule set
CPU idle (%)	CPU idle < 10%
Processor load	CPU load avg 1min > 8
Running processes	# of processes > 1200
Host status	Application and DB server availability: 0 - Unavailable; 1 - Available
Memory (%)	Free memory < 5% total memory
Services	Glassfish, MySQL and SSH availability: 0 - Unavailable; 1 - Available
Disk space (%)	Free disk space < 5% of total disk
Inodes (%)	Free number of inodes < 5% total filesystem inodes
Swap space (%)	Free swap < 5% of total swap memory
File Descriptors (%)	Available file descriptors < 5% total file descriptors
JVM memory (%)	Glassfish memory used > 90% JVM heap
Busy Threads (%)	# Glassfish busy threads > 80% total threads
HTTP timeouts	HTTP timeouts > 0 (Glassfish logs)
HTTP refusals	HTTP refusals > 0 (Glassfish logs)
HTTP error 4XX	HTTP errors 4xx > 0 (Glassfish logs)
HTTP error 5XX	HTTP errors 5xx > 0 (Glassfish logs)
DB connections	Available DB connections < 10

The first ten rows in Table 3 represent resources of the operating system. They are predefined in the Zabbix templates and we just have fine-tuned the polling interval and the trigger rule. The other rows refers to the application and database server parameters. For these, the local Zabbix agent was configured to request data through the REST API (provided by Glassfish) and through the `mysqladmin` command-line interface tool. The parameters were set to be monitored by Zabbix in every 10 seconds.

The log-analysis provided by the Swatch daemon is quite different. It actively monitors the Glassfish log files, and watches for the occurrence of expressions like *"SEVERE"*, *"OutOfMemory"*, *"Exception"*, *"Timeout"* or *"Error"*. If any of the configured expressions is found then an alert message is triggered to the system operator.

The end-to-end monitoring was configured to run in every 60 seconds. It checks for the HTTP status code, measures the transactions response time and scans the HTTP file, to check if it contains some of the given expressions (e.g., *"error"*, *"exception"*). If a HTTP error code or a predefined expression is found, or if the response time delay is superior to 2 seconds, then an alert message is sent to the system operator.

SHõWA triggers a performance anomaly message when the impact of dissociation observed is superior to 10. A workload variation is triggered when the change in the correlation level is greater than 0.1 degrees. According to the data analysis presented in Chapter 3, an impact of dissociation greater than 10, indicates situations like: a response time delay of 500 millisecond or a response time delay of 100 milliseconds while the number user-transactions processed decreased at least 50%.

4.3.1.4 Anomaly scenarios injected

With the test environment ready we defined a set of representative anomaly scenarios. These scenarios combine fault-load, resource contention, workload variation and application changes. They were chosen according to the common causes of failures in Web applications, described in [Pertet 2005].

Rather than implement all the anomaly scenarios from scratch we decided to use some available tools. We used JAFL [Silva 2008] to inject some faults. JAFL is implemented in Java and allows the simulation of some of the most common types of failures in Web applications. We used the following modules:

- **Memory Consumption:** used to consume a given amount of the Glassfish JVM memory along the time. If a stop criteria is not used, then all the available memory is consumed and an *"OutOfMemory"* error is throw;
- **Thread Consumption:** this module allows to consume a certain number of threads along the time;
- **File Handler Consumption:** with this module we can consume file descriptors associated with the Glassfish process running on top of the operating system;
- **Database Connections Consumption:** This module allows to consume database connections. When the maximum number of database connections is reached, no one can access to the database;
- **Database Table Lock:** this module locks periodically a database table. The table remains locked for a given amount of time.

To impose load on the system we used the `stress` [Stress] tool. This is a very simple tool and it was used to evaluate how the monitoring techniques perform considering a gradual **CPU consumption** and a scenario of **IO load** (read/write operations to the disk). The `dd` Unix operating system command was also used to consume the filesystem space, until the TPC-W database stops due to an out-of-disk space error (**Storage Consumption**).

The *httperf* [Mosberger 1998] tool provides a flexible facility for generating various HTTP workloads and measure the Web server performance. We used it to stress the application server (**Requests Overload**) and observe which monitoring system is able to detect the overload scenario before the maximum nominal capacity of the application server is reached.

Scenarios of **Database Load** and **Application Change** were also considered in this experimental study. The **Database Load** was imposed by a set of ad-hoc queries retrieving data from non-indexed tables. The **Application Change** consists on running a modified version of the TPC-W application. In the modified version there is a given user-transaction that becomes slow as the time goes.

Table 4 shows the configuration details for the anomalies used.

Table 4: Anomaly scenarios (fault-load, resource consumption, workload variation and application change)

Anomaly	Level	Description
Memory Consumption	glassfish	Consume 1MB of JVM Heap in every second
Thread Consumption	glassfish	Consume 30 threads per second
File Handler Consumption	glassfish	Consume 25 file handlers per second
CPU consumption	system	Consume 100% of CPU (gradual)
IO load	system	Read/Write 16MB per second
Storage Consumption	system	Consume 8MB of disk space in every 15 seconds. The consumption is performed in the database server
Database Connections Consumption	TPCW	Consume 2 connections per second
Database Load	TPCW	Runs eight concurrent queries in every 30 sec
Database Table Lock	TPCW	Lock a database table for 30 seconds
Application Change	TPCW	Transactions response time increases 1 millisecond in every 1 second
Requests Overload	TPCW	Starts with 10 concurrent users performing 10 calls each. Per 2000 requests executed the number of concurrent users is increased in 5

4.3.2 Experimental results

Our goal is to study the coverage, detection latency and effectiveness of different monitoring systems. The coverage analysis allows to understand which sort of anomalies are more likely to escape to the different monitoring systems. The detection latency yields the ability to timely detect an anomaly. The effectiveness analysis aims to observe the anomaly effects, as experienced by the end-users, before the anomaly detection phase. It points the number of visible errors and the number of user affected by a slow response time, emphasizing the advantages of early anomaly detection. The performance impact induced by the monitoring systems is also considered in our analysis.

Each of the experiments consisted of injecting the anomaly scenarios described in the previous subsection. For each of the anomaly scenario we measured the coverage and detection latency provided by the different monitoring systems. Each experimentation was repeated 15 times.

4.3.2.1 Coverage analysis

The coverage analysis is presented in Table 5. By monitoring technique and anomaly scenario: when the anomaly was detected it is identified with “1”, otherwise it is identified with “0”.

Table 5: Anomaly detection coverage

	System-level Zabbix	End-to-End JMeter script	Log Analyzer Swatch	App-Level *SHõWA*
Memory Consumption	1	1	1	1
Thread Consumption	1	1	1	1
File Handler Consump.	1	1	0	1
CPU Consumption	1	0	0	1
IO load	1	1	1	1
Storage Consumption	1	1	1	1
Database Connections Consumption	1	1	1	1
Database Load	0	1	0	1
Database Table Lock	0	1	1	1
Application Change	0	1	1	1
Requests Overload	0	1	1	1
Coverage	64%	91%	73%	100%

The anomaly scenarios were selected at an early stage, taking into consideration the report provided by Soila Pertet and Priya Narasimhan in [Pertet 2005]. That report investigates the causes and prevalence of failure in Web applications and it concludes that system overload, resource exhaustion and software upgrades are significant causes of software failures. Considering the anomaly scenarios used, the coverage results presented in Table 5 are very interesting. The *SHõWA* monitoring was the only technique which has detected all the anomalies. The anomalies were detected by multiple monitoring techniques: five of the anomalies were detected by all the monitoring techniques; four of the anomalies were detected by three of the monitoring techniques; two anomalies were detected by two monitoring techniques.

The results obtained by Zabbix are within the expectations. The system-level monitoring tool was efficient to detect all the anomalies injected at the system-level and at the application server level. The anomalies related with changes in the application and related with the database were not detected. The detection of such anomalies requires a more specific analysis on the application, which is beyond the focus of the system-level monitoring tools.

4.3.2.2 Detection latency

The detection latency is a fundamental aspect to accelerate the recovery procedure and mitigate the impact of anomalies. Table 6 contains the detection latency, in seconds, achieved by the different monitoring techniques and anomaly scenarios.

Table 6: Anomaly detection latency (seconds)

	System-level Zabbix	End-to-End JMeter script	Log Analyzer Swatch	App-Level *SHõWA*
Memory Consumption	827	*1193*	1176	**650**
Thread Consumption	*1761*	**1166**	1177	1176
File Handler Consump.	1271	*1464*	—	**1192**
CPU Consumption	372	—	—	**312**
IO load	694	1140	1594	**640**
Storage Consumption	1480	*1915*	*1649*	**104**
Database Connections Consumption	**529**	893	852	*1753*
Database Load	—	87	—	**9**
Database Table Lock	—	43	869	**33**
Application Change	—	806	1493	**19**
Requests Overload	—	613	880	**148**

There are situations when the anomaly occurs immediately after being injected and other situations when it is manifested only some time after being injected. Since we intend to compare the detection latency achieved by each monitoring system, we present the detection latency as being the difference between the time of injection and the moment when it is detected by the monitoring system. For facilitate the reading of table, the fastest anomaly detector appears highlighted in bold. Values in italics and underlined indicate the cases when it was observed the existence of end-user failures.

The detection latency results presented in Table 6, shows that in 9 of the 11 anomaly scenarios the *SHõWA* framework has provided the lowest detection latency. For a considerable number of this scenarios is also remarkable the difference between the detection latency achieved by the *SHõWA* framework and the detection latency achieved by the other tools.

In the **Thread Consumption** scenario the detection latency achieved by *SHõWA* is very close to the detection latency achieved by the end-to-end monitoring. Regarding the little difference on the detection time of these techniques, we did a detailed analysis and we observed that for a considerable number of times we run the experiments, the application-level monitoring was the first technique to detect the anomaly. This happens because while the end-to-end monitoring is performed in every 60 seconds the application-level monitoring is continuously performing the data analysis.

We can not say that *SHõWA* will detect all the anomalies. However, according to the results, we can say that it is useful to detect a number of performance anomalies that occur frequently and affect the behavior of Web-based applications.

4.3.2.3 Effectiveness analysis

The effectiveness analysis provides the relationship between the detection latency, the number of visible errors and the number of users who have experienced a response time slowdown. The results are presented in Table 7. The values in bold are used to establish the association between the effectiveness and the fastest detection technique.

The average number of slow transactions experienced by the end-users was appraised in three distinct intervals. The first interval contains the average number of user-transactions that, compared to the baseline values observed previously, suffered a delay in response time between 100 and 500 milliseconds. The second interval refers to the number of user-transactions that have experienced a delay in response time between 500 milliseconds and 2000 milliseconds. The third interval, gauges the average number of user-transactions with a response time latency higher than 2000 milliseconds. These intervals were chosen according to the response time impact reported in [Linden 2006] and [Schurman 2009]. According to Greg Linden in [Linden 2006] an increase of 500 milliseconds in the access latency to Google may result in a loss of 20% of its traffic. Likely, an increase of just 100 milliseconds in the latency to Amazon may result in a reduction of 1% of sales. Eric Schurman from Bing and Jake Brutlag from Google, report in [Schurman 2009] a linear impact on the business metrics, as the delay increases from 200 milliseconds to 2000 milliseconds.

Table 7: Average number of visible errors and average number of end-user transactions affected by a response time slowdown
(**BOLD**: first detector to detect the anomaly)

Anomaly	sys+container+db: Zabbix				End-to-end: JMeter script				Log Analyzer: Swatch				App-level: *SHõWA*			
	# Errors	#slow responses			# Errors	#slow responses			# Errors	#slow responses			# Errors	#slow responses		
		[100: 500[ms	[500: 2000[ms	[2000: Inf[ms		[100: 500[ms	[500: 2000[ms	[2000: Inf[ms		[100: 500[ms	[500: 2000[ms	[2000: Inf[ms		[100: 500[ms	[500: 2000[ms	[2000: Inf[ms
Memory Consumption	0	0.6	8	0	*2*	19.8	143.1	16.2	0	17.7	103.4	4.2	**0**	**0**	**3.2**	**0**
Thread Consumption	*392*	15.3	1.2	68.6	**0**	**13.6**	**0**	**3.6**	0	13.6	0	4.1	0	13.6	0	4.1
File Handler Consumption	0	1.7	7.9	0	*6*	1.7	8.4	0	6	1.7	8.4	0	**0**	**1.7**	**6.6**	**0**
CPU Consumption	0	247.2	0	0	0	4308	612.5	0	0	4308	612.5	0	**0**	**48.6**	**0**	**0**
IO load	0	18	0.8	0	0	41	0.8	0	0	70.6	2.7	0	**0**	**11.4**	**0.8**	**0**
Storage Consumption	0	27.5	0	0	*45*	33.7	0	0	*5*	28.2	0	0	**0**	**20.2**	**0**	**0**
DB Conn Consumption	**0**	**0**	**0**	**0**	0	0.7	0	0	0	0.7	0	0	*335190*	0.7	0	6.2
DB Load	0	385.7	31.3	262.4	0	30.1	1.9	22.3	0	385.7	31.3	262.4	**0**	**14.8**	**1.6**	**6.4**
DB Table Lock	0	67.1	0	909.9	0	1.6	0	7.5	0	22.1	0	223.5	**0**	**1.6**	**0**	**7.5**
Application Change	0	1119	6311	2044	0	1107	1002	0	0	1107	4518	0	**0**	**0**	**0**	**0**
Requests Overload	0	17.6	0.6	1353	0	0	0	0	0	13.4	0	2.2	**0**	**0**	**0**	**0**

This first row in Table 7 presents the results when the application suffered from a memory consumption anomaly. All the detectors were able to detect the anomaly and *SHõWA* was the fastest detector to report the anomaly. When the anomaly was detected by *SHõWA* there were only three requests affected by a response time increase superior to 500 milliseconds. Zabbix has triggered the alert almost three minutes later. As consequence the number of end-users experiencing a slow response is higher. The end-to-end monitoring presented the highest detection latency. It needed almost twice of the time to detect the anomaly. Consequently, there is a significant number of users that have experienced a slow response time and also users affected by HTTP 5XX errors.

Like in the previous anomaly scenario, if the thread consumption anomaly is not detected and recovered in time, a crash-failure occurs. The thread consumption scenario was detected by all the techniques. The end-to-end, log analyzer and *SHõWA* monitoring techniques provided similar detection latency results. For the corresponding techniques, the number of user-transactions affected was also very low. Zabbix presented the highest detection latency. It have issued some intermittent alerts advising for Glassfish and SSH unavailability and some attempts to access the system were rejected by the operating system, since it was unable to fork more child processes. When the anomaly was detected by Zabbix (nine minutes after the other techniques), there were already 392 end-users who experienced HTTP errors and 85 others who have experienced a slow response time.

In the third row of Table 7 we see the results achieved when the number of open file descriptors increases until the number of open files per-process limit is reached. At this point the server stops accepting new requests. The *SHõWA* monitoring was the fastest technique to detect the anomaly. Consequently it is the technique with the lowest number of end-users experiencing slow response times. During the tests, Zabbix has issued some false and intermittent alarms advising for *"LOW"* and *"OK"* JVM Heap memory available. A clear message reporting the file handlers problem was sent by the Zabbix only 80 seconds after the detection provided by *SHõWA*. The end-to-end technique has presented the highest detection latency. Its detection time corresponds to the moment when the application server has stopped responding to the incoming requests.

The CPU consumption scenario is presented in the fourth row. Only two of the monitoring techniques were able to detect this anomaly. The *SHõWA* monitoring was the fastest technique to detect the anomaly. It was followed by Zabbix which, 60 seconds later, has sent an alert message indicating "high CPU load". Concerning to how the end-users have experienced this anomaly, the results makes clear the advantages of early detection. A difference of just 60 seconds between the first two detectors has resulted in an increase from 48 to 247 slow user-transactions. The end-to-end and log analyzer techniques were unable to detect this anomaly, consequently the number of slow user-transactions observed at the end of the experiment ascended to 4920. A similar result can be observed in the fifth row. The *SHõWA* monitoring have detected the IO load impact on the application before the other monitoring techniques did. As result, the number of end-users affected by slow user-transactions was lower.

The sixth row in Table 7 presents the number of visible errors and user-transactions experiencing slow response times when the filesystem space of the TPCW database is consumed until an *"out of space"* error occurs. *SHõWA* was the fastest technique to detect the anomaly. It detected the anomaly before the disk space becomes totally used. This happened because the IO write operations originate an increase in the time necessary to retrieve the data from the storage subsystem. This has affected the database queries response time. Zabbix detected and triggered the corresponding alert message. The end-to-end and log analyzer techniques have detected the anomaly later. In these cases, there were users that experienced HTTP errors. The existence of end-user failures reinforces the importance of early detection of performance anomalies. By detecting performance anomalies earlier it is possible to execute recovery actions preventing the occurrence of severe failures.

The scenario in which the application is affected by the number of database connections is presented in the seventh row. TPCW is provided with a basic database connections pool management. If the connection pool is empty and if the number of database connections is bellow the value defined in the database configurations then a new connection is created, otherwise a *"too many connections"* error is throw. The anomaly was detected by all the monitoring systems. Zabbix was the fastest technique to detect a low number of database connections available. For the first time the *SHõWA* monitoring was the slowest detector, as well, it accounted for a considerable number of end-users who have experienced HTTP errors. This result is justified by the type of anomaly: if there are database connections available, then everything will work normally, otherwise the new user-transactions will fail. Since it does not originate a response time degradation it is quite difficult to be timely detected by *SHõWA*.

A different result is observed for the database load and table lock scenarios. *SHõWA* was the fastest technique to detect the anomalies. As consequence the number of end-users affected by the slowdown is lower when compared with the other monitoring techniques.

The application change that motivates a performance slowdown is presented in row number ten. *SHõWA* has detected the anomaly 19 seconds after it was initiated. The end-to-end monitoring and log analyzer techniques have also reported the anomaly, but when they did it, the number of end-users experiencing a slow response time has already increased considerably. This anomaly was gone unnoticed by Zabbix.

Finally, in the last row, we present the results achieved the monitoring systems when the request overload anomaly was injected. This anomaly was detected by the end-to-end, log analyzer and *SHõWA* monitoring techniques. The *SHõWA Workload*

Variation Analysis module detected the workload variation before the anomaly was detected by the other tools. Since we did not apply any repair action, the application-level monitoring reported the problem after 1169 seconds of its injection. At this time the dissociation between the response time and the number of user-transactions was very evident: 17.7 users have experienced a slowdown between 100 milliseconds and 500 milliseconds; 0.6 users in average have experienced a slowdown between 500 milliseconds and 2000 milliseconds; and 113.8 users have experienced a response time slowdown superior to 2000 milliseconds. This result confirms the importance of the early detection of workload variations. By detecting this situations, it is possible to redimension the system, preventing end-users from being affected by performance anomalies.

4.3.3 Performance overhead analysis

To evaluate the performance impact introduced by the monitoring techniques we conducted an experimental analysis per each of the monitoring techniques. The experimentation considers different levels of system utilization and evaluates the user-transactions response time latency and the server throughput impact induced by the different tools.

Initially, we conducted an experimentation to determine the maximum capacity of the server in terms of response time and throughput. In this experimentation we did not used any monitoring system. After this, we performed tests to measure the performance impact induced by each of the monitoring systems.

Each experimentation was repeated 15 times. For each group of experiments conducted, we started with an execution using the TPC-W RBE followed by a test with the *httperf* tool. The *httperf* [Mosberger 1998] is a general purpose tool useful to measure the application and server limits. The application workload performed with the *httperf* tool mimics real end-users sessions and it was executed varying the number of concurrent user requests from 29 and 804 requests per second. According to the baseline results, the application server can process up to 517 requests per second, until the response time is affected by a severe slowdown.

To facilitate the analysis the results were split in four intervals. These intervals correspond to the system utilization: 0% to 25% corresponds to requests load varying between 29 and 144 requests per second; 25% to 50% corresponds to requests load varying between 144 and 287 requests per second; 50% to 75% corresponds to requests load varying between 287 and 431 requests per second; and 75% to 100% corresponds to requests load varying between 431 and 517 requests per second.

The response time latency is summarized in Table 8. It corresponds to the difference between the response time achieved while using each monitoring system and the baseline response time. The latency time is displayed in milliseconds per request. The values are presented in milliseconds because we observed that the response time latency is independent from the duration of the transactions, i.e., the response time latency value is very similar regardless of the transaction takes 10 or 2000 milliseconds to run.

Table 8: Response time latency induced by the monitoring systems considering different levels of system utilization

	Latency penalty (ms/req) per system load			
	0%-25%	25%-50%	50%-75%	75%-100%
System-level: Zabbix	0.05	0.03	0.05	0.15
End-to-end: JMeter script	0.03	0.03	0.58	12.53
Log Analyzer: Swatch	0	0.03	0.13	0.73
App-level: *SHõWA*	0.0	0.14	0.35	2.21

The throughput impact is presented in Table 9. The results are presented as the percentage of user-requests that the server was unable to process when compared with the baseline values.

Table 9: Throughput penalty - reduction in % of req/sec - considering different levels of system load

	Reduction in % of req/sec per system load			
	0%-25%	25%-50%	50%-75%	75%-100%
System-level: Zabbix	0%	0%	0%	0.07%
End-to-end: JMeter script	0%	0%	0.02%	0.80%
Log Analyzer: Swatch	0%	0%	0.02%	0.12%
App-level: *SHõWA*	0%	0%	0.02%	0.89%

As presented in Table 8 and in Table 9, the application-level monitoring (*SHõWA Sensor* module) does not induce a significant performance penalty. The response time latency per user-transaction is only 2 milliseconds and the throughput impact is inferior to 1%. It should be noted that behind these values are the selective and adaptive monitoring algorithms included in the *SHõWA Sensor* module.

4.3.4 Summary of results

The study has provided an experimental analysis of different techniques available for monitoring and detecting anomalies in Web-based applications. The results focus the coverage and the detection latency provided by each monitoring technique. The number of visible errors and the number of users affected by a slow response time was also evaluated.

From the results stands out:

- The application-level monitoring provided by *SHõWA* was the best detection technique. It detected 100% of anomalies used in the experimental study and, for 82% of them it was the technique with the lowest detection latency;
- The system-level monitoring provided by Zabbix was the technique with lower coverage. It only detected 64% of the anomalies. These results expose its limitation to detect anomalies that are out-of-the system-level scope;
- The end-to-end monitoring, provided by the JMeter script has detected a considerable number of anomalies. In three of the scenarios, the anomaly was only detected after the end-users were affected by HTTP errors;
- The log analyzer, provided by the Swatch tool, has detected some symptoms of performance anomalies, mainly due to the existence of HTTP keep alive timeouts. A keep alive timeout has nothing to do with visible failures, but a highly number of keep alive timeouts in a short time may be interpreted as a symptom of performance degradation;
- Regarding the effectiveness analysis it becomes clear that every second matters. Timely detection, followed by recovery contributes to reduce the number of end-users experiencing slow response times or visible errors.

Another important conclusion can also be taken from the performance impact analysis. The application-level monitoring provided by *SHõWA* has just introduced a response time latency that, according to the system load, varies between 0 and 2.21 milliseconds per request. The throughput impact is not affected when the system load is low. It is affected in less than 1% when the system load is close to its maximum capacity. Considering the coverage and detection latency achieved by the *SHõWA* monitoring and the performance impact results we are convinced that it is viable to adopt *SHõWA* to detect anomalies in production environments.

4.4 Conclusions

A monitoring system is a fundamental tool to gather information from the system and verify if it is performing as expected. In a self-healing framework, as we propose, the monitoring activity is also very important. Sensors should collect data from the managed resource and turn the data available for the analysis carried out by the autonomic manager.

In this chapter we have presented the different elements of a monitoring system and described the different monitoring techniques that are commonly adopted for Web-based applications. A coverage analysis, a detection latency and an effectiveness study was also presented. This study considers different monitoring techniques and different anomaly scenarios. One of the tools adopted in the study was the application-level technique implemented in the *SHõWA* framework. It consists on a program, implemented according to the Aspect-Oriented Programming (AOP) principles, that is responsible for measuring the duration of the real user-transactions, collect different server and application server parameters and measure the duration of the calls belonging to the user-transactions (transactions profiling). This program performs bytecode instrumentation, do not require the application recompilation neither knowledge about its implementation details.

The experimental results are very encouraging. From the different monitoring techniques, the monitoring performed by the *SHõWA* framework was able to detect all the anomalies and it was the first technique to detect 9 of the 11 anomalies injected. It was also the only technique to detect scenarios of workload variation, before the end-users transactions are affected by a severe slowdown.

Considering the monitoring approach followed by the *SHõWA Sensor* module (application profiling), is also notable the low performance impact induced by the *Sensor* module. Behind this happening, are the selective and adaptive monitoring algorithms that dynamically and autonomously adapt the sampling frequency and the number of user-transaction calls to be intercepted. A complete description about these algorithms and a comparison between different application-level monitoring tools is presented in the next chapter.

Chapter 5
Self-adaptive Monitoring for Pinpointing Anomalies in Web-based Applications

KEY POINTS

◇ From the monitoring systems tested in the previous chapter, the application-level monitoring stands out for its degree of anomaly coverage and shorter detection latency.

◇ Application-level monitoring allows to profile the application by collecting application parameters (e.g., user-transactions response time, calls response time, interactions between components) and to monitor these parameters to detect changes and analyze whether these changes are motivated by anomalies.

◇ The level of detail that is achieved through application-level monitoring is useful to pinpoint the anomalies. However, doing this type of monitoring at runtime may impose a significant performance impact.

◇ Many of the application-level monitoring solutions, abdicate to intercept a substantial portion of the application, thereby reducing the performance impact induced, but in this way it is not possible to pinpoint anomalies originating from the components that are not monitored.

◇ In this chapter, we evaluate the performance impact induced by different application-level monitoring systems and we propose self-adaptive and selective mechanisms that allow to profile all the application calls with low performance impact.

Application-level monitoring receives a considerable attention from the scientific community. It is commonly applied by the developers as part of the development cycle to identify where the system resources burdens are located and how to suppress them. Although, and considering that such offline analysis do not capture runtime anomalies, its adoption for runtime monitoring, anomaly detection and root-cause analysis is considered fundamental to reduce the mean-time-to-repair (MTTR), improving the systems availability and performance. A major challenge with runtime application profiling is the performance impact induced in the application or system under monitoring.

In this chapter we describe the self-adaptive and selective monitoring algorithms used to regulate the application-level profiling. We present a comparison between application-level monitoring tools and the performance impact induced by them. We also present some considerations about the tradeoff between the amount of data provided by the application-level monitoring tools and the ability to timely detect and pinpoint anomalies.

5.1 Application-level monitoring tools

Collecting application-level data may be achieved by means of black-box or white-box monitoring. The black-box monitoring treats the application as a monolithic structure, i.e., it only considers the interactions with the external environment without any attempt to model the internal structure. There are some remarkable projects which have adopted black-box techniques to detect anomalies in Web-based applications. Aguilera et. al, in [Aguilera 2003], have intercepted the network information to derive the call-graph between the black-boxes (nodes) without any knowledge of node internals or message semantics. Such approach do not require modifications to applications, middleware, or messages and allows to detect and identify the casual path patterns accusing higher latency, as well, the corresponding nodes. The Pinpoint project [Chen 2002] takes a black-box approach to instrument the application server to audit the components and record the request execution paths, without the overhead of writing additional code for each component. The Magpie project [Barham 2004] is another sound work in the area. It collects fine-grained traces from all software components, combines those traces across multiple machines and tracks the resources usage to build a probabilistic model of request behavior that is then used to detect anomalies.

The white-box monitoring requires knowledge about the implementation details in order to devise techniques to instrument applications to model the application structure (application profiling) and disclose significant events. The instrumentation is said static when it is previously defined by the programmer before the compilation or execution. It is considered dynamic when it can be included at runtime. It is also common the division between source code and binary/bytecode instrumentation. Source code instrumentation is made directly over the application source code. In the binary/bytecode, the instrumentation hooks operate (statically or dynamically) at the virtual execution environment.

A useful approach for binary/bytecode instrumentation is provided by the Aspect-Oriented Programming (AOP) paradigm [Kiczales 1997]. AOP supports the modularization of concerns at the source code level and it is being largely adopted to integrate profiling code without requiring changes in the original application source code (binary/bytecode weaving). There is a significant number of tools adopting AOP to instrument Web-based applications. Most of them adopt static instrumentation and, while some of them provide online visualization for the metrics of interest (e.g. execution times, available memory, CPU time) at runtime, others only provide a final report when the application under monitoring is stopped.

DJProf [DJProf] is an example of a such AOP-based tool. It allows to intercept the execution of a Java program to collect the memory usage, objects lifetime, wasted CPU and the total response time. It is easily integrated with standalone programs, but the lack of compatibility libraries makes difficult its integration with application containers. Profiler4J [Profiler4j] is another tool. It allows to write the rules to instrument the packages, classes, or methods belonging to the application. The instrumented code can be seen online, through a graphical user interface. Like Profiler4J, JRat [JRat] also allows to specify the monitoring rules, however the instrumented code can only be visualized in offline. Hence the application is defined to be profiled it do not allow to remove the instrumentation code from the class files. JIP [JIP] is a another AOP-based tool. It adds AOP aspects to every methods and allows to collect the network time and objects allocation. The collected data is stored in memory and it is dumped to a file when the JIP profiler is turned off.

The jeeObserver [jeeObserver], InfraRED [InfraRED] and JavaMelody [JavaMelody] tools also allow the collection of application-level data without requiring application changes. The data can be visualized online, through a Web browser interface. By default these tools allow the interception of user-transactions and the corresponding methods. The collected data is kept in memory and aggregated according to a predefined time interval. Through the web page is possible to access the response time of each user-transaction, the memory usage or the number of running threads.

The *SHõWA* framework includes the data gathering process, the data preparation and the workload variation and performance anomaly analysis. These steps are performed online, i.e., the data is continuously collected by the *Sensor* module, prepared by the *Data Preparation* module and submitted for analysis. The *Sensor* module makes use of Aspect-Oriented Programming (AOP) [Kiczales 1997], more specifically using the AspectJ AOP extension [Laddad 2009]. The *Sensor* module is attached to the application server using the Load-Time Weaving [LTW] features. Thus, through the *Sensor* module, it becomes possible to collect application-level data without having to change the application source code. The application is monitored as its code is executed by the application virtual machine. The *Sensor* module collects data with two different levels of granularity: *user-transaction* level monitoring and *profiling* level monitoring. In the *user-transaction* level it includes a pointcut that intercepts and measures the server response time of the user-transactions and gathers different system and application server parameters (e.g., CPU load, JVM heap memory, number of open files, number of running threads). In the *profiling* level it makes use of a pointcut (`call(*`

`*(..))` that allows to intercept and measure the execution time of the calls involved in the user-transactions call-path. The data collected at the *user-transaction* level is used to detect slow user-transactions and identify if there is a system or application server parameter change that is associated with the performance anomaly. The application *profiling* data is used to pinpoint the calls/components associated with a performance anomaly.

Since collecting application level data can induce a high performance penalty in the system, it is fundamental to devise mechanisms that allow to do this type of monitoring without compromising the application performance. In this context, we propose algorithms that allow the *Sensor* module to adapt dynamically the frequency of monitoring and select the list of calls to be intercepted. The performance impact induced by *Sensor* module, while using these algorithms, is analyzed in this chapter through an experimental study.

5.2 Adaptive and selective monitoring

In the literature we find some examples that focus on the monitoring adaptation. Rish et al., in [Rish 2005], describe a technique, called active probing. It combines probabilistic inference with active probing to yield a diagnostic engine able to "asks the right questions at the right time", i.e., dynamically selecting the probes that provide maximum information gain about the current system state. From several simulated problems, and in practical applications, the authors observed that active probing requires in average up to 75% less probes than the pre-planned probing. In [Kumar 2006] the authors apply transformations to the instrumentation code to reduce the number of instrumentation points executed, as well, the cost of instrumentation probes and payload. With the transformations the authors have improved the profiling performance by 1.26x to 2.63x. A technique to switch between instrumented and non-instrumented code is described in [Arnold 2001]. The authors combine code duplication with compiler inserted counter-based sampling to activate instrumentation and collect data for a small, and bounded, amount of time. According to the experimentation, the performance overhead induced by the sampling framework was reduced from 75% (in average) to just 4.9%. According to the authors, the final accuracy of the analysis was not affected. In [Munawar 2006] the authors argue that a monitoring system should continuously assess current conditions by observing the most relevant data, it must promptly detect anomalies and it should help to identify the root-causes of problems. Adaption is used to reduce the adverse effect of measurement on system performance and the overhead associated with storing, transmitting, analyzing, and reporting information. Under normal execution, only a set of healthy metrics is collected. When an anomaly is found, the adaptive monitoring logic starts retrieving more metrics till a faulty component is identified. Unfortunately the authors did not present results about the performance impact induced by their approach.

The *SHõWA Sensor* module includes self-adaptive and selective algorithms. These algorithms provide the self-adaptation of the level of application profiling in the sense that the sampling frequency is adjusted *on-the-fly*. They are selective since the adaptation only applies to user-transactions reporting symptoms of performance anomaly and there is only a subset of calls/components which are intercepted. Achieve the equilibrium between timely detection and pinpointing of anomalies and a low performance penalty is the motivation behind the adoption of adaptive and selective monitoring techniques.

5.2.1 Linear, exponential and polynomial adaptation

Currently the *Sensor* module includes three adaptive algorithms that allow to adjust, dynamically, the sampling frequency at which the user-transactions are profiled: linear adaptation algorithm; exponential adaptation algorithm; polynomial adaptation algorithm.

The three algorithms follow the same basic principle: if there is no symptom of performance anomaly, then the frequency at which the user-transactions are profiled is smaller. If a symptom of performance anomaly is observed, then the user-transactions are profiled more frequently. Being more frequent, it allows to collect more data, improving the detection and pinpointing of performance anomalies. The symptom of performance anomaly is given by the *Performance Anomaly Analysis* module. This module computes the Spearman's rank correlation coefficient - $\boldsymbol{\rho}$ - between the accumulated server response time of each of the user-transactions and the number of the corresponding user-transactions processed in the same interval of analysis. A strong and stable correlation degree indicates that the response time is varying with the number of requests. When the correlation degree decreases, such relationship becomes compromised, evincing a symptom of performance anomaly. Therefore, taking into account variations in the correlation degree, when the correlation degree remains stable and strong the frequency of user-transactions profiling can be low. When the correlation degree decreases, the frequency at which the user-transactions are profiled increases in order to provide more data for the anomaly detection and pinpointing analysis.

In the linear adaptation algorithm, the sampling frequency adjustment is proportional to decrease or increase verified in the correlation degree computed by the *Performance Anomaly Analysis* module. With the exponential algorithm, when the correlation degree decrease is small then the sampling frequency will remain high. As the correlation degree approximates the minimum value, the algorithm provides a rapid adjustment allowing to gather more data from the application. The polynomial adaptation algorithm fits between the linear and exponential algorithms. It is composed by different exponential functions and it allows to keep the sampling large until a given amount/percentage of correlation degree decrease is observed. After that point the sampling frequency is adapted more frequently to provide enough data for the analysis.

We are using these adaptive algorithms to determine the sampling frequency used by the *Sensor* module, but the algorithms can be adopted for other adaptive scenarios (e.g. time-based adaptation, parameter based adaption).

5.2.1.1 Adaptability analysis

The linear, exponential and polynomial algorithms presented return different sampling frequencies. Taking as an example: a value $\boldsymbol{M} = \mathbf{80}$ as maximum value for sampling frequency; a value $\boldsymbol{m} = \mathbf{2}$ as the minimum sampling frequency; $\boldsymbol{\alpha} = \mathbf{0.2}$ as the maximum correlation degree decrease to be tolerated; $\boldsymbol{\beta} = \mathbf{50\%}$ as the break point used by the polynomial algorithm, in Figure 14 we illustrate how the sampling frequency is adjusted. The analysis considers both slow and abrupt correlation degree variations.

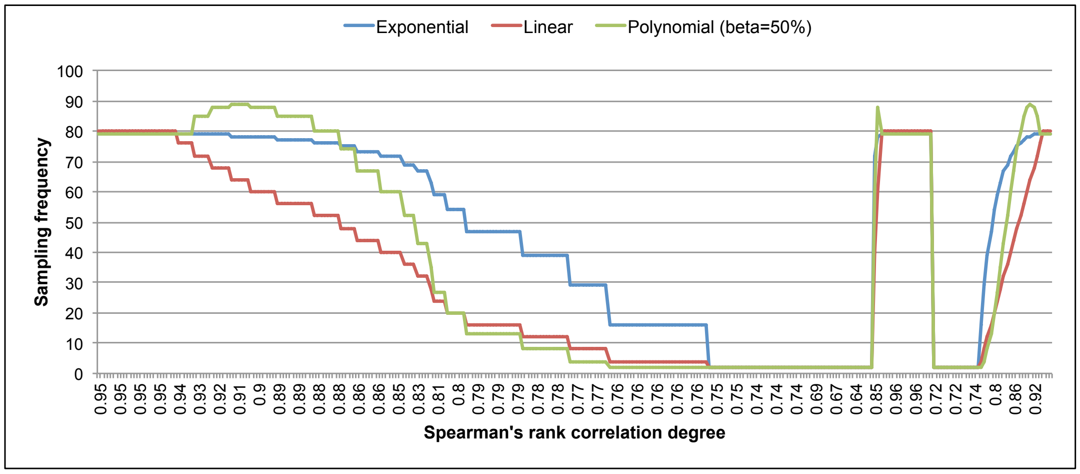


Figure 14: Adaptive monitoring - adaptability provided by the linear, exponential and polynomial algorithms

As illustrated in Figure 14, with the linear adaptive algorithm the sampling frequency is adjusted according to the decrease observed in the correlation degree. The exponential algorithm is more conservative: it does not adapt the sampling frequency when the correlation degree variation is small, but for higher correlation degree variations it quickly adjusts the sampling frequency to a higher rate, increasing the number of times a user-transaction is profiled.

The third degree polynomial algorithms provides an interesting feature. When the correlation degree decrease is below a given threshold (determined by $\boldsymbol{\beta}$), the sampling frequency decreases. This is very interesting, because it allows to verify if the correlation degree decrease was motivated by the application profiling itself. When the decrease on the correlation degree is between the minimum and maximum threshold, the sampling frequency is adapted in a moderate fashion. If the decrease on the correlation degree is superior to the $\boldsymbol{\beta}$ threshold, then the sampling frequency is rapidly adjusted, allowing the *Sensor* module to collect more data about the application calls.

From Figure 14 is also visible, that all the algorithms react very quickly when an abrupt correlation degree decrease or increase is observed. When the correlation degree increases, the algorithms return a higher sampling frequency, reducing the performance impact induced by the *SHõWA Sensor* module.

5.2.2 Selective monitoring

The implementation of the *Sensor* module allows to intercept, by default, all the user-transactions and the corresponding application calls. This is done through the pointcut `call(* *(..))` included in the monitoring program.

If, on one hand, collecting all the application calls improves the pinpointing of anomalies, on the other hand, it becomes too intrusive for the application. In this context, limiting the monitoring only to a subset of the application can reduce the performance impact induced by monitoring. However, the pinpointing of the anomalies is compromised when are involved application calls that are out of the monitoring scope. Taking this into consideration, we adopt a mechanism that allows the *Sensor* module to select, at runtime, the list of application calls to be intercepted, without discard any application call.

The selective mechanism takes into account the weight that every application call represents for the response time of the user-transactions. If the response time exceeds a certain $\boldsymbol{X\%}$ then the application call is intercepted and its processing time is measured. Otherwise the call is not intercepted. In certain moments, all the calls are intercepted.

Once the response time of an application call can vary, for example motivated by a resource contention or due to a change in the application, the list of calls to be intercepted is refreshed: whenever the Spearman's rank correlation coefficient - $\boldsymbol{\rho}$ - decreases by at least 0.05 degrees; periodically, from $\boldsymbol{K*R}$ times, being $\boldsymbol{K}$ the sampling frequency and $\boldsymbol{R}$ a refresh factor. The $\boldsymbol{K}$ is determined by the adaptive algorithms and $\boldsymbol{R}$ can be a predefined number or based-on the current number of application calls intercepted.

5.3 Application-level monitoring: performance study

An application may, at any time and for various reasons, behaving differently from what is expected, affecting its users and consequently the revenue and reputation of the companies that use the application within their business processes. For this reason, monitoring applications at runtime in order to quickly detect and recover from anomalies is an important aspect. For the recovery process to be faster and more efficient it is crucial that the detection is done quickly and indicate the root-cause of the anomaly.

Application-level monitoring allows to keep track about what is going on inside of the application. What are the main entry points, what database queries are involved or which parts of the application are slow, are some examples of data that can be provided by an application-level monitoring tool.

Naturally, instrument the application to capture this data introduces a performance penalty. In this chapter we address the performance impact introduced by different application-level monitoring tools. To reduce the performance impact introduced by the application-level monitoring, included in the *SHõWA* framework, we adopted self-adaptive and selective monitoring algorithms. We conducted an experimental study to access:

- *What is the role of application-level monitoring in the context of anomaly detection and localization?*;
- *What is the performance impact induced by different application-level monitoring tools?*;
- *How efficient are the proposed algorithms?, particularly*:
  - *How much does they contribute to reduce the performance impact introduced with runtime application profiling?*;
  - *Do they still providing the timely detection and pinpointing of anomalies?*

To conduct the performance study, we prepared a framework composed by a Web-based application and different application-level monitoring tools. We used some representative workloads to evaluate the latency and throughput of the server under test. For the performance analysis we considered the amount of source code covered by the application-level monitoring tools. The set of adaptive and selective algorithms presented in this chapter were considered to reduce performance penally induced with application-level profiling. The tradeoff between the amount of data provided by the application-level monitoring tools and the ability to timely detect and pinpoint anomalies is also considered in this study.

5.3.1 Experimental setup

The experimental environment comprises a three-tier infrastructure running a benchmark application, different application-level monitoring tools (JRat, InfraRed, JeeObserver, JavaMelody, profiler4J, *SHõWA*), dedicated servers to store the monitoring data, servers carrying out the statistical analysis for anomaly detection and emulated clients performing requests.

In the first phase of the experimental study, we analyze the impact induced by the *Sensor* module without considering the use of adaptive and selective monitoring algorithms. Then we evaluate the performance impact induced by other application-level monitoring tools and mention the main differences between these tools and the monitoring provided by the *SHõWA Sensor* module. Finally we present results on the performance impact caused by the *Sensor* module, when it uses the adaptive and selective algorithms.

For a better understanding of the role of dynamic adaptation, we present a comparison between the selective and adaptive monitoring and the monitoring performed while using static sampling frequencies. This analysis allows us to evaluate the relationship between the frequency of monitoring and the time required to detect and pinpoint anomalies.

5.3.1.1 Testbed

The testbed is illustrated in Figure 15.

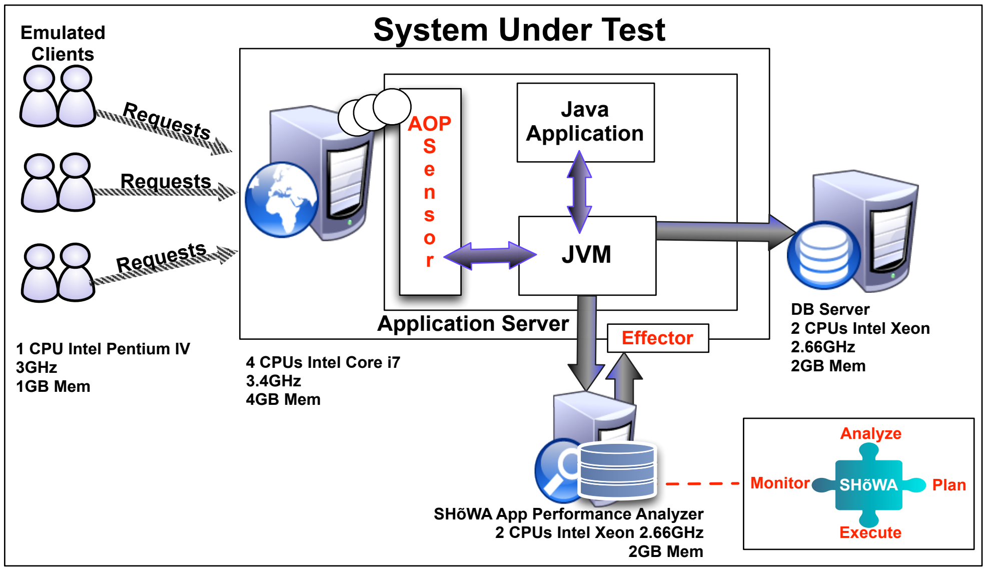


Figure 15: Test environment

The System Under Test consists of an application server (Oracle Glassfish Server [Oracle Glassfish Server]) running the TPC-W [Smith 2001] benchmark application. The TPC-W benchmark simulates the activities of a retail store website. It uses a MySQL database consisting of many tables with a wide variety of sizes, attributes, and relationships as a storage backend. The user requests are simulated using several remote emulated browsers (RBEs) provided by the TPC-W benchmark application.

The benchmark allows the execution with different concurrent users, three different traffic mixes (browsing, shopping and ordering), different session length, user think-times and ramp-up/ramp-down periods.

The testbed is adjusted to accommodate the different application-level monitoring tools adopted in this study. Most of the tools are installed directly with the application. The *SHõWA* framework combines the *Sensor* module (installed on the application server), with the *Workload Variation Analysis*, *Performance Anomaly Analysis*, *Anomaly Detector* and *Root-cause Failure Analysis* modules, which are installed on the "*SHõWA* Application Performance Analyzer" server. This separation is important to avoid performance impacts motivated by the data analysis process.

The database management system used in the "*SHõWA* Application Performance Analyzer" server to store the data gathered by the *Sensor* module is H2 [H2 Database]. H2 is a high performance database system. It allows to keep database tables in memory and provides a SQL API to interact with the data.

The test environment is composed by several machines. It includes four workload generator nodes with a 3GHz Intel Pentium CPU and 1GB of RAM each. The "*SHõWA* Application Performance Analyzer" machine includes two 2.66GHz Intel Xeon CPUs and 2GB of RAM. The application server machine includes four 3.4GHz Intel Core i7 CPUs and 4GB of RAM and the database server machine includes two 2.66GHz Intel Xeon CPUs and 2GB of RAM. All the nodes run Linux with 2.6 kernel version and are interconnect through a 100Mbps Ethernet LAN.

5.3.1.2 Workload characterization

To study the response time latency and throughput penalty, induced by the application-level monitoring techniques it is important to determine first the performance baseline values. For this purpose we performed experiments without any monitoring system enabled and considering the system under different levels of application workload. The baseline values are used to make comparisons of performance between the different application-level monitoring tools and to evaluate the benefits of using adaptive and selective monitoring algorithms.

The TPC-W contains its own workload generator (RBE) that simulates the activities of a business oriented transactional Web server and exercises a breadth of system components associated with such environments. It reports metrics, such as, response time and the application throughput. Despite its usefulness, the RBE implementation presents some challenges to determine the server's maximum capacity. An alternative, and widely used tool to find out the application and server limits, is *httperf* [Mosberger 1998]. In its simplest form, it allows to choose a given URI, or a list of URIs, to be accessed by a given number of parallel requests. With some efforts, it also allows to submit a workload that is more representative of the end-users behavior.

Taking this into consideration, for each group of experiments we started with an execution using the TPC-W RBE followed by a test with *httperf*. This was achieved by extracting the session requests sequence within each session from the log file generated from the TPC-W RBE client, and saving that information to a "session_file" for use with *httperf*. The "session_file" was replayed by *httperf* varying the number of concurrent requests from 29 and 804. The number of concurrent requests per second increases during the experiment, until the server reach its maximum capacity.

The baseline values for the server throughput and the user-transactions response time are illustrated in Figure 16.

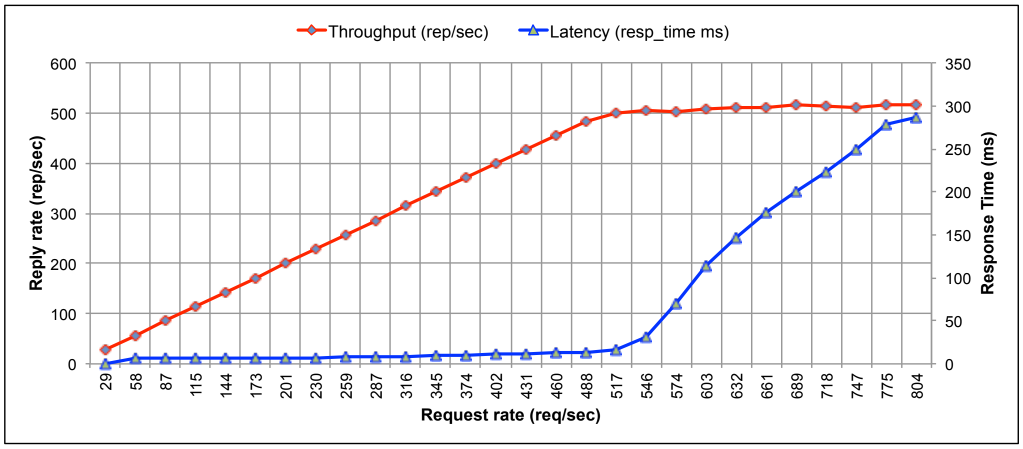


Figure 16: Baseline values for throughput in replies per second and the response time latency in milliseconds

From the results illustrated in Figure 16 is visible that the server is able to process up to 517 requests per second without compromising the response time. After that point the server becomes overloaded and the throughput and response time is severally affected. The 517 requests per second will be used for the remaining experimentation as the reference value for the maximum server capacity/load.

5.3.2 Performance impact induced by the *SHõWA* framework without adaptive and selective monitoring

The first analysis about the performance impact induced by the application-profiling performed by the *Sensor* module took into account the use of the *SHõWA* framework without the adaptive and selective algorithms. It was executed the same workload that was used to achieve the baseline values.

The results are illustrated in Figure 17. The values correspond to the absolute difference between the baseline results and the results obtained in this experimentation.

The figure contains three scenarios on the use of the *SHõWA Sensor* module: “full-profiling”, “full-profiling with selective" and “full-profiling without persistence”. In the “full-profiling” scenario, all the user-interactions and calls are intercepted and stored in the database for analysis. In the “full-profiling with selective" scenario, the user-interactions are intercepted all the time, but only the calls which have a duration greater than 10% of the user-transaction processing time are intercepted and stored in the database for posterior analysis. In the “full-profiling without persistence” scenario, all the user-interactions and calls are intercepted but they are not stored in the database.

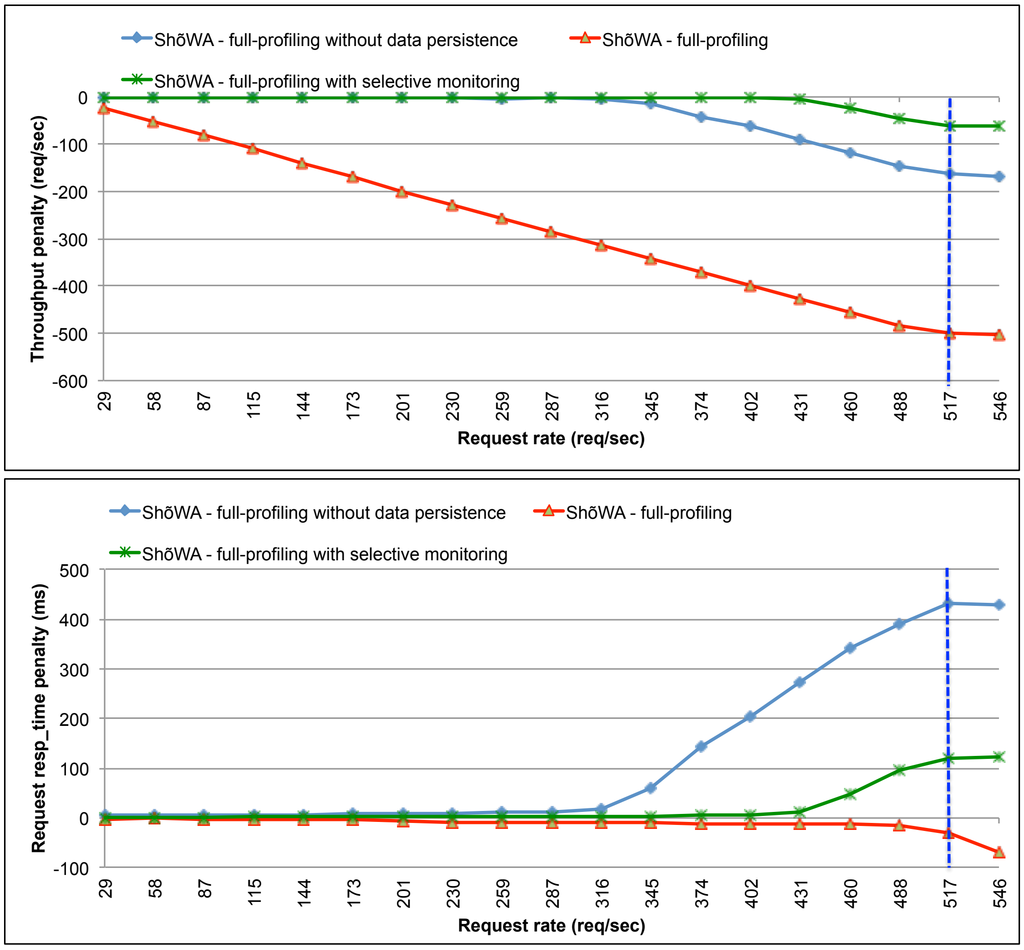


Figure 17: Throughput penalty (top) and response time penalty (down) induced by the *SHõWA* framework before the adoption of adaptive and selective monitoring

The results presented in Figure 17 are clear about the performance impact induced by the *SHõWA* framework when all the user-transactions and calls are intercepted. In this case it is not observed an increase in response time because, as can be seen from figure containing the throughput analysis, the server throughput is severally affected and almost all the requests failed due to HTTP timeout errors.

From the figure it is also perceptible that a substantial part of the performance impact is due to the data persistence. In this scenario the response time increased between 4.9 milliseconds and 431.3 milliseconds per request. The server throughput has reduced in 163 requests per second. Likewise it can be seen that the number of intercepted calls is an important factor for the performance. In the "full-profiling with selective" scenario the response time increased between 1.2 milliseconds and 120 milliseconds per request. The server throughput impact lies between 0 and 60.5 requests per second depending on system load.

As shown by the results, intercept all transactions and corresponding calls imposes a severe performance degradation. The selective approach reduced the performance impact and now we expect to get even better results by combining adaptive and selective algorithms. Before proceeding with such analysis we measure the performance impact induced by other application-level monitoring tools.

5.3.3 Comparing the performance impact induced by different application-level monitoring tools

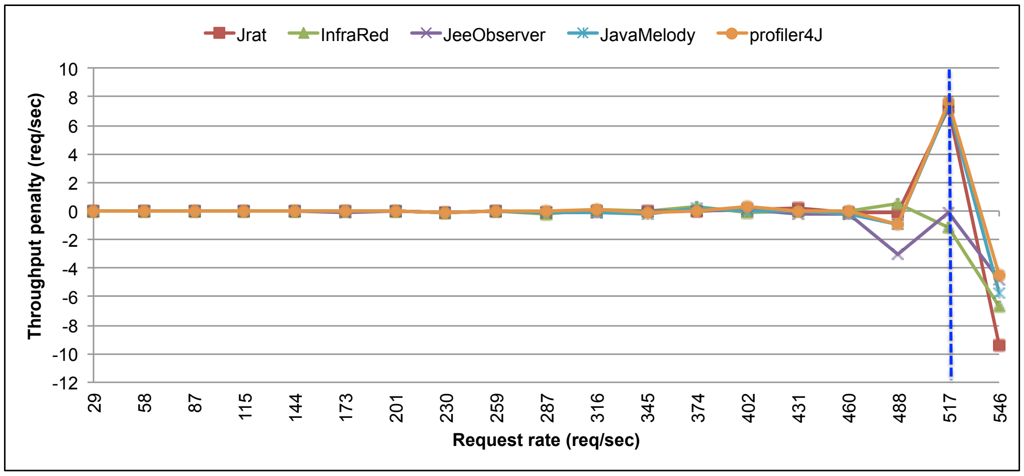
In this section we present the performance impact induced by different application-level monitoring tools. For the comparison we used the JRat, InfraRed, JeeObserver, JavaMelody and profiler4J tools. The results are illustrated in Figure 18.

As in the previous analysis, the results presented in Figure 18 represent the absolute difference between the baseline values and the response time and server throughput results obtained with each application-level monitoring tool.

Generally the results show that the performance penalty induced by these tools is low. For example, InfraRed (tool which obtained the best results) induce a delay in response time between 0 and 2.7 milliseconds (depending on system load). The throughput impact is only 1 request per second, when the server is near to the maximum capacity. These results are decidedly better than those obtained with the *SHõWA* framework prior to use adaptive and selective monitoring algorithms. A reason for the low performance impact observed is, as described next, the lower portion of code covered by these tools.

5.3.4 Code coverage analysis

The amount of source code covered by an application-level monitoring tool is important for runtime anomaly detection. Scenarios like application changes or remote service changes can interfere with the application execution and lead to performance anomalies difficult to detect and pinpoint. A possible solution to cope with this type of scenarios is to profile the application call-tree and keep track of the calls execution time.



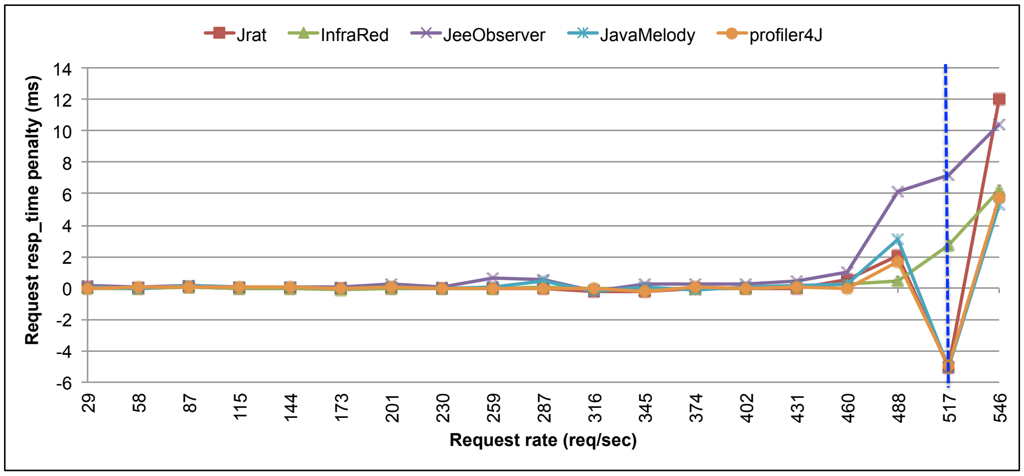


Figure 18: Throughput penalty (top) and response time penalty (down) induced by different application-level monitoring tools

To measure the amount of executed code that is intercepted by the application-level monitoring tools, we have executed a test-run and used a code coverage tool (EMMA [EMMA]). EMMA provides a report with the number of source code lines and methods touched during the test-run. With its output we performed a line-by-line and method-by-method comparison to account for how many of the executed lines and methods are intercepted by the application-level monitoring tools. The result is summarized in Table 10.

The set of methods or constructors to be intercepted by the JRat, InfraRed, JeeObserver, JavaMelody and profiler4J tools are predefined and once defined it is not possible to change them at runtime. While useful to reduce the performance impact induced, as presented in Table 10, these tools do not intercept a considerable number of calls

Table 10: AOP-based tools: Code coverage analysis

	% lines	**% methods**
SHõWA Sensor module	81.2%	100%
Best coverage scenario (JRat, InfraRed, JeeObserver, JavaMelody, profiler4J)	2.4%	3.7%

inhibiting the detection of anomalies, particularly when these are motivated by the calls that are out of the monitoring scope.

The separation of concerns provided by the AOP paradigm provides an easy and clean way to monitor a large portion of code being executed. The AspectJ pointcut `call(* *(..))` allows to intercept all the method-calls without requiring any knowledge about the application implementation details. As indicated above, the *Sensor* module implemented in the *SHõWA* framework takes advantage of this pointcut to intercept all the methods (call-tree) involved in the user-transactions execution. The biggest challenge arises in performance penalty induced. Hence the proposal for the adoption of mechanisms to dynamically adjust the monitoring subsystem without losing coverage and simultaneously without damaging the system performance.

5.3.5 Performance impact induced by the *SHõWA* framework with adaptive and selective monitoring

By adapting dynamically the number of times a user-transaction is intercepted for profiling, and the list of calls that will be intercepted, we expect a significative reduction on the performance impact induced by the application profiling.

To evaluate the performance impact induced by the *Sensor* module and the advantage of using adaptive and selective monitoring we decided to compare the performance impact induced while using constant sampling frequencies ($\boldsymbol{K = 80}$, $\boldsymbol{K = 2}$) versus the adoption of adaptive monitoring.

The upper limit for the sampling frequency we used was $\boldsymbol{K = 80}$, i.e., the call tree for each of the user-transactions will be captured every 80 times the user-transaction executes. This value was chosen because we performed some earlier experiments and observed that it provides a low performance penalty. In practice it can be any number such as, under the expected application behavior, the performance penalty is minimal. This number will then be adjusted by the adaptive algorithm. The lower limit for the sampling frequency used was $\boldsymbol{K = 2}$ (the call tree for each of the user-transactions will be captured in every 2 times the user-transaction executes). By using $\boldsymbol{K = 2}$, there are 50% of the calls which are not profiled, allowing to verify if the performance anomaly persist over the time periods.

The list of calls to be intercepted is updated every time the $\boldsymbol{K}$ decreases or when the

$\boldsymbol{mod(NumExecutedTrans, K, R) = 0}$. $\boldsymbol{R = 20}$ was used as the default selective refresh factor for all of the experiments.

The results are illustrated in Figure 19.

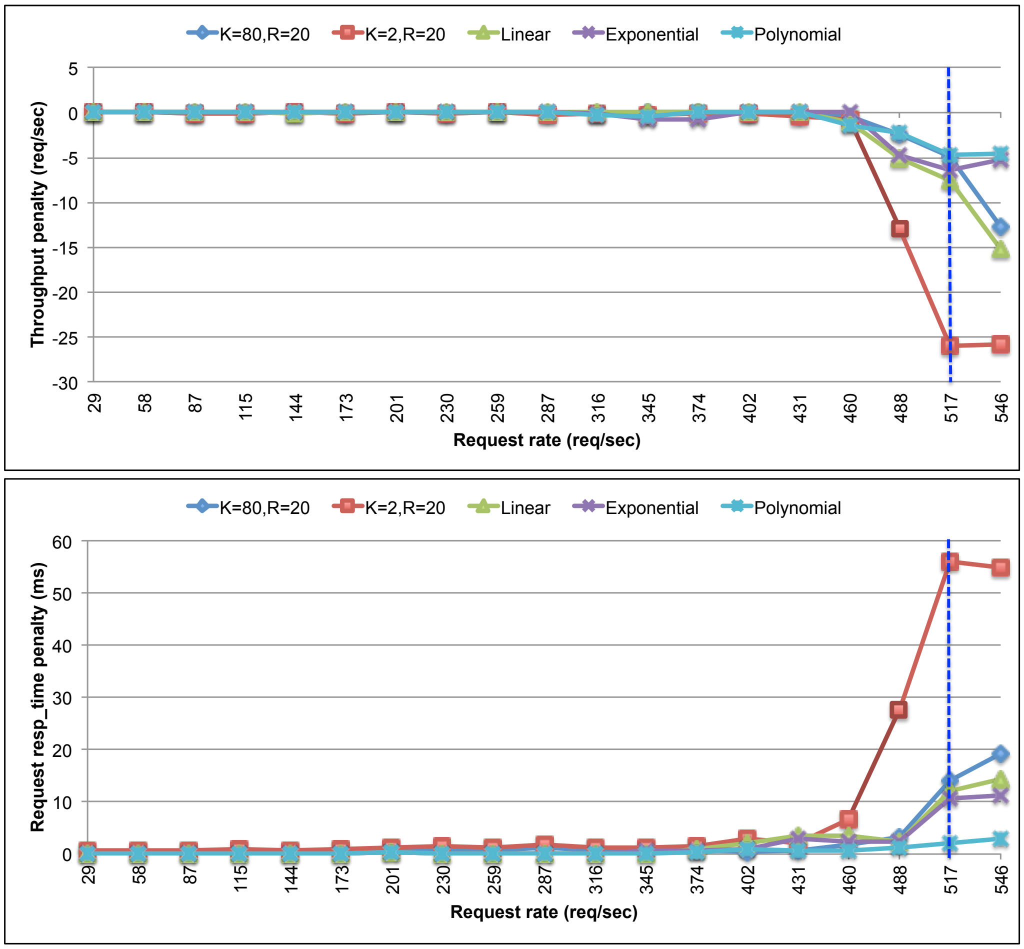


Figure 19: Throughput penalty (top) and response time penalty (down) considering the adoption of adaptive and selective monitoring

The results highlight the role of sampling and the use of adaptive and selective algorithms for application-profiling:

- When $\boldsymbol{K = 80}$ (sampling frequency) and $\boldsymbol{R = 20}$ (refresh factor), the response time penalty is small and the server throughput penalty is low: the response time penalty did vary between 0.6 milliseconds, when the number of concurrent requests is low and 14 milliseconds when the server is close to its maximum capacity. The throughput reduction reached 5 requests per second, when the server is near to the maximum capacity;

- When $\boldsymbol{K} = \mathbf{2}$ and $\boldsymbol{R} = \mathbf{20}$, the response time penalty did vary between 0.7 and 56.2 milliseconds and when the server is near to the maximum capacity, the throughput was reduced in 26 requests per second;
- With the linear adaptive algorithm, the response time penalty did vary between 0 and 12.2 milliseconds and the server throughput was reduced in 7.5 requests per second, when the server is near to the maximum capacity;
- With the exponential adaptive algorithm, the response time penalty did vary between 0 and 10.8 milliseconds and when the server is near to the maximum capacity, the throughput was reduced in 6.4 requests per second;
- The polynomial adaptive algorithm was what achieved better results: when the number of concurrent requests is low, then the response time did not vary. When the server is close to its maximum capacity, then the response delay observed was just 2 milliseconds per request. The throughput reduction reached 4.7 requests per second, when the server is operating near to its maximum capacity.

The selective property has also played an important role. According to our analysis it is responsible for a reduction of almost 96% of the calls intercepted. It should be noted that any call can, at any time, be intercepted. The *Sensor* module intercepts all the calls, since its execution time exceeds 10% of the user-transaction execution time.

5.3.6 Adaptive and selective monitoring: timely detection and pinpointing of anomalies

Despite the low performance impact achieved with the adoption of adaptive and selective monitoring algorithms, a monitoring system should be able to timely detect anomalous scenarios.

The detection latency results and the effectiveness analysis, presented in Chapter 4, are very clear about the importance of early detection. By detecting workload variations or anomalies earlier it is possible to take the appropriate recovery actions while the number of end-users affected by the anomaly is still low. To evaluate this aspect, we conducted an experimentation to verify how timely is the detection and localization of anomalies. The analysis takes into consideration the amount of data collected by the *Sensor* module when different sampling strategies are used to profile the application.

The performance anomaly scenario considered in this analysis results from an application change. It consists on running a modified version of TPC-W. In the modified version there is a user-transaction that suddenly starts performing more slowly than usual. From the analysis point of view, a sudden degradation corresponds to a worst-case scenario: when the sampling frequency is low and a sudden performance anomaly arises the amount of data available for the analysis might be not sufficient to support the pinpointing analysis, delaying the recovery and leaving more users exposed to the anomaly. The *Sensor* module adaptation is fundamental to cope with this type of degradation. It is important to refer that this anomaly is not detected by the other monitoring tools described in this chapter. It involves a call that is not monitored by those tools.

The results achieved for the slow degradation scenario are illustrated in Figure 20 and Figure 21.

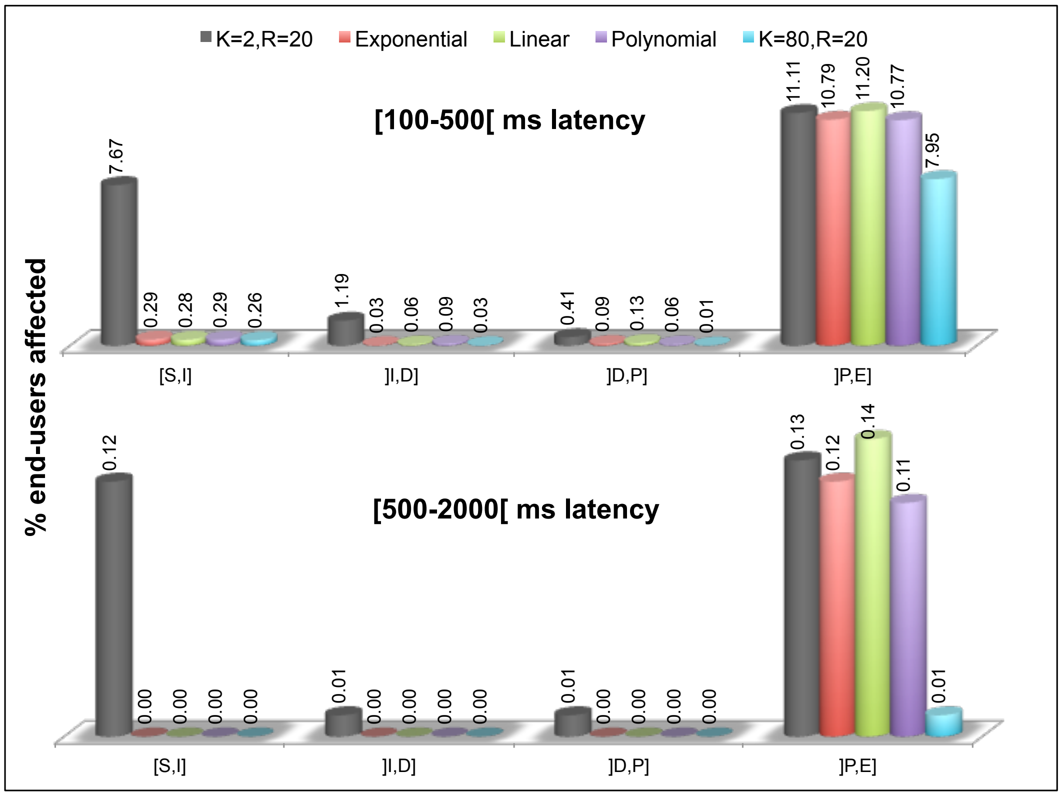


Figure 20: Slow degradation: percentage of end-users experiencing a slowdown increase between [100,500[ms and [500, 2000[ms

The figures illustrate the results considering the adoption of different adaptation strategies: exponential, linear, polynomial and fixed sampling ($\boldsymbol{K = 2}$ and $\boldsymbol{K = 80}$). They also consider four intervals of analysis: (1) the $[\boldsymbol{S, I}]$ contains the percentage of end-users affected by the response time slowdown between the Start of the experiment and the anomaly Injection period; (2) The $]\boldsymbol{I, D}]$ comprises the end-users affected between the Injection and the Detection period; (3) the $]\boldsymbol{D, P}]$ corresponds to the percentage of end-users affected between the Detection and the Pinpointing of the performance anomaly; (4) the last interval $]\boldsymbol{P}, \boldsymbol{E}]$ accounts for the percentage of users affected between the Pinpointing and the End of the experiment. The results are better as the smaller is the percentage of end-users affected.

While Figure 20 illustrates the percentage of end-users experiencing a slow response time motivated by the user-transaction change, in Figure 21 we illustrate the percentage of end-users experiencing a slow response when requesting unchanged user-transactions. This is useful to understand how the selective properties can isolate the performance penalty introduced with application-level profiling.

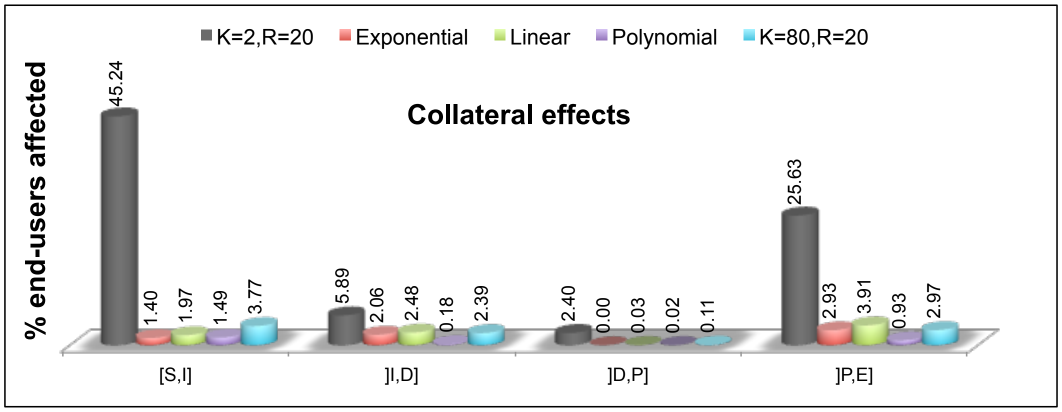


Figure 21: Slow degradation: percentage of user-transactions indirectly involved and which have experienced a response time increase larger than 100 ms

From the results presented in Figure 20 and Figure 21, is noticeable the advantage of using adaptive and selective monitoring. When $\boldsymbol{K = 2, R = 20}$ (i.e., in every 2 times a user-transaction is executed it is profiled), the percentage of end-users affected, even before the anomaly was injected, is very high.

With $\boldsymbol{K = 80, R = 20}$, or with the sampling frequency provided by the exponential, linear or polynomial algorithms, the percentage of end-users affected by the slowdown is low. Likely, as illustrated in Figure 21, the performance impact on the other user-transactions is minimal. For this anomaly scenario the difference on the percentage of end-users affected during the $]\boldsymbol{I}, \boldsymbol{D}]$ and $]\boldsymbol{D}, \boldsymbol{P}]$ periods is not significant. This happened because all the sampling frequencies have had time to collect the amount of data necessary to support the analysis.

In presence of a higher degradation rate the results are expected to be different. In this scenario, the lower the frequency of monitoring is, the smaller is the amount of data available for analysis. The results for the fast degradation scenario are illustrated in Figure 22 and Figure 23.

From the figures is clear the advantage of using adaptive and selective monitoring. The ability to adjust the sampling frequency and dynamically select the list of calls to be

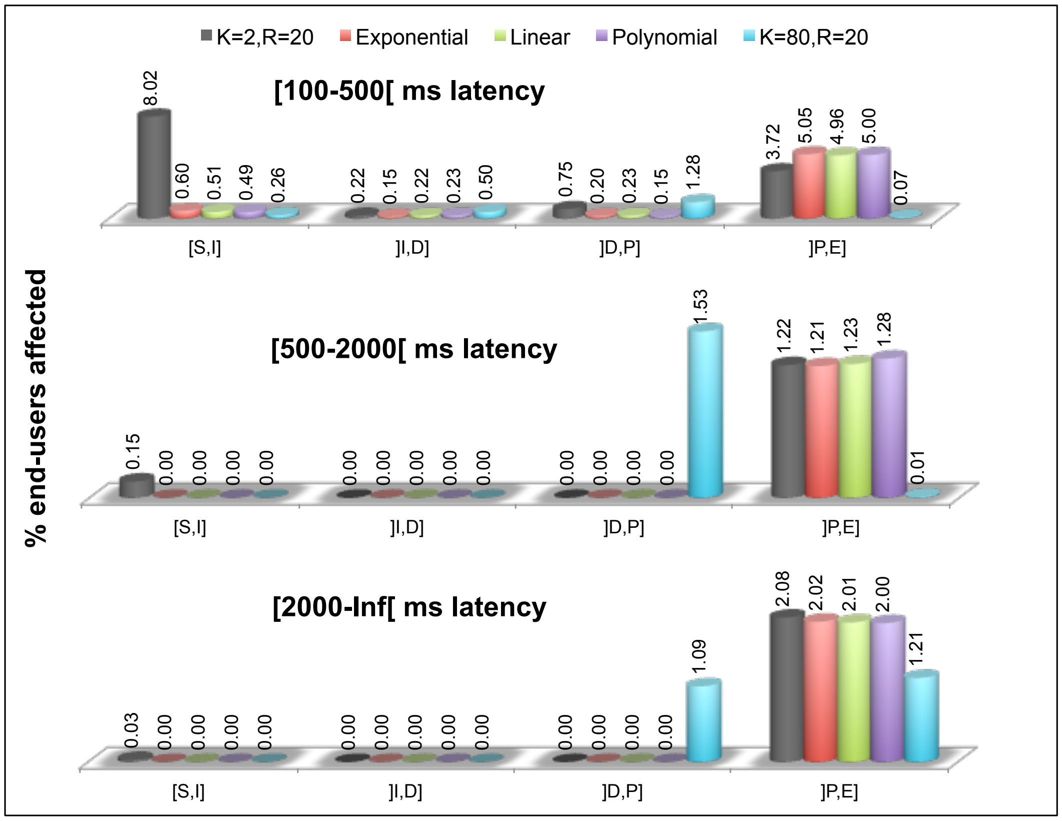


Figure 22: Fast degradation: percentage of end-users experiencing a slowdown increase between [100,500[ms, [500, 2000[ms and larger than 2000 ms

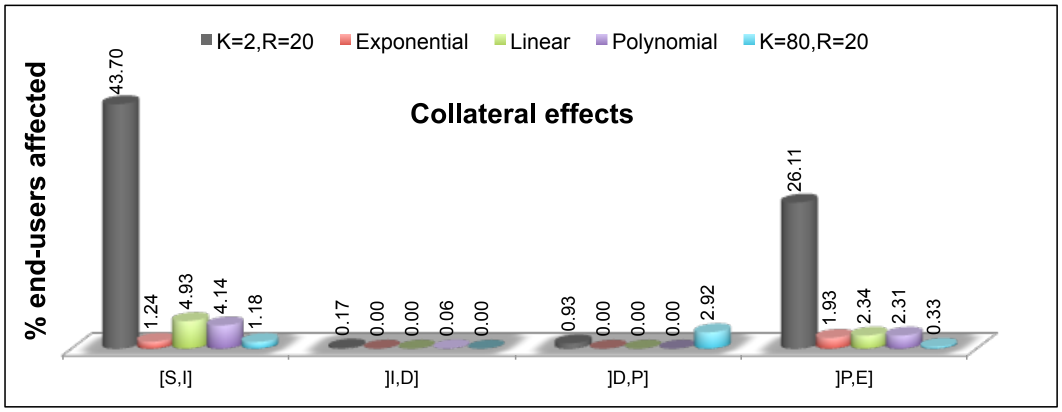


Figure 23: Fast degradation: percentage of user-transactions indirectly involved and which have experienced a response time increase larger than 100 ms

intercepted, allowed the detection and pinpointing of the anomaly while the percentage of end-users experiencing the response time slowdown is still low. When $\boldsymbol{K} = \mathbf{2}$ the percentage of end-users experiencing a slowdown was very high. That happened due to the performance impact introduced by the monitoring itself. When $\boldsymbol{K} = \mathbf{80}$ and the degradation occurs rapidly, it is required more time to collect the data necessary to support the analysis. This situation leaves more users exposed to the performance anomaly. From the three adaptive algorithms, the exponential has achieved better results. This is given to its adaptability: it keeps the sampling frequency low during more time and adjusts it very quickly when a performance anomaly is detected.

The self-adjustment of the sampling frequency and the dynamic selection of the list of calls to be intercepted allows the *Sensor* module, implemented in the *SHõWA* framework, to collect few data when there are no symptoms of performance anomaly, reducing the performance impact. In presence of symptoms of performance anomaly the sampling frequency increases. Thus it is possible to collect more data, that is then used by the *Root-cause Failure Analysis* module, to pinpoint the call (or set of calls) associated with a performance anomaly.

5.3.7 Discussion

The experimental study has described the role and importance of application-level monitoring for timely detection and localization of anomalies in Web-based applications. The experimental results achieved so far let us conclude the following:

- The performance impact introduced by the available application-level monitoring tools seems to be negligible for the application. Such low performance impact is the result of a low number of application calls intercepted. According to our analysis, these tools only intercept 2.4% of the TPC-W source code lines, which corresponds to 3.7% of invoked methods;
- Taking advantage of AOP to devise an application-level monitoring tool able to intercept 100% of invoked methods is quite easy. The challenging part is how to reduce the performance impact introduced with such approach;
- The adoption of adaptive and selective algorithms allow to intercept all the application call with a low impact on the system performance. According to our analysis:
  - In the absence of symptoms of performance anomalies, the frequency to which the user-transactions are profiled is lower. Thus, the server throughput is affected in less than 1% and the response time penalty per request is about 0.5 milliseconds (when the server load is below 75% of its maximum capacity); and 2 milliseconds (when the server load is above 75% of its maximum capacity);

  - When the sampling is set to the minimum frequency allowed, the server throughput is affected in less than 5%, and the response time penalty varies between 0.7 milliseconds (when the number of concurrent requests is low); and 56.2 milliseconds (when the server is close to its maximum capacity).

- The adaptability does not affect the ability to detect and pinpoint anomalies. The dynamic adaption of the sampling frequency and the selective approach adopted, reduce the performance impact induced by the application profiling. It allows simultaneously to collect detailed data to support the detection and localization of anomalies while the number of end-users experiencing the response time slowdown stills low.

The adoption of self-adaptive and selective techniques to reduce the performance impact induced by the application-level profiling and the ability to timely detect and pinpoint anomalies is an important achievement for the establishment of the *SHõWA* framework.

5.4 Conclusions

In this chapter we did present the self-adaptive and selective algorithms adopted by the *SHõWA Sensor* module. These algorithms are targeted for the reduction of the performance impact induced by the application profiling. The application profiling is performed at runtime to measure the response time of the calls belonging to each of the user-transactions. This measurement allows to pinpoint the call, or set of calls, associated to a performance anomaly.

In a system the interoperability between components can be high. For example, a simple user-transaction can performs queries on the database, interact with remote services and invoke several system call functions. In this context, accessing which of the calls is related with a performance anomaly is fundamental to determine the root-cause of the problem and decide on the most appropriate recovery plan.

The results presented in this chapter are a motivation for the development of a self-healing framework for Web-based production environments. The adoption of adaptive and selective monitoring algorithms allows to profile the application at runtime without compromise its performance. The ability of these algorithms to self-adapt the application-profiling frequency, depending on the detection of performance anomalies and the implementation of a monitoring system that do not require application changes or previous knowledge about its implementations details is an important aspect. It automates the monitoring tasks and reduces the need for human intervention.

In the next chapter we present a study that focus the ability to detect and pinpoint anomalies achieved by the *SHõWA* framework.

Chapter 6
Detecting and pinpointing anomalies with *SHõWA*

KEY POINTS

- ◇ The ability to detect and pinpoint anomalies in runtime is a central aspect of any monitoring or self-healing system. It involves the analysis of the collected data to detect the occurrence of anomalous situations that may cause a negative impact on the normal functioning of the service.
- ◇ In the context of the Web-based applications, the detection of anomalous behaviors can be explored in different ways. Systems capable of modeling the interactions between the application components and detect changes in those interactions, systems that are attentive to the manifestation of errors and systems capable of detecting changes in the performance of the application, are some examples of the approaches explored by the scientific community.
- ◇ In this work we focus on the detection and localization of performance anomalies. Our aim is to detect performance anomalies, in which the end-users are being affected by a degradation of the quality of service .
- ◇ The data analysis for the detection of performance anomalies presents several challenges. It should distinguish between performance variations due to changes in workload and performance anomalies. It should verify if the performance anomaly is driven by changes at the system level, changes in the application server or changes in the application. It should detect and pinpoint anomalies in a timely manner. The detection of anomalies must have a high precision rate. And, in the case of a self-healing system, the detection and location of anomalies must be as autonomous as possible, minimizing the need of human intervention;
- ◇ In this chapter we evaluate how *SHõWA* can contribute to address the challenges listed above.

A monitoring system comprises three fundamental aspects: monitoring, events/alerts and actions. The monitoring is responsible for the data gathering and data analysis process. The events/alerts are triggered when one undesired condition is met. The actions are used to handle with the event, promoting the recovery of the system state.

In the context of online monitoring, the events and actions taken by a monitoring system are considered fundamental entities to detect and answer to the unexpected behaviors that affect the quality of the service. To support the detection of service deviations are adopted techniques, that range from *reactive* approaches based-on predetermined and static thresholds, to more sophisticated techniques that adopt different types of analysis to continuously access the system or application behavior.

Considering the increasing complexity of the systems and applications, define rules to cope with the different aspects that may affect the quality of the service is a daunting task. For this reason, the existence of techniques that are able to learn the expected behavior of the system and that are capable to detect service deviations are considered more appropriate. These techniques are more able to cope with the miscellany of factors that may influence the dependability of the systems or applications. They are also important to reduce the human dependency on defining the controls that should be meet to keep the service fully operational.

In this chapter we evaluate the approach used by the *SHõWA* framework to detect and pinpoint anomalies in Web-based applications. The evaluation consists on an experimental study with two benchmark applications and different types of anomalies.

6.1 Anomaly detection and localization

In the literature we can find several research projects focusing the detection and pinpointing of anomalies in Web-based applications. The approaches followed by those projects can be roughly divided in three types:

1. Attentive to the usage pattern;
2. Responsive to errors manifestation;
3. Watchful of performance variations.

6.1.1 Usage pattern analysis

The usage patterns analysis aims to observe how the system or service is used and how much it differs from the expected usage, that was previously defined, or learned. The usage patterns can be observed with respect to the resources consumption, user's behavior or level of interaction between the components. The type of analysis is commonly based-on time series models, machine learning and structural models, i.e., models watchful to changes such as a different execution path taken by the transactions.

Li et al., in [Li 2002], describe a measurement approach based-on time series analysis to detect the aging phenomena and to estimate the time-to-exhaustion of the system resources. They considered several parameters, from an Apache Web server and Linux O.S., collected during a long time period of time, and represented in a time series format. The statistical Autoregressive Moving Average (ARMA) model was applied, to analyze and forecast the time series data and estimate the time-to-exhaustion of the resources. Based-on the predicted time-to-exhaustion some recovery actions can be taken, reducing the time and efforts to detect and recover the system in case of failure.

Efforts to improve the anomaly detection and localization, and reduce the Mean-Time-To-Repair (MTTR) are also presented in [Bodíc 2005]. Based-on the available web page access logs, the authors modeled the normal behavior of the requests of the 40 mostly accessed web pages. Then, they use a chi-square test to compute a vector containing the web page hit counts during a time interval and analyze if the hit count follows the same distribution as the modeled data. If the chi-square significance test is higher than 0.99, then the interval being analyzed is marked as anomalous and an anomaly score is assigned to the interval. The analysis is complemented by a Naïve Bayes classifier that was trained using two different methods: unweighted learning and weighted learning. In the unweighted learning, the mean and variance is estimated with the assumption that every previous time interval is normal. In the second method, the mean and variance is weighted by the probability that is normal, thus, the more anomalous a time interval appears, the less it is incorporated into the model of the normal behavior. Both methods have some disadvantages: the first will absorb the anomalous time periods, treating them as normal and the second may require more time to adjust to new steady state that are normal.

Another usage pattern methodology is presented in [Candea 2006], [Chen 2004] and [Kiciman 2005]. The authors propose a monitoring improvement by modeling the execution path and the component interactions for posterior analysis. To detect failures, the analysis consists on comparing the historical reference model with the current usage pattern. In [Candea 2006] the interaction between a component and a set of other components is modeled through a weight value, representing the proportion of runtime call paths that enters or leaves the component. At runtime, the chi-square test is used to detect changes in the weights of the links. In [Chen 2004] the path-shape is modeled and used to detect failures in the path structure. The path-shape is represented as a call-tree structure, and a set of grammar rules are deduced by means of probabilistic context-free grammar (PCFG) analysis. The PCFG for normal path behaviors is a set grammar rules inferred from the component interactions represented in the path-shape, and the probability that any given component will call a particular set of other components. The normal path behaviors are compared with the runtime path behaviors and an anomaly is detected when the expected probability differs from the observed probability. The work presented in [Kiciman 2005], combines the analysis presented in [Chen 2004] and [Candea 2006]. The interactions between an instance of a component and each class of components in the system is represented as a set of weighted links, i.e., the proportion of runtime paths that enter or leave a component through each interaction. Weight link deviations between a single component's behavior and the reference value is considered an anomaly. Besides the analysis of interaction between components, the detection of anomalies is complemented with a path-shape analysis. While the components interaction analysis observes the components behavior, the path-shape analysis provides an orthogonal view of the system, allowing to detect anomalies in the individual paths. The path-shape was represented as a call-tree structure, and, like in [Chen 2004], a set of grammar rules were deduced by probabilistic context-free grammar (PCFG) analysis: the path-shape and the corresponding probabilistic value of a given sentence (path) occur. To observe whether a subsequently observed path-shape is anomalous, each transition is compared with its corresponding rule and a total anomaly score is determined by the difference between the probability of the observed transition and the expected probability for the transition. The PCFG analysis was used to build a decision tree. This tree is used to localize the components associated with the anomaly.

The Pinpoint project, presented in [Chen 2002], is a good example in how the structural analysis can be efficient to timely detect and pinpoint the anomalies, reducing the downtime in case of a failure. This project relies on a modified version of the application server [Candea 2003a] to record the request execution paths and build a model of the expected interactions between components. In runtime, differences between the expected and observed interactions are regarded as failures and recovery procedures are initiated. Regarding to the experimental results, authors have concluded that Pinpoint has a very high accuracy and precision for single-component faults, but not a so good accuracy when there are three or more component-faults. In [Jiang 2006] is presented another approach based-on structural changes. By using commonly available monitoring data, authors extracted the system invariants, i.e., the flow intensities that hold during all the time and vary according to the number of user requests. These invariants corresponds to the number of interactions that occur between the components while the requests are being processed. Abrupt changes in those invariants are considered symptoms of problems. From the conducted experimentation, using a real three-tier system and simulating different types of faults, authors show that invariants do exist in distributed transaction systems and that changes in those invariants are useful to interpret the healthiness of operational systems.

6.1.2 Error manifestation analysis

The error manifestation and error propagation analysis is explored by several authors to detect and pinpoint the components responsible for a failure in a Web-based application.

In [Li 2007] is presented a study focusing the types of failures occurring in open source J2EE implementations. This study reveals that two-thirds of the failures analyzed have thrown an exception or error message. This result emphasizes the exception and error analysis, as an important mean to monitor, recover or repair from application-level failures.

Candea et al., in [Candea 2003b], used a modified version of the JBoss application server to discover the set of exceptions declared in each component. These exceptions were then completed with other exceptions corresponding to environmental faults that could occur. Using reflection facilities, the authors have modified the runtime behavior of the application, by injecting some exceptions to build the failure propagation maps. At runtime, these failure maps are used to establish the causality between a failure and the component. That work enhances the ability to determine the minimal subset of the system that must be recovered.

In [Bellur 2007] the failure dependencies are modeled for root-cause isolation of anomalies in J2EE environments. This is achieved from: the application topology graph (a set of components within their dependencies); a usage strength value that represents the probability of the component $\boldsymbol{A}$ use the component $\boldsymbol{B}$ in the application; and a failure propagation probability derived from a static analysis and that represents the probability of an exception $\boldsymbol{E_1}$ propagates from one component $\boldsymbol{A}$ to another component in the form of an exception $\boldsymbol{E_2}$. The information is modeled using the Bayesian belief network methodology. The model is then used to infer the failure states of system components. This approach benefits from the previously failure knowledge, so as more data about the failures becomes available, better will be the a-priori failure propagation probabilities included in the Bayesian belief network and consequently better will be the root-cause isolation analysis.

6.1.3 Performance variation analysis

Both, tracking structural changes and model the error manifestation are interesting approaches but they are unable to detect situations when the low performance of a component causes a strong impact on the service. In this context, tracking a *customer-affecting metric* (such as the user-transactions response time) is useful to monitor the service. Along with the *customer-affecting metric*, tracking system or environmental metrics enhances the detection and anomaly localization ability.

The work presented by Cherkasova et al., in [Cherkasova 2008] and [Mi 2008] is an example where the performance variation analysis is taken into consideration to detect performance anomalies. The authors combine a regression model and an application signature to identify if a response time variation is due to a workload change, due to an application change or due to an anomaly. The regression-based model is used to model the CPU demand of the application transactions across different time segments. The anomalous segments describe the scenario where the observed CPU utilization cannot be explained by the application workload. During the modeling phase these segments are deliberately removed from the regression-based model to avoid corruption on the regression estimations of the segments with normal behavior. The application performance signature is used to uniquely reflect the application transactions and their CPU requirements, independently from the workload types. While the regression-based approach is useful to accurately detect a difference in the CPU consumption model of application transactions and alarms about a performance anomaly or a possible application change, the application performance signature allows to compare the new application signature against the old one and detect which of the transactions was affected by a performance change.

The Magpie framework [Barham 2003] is also an interesting work in the area. It adopts fine-grain analysis to characterize the transaction resource footprints in fine detail. Then, clustering and probabilistic state machine techniques are used to observe performance changes and identify which events, or event sequences, are behind an anomalous behavior. Preliminary results presented by the authors shown that Magpie has good potential, however extra work needs to be done, in particular, to deal with high intra-requests concurrency.

The data analysis included in the *SHõWA* framework fits the performance variation analysis approach. From the above works, the ones more related to our approach are [Cherkasova 2008] and [Mi 2008]. The *Workload Variation Analysis* and the *Performance Anomaly Analysis* modules keep track of the correlation between the response time and the mix and load of user-transactions. If a misalignment is observed (the users-response time is not aligned with the workload), then the *Anomaly Detector* module, verifies if there is any system or application server change that is correlated with the response time variation. Simultaneously, the *Root-cause Failure Analysis* module measures the degree of association between the transactions response time and the set of calls participating in the transaction servicing, in order to pinpoint the call, or set of calls, associated with a performance change. This analysis allows to verify if a performance variation is motivated by an application or remote service change (e.g., database, Web Service).

6.2 *SHõWA*: overview of the data analysis used to detect and pinpoint performance anomalies

The data analysis performed by the *SHõWA* framework targets the detection and localization of performance anomalies avoiding, to the extent possible, the negative consequences that those anomalies have for the business.

The data analysis relies on the data collected by the *SHõWA Sensor* module and prepared by the *Data Preparation* module. The *Sensor* module adopts AOP bytecode instrumentation to collect data online. It gathers data with different granularities: *user-*

transaction level and *profiling* level. In the *user-transaction* level the user-transactions are intercepted and the server response time is measured ($\boldsymbol{endtime - starttime}$). Simultaneously, a set of system and application server parameters is collected. This monitoring mode is always enabled, i.e., all the transactions processed by the application server are intercepted. In the *profiling* level, is measured the execution time of each call (methods execution time), belonging to each of the user-transactions. The *Data Preparation* module aggregates the data in periods/intervals and groups the mix of user-transactions by a key.

The analysis is provided by the *Workload Variation Analysis*, *Performance Anomaly Analysis*, *Anomaly Detector* and *Root-cause Failure Analysis* modules. These modules kept track of the statistical correlation between two variables $\boldsymbol{(X, Y)}$ with the purpose of: (1) verify if a performance variation is due to a workload change or if it is a symptom of performance anomaly; (2) verify if there is a system or application server change associated with the performance anomaly; (3) verify if there is an application or remote service change associated with the performance anomaly.

For each time interval of analysis, it is built a bi-dimensional dataset. The dataset contains the historical values of each variable ($\boldsymbol{X}$,$\boldsymbol{Y}$) and it is grouped by the workload mix key. The vectors $\boldsymbol{X}$ and $\boldsymbol{Y}$ are then used in different contexts of analysis.

In the following items we present an overview about the data analysis included in the *SHõWA* framework and present the threshold values used to detect and pinpoint anomalies.

- *Performance Anomaly Analysis* module: To detect if a performance variation is a symptom of performance anomaly: $\boldsymbol{X}$ is a vector of accumulated response time, aggregated per period and user-transaction, and $\boldsymbol{Y}$ is a vector containing the absolute number of transactions processed per period and user-transaction. The Spearman's rank correlation coefficient - $\boldsymbol{\rho}$ - is measured and, according to the Algorithm 4 presented in Chapter 3, the impact of dissociation between the response time and the number of user-transactions processed is determined. According to Table 2, a value of dissociation equal or higher than 10 corresponds to a response time increase of approximately 500 milliseconds or to a decrease of more than 50% in the number of requests. Taking this values in consideration, in the experimental study presented in this chapter a performance anomaly is detected when the value of dissociation is greater than or equal to 10.

- *Workload Variation Analysis* module: To detect if a performance variation is motivated by a workload contention: $\boldsymbol{X}$ is a vector of accumulated response time, aggregated per period and $\boldsymbol{Y}$ is the total number of requests waiting in the application container queue. If a strong correlation is verified, then it means that there are requests waiting in the application server queue, suggesting the existence of an extra load. The extra load could be detected together with the occurrence of a performance anomaly. If this happens, it becomes clearer the impact that the extra load has on the user-transactions response time. According to preliminary tests, a $\boldsymbol{\rho}$ variation greater than 0.1 degrees is indicative of the existence of at least twice the requests waiting in the server queue, compared to the normal behavior.

- *Anomaly Detector* module: To verify if there is a system or application server parameter associated with a workload change or a performance anomaly: $\boldsymbol{X}$ is defined as the total number of user-transactions per period and $\boldsymbol{Y}$ is the cumulative value of each parameter (e.g., amount of JVM memory used) per period. If the number of requests increases, then the cumulative value of each parameter should also increase. If, for some reason, the cumulative value of a parameter increases, and that is not motivated by an increase in the number of requests, then the $\boldsymbol{\rho}$ degree will decrease, highlighting the parameters associated with the performance anomaly. According to preliminary tests, a decrease superior to 0.1 degrees is sufficient to identify the parameter(s) associated with a performance anomaly.

- *Root-cause Failure Analysis* module: This module uses the *profiling* level data, collected by the *Sensor* module. For each user-transaction intercepted and corresponding list of calls (methods executed), the data is organized as follows: $\boldsymbol{X}$ is defined as the frequency distribution of the transactions response time and $\boldsymbol{Y}$ as the processing time frequency distribution of the calls belonging to the transaction call-path. When a performance anomaly is detected, affecting one or more user-transactions, the vectors $\boldsymbol{X}$ and $\boldsymbol{Y}$ are submitted to a correlation analysis, via the Spearman's rank correlation $\boldsymbol{\rho}$ [Zar 1972]. If, from the result of the analysis, it is found a call or set of calls, whose correlation index has risen, then it means that there is a correlation between the user-transaction response time increase and the processing time of the calls. Through the call list highlighted by analysis, or more specifically with the call signatures highlighted, it becomes possible to verify whether the delay in response time is due to any recent application change or to a remote service change (e.g., the invocation of a Web Service, the interaction with the database).

6.3 Detecting and pinpointing anomalies with *SHõWA*

Made a summary on the operation of the *SHõWA* framework, in this section we present an experimental study to assess the detection and pinpointing ability provided by the framework. More specifically we seek answers for the following questions:

- *When there is a performance variation was it caused by a change in the workload* *or is it a result of some internal anomaly?*

- *How to distinguish between a performance anomaly and workload variations?*
- *How to obtain useful information about the potential root-cause?*
  - *Is the performance anomaly motivated by a resource contention in the server?;*
  - *Is the performance anomaly motivated by an application server contention?;*
  - *Is the performance anomaly associated with an application change?;*
  - *Is the performance anomaly associated with a remote service change?.*
- *How timely is the detection and pinpointing of anomalies?;*
- *How accurate is the detection and pinpointing of anomalies?;*

To answer these questions we prepared a test environment and conducted an experimental study. The study considers two benchmark applications and different anomaly scenarios. The anomalies were injected on both of the benchmarking applications. To facilitate the presentation, the detection and pinpointing of anomalies, provided by *SHõWA*, are presented by means of figures and tables.

6.3.1 Benchmark applications

For the experimental study we have adopted two J2EE benchmarking applications: TPC-W [Smith 2001] and RUBiS [Cecchet 2002].

6.3.1.1 TPC-W

The TPC-W benchmark simulates the activities of a retail store website. It defines 14 user-transactions which are classified as either of browsing and ordering types.

The TPC-W benchmark uses a MySQL database as a storage backend and contains its own workload generator that emulates the behavior of end-users.

The database can hold 1,000, 10,000, 100,000, 1,000,000 and 10,000,000 books or product items. For each of the emulated users, the database maintains 2880 customer records and the corresponding orders data. Thus, as the number of emulated users increases, the workload and the database size increases accordingly.

TPC-W includes three workload mixes. A browsing mix, where 95% of the web pages are browsing pages and 5% of the web accesses are to ordering pages. This mix places more pressure on the front-end Web servers, image servers and Web caches. In the ordering mix, 50% of the web pages accessed are browsing pages and the other 50% are ordering pages. This mix puts more pressure over the database server. In the shopping mix, 80% of the accesses are to browsing pages and 20% are to ordering pages.

6.3.1.2 RUBiS

The RUBiS (Rice University Bidding System) benchmark application models an auction site. Loosely modeled on eBay, it defines 26 user-transactions. Among the most important user-transactions are: browsing items by category or region, bidding, buying or selling items, leaving comments on other users and consulting one's own user page. Browsing items also includes consulting the bid history and the seller's information.

The RUBiS database contains 33,000 items for sale, distributed among 40 categories and 62 regions. A history of 500,000 past auctions is kept and there is an average of 10 bids per item (330,000 entries in the bids table). The users table has 1 million of entries and it is assumed that the users give feedback for 95% of the transactions. The tables containing the new and old comments holds about 31,500 and 475,000 comments, respectively.

RUBiS includes two workload mixes: a browsing mix made up of 100% of read-only interactions and a bidding mix that includes 15% of read-write interactions and 85% of read-only interactions.

6.3.2 Workloads

The TPC-W contains its own workload generator that simulates the activities of a business oriented transactional Web server. The workload exercises a breadth of system components associated with such environments and reports metrics, such as, the response time and the server throughput. It applies the workload via emulated browsers (RBE) and allows the execution with different concurrent users, three different traffic mixes, different session length, user think-times and ramp-up/ramp-down periods. The RBE simulates the users think-time, by sleeping for a number of seconds based upon a distribution with an average think-time of 7 seconds and a maximum of 70 seconds.

According to the TPC-W specification the number of emulated browsers and workload mix is kept constant during the experiment. However, since real e-commerce applications are characterized by dynamic workloads, we created workloads that change the number of users and traffic mix during the execution. The workload duration is 8400 seconds, and while it is executing the workload mix changes at every 5 to 15 minutes and the number of emulated-users varies between 30 and 500.

Like TPCW, RUBiS includes a client-browser emulator. It defines a session as a sequence of interactions for the same customer. The client emulator opens a persistent HTTP connection to the Web server and closes it at the end of the session. Each emulated client waits for a certain think-time before initiating the next interaction. The next interaction is determined by a state transition matrix that specifies the probability to go from one interaction to another. The think-time and session time for all benchmarks are generated from a negative exponential distribution with a mean of 7 seconds and 15 minutes, respectively. For this study we have adopted the RUBiS bidding mix workload and each client node has emulated 150 users.

6.3.3 Testbed

The environment used for the experimental tests is illustrated in Figure 24. It is composed by a typical three-tier architecture: the clients interact with the application through a Web browser; the requests are processed by an application server; the application server interacts the database server(s) where the application data is stored.

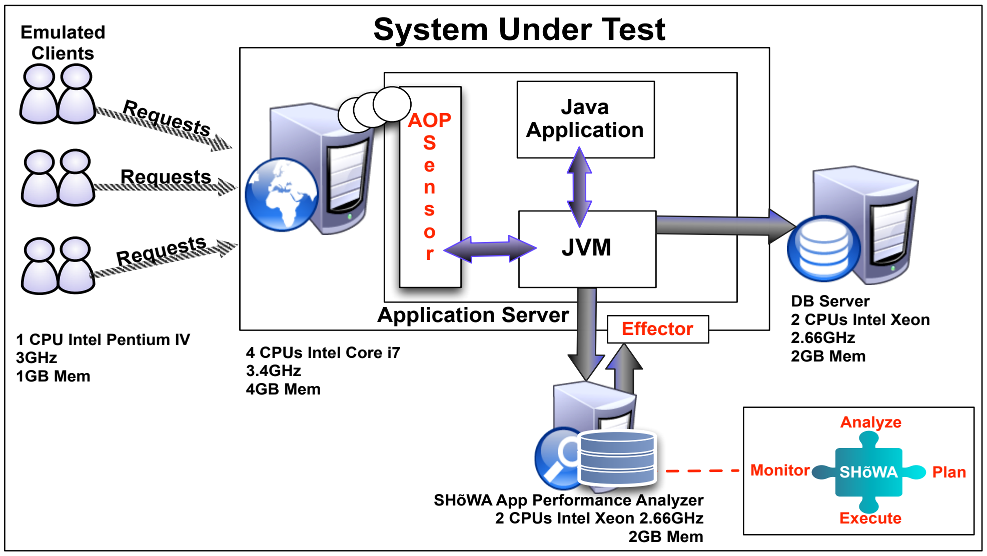


Figure 24: Test environment

The System Under Test consists of an application server (Oracle Glassfish Server [Oracle Glassfish Server]) running the benchmarking applications. Only the application under test is deployed in the server. The user requests are simulated using several emulated clients. The benchmarking application interacts with a MySQL database as a storage backend. The *Sensor* module consists on a Java Archive File (JAR) that is placed in the application server libraries folder. It is loaded with the application server startup, and it contains the AOP pointcuts and the code necessary to collect the system, application server and application parameters. As the data is collected, it is sent to the "*SHõWA* Application Performance Analyzer" server. The "*SHõWA* Application Performance Analyzer" server contains the *Data Preparation*, *Workload Variation Analysis*, *Performance Anomaly Analysis*, *Anomaly Detector* and *Root-cause Failure Analysis* modules. It runs on a separated server to avoid the performance impact motivated by the data analysis process.

The test environment is composed by several machines. It includes four workload generator nodes with 3GHz Intel Pentium CPU and 1GB of RAM each. The "*SHõWA*

Application Performance Analyzer" machine includes two 2.66GHz Intel Xeon CPUs and 2GB of RAM. The application server machine includes four 3.4GHz Intel Core i7 CPUs and 4GB of RAM and the database server machine includes two 2.66GHz Intel Xeon CPUs and 2GB of RAM. All the nodes run Linux with 2.6 kernel version and are interconnect through a 100Mbps Ethernet LAN. The operating system, the TCP/IP and the application server parameters were adjusted to the recommended settings.

6.3.4 Anomaly scenarios

If the performance changes are due to workload variations, then this is a normal situation and the most that can be done is to work on the dynamic dimensioning of the system to cope with more requests or different workload demands. However, if the performance change is caused by some anomaly, then it would be mandatory to raise some inspection and recovery actions. These actions may avoid a visible performance slowdown and/or anticipate the probability of a failure.

To evaluate the ability of the *SHõWA* framework to detect and pinpoint performance anomalies, and distinguish performance anomalies from workload variations, we decided to test five different scenarios. These scenarios include some of the software failures included in the report "*Causes of Failure in Web Applications*" [Pertet 2005]. The anomalies were chosen in order to cover situations in which the applications are affected by a system overload, resource exhaustion or failed software upgrades.

The anomaly scenarios used in this study include:

Scenario A - CPU load: In this scenario the application performance is affected by a CPU contention. A CPU contention scenario can have multiple origins. Phenomenas like software aging [Garg 1998], unplanned or untested changes (O.S. or application upgrades, backup's window, maintenance/systems operations) can lead to CPU consumption scenarios potentially affecting the user-transactions response time.

Scenario B - workload contention: In this scenario the number of user-transactions starts increasing. A situation like this might occur due to a denial of service attack and it may cause a contention on the application server queue, and consequently, the rejection of new requests. Detecting scenarios of workload variation, with potential impact on the user-transactions response time, is of utmost importance to improve the application availability.

Scenario C - remote service change: In this scenario the database starts performing slowly and affects the application performance. The occurrence of adhoc queries directly on the database without concerning their optimization and performance or missing of indexes are examples of scenarios that may affect the performance of the database server and, consequently, the application behavior.

Scenario D - memory consumption: In this scenario we simulate a memory leak. During the experiment, the amount of heap memory available in the application server is consumed, forcing the garbage collector to be more intervenient and causing a performance degradation in the system. If all the memory is consumed, then an out-of-memory error is issued and the application server hangs. Memory leaks commonly occur in several scenarios. In many cases, the effects of a memory leak fault are subtle but if it is not detected it will, sooner or later, lead to a failure.

Scenario E - application change: In this scenario we simulate a performance anomaly motivated by an application change. As mentioned in [Pertet 2005], software upgrades represent a large number of non-malicious failures. These upgrades may introduce software bugs (e,g,. logic errors, deadlocks, race conditions) or lead to performance degradation, either because of bad programming, either by unexpected behavior on interoperability between modules.

These scenarios were injected in both of the benchmark applications: TPC-W and RUBiS. In Table 11 we present the configuration details about the anomaly scenarios.

Table 11: Anomaly scenarios - configuration details

Anomaly	Level	Description
CPU load	System	O.S. processes that provoke a gradual CPU consumption
Workload contention	Application server	Add, to the executing workload from 50, up to, 1500 new requests per second
Remote service change	DB	Miss of database indexing (RUBiS) and ad-hoc queries executed at every minute (TPC-W)
Memory consumption	Application server	Consume 1MB of JVM heap memory per second
Application change	Application	An user-transaction is affected by a gradual response time delay

The results about the detection and pinpointing of anomalies, achieved by the *SHõWA* framework, are presented in the next subsections.

Since the anomalies were detected and pinpointed by *SHõWA* in a very similarly manner in both of the benchmark applications, we only present the results obtained for each anomaly. The results presented for the CPU load, workload contention and remote change, concern the adoption of TPC-W as benchmark application. The results presented for the memory contention and application change, concern the adoption of RUBiS as benchmark application.

6.3.5 Scenario A: CPU load

In this experimentation we expect that our framework detects a performance anomaly and identifies the CPU load as the potential cause. Since a CPU contention may affect the response time of several user-transactions and inner components, the *Root-cause Failure Analysis* module may spot a set components that change together with the user-transactions response time.

To impose load on the system we used the `stress` [Stress] tool. This is a very simple tool and it was used to evaluate how the *SHõWA* framework performs, considering a degradation on the transaction response time motivated by a gradual CPU consumption. The benchmark application used in this experimentation was TPC-W. The CPU consumption has started 60 minutes after the beginning of the experiment.

In Figure 25 we present the results provided by the *Performance Anomaly Analysis* module. The anomaly was injected around period 588. The dissociation between the response time and the number of user-transactions was detected around period 645. The `TPCW_home_interaction` was the first user-transaction affected by the anomaly.

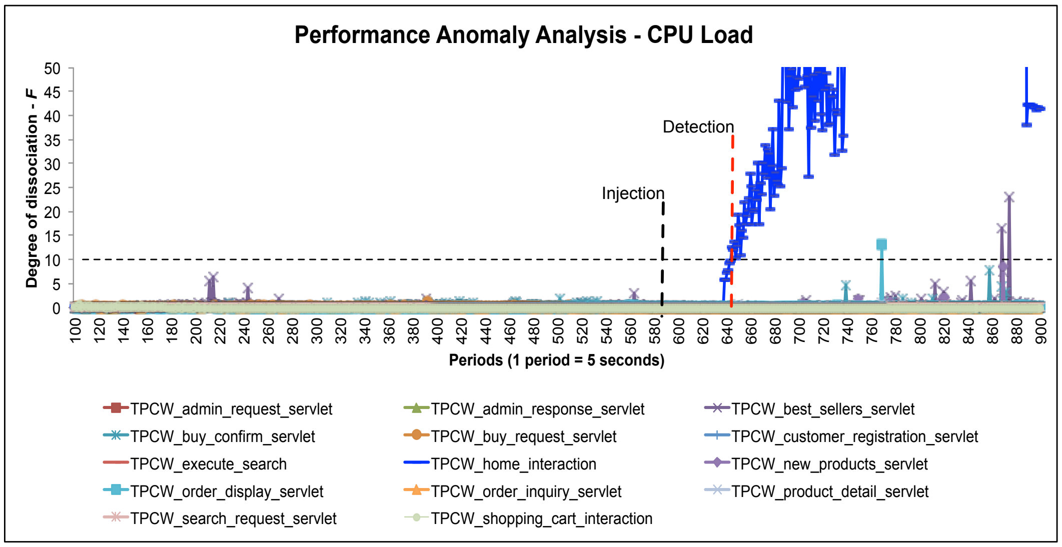


Figure 25: CPU Load - Anomaly Detection Analysis: dissociation between the response time and the number of user-transactions processed

At this stage the dissociation detected might be due to a workload change, a resource contention or application change scenario. The *Workload Variation Analysis* module aims to detect if there is a workload variation that may be causing the performance slowdown. The result about the workload variation analysis is illustrated in Figure 26. The Spearman's rank correlation corresponds to the degree of association between the response time and the number of requests in the application server queue.

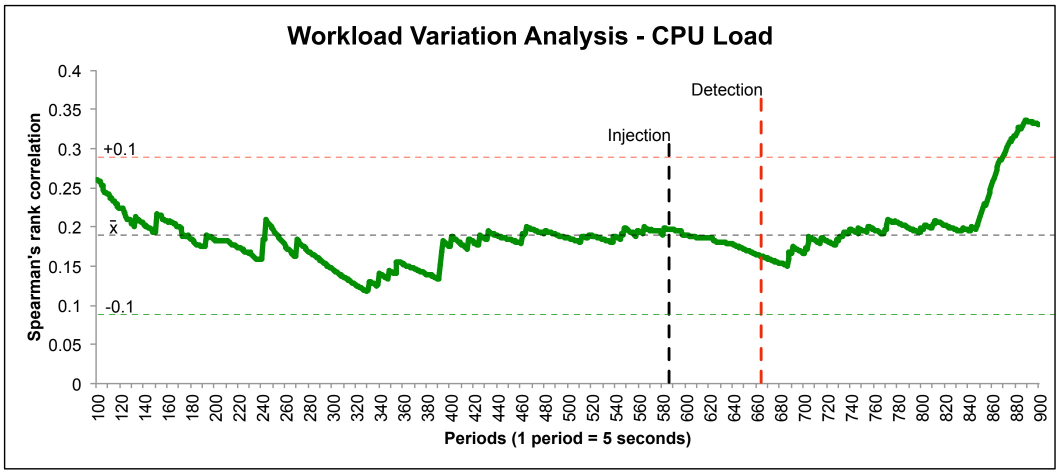


Figure 26: CPU Load - Workload Variation Analysis: dissociation between the response time and the number of requests in the application server queue

From the results illustrated in Figure 26, the correlation degree is low and remains almost stable across the periods of analysis. This means that the response time is not being impacted by the number of requests in the application server queue. The next step is performed by the *Anomaly Detector* module. This module verifies if there is any system or application server contention related with the performance variation observed. The results are illustrated in Figure 27.

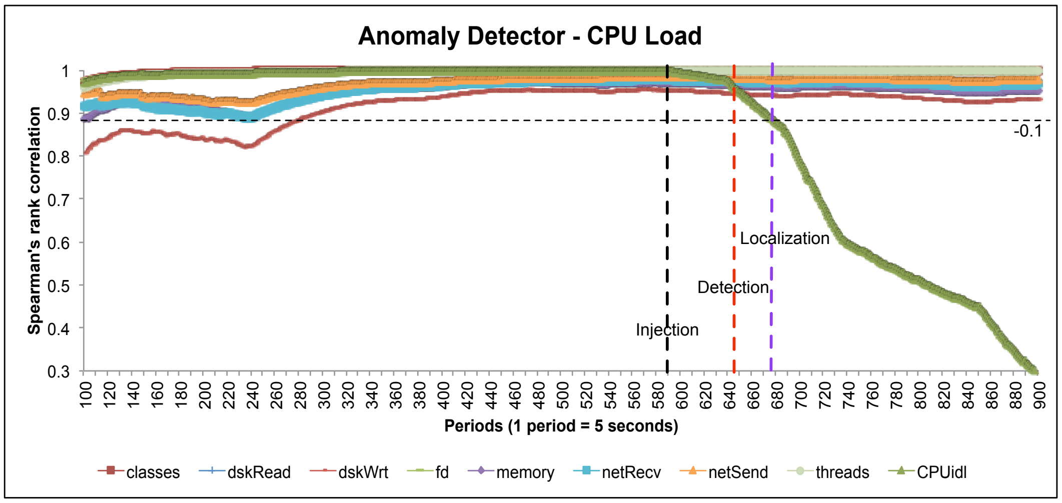


Figure 27: CPU Load - Anomaly Detector: identify if any of system and application parameters is associated with the performance variation

The Spearman's rank correlation, illustrated in Figure 27, denotes a dissociation between the number of requests processed and the amount of CPU used. Considering the sequence of analysis provided by the *SHõWA* framework so for: it was detected a response time variation, the variation is not associated with a workload contention; there is a significative change in the CPU load (period 667) that suggests that CPU is influencing the user-transactions response time.

When a performance anomaly is detected, the *Root-cause Failure Analysis* module verifies if there is any application call, or set of calls, associated with the performance anomaly. Using the *profiling* data, collected by the *Sensor* module, the *Root-cause Failure Analysis* module has picked all the calls that belongs to the `TPCW_home_interaction` user-transaction to perform the correlation analysis. The results are illustrated in Figure 28.

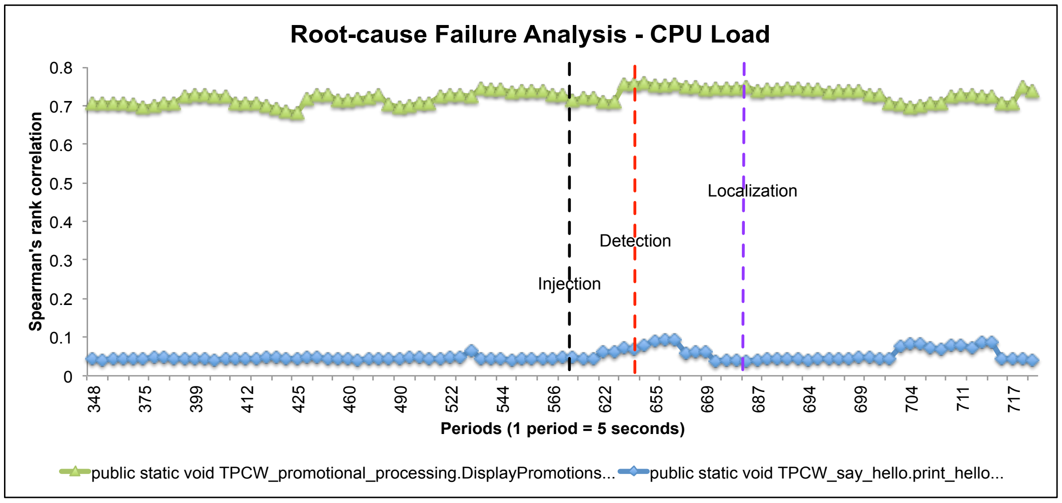


Figure 28: CPU Load - Root-cause Failure Analysis: list of application calls associated with the performance anomaly

The stability verified (before the anomaly was injected and until its localization) in the Spearman's rank correlation, illustrated in Figure 28, means that the response time of the calls belonging to the `TPCW_home_interaction` user-transaction call-path are not influencing the response time of the user-transaction. Such results allow us to conclude that the performance anomaly was motivated by a CPU consumption, and that the consumption was not motivated by the application. Excluded this possibility, the focus of attention goes to the operating system where, in fact, there exist some processes consuming the CPU.

6.3.6 Scenario B: workload contention

A contention motivated by an application workload change might occur due to a denial of service attack or when some special event (e.g., commercial promotion) occurs.

In this experimentation, TPC-W was used as the benchmarking application and after 45 minutes, counting from the start of the experiment, the number of users requesting the `TPCW_home_interaction` has started to increase gradually.

In Figure 29 we illustrate the results provided by the *SHõWA Performance Anomaly Analysis* module.

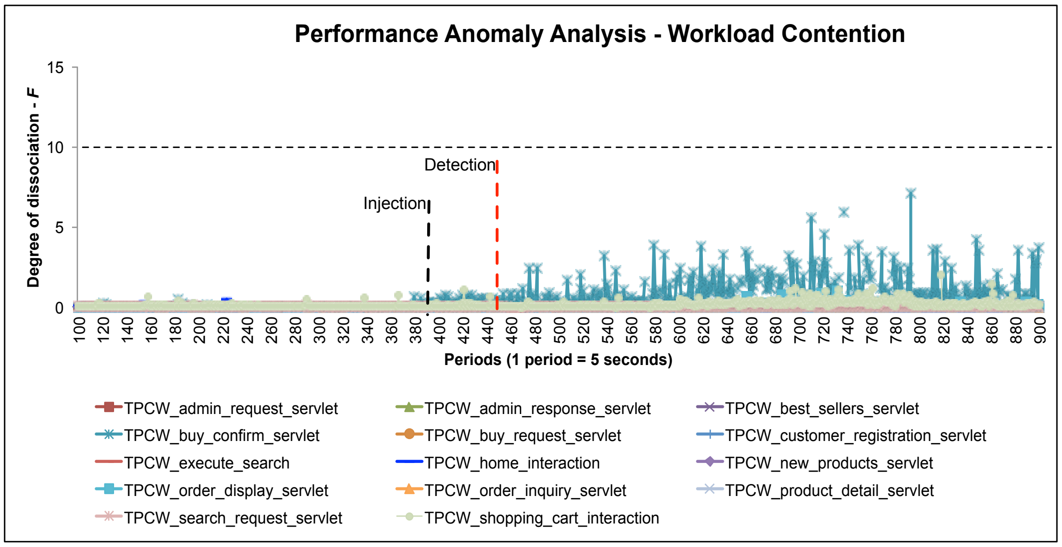


Figure 29: Workload contention - Anomaly Detection Analysis: dissociation between the response time and the number of user-transactions processed

As illustrated in Figure 29, the degree of dissociation between the number of transactions and the corresponding processing time do not goes over 10 degrees. According to the threshold we defined, this means that there are no symptoms of performance anomaly. This occurs because, independently from the state of the application server queue, the requests processed were not affected by a significant response time delay.

When analyzing the workload variation the result is different. As illustrated in Figure 30, after the number of requests starts increasing (period 389), the Spearman's rank correlation has also increased, making known the existence of a relationship between the response time and the number of requests waiting in the application server queue.

The *Workload Variation Analysis* module has detected the workload contention around period 443. The results provided by the *Anomaly Detector* are illustrated in Figure 31. The results allow to verify if the increasing number of requests is due to a

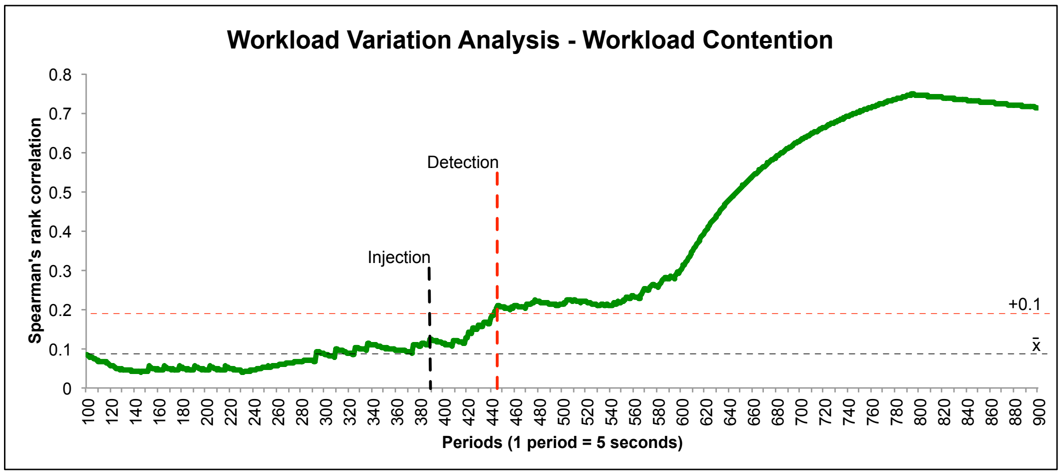


Figure 30: Workload contention - Workload Variation Analysis: dissociation between the response time and the number of requests in the application server queue

system or application server contention.

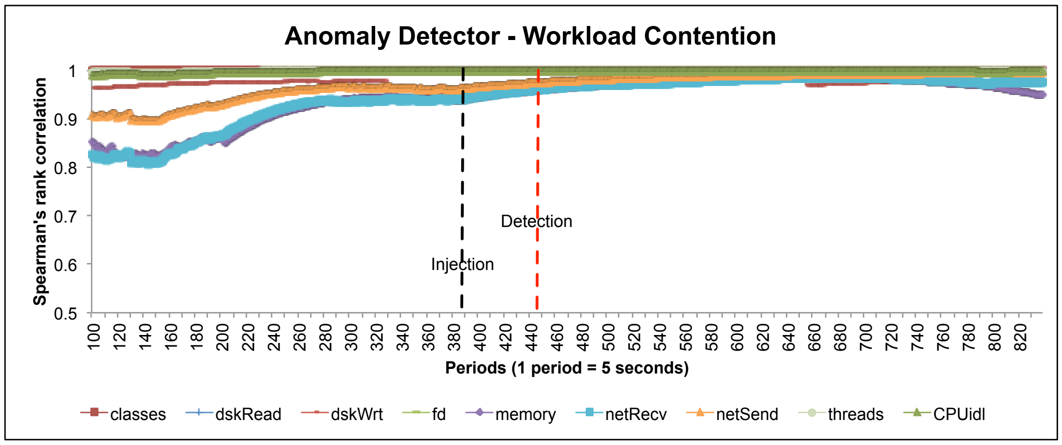


Figure 31: Workload contention - Anomaly Detector: identify if any of system and application parameters is associated with the workload variation

The results illustrated in Figure 31, more specifically the Spearman's rank correlation stability observed between the number of requests processed and the system and application server parameters, allow us to say that the workload contention is not motivated by a system or application server contention. According to the results, the server did not introduce a slowdown during the processing phase. Thus, according to the analysis, it can be said that the root-cause for the workload contention is the number of requests waiting in the application server queue, and that this problem is not caused by any apparent resource contention.

6.3.7 Scenario C: remote service change

In this scenario we have executed several ad-hoc queries directly on the database server. These queries affect the application performance. From the analysis provided by the *SHõWA* framework we expect to observe if: (1) it detects some performance degradation affecting the user-transactions response time; (2) such performance anomalies are not explained by changes at O.S. or application server levels; (3) the *Root-cause Failure Analysis* module is able to identify which calls/components are associated with the observed performance anomaly.

We used TPC-W for this analysis.

The anomaly was inject around period 425 and, as illustrated in Figure 32, an indication of performance anomaly affecting several user-transactions was detected around period 426.

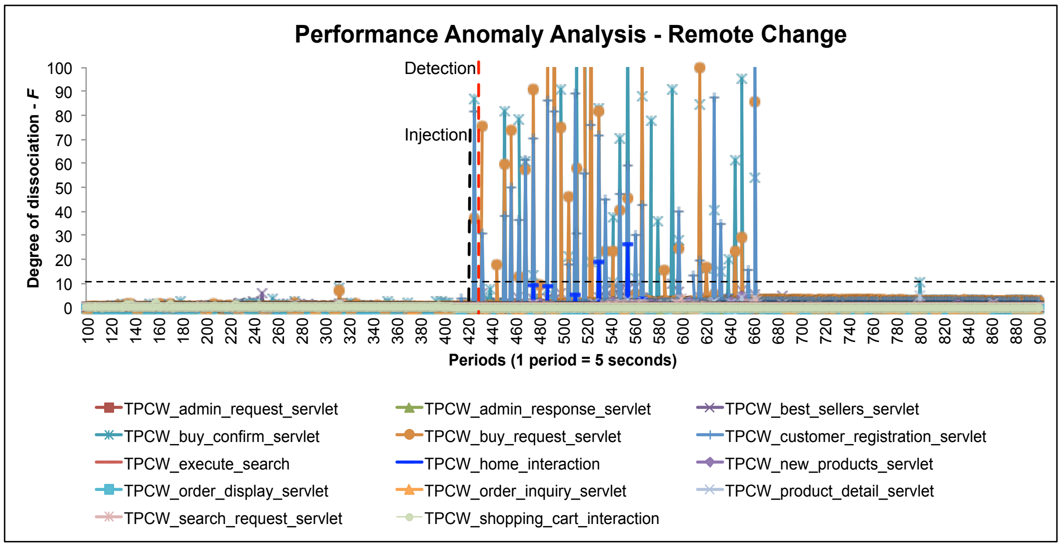


Figure 32: Remote change - Anomaly Detection Analysis: dissociation between the response time and the number of user-transactions processed

The workload variation analysis is illustrated in Figure 33. From the results is visible that, between the anomaly injection period and its detection, the degree of association between the response time and the number of requests in the application server queue does not exceed 0.1 degrees. The degree exceeds 0.1 only around period 530, i.e., almost 10 minutes after the anomaly was detected. This means that the response time variation was not motivated by a workload contention. The workload contention was a consequence.

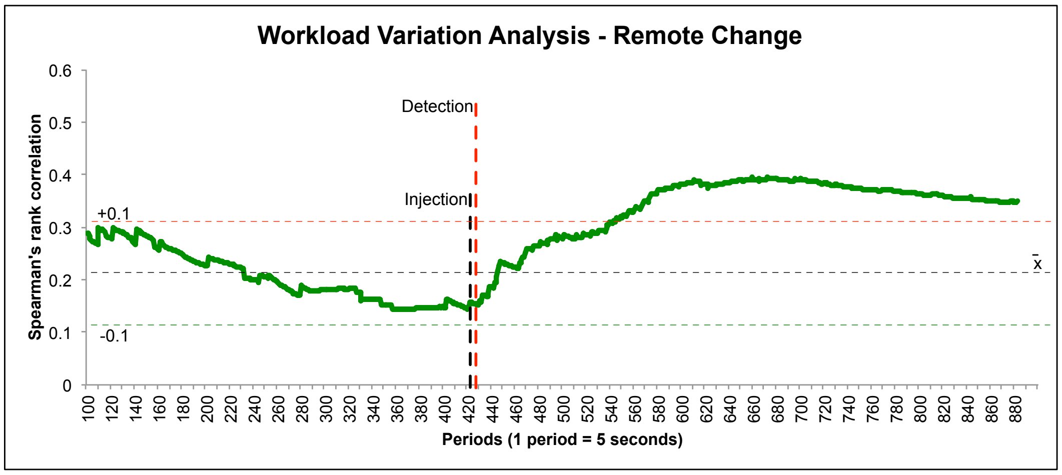


Figure 33: Remote change - Workload Variation Analysis: dissociation between the response time and the number of requests in the application server queue

The next step of the analysis consists on verifying if there is any system or application server change associated with the response time variation. The result is illustrated in Figure 34.

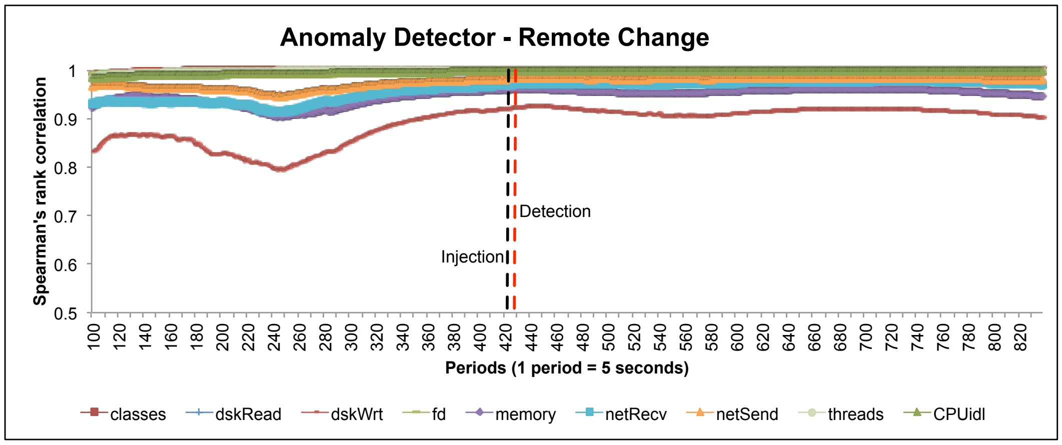


Figure 34: Remote change - Anomaly Detector: identify if any of system and application parameters is associated with the workload variation

Since the cause of the anomaly is located at the database server the correlation analysis between the application workload and the system/application server parameters, illustrated in Figure 34, do not reveal any indication about the potential cause (Spearman's rank correlation remains large).

The *Root-cause Failure Analysis* module receives the list of user-transactions which have reported a performance variation (`TPCW_customer_registration`, `TPCW_buy_request` and `TPCW_buy_confirm`) and investigates if there is any call, or set of calls, in the call-path of these user-transactions that is contributing to the performance degradation.

The analysis performed by the *Root-cause Failure Analysis* module is done at two levels. First it observes the list of calls that are accessed directly by the user-transaction and then it follows the cascade of calls in the call-path. The results achieved by the *Root-cause Failure Analysis* module for the `TPCW_customer_registration` are illustrated in Figure 35 and Figure 36.

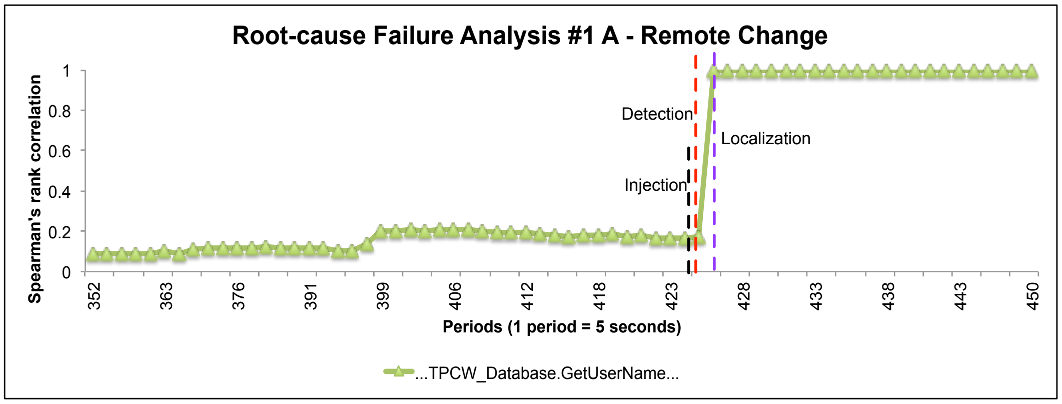


Figure 35: Remote change - Root-cause Failure Analysis: list of application calls associated with the performance anomaly - (`TPCW_customer_registration` $\longrightarrow$ `TPCW_Database_GetUserName`)

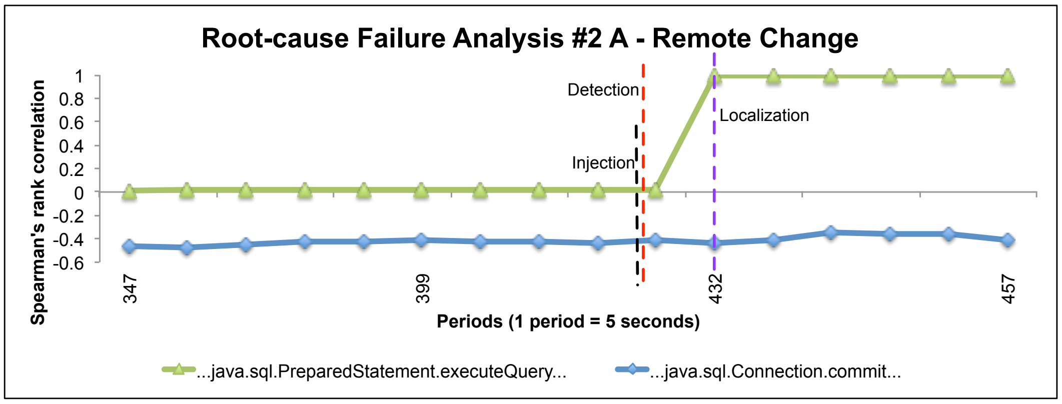


Figure 36: Remote change - Root-cause Failure Analysis: list of application calls associated with the performance anomaly - (`TPCW_customer_registration` $\longrightarrow$ `TPCW_Database_GetUserName` $\longrightarrow$ `java.sql.PreparedStatement.executeQuery`)

According to the results illustrated in Figure 35 and Figure 36, the anomaly was pinpointed around period 432, i.e., 50 seconds after it was detected by the *Performance Anomaly Analysis* module. The same results indicate that the `...executeQuery` call is highly associated with the performance slowdown observed.

The first round of results are illustrated in Figure 37. The results highlight the calls associated with the performance slowdown affecting the `TPCW_buy_request` user-transaction: `TPCW_Database_GetCustomer`; `TPCW_Database_refreshSession`.

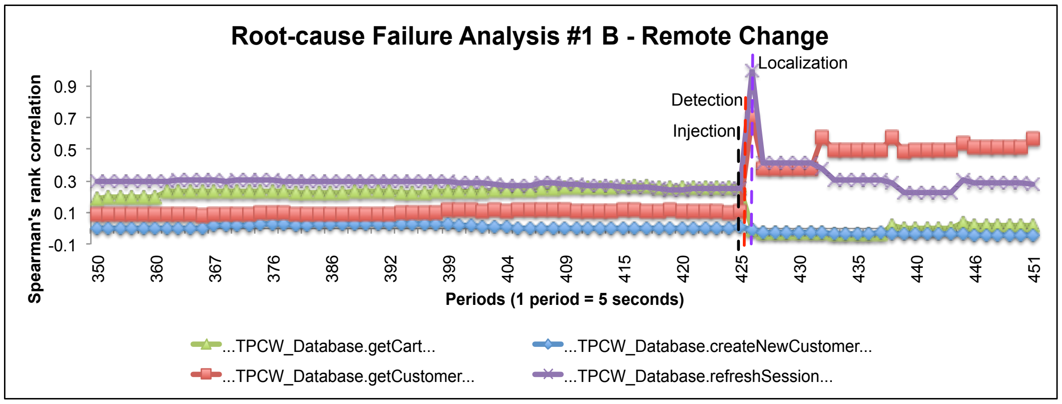


Figure 37: Remote change - Root-cause Failure Analysis: list of application calls associated with the performance anomaly - (`TPCW_buy_request` $\longrightarrow$ `TPCW_Database_GetCustomer` and `TPCW_Database_refreshSession`)

We illustrate the second round of results in Figure 38 and Figure 39.

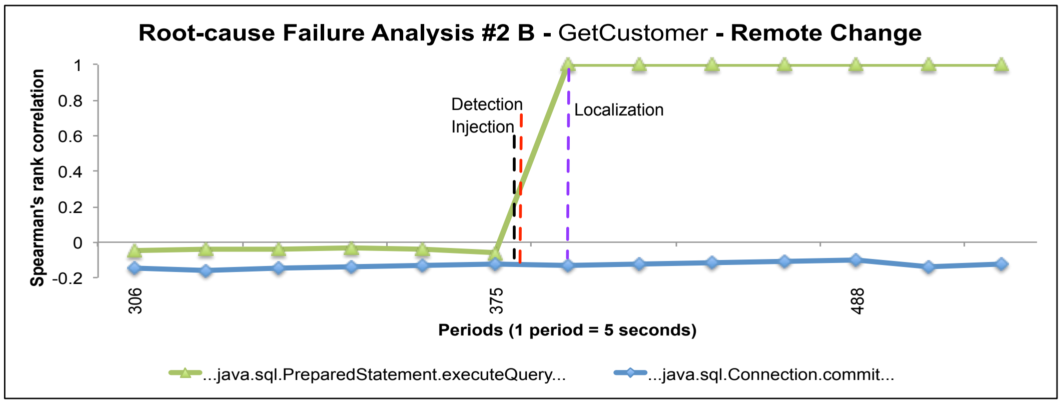


Figure 38: Remote change - Root-cause Failure Analysis: list of application calls associated with the performance anomaly - (`TPCW_buy_request` $\longrightarrow$ `TPCW_Database_GetCustomer` $\longrightarrow$ `java.sql.PreparedStatement.executeQuery`)

The results indicate the calls associated with the performance slowdown observed. These results are surprising. From the numerous list of calls in the application, the

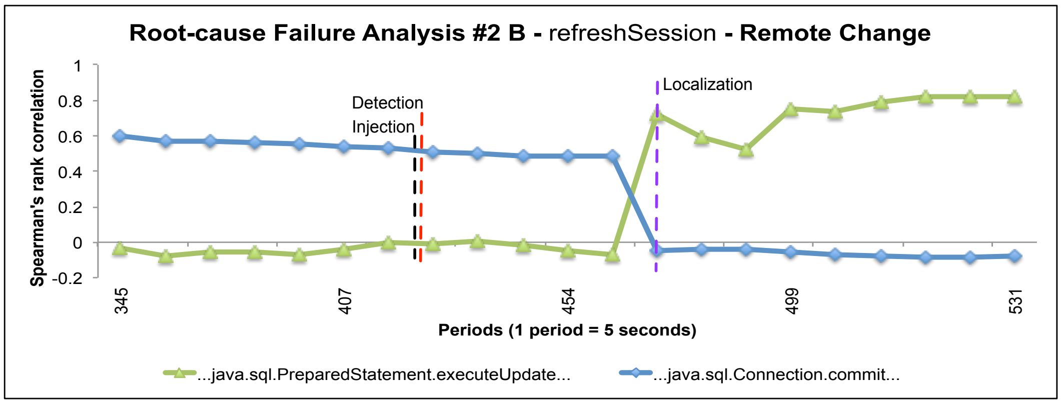


Figure 39: Remote change - Root-cause Failure Analysis: list of application calls associated with the performance anomaly - (`TPCW_buy_request` $\longrightarrow$ `TPCW_Database_refreshSession` $\longrightarrow$ `java.sql.PreparedStatement.executeUpdate`)

`..._executeQuery()` and `..._executeUpdate()` calls were pinpointed by the analysis, suggesting problems in the database server.

The results achieved by the *SHõWA* framework while analyzing the user-transaction `TPCW_buy_confirm` are very similar. The *Root-cause Failure Analysis* module has also pinpointed the `..._executeQuery()` call from the list of calls belonging to the user-transaction call-path.

6.3.8 Scenario D: memory consumption

If a program has a memory leak and its memory usage is steadily increasing, then it will sooner or later exhaust the amount of available memory causing or contributing to software aging. The memory leak symptoms include a diminish of performance that, if it is not detected and repaired in time, then all or part of the system stops working correctly and the application fails.

In our scenario, a Web-based application installed in the same application server, consumes 1 MB of memory per second. The memory consumed is never released. As consequence it originates an *"OutOfMemory"* error and the RUBiS application, that is also running under the application server, hangs.

As the amount of JVM heap memory is consumed, the garbage collection takes more time and executes more often, causing a performance slowdown. Using the *SHõWA* analysis, we expect to detect the slowdown and identify the system or application server parameter associated with it.

In Figure 40 we illustrate the results provided by the *Performance Anomaly Analysis* module. The memory consumption anomaly was injected around period 447 and it was detected by the *SHõWA* framework around period 548.

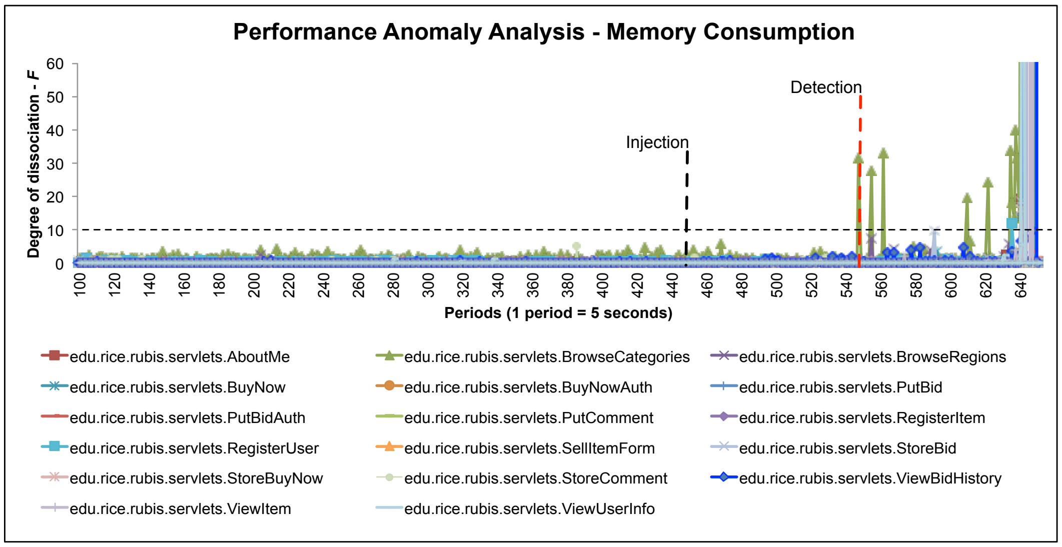


Figure 40: Memory consumption - Anomaly Detection Analysis: dissociation between the response time and the number of user-transactions processed

The *Workload Variation Analysis* module verifies if the dissociation between the response time and the number of user-transactions processed is motivated by a workload contention. We illustrate the results in Figure 41. As can be seen in the figure, the analysis do not reveal a correlation between the workload variation and the performance variation detected (the correlation degree remains low and stable).

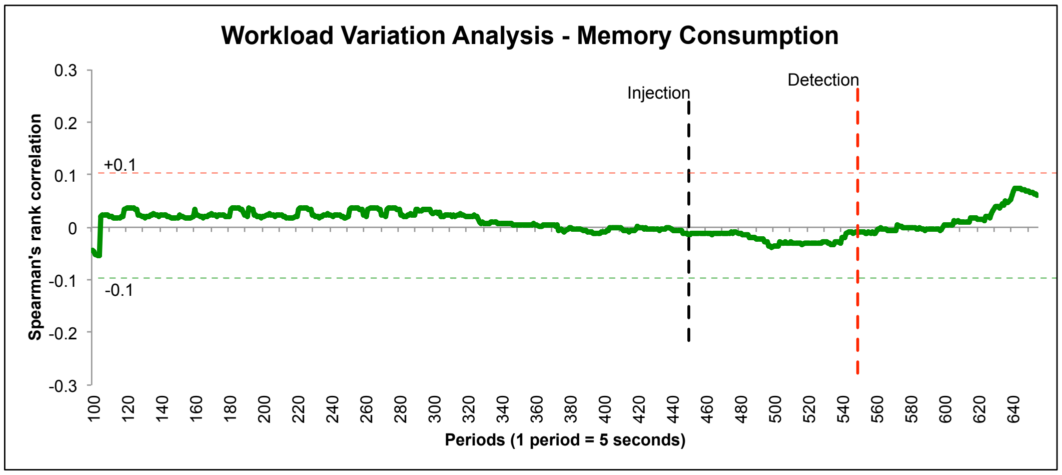


Figure 41: Memory consumption - Workload Variation Analysis: dissociation between the response time and the number of requests in the application server queue

The next step is to analyze the parameters of the system, to check whether there are some parameters related with the performance anomaly detected by the analysis. In Figure 31, we illustrate the results achieved by the *Anomaly Detector* module.

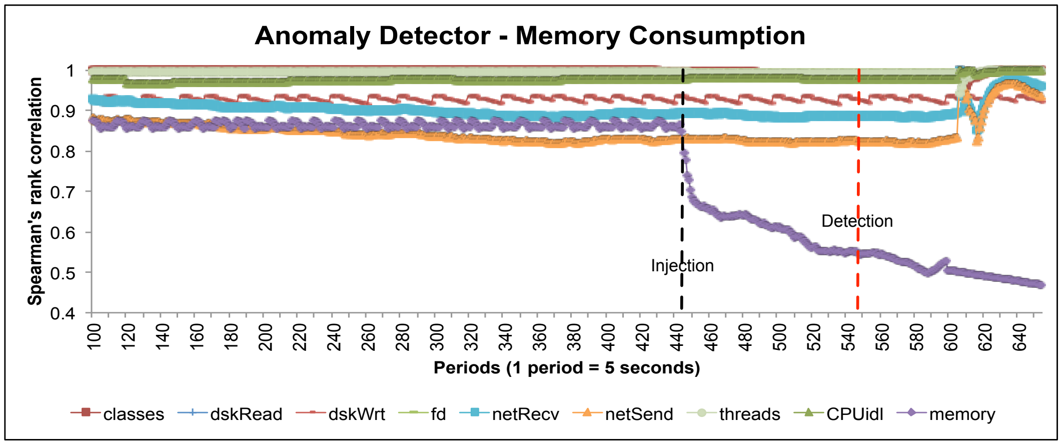


Figure 42: Memory consumption - Anomaly Detector: identify if any of system and application parameters is associated with the workload variation

From the set of system and application server parameters, as illustrated in Figure 31, the *Anomaly Detector* module has pinpointed the JVM heap memory as the parameter associated with the performance anomaly observed. Since we do not apply any recovery action, around period 642 the JVM heap memory was totally consumed and the application server has hanged.

6.3.9 Scenario E: application change

For this experimentation we manually changed the application source code to include a function into a user-transaction that sleeps for a certain amount of time. Rather than slowing down the response time continuously, this function intercalates periods of slowdown, from 10 to 10 seconds with periods of normal execution also from 10 to 10 seconds. The slowdown starts with 3 milliseconds and it is increased by an increment of 3 milliseconds in every minute. We used RUBiS for this experimentation.

From the *SHõWA* analysis it is expected that: (1) the *Performance Anomaly Analysis* module detects the user-transaction affected by the performance anomaly; (2) the *Workload Variation Analysis* module do not reveal a workload contention; (3) the *Anomaly Detector* module do not identify any system or application server parameters related with the performance anomaly; (4) the *Root-cause Failure Analysis* pinpoints the call, or set of calls, associated with the response time slowdown.

The results provided by the *Performance Anomaly Analysis* module are illustrated in Figure 43. The anomaly was injected around period 451 and the degree of dissociation between the response time and the number of user-transactions processed becomes higher than 10 around period 661.

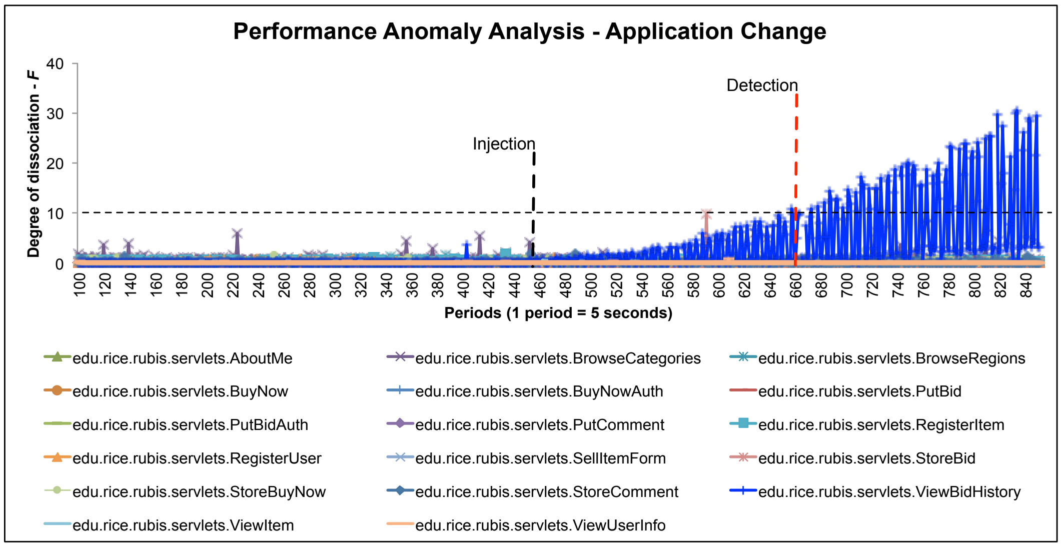


Figure 43: Application change - Anomaly Detection Analysis: dissociation between the response time and the number of user-transactions processed

To verify if the response time slowdown is motivated by some workload contention, the *Workload Variation Analysis* module evaluates if the response time is influenced by the application server queue status. In Figure 44, we illustrate the result achieved by the *Workload Variation Analysis* module.

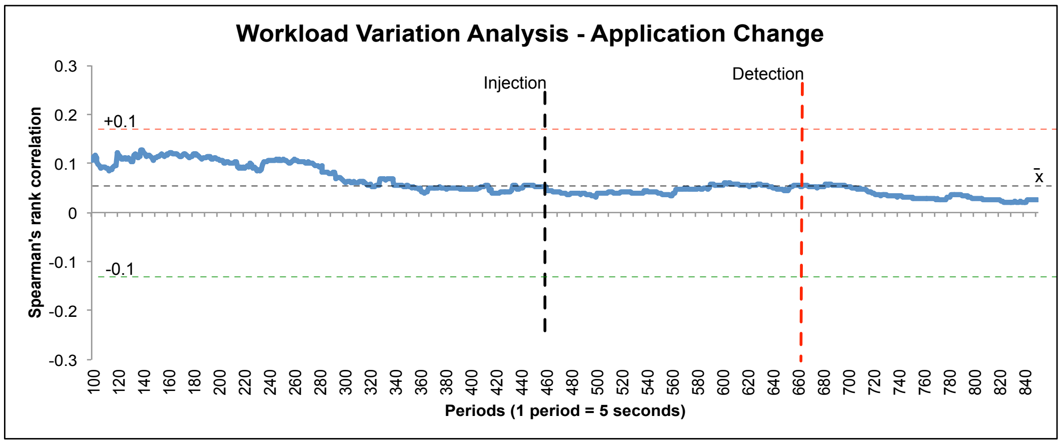


Figure 44: Application change - Workload Variation Analysis: dissociation between the response time and the number of requests in the application server queue

The results illustrated in Figure 44 show that there is no association between the response time and the number of requests in the queue. Thus it can be said that, the number of requests is not the main reason for the performance anomaly detected by the *Performance Anomaly Analysis* module. The next step regards the detection of system or application server changes. From the results, illustrated in Figure 45, is visible that the *Anomaly Detector* module did not notice a correlation degree variation.

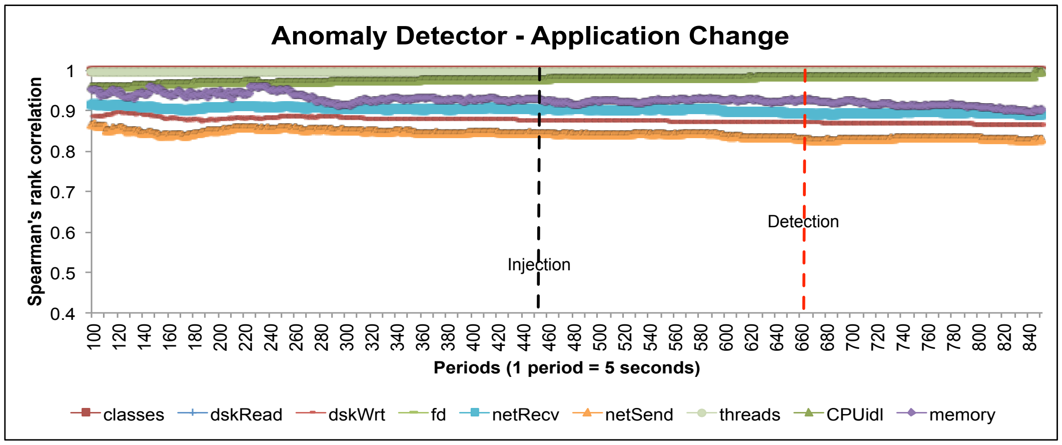


Figure 45: Application change - Anomaly Detector: identify if any of system and application parameters is associated with the performance variation

The *Root-cause Failure Analysis* module verifies the association between response time of the calls belonging to the user-transactions call-path and the response time of the user-transaction. The analysis relative to the `rubis_servlets_ViewBidHistory` user-transaction, is illustrated in Figure 46 and in Figure 47.

In Figure 46 we illustrate the set of calls with a higher weight in the response time of the user transaction under analysis (`rubis_servlets_ViewBidHistory`). By the figure, it is verified that after the anomaly injection, the response time of the `viewBidHistory.listBids` call and the user-transaction response time becomes more correlated.

In Figure 47 we illustrate the set of calls belonging to the `viewBidHistory.listBids` call-path. From the list of calls belonging to the `viewBidHistory.listBids` call-path, the analysis has identified a strong relationship between the response time slowdown affecting the `java.lang.Thread.sleep` call and the performance anomaly detect by the *Performance Anomaly Analysis* module, involving the `rubis_servlets_ViewBidHistory` user-transaction.

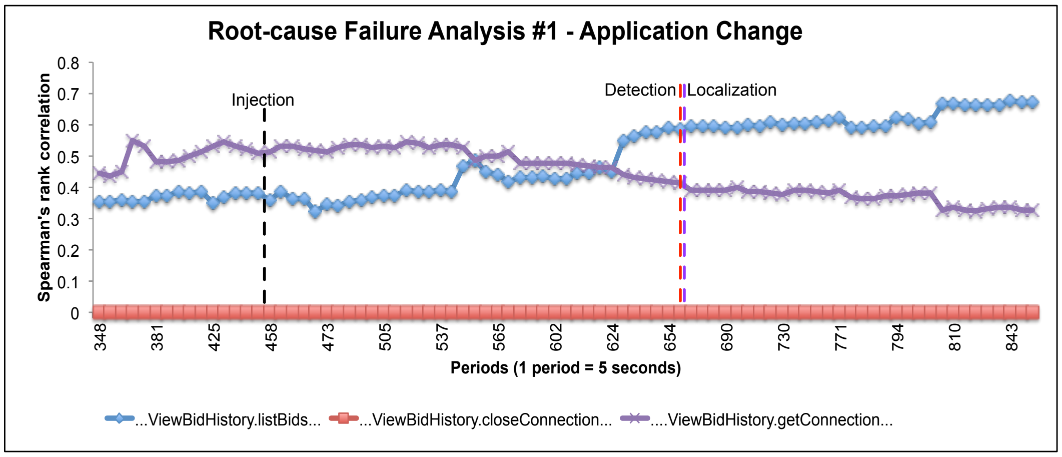


Figure 46: Application change - Root-cause Failure Analysis: list of application calls associated with the performance anomaly

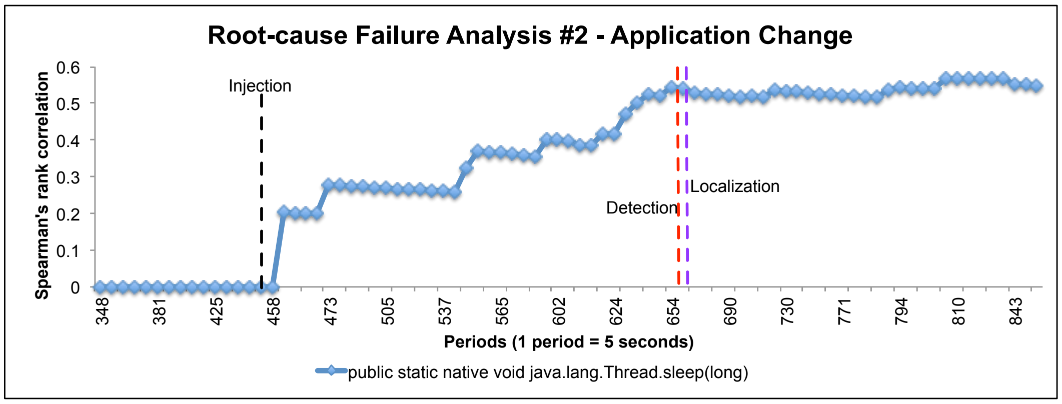


Figure 47: Application change - Root-cause Failure Analysis: list of application calls associated with the performance anomaly

6.3.10 Detecting and pinpointing anomalies with *SHõWA*: experimental study conclusions

The results presented in this section refer to the injection of five different anomaly scenarios. The anomaly scenarios adopted cause a performance degradation which is detected by the data analysis included in the *SHõWA* framework.

Each of the anomaly scenario was injected in two different benchmarking applications. According to the results, *SHõWA* is able to detect and pinpoint all the anomalies used in this study. To quantify the efficiency of the detection and anomaly pinpoint, provided by the framework, we present in Table 12 the percentage of user-requests affected by a response time slowdown and the number of user-visible errors verified. Like in the previous chapter, the number of slow transactions experienced by the end-users were appraised in three distinct intervals. The first interval contains the average number of transactions that, comparatively to the response time baseline previously measured, were affected by a response time delay between 100 milliseconds and 500 milliseconds. The second interval refers to the transactions that suffered a response time delay between 500 milliseconds and 2000 milliseconds. The third interval gauges the average number of user-transactions that have suffered a response time delay higher than 2000 milliseconds. These values were chosen based-on the performance impacts reported in the field (e.g., [Linden 2006] and [Schurman 2009]). The results also consider four intervals of analysis: (1) the $[\boldsymbol{S}, \boldsymbol{I}]$ contains the percentage of end-users affected by the response time slowdown between the Start of the experiment and the anomaly Injection period; (2) The $]\boldsymbol{I}, \boldsymbol{D}]$ comprises the end-users affected between the Injection and the Detection period; (3) the $]\boldsymbol{D}, \boldsymbol{P}]$ corresponds to the percentage of end-users affected during the Detection and Pinpointing of the performance anomaly period; (4) the last interval $]\boldsymbol{P}, \boldsymbol{E}]$ accounts for the users affected between the Pinpointing and the End of the experiment. Since we did not make any recovery attempt it is expected a high percentage of end-users affected in the interval $]\boldsymbol{P}, \boldsymbol{E}]$.

Table 12: Percentage of end-users affected by a slow response time, before, during and after the detection and pinpointing of the anomalies injected

	Slowdown (ms)	[S,I]	]I,D]	]D,P]	]P,E]
CPU Load	[100:500[	0.10%	0.37%	0.89%	12.55%
	[500:2000[	0.00%	0.00%	0.01%	7.91%
	[2000:∞[	0.00%	0.00%	0.00%	24.51%
Workload contention	[100:500[	0.11%	0.82%	0.00%	0.63%
	[500:2000[	0.00%	0.00%	0.00%	28.94%
	[2000:∞[	0.00%	0.00%	0.00%	3.45%
Remote change	[100:500[	0.18%	0.17%	0.24%	5.02%
	[500:2000[	0.00%	0.07%	0.81%	23.16%
	[2000:∞[	0.00%	0.78%	0.86%	11.51%
Memory consumption	[100:500[	0.08%	0.02%	0.00%	1.78%
	[500:2000[	0.00%	0.01%	0.00%	0.24%
	[2000:∞[	0.00%	0.00%	0.00%	0.40%
Application change	[100:500[	0.07%	0.14%	0.00%	7.83%
	[500:2000[	0.00%	0.02%	0.00%	0.00%
	[2000:∞[	0.00%	0.00%	0.00%	0.00%

From the percentage of end-users affected by a response time slowdown, presented in Table 12, it can be seen that between the injection and detection of the anomaly and between the detection and pinpointing the percentage of user-transactions affected is inferior to 1%. Since, in this experiment, we do not trigger any recovery action, the percentage of end-users affected after the anomaly pinpointing becomes very high, when compared with the previous values.

It is also important to note that after the memory consumption there were 37 HTTP errors and the application server has hanged.

6.4 *SHõWA*: Accuracy analysis

The accuracy achieved by the *SHõWA* framework for detecting and pinpointing anomalies is presented in two levels.

At the first level we check the following:

- **A:** *How often does the SHõWA has detect anomalies when we did not injected anomalies?*

- **B:** *How often does the SHõWA has detect anomalies in presence of anomalies?*

At the second level of analysis we check:

- **C:** *Knowing that SHõWA did not detected any anomaly and that we did not injected anomalies: what is the percentage of user-transactions affected by a response time delay superior to 500 milliseconds?*

- **D:** *Knowing that SHõWA has detected the anomaly and that we have injected anomalies: what is the percentage of user-transactions affected by a response time delay superior to 500 milliseconds?*

To answer the questions A and C we analyzed 24 hours of data. During these 24 hours were executed dynamic workloads totaling 3,784,982 requests. We did not injected any type of anomalies. Considering the results, the answers that we observed to the question A is 0% and to the question C is 0.038%.

To answer the questions B and D we also analyzed 24 hours of data. During these 24 hours were executed dynamic workloads totaling 2,371,233 requests. During the experiment we injected different types of anomalies: system overload, resource exhaustion (CPU and memory) and application changes. Considering the results, the answers that we observed to the question B is 100%. The answer to question D is separated in two periods:

- 15 minutes before the detection of the anomaly: 0.042%

- 15 minutes after the detection of the anomaly: 0.71%

These results highlight two important aspects. *SHõWA* has detected all the anomalies we have injected and when it detects it thats because there are users affected by the anomaly. When there are no anomalies *SHõWA* did not raise any false alarm.

The ability to detect anomalies while the number of end-users affected is low and the accuracy results presented in this section, makes the *SHõWA* framework a key tool for the automatic detection and recovery of performance anomalies in Web-based applications.

6.5 Conclusions

In this chapter we evaluated the data analysis process, performed by the *SHõWA* framework to detect and pinpoint performance anomalies in Web-based applications. An experimental analysis was conducted to study the feasibility of the analysis mechanism.

The data analysis involves different modules that cooperate each other to detect performance anomalies, distinguish performance anomalies from workload variations and to pinpoint the cause behind the anomaly. The data used as input is collected by the *Sensor* module and prepared by the *Data Preparation* module. The experimentation conducted includes five anomaly scenarios and two different benchmarking applications. The applications were exercised through highly dynamic workloads.

The experimental results achieved so far let us to conclude the following:

- The adoption of the Spearman's rank correlation enhances the association between the variables independently from its scale. This allows to generalize the analysis to different applications and the definition of a single threshold value for all the user-transactions;

- The analysis is done from the perspective of the application, i.e., it focus application performance anomalies and strives to pinpoint the causes behind the performance variation observed;

- *SHõWA* was able to detect and spot all the anomalies we injected and it has distinguished them from workload variations;

- When there is a very dynamic workload but there are no faults the tool was able to detect this situation and did not raise any false alarms, which was also an important result;

- The anomalies were detected and pinpointed while the number of end-users affected is low.

The low performance impact induced by *SHõWA* and its ability to detect and pinpoint different types of anomalies automatically, are important aspects in the implementation of a self-healing system. Taking advantage of these results, in the next chapter we evaluate the recovery mechanism included in the *SHõWA* framework.

Chapter 7 Recovery

KEY POINTS

- ◇ The time needed to detect and recover from an anomaly is crucial to the availability of a computational system. Reduce these times is a topic that captures the attention of the scientific and industry communities.
- ◇ A Web-based infrastructure is usually provided with mechanisms of fault tolerance and high-availability. Data redundancy and clustering configurations are widely used to ensure the provision of the service in case of failure of one or more systems.
- ◇ Regardless of the advantages of high-availability solutions, they are not prepared to deal with scenarios of performance anomalies. Like an outage scenario, the occurrence of performance anomalies affects the normal operation of the service and leads to severe revenue and reputation losses.
- ◇ Recovery mechanisms, such as those presented in the software aging and rejuvenation field and the *microrebooting* approach presented with the project Pinpoint, extend the traditional mechanisms for fault tolerance and high-availability.
- ◇ In line with those projects, in this chapter we evaluate the usefulness of the *SHõWA* framework to recover from performance anomalies in Web-based applications.

The availability of a system corresponds to the probability that it will be operational when it is needed. It is given by Equation 7.0.1, and it depends on the mean time to failure ($\boldsymbol{MTTF}$) and on the mean time to repair ($\boldsymbol{MTTR}$), with the latter being the most important factor in the assessment of the availability of a system.

$$\boldsymbol{Availability} = \frac{\boldsymbol{MTTF}}{\boldsymbol{MTTF} + \boldsymbol{MTTR}} \tag{7.0.1}$$

Increase $\boldsymbol{MTTF}$ and decrease $\boldsymbol{MTTR}$ are two sides of the same coin that argue in favor of greater availability. These topics capture a lot of attention from the scientific community, solution providers and from the IT/IS consumers.

In the last years we have seen a change on the causes of failures in Web-based applications. In place of hardware problems, that were until few years ago the main cause of failures, we have now software failures and human errors as the dominant causes of system failures. This change occurs partly due to the steady increase in hardware reliability, and on the other hand, due to the increase in software complexity which make it more error-prone and more difficult to maintain. This complexity makes the task of problem diagnosis and recovery more time-consuming and requires more expertise from the IT staff.

In this chapter we describe some of the most relevant work focused in automatic detection and recovery from software failures. We also evaluate the functioning of the *SHõWA* framework to recover from different types of performance anomalies in Web-based applications. For this study we consider an execution environment equal to that found in production environments.

7.1 Recovering from anomalies - related work

There are various approaches in the literature that focus on failure recovery.

One of the more active approaches that is being employed to proactively prevent unplanned, and potentially expensive, system outages is known as "software rejuvenation". This approach was proposed by Huang et al. in [Huang 1995]. It involves, in a periodic or predicted/measurement manner, stop the software application to remove the accrued error conditions, freeing up operating system resources and then restarting the application in a clean state. While the periodic-based rejuvenation assumes a known degradation rate of the system, the measurement-based approach collects data, at runtime, to determine the health of the executing software and initiate a recovery action if necessary. It is widely understood that the measurement-based technique maximizes the availability of the system.

In [Trivedi 2004] authors present a measurement-based methodology to detect software aging and estimate its effect on various system resources. The authors adopt time and workload-based estimation to verify the parameters consumption and trigger the rejuvenation actions. Their approach is being used in the IBM xSeries line of cluster servers. In [Castelli 2001] the authors perform rejuvenation actions, taking advantage of the cluster service. Rejuvenation is triggered according two types of policies: periodic versus predicted-based. In the periodic policy, the computing nodes are restarted alternately after a certain period of time. In the prediction-based policy, the time to rejuvenate is estimated based-on the collection and statistical analysis of system data. The rejuvenation of the system involves the restart of services or rebooting the affected node, leaving the application failover and recovery in charge of the clustering services. In [Li 2002] the authors take advantage of the Apache Web Server configuration parameters to rejuvenate the server by killing and recreating processes after a certain numbers of requests have been served.

Candea et al., in [Candea 2001] and in [Candea 2002], presents an approach, known as recursively restartable system. The recursive restartability aims to improve the availability ratio by reducing the mean time to repair (MTTR). It consists on reactively restart the failed subsystems. The technique is viable for systems that gracefully tolerates successive restarts at multiple levels avoiding long downtimes and potential data losses.

Another sound work in the area is proposed by the Pinpoint project, in [Chen 2002], [Patterson 2002] and [Candea 2006]. This project relies on a modified version of the application server to monitor the application structure and pinpoint application-level failures, without requiring a priori knowledge about the application. The request execution is intercepted to model the interactions between components. In runtime, differences between the expected and observed interactions are regarded as failures and recovery procedures are initiated. The project was subject of several experiments related to recovery. The adoption of a server restart and recursive restartability appears as one of the first approaches. Later, and rather than doing a reboot by default, authors have developed the notion of a component-level *microreboot* [Candea 2004b], as a way to reduce the functional disruption motivated by a server or process restart.

In [Sultan 2003], [Bohra 2004] and [Sultan 2005] authors take advantage of the backdoor architecture to remotely monitor and recover from operating system failures. Using the programmable network interfaces to obtain remote access, the liveness of the targeted systems is continuously monitored and the sessions information is stored. In case of a server failure, the clients are redirected to another server and the sessions are restored.

In [Silva 2006] is presented a SLA-based recovery mechanism targeted for SOAP-based servers. Online monitoring is used to collect system parameters from different layers of the monitored system (operating system, HTTP server, SOAP-router). The collected data is analyzed by a evaluator module that, according to the predefined SLA contract, triggers rejuvenation actions. The recovery module performs a selective restart in the most appropriate layer. The results show that the approach is useful for avoiding system crashes due to software aging and yet to maintain the service with a good performance level. Another software rejuvenation solution that has provided interesting results is presented by Silva et al. in [Silva 2009]. The authors have used virtualization services to improve software rejuvenation of application servers. Off-the-shelf monitoring is used to detect the resources consumption, and if a predefined SLA is violated then the service is commuted to a new virtual machine. According to the results, with virtualization it is possible to rejuvenate the software with zero errors, without loose any work-in-progress and without affect the application performance.

A self-healing technique targeted for component-based applications is presented in [Chang 2008]. The self-healing technique masks software faults by intercepting and changing the methods call before execute or retry the operations. From the experimentation, authors have confirmed the usefulness of the approach to avoid some application-faults before they result in application failures. Undo, retry and substitution operations are also presented in [Patterson 2002], [Brown 2003] and [Pernici 2007] as viable mechanisms to cope with failures in Web-based and component-based applications.

7.2 The *SHõWA* recovery module

The recovery service provided by the *SHõWA* framework involves three modules: the *Recovery Planner*, the *Executor* and the *Effector* modules.

The *SHõWA* recovery service, faced with the detection and localization of anomalies performed by the *Performance Anomaly Analysis*, *Anomaly Detector* and *Root-cause Failure Analysis* modules, is responsible for restore the normal functioning of the system, as quickly as possible and preferably with minimal impact to end-users. For this purpose, the *Recovery Planner* module selects a recovery procedure. The recovery procedure is then executed by the *Executor* module. The *Executor* module implements the recovery actions in the managed resource through the *Effector* module, and receives feedback on its implementation.

Summarizing the rationale of the data analysis presented in Chapter 3, and according to the results presented in the previous chapter:

- **A performance anomaly is detected when -** the degree of dissociation (provided by Algorithm 4) between the response time and the number of user-transactions processed, is greater than or equal to 10;

- **A performance anomaly motivated by a system or application container change is pinpointed when -** the Spearman's rank correlation between the total number of user-transactions and the cumulative value of each parameter decreases at least 0.1 degrees;

- **A performance anomaly motivated by an application change is pinpointed when -** the Spearman's rank correlation between the frequency distribution of the transactions response time and the response time frequency distribution of the calls belonging to the transaction call-path increases significantly for some of the calls, while for the other calls the correlation degree stands or decreases.

In the assessment of the recovery mechanism, presented in this chapter, upon the detection and pinpointing of a performance anomaly, a recovery procedure is automatically selected and executed. In the current version of *SHõWA*, the recovery procedure is selected using a conditional rule: `if (anomaly == <type-of-anomaly>) then pick-ruleA`. The recovery procedures included in the *Recovery Planner* module were defined manually by a human operator.

7.3 Recovering from anomalies using the *SHõWA* framework

In this chapter, we evaluate the ability of the *SHõWA* tool to recover from performance anomalies in Web applications. More specifically, through an experimental study, we seek answers to the following set of questions:

- *What are the advantages of using the SHõWA to detect, pinpoint and recover from different performance anomaly scenarios?*
- *What advantages can we take from combining SHõWA with cluster and load balancing services to mitigate performance anomalies?*;
- *What is the time-to-repair achieved by SHõWA when compared with other monitoring tools?*.

Since most of the Web-based applications runs in an environment equipped with high-performance and high-availability services, we decided to prepare a test environment as similar as possible to a productive environment. The system under test (SUT) consists in a Web-based application running in a high-availability cluster with load balancing services.

The SUT was submitted to different anomaly scenarios. For each anomaly scenario we present the results achieved by the *SHõWA* framework versus a non-recovery scenario. Similarly, we present results that show the advantages in performing controlled recovery actions versus a cold-recovery strategy. Whereas the system is provided with high availability mechanisms, the cold-recovery analysis aims to assess to what point these mechanisms are sufficient to tolerate failures resulting from a recovery action. We also show how timely is the recovery achieved by the *SHõWA* framework when compared to a system-level monitoring tool.

7.3.1 Experimental setup

In this section we present the experimental environment used to evaluate the detection, pinpointing and recovery mechanism included in the *SHõWA* framework. The workload, the anomaly scenarios and the recovery plans adopted in this study are also described.

The experimental environment comprises a three-tier infrastructure running a benchmark application in a cluster configuration, with load balancing services, dedicated servers to store and carry out the data analysis and multiple emulated clients interacting with the application.

The testbed is illustrated in Figure 54. It consists of an application server (Oracle Glassfish Server) running a benchmark application (TPC-W [Smith 2001]). The TPC-W benchmark simulates the activities of a retail store website. It uses a MySQL database as a storage backend and contains its own workload generator (RBE) that emulates the behavior of the end-users.

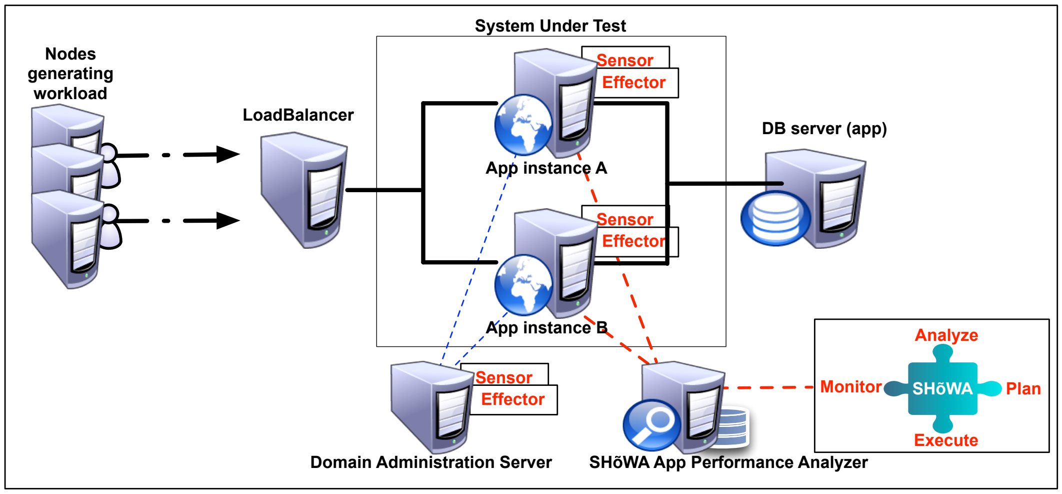


Figure 48: Test environment: Web-based application in a high-availability cluster configuration with load balancing

The application is running in a cluster composed by two application server instances. The client requests are sent to a layer-7 load balancer (HaProxy [HAProxy]) that forwards the requests to the active application servers. The cluster service provides session replication, allowing the user-transactions to continue processing even when one of the instances becomes unavailable.

Each application server instance includes a *Sensor* module that collects the application, system and container parameters and sends the collected data to the "*SHõWA* Application Performance Analyzer" server. The data is prepared and analyzed continuously in a separate machine to avoid performance impacts. When a performance anomaly is detected, the "*SHõWA* Application Performance Analyzer" server interacts with the affected instance through the *Effector* module.

The test environment is composed by several machines. The nodes responsible for generating workload include a 3GHz Intel Pentium CPU and 1GB of RAM. The "LoadBalancer" machine includes two 3.4GHz Intel i7 CPU cores and 2GB of RAM. The "Domain Administrator Server" is responsible for the cluster instances and resources management. It includes four 3.4GHz Intel i7 CPU cores and 4GB of RAM. The two application server instances and the "*SHõWA* Application Performance Analyzer" virtual machines, include four 3.4GHz Intel i7 CPU cores and 4GB of RAM. The database server machine includes four 2.66GHz Intel Xeon CPUs and 2GB of RAM. We used the XEN virtualization technology. All the nodes run Linux 2.6 kernel version and are interconnect through a 100Mbps Ethernet LAN.

7.3.2 The Workload-test tool

TPC-W contains its own workload generator (RBE) that emulates the behavior of the end-users and reports metrics, such as, the user-transactions response time and the server throughput. According to its specification, the number of emulated browsers is kept constant during the experiment.

Since real e-commerce applications are characterized by a dynamic number of users and requests, we decided to create a dynamic workload. It runs for 7200 seconds, and during the execution, the type of traffic mix and the number of emulated-users varies periodically. The number of concurrent users varies between 50% and 80% of the maximum system throughput. These values were chosen to analyze the performance impact in the application server instance that remains active, while the other is under rejuvenation. The maximum capacity of the server was previously determined by running workloads, in which the number of concurrent sessions increased almost linearly. The maximum capacity corresponds to the maximum number of concurrent requests that the system can process before the response time is severely affected.

7.3.3 Anomaly scenarios tested

In this experimental evaluation we consider four anomaly scenarios: JVM memory consumption, CPU consumption, Threads consumption and application changes.

To consume JVM memory we used JAFL [Rodrigues 2008]. JAFL is implemented in Java and allows the simulation of some of the most common types of failures in Web applications. The **Memory Consumption** module was used to consume 1 MB of JVM memory per second. When all the JVM memory is consumed, an *"OutOfMemory"* error is throw. The JAFL **Thread Consumption** module was also used to consume a certain number of threads along the time. If a recovery action is not performed, then the number of user processes will become exhausted and the system hangs/stops responding.

To impose load on the system we used the `stress` [Stress] tool. This is a simple tool which allows to inject a gradual consumption of CPU. In this case, we use it to evaluate the ability of the *SHõWA* framework to detect and recover from the performance anomaly caused by the **CPU consumption**.

A performance anomaly motivated by an **Application Change** was also considered in this study. The **Application Change** scenario consists in running a modified version of the TPC-W application. In the modified version there is an user-transaction that becomes slow as the time goes: the response time of the user-transactions increases 1 millisecond per second.

Like other anomaly scenarios used throughout this work, the anomaly scenarios used for this experimentation cover situations in which applications are affected by system overloads, resources exhaustion or failed upgrades. In Table 13 we present the configuration details about the anomaly scenarios used in this experimental analysis.

Table 13: Perturbation scenarios (resource consumption and application change)

Anomaly	Level	Description
Memory Consumption	Glassfish	Consume 1MB of JVM Heap per second
CPU consumption	System	O.S. processes that provoke a gradual CPU consumption
Thread Consumption	Glassfish	Consume 30 threads per second
Application Change	TPCW	Transactions response time increases 1 millisecond/second

7.3.4 Recovery procedures

The *Recovery Planner* module contains several recovery procedures. Given the detection of an anomaly, the *Executor* module selects the recovery procedure. It uses a conditional rule that tells which plan should be adopted to mitigate a certain anomaly. The current implementation gathers data about the execution of the recovery actions. This data can be used further to make the recovery process more autonomous.

The layer-7 load-balancer is used to control the availability of the application server instances and perform controlled recovery actions. Each application server contains a check file, and if this file can be retrieved by the load-balancer then it considers the application server available, otherwise it do not forward requests to that server.

The recovery procedure defined for the **Memory Consumption** anomaly is the following:

Memory Consumption - recovery procedure

```
Decrement the number of appserver instances available (synchronized)
if (number of instances available >= 1) then
   Remove the control file that is checked by the load-balancer
   Stop the appserver instance which is reporting the anomaly /*REPAIR ACTION*/
   if (previous steps takes > 30 seconds to complete) then
      Force the appserver instance termination (kill -9) /*REPAIR ACTION*/
   end if
   Start the appserver instance /*REPAIR ACTION*/
   Restore the control file that is checked by the load-balancer
end if
if (appserver instance under recovery is available) then
   Increment the number of appserver instances available
else
   Send an alert to the console
end if
```

The recovery procedure defined for the **CPU consumption** and **Thread Consumption** anomaly is the following:

Thread Consumption and **CPU consumption** - recovery procedure

```
1: Decrement the number of appserver instances available (synchronized)
2: if (number of instances available >= 1) then
3:    Remove the control file that is checked by the load-balancer
4:    Reboot the application server that is reporting the anomaly /*REPAIR ACTION*/
5:    Restore the control file that is checked by the load-balancer
6: end if
7: if (appserver instance under recovery is available) then
8:    Increment the number of appserver instances available
9: else
10:    Send an alert to the console
11: end if
```

The recovery procedure for the **Application Change** anomaly is the following:

Application Change - recovery procedure

```
1: Decrement the number of appserver instances available (synchronized)
2: if (number of instances available >= 1) then
3:    Deploy a previous version of TPC-W /*REPAIR ACTION*/
4:    Remove the control file that is checked by the load-balancer
5:    Do a rolling downgrade to switch the TPC-W version /*REPAIR ACTION*/
6:    Restore the control file that is checked by the load-balancer
7: end if
8: if (appserver instance under recovery is available) then
9:    Increment the number of appserver instances available
10:    Repeat the recovery process on the other appserver instances /*REPAIR ACTION*/
11: else
12:    Send an alert to the console
13: end if
```

The *SHõWA* framework was prepared to trigger the recovery actions when the impact of dissociation between the response time and the number of user-transactions processed (determined according to the Algorithm 4) exceeds 10 degrees. According to the rationale of the analysis, presented in Chapter 3 and the detection and anomaly pinpointing results presented in Chapter 6, this value is representative of a significant response time slowdown that is not justified by the number and mix of user-transactions processed. A workload variation is detected when the *Workload Variation Analysis* module returns a variation of $\boldsymbol{\rho}$ (Spearman's rank correlation) higher than 0.1 degrees. For the purpose of pinpointing anomalies, we defined as threshold value a Spearman's rank correlation variation higher than 0.1 degrees.

7.4 Experimental results

In this section we present the experimental results, considering the anomalies and recovery procedures defined. Each experiment was repeated 10 times and the results presented correspond to the average values obtained. The results provide:

- A comparison between the adoption of the *SHõWA* framework against a non-recovery scenario;
- The throughput and response time impact observed during the recovery process;
- The time to repair and the number of errors experienced by the end-users;
- A comparison between *SHõWA* and a system-level monitoring tool.

7.4.1 Memory consumption scenario

The memory consumption simulates a memory leak. During the experiment a given amount of the JVM heap memory in the application server instance A is allocated and it is never freed. After some time of activity, the amount of free memory is reduced and this causes the garbage collector to become more active in trying to free memory. The effort to free memory has an impact in terms of computational resources, leading to a performance variation that is detected by *SHõWA* framework.

In Figure 49 is illustrated the JVM memory consumption on each of the servers.

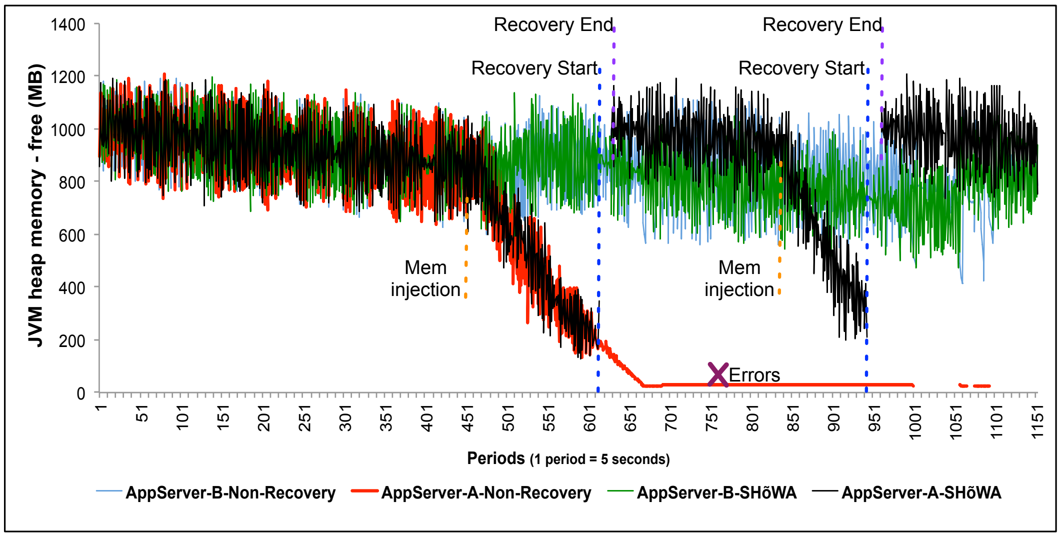


Figure 49: JVM heap memory consumption per application server instance: *SHõWA* recovery versus without recovery

From Figure 49 is visible the effect of doing recovery. Without recovery the JVM heap memory of the application server instance A was totally consumed, leading to *"OutOfMemory"* errors and consequently to the server hang (epoch 660). With the *SHõWA* framework the anomaly was detected and the recovery procedure was performed automatically, avoiding the JVM heap memory exhaustion.

In Table 14 we present the average response time in milliseconds and number of requests processed by application server instance. The analysis considers the adoption of the *SHõWA* framework versus a non-recovery scenario and it allows to analyze the impact on the response time and the impact on the server throughput. We present the results in three periods of analysis: $[\boldsymbol{S} : \boldsymbol{I}]$ corresponds to the average response time and to the average server throughput observed between the start of the experiment and the period when the anomaly was injected; the $]\boldsymbol{I} : \boldsymbol{R}]$ interval comprehends the periods between the anomaly injection until the recovery was executed by the *SHõWA* framework; the $]\boldsymbol{I} : \boldsymbol{E}]$ points out the periods between the anomaly injection and the end of the experiment (it represents the non-recovery scenario).

Table 14: Memory Consumption - Response time and Throughput impact per instance: *SHõWA* recovery versus without recovery

	Avg resp_tm (ms)			Avg throughput (req/sec)		
	[S:I]	]I:R]	]I:E]	[S:I]	]I:R]	]I:E]
AppServer-A-Non-Recovery	11	–	*782*	1124	–	*688*
AppServer-B-Non-Recovery	11	–	16	1123	–	1157
AppServer-A-*SHõWA*	11	13	–	1125	1111	–
AppServer-B-*SHõWA*	11	16	–	1123	1724	–

The results presented in Table 14 show that, without recovery, the user-transactions latency and the server throughput is severely affected by the anomaly. The response time has increased from 11 to 782 milliseconds per request (with more than 18 seconds of standard deviation). The server throughput, that should be similar across the application server instances, has dropped to 688 requests per second, and this drop was not compensated by the application server instance B. With *SHõWA* the impact is much lower. During the 80 seconds taken by the recovery procedure to rejuvenate the application server instance A, all the user-transactions were redirected to the application server instance B and the response time has increased just in 2 to 5 milliseconds per request. Broadly speaking, the response time in the scenario of non-recovery was 27 times higher than in the scenario where the anomaly was detected and recovered by the *SHõWA* framework. Similarly it was found that, in the setting of non-recovery, the server capacity was 35% lower in comparison with the recovery scenario.

In Table 15, we present the number of HTTP client errors (4XX), HTTP server errors (5XX) and lost sessions, experienced by the end-users and motivated by the recovery process. To verify if the high-availability service is sufficient to support the recovery of the service without introduce errors, we present the results obtained with the implementation of a cold-recovery scenario. In the cold-recovery scenario the recovery is performed without notice the load-balancer first.

Table 15: Memory Consumption: Number of errors considering non-recovery, *SHõWA* recovery and cold-recovery scenario

	HTTP 4XX	HTTP 5XX	Sessions Lost
Non-Recovery	0	536	0
SHõWA-Recovery	0	0	0
Cold-Recovery	0	571	0

The results presented in Table 15 reveal the importance of doing recovery and implement controlled recovery mechanisms. The recovery process performed by the *SHõWA* framework was the only one that did not registered any visible error, experienced by the end-users. The number of errors obtained with the cold-recovery and the non-recovery is very similar. These errors correspond to requests that were being processed on the application server A when the available memory becomes exhausted. The cold-recovery provides faster recovery time, but it does not prevent the occurrence of failures.

7.4.2 CPU consumption scenario

Through the `stress` [Stress] tool, we have launched some processes that cause a higher CPU consumption. The processes were launched in the application server instance A.

In Figure 50 is illustrated the variation on the amount of CPU idle verified during the experiment. In the figure is included the scenario where the anomaly was recovered by the *SHõWA* and the scenario where the anomaly persisted without any recovery attempt. Are illustrated two occurrences of CPU consumption. From the results, it is visible that without recovery the CPU on the application server instance A is totally consumed. This, as presented in Table 16, leads to a response time delay and to a server throughput degradation: during the anomaly manifestation ($]\boldsymbol{I} : \boldsymbol{E}]$), and when compared to the results achieved with *SHõWA*, the response time has increased up to 71 milliseconds per request (with 42 milliseconds of standard deviation) and the system throughput has dropped by more than 16%.

Taking advantage of the *SHõWA* framework, the performance anomaly, caused by the CPU load, is detected and recovered in time to avoid the impact. The recovery took about 60 seconds and, as presented in Table 16, between the anomaly injection until the recovery period $]\boldsymbol{I} : \boldsymbol{R}]$, the response time has increased just 2 milliseconds per request. During the recovery period, all the requests were processed by the application

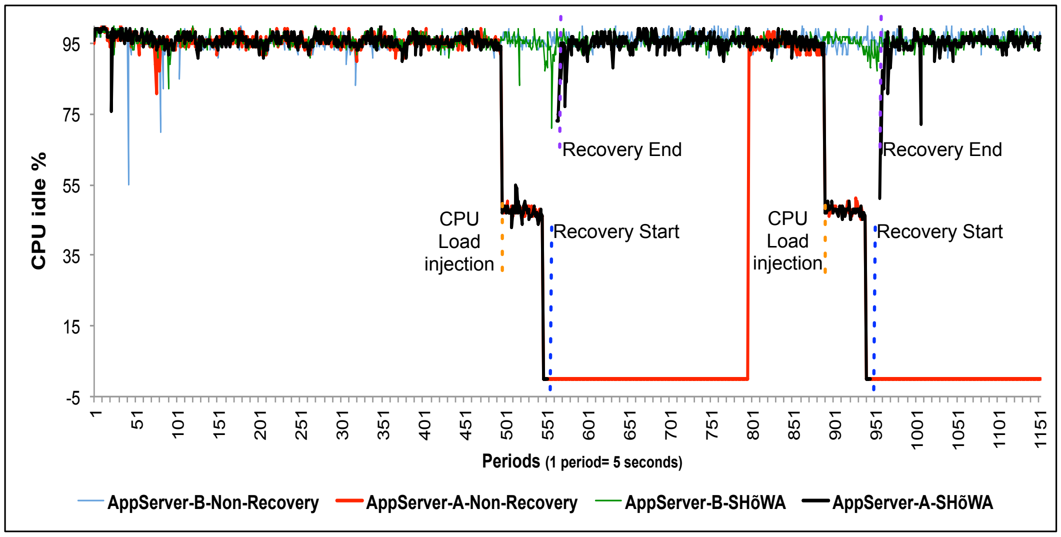


Figure 50: CPU parameter consumption per application server instance: *SHõWA* recovery versus without recovery

server instance B, and therefore, there were no significant losses in terms of throughput.

Table 16: CPU Consumption - Response time and Throughput per instance: *SHõWA* recovery versus without recovery

	Avg resp_tm (ms)			Avg throughput (req/sec)		
	[S:I]	]I:R]	]I:E]	[S:I]	]I:R]	]I:E]
AppServer-A-Non-Recovery	11	–	*71*	1142	–	*973*
AppServer-B-Non-Recovery	11	–	14	1137	–	*976*
AppServer-A-*SHõWA*	11	14	–	1122	1110	–
AppServer-B-*SHõWA*	11	13	–	1126	1222	–

Regarding to the amount of errors motivated by the type of recovery (*SHõWA* recovery, cold-recovery or without recovery), the cold-recovery was the only type which registered errors. The restart of the application server instance A, without notifying the load-balancer, has originated 45 HTTP server errors (status code 5XX).

7.4.3 Threads consumption scenario

The Threads consumption was injected two times during the experiment. At a given point, during the experiment, the number of running threads starts increasing. When the number of running threads reaches the maximum number of user processes defined in the operating system, the system rejects the new requests.

The anomaly was injected in the application server instance A. In Figure 51 is illustrated the consumption of processes, without considering the system recovery.

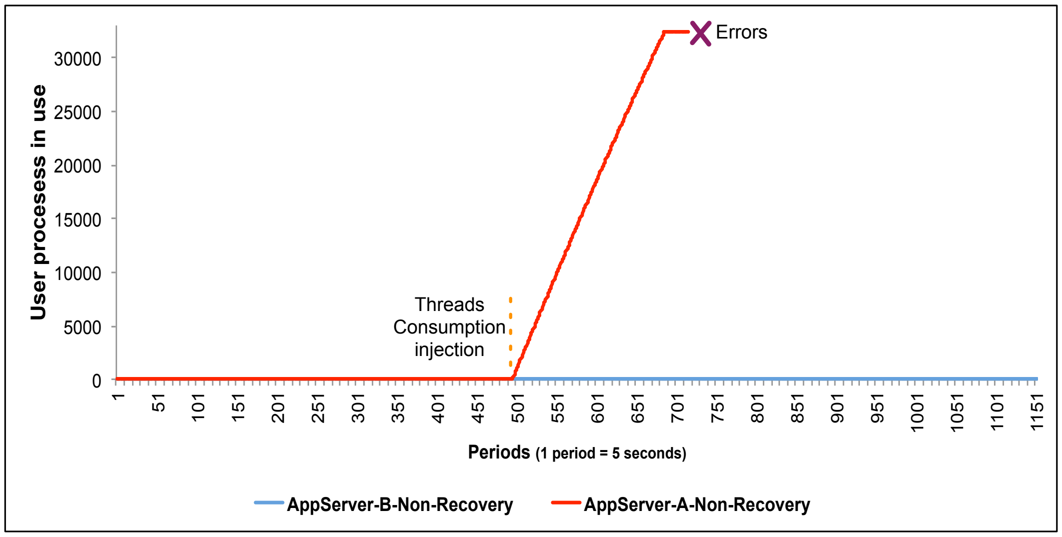


Figure 51: Number of running processes/threads: without recovery

In Figure 52 is illustrated the consumption of processes, taking into account the recovery provided by *SHõWA*.

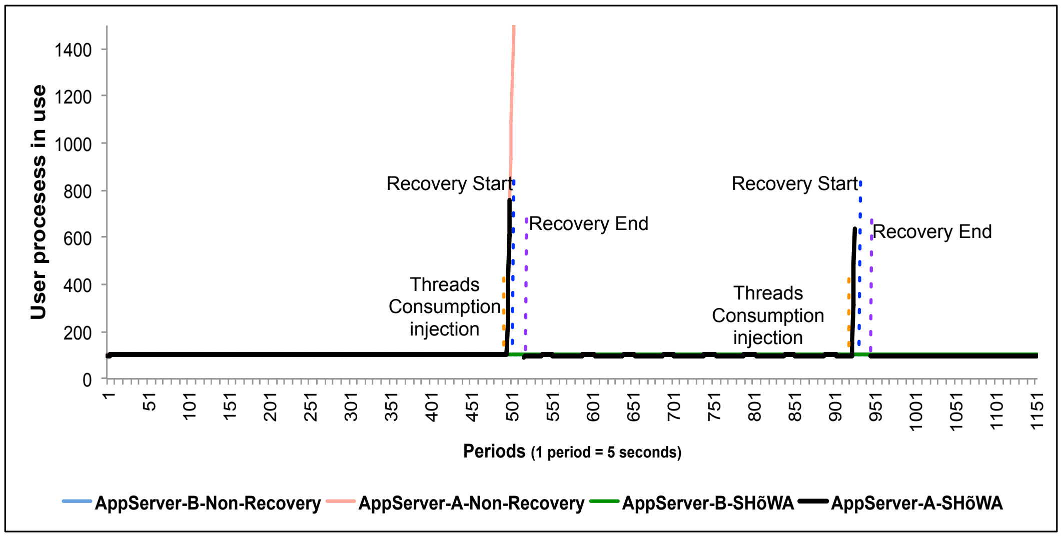


Figure 52: Number of running processes/threads: with *SHõWA* recovery

From Figure 51 and Figure 52 is visible the effect of applying recovery. Without recovery the number of maximum processes is reached and the application server instance A stops processing the requests (epoch 716). With the *SHõWA* framework, the deviation between the number of application processes and the number of busy processes is detected (epoch 500 and epoch 927)) and recovered by the recovery procedure (epoch 517 and epoch 947 respectively).

The response time and server throughput analysis, considering the recovery provided by the *SHõWA* framework versus a non-recovery approach, is summarized in Table 17.

Table 17: Threads Consumption - Response time and Throughput per instance: *SHõWA* recovery versus without recovery

	Avg resp_tm (ms)			Avg throughput (req/sec)		
	[S:I]	]I:R]	]I:E]	[S:I]	]I:R]	]I:E]
AppServer-A-Non-Recovery	12	–	*63*	1096	–	*1040*
AppServer-B-Non-Recovery	12	–	13	1095	–	*2295*
AppServer-A-*SHõWA*	12	14	–	1097	918	–
AppServer-B-*SHõWA*	12	12	–	1096	1690	–

The results presented in Table 17 show that, without recovery, the user-transactions response time has increased up to 63 milliseconds in average (with more than 229 milliseconds of standard deviation). Without recovery the server throughput on instance A was not significantly affected, but as soon as the number of processes available decreases in the application server instance A, the application server instance B begins to have more requests to process. With the *SHõWA* recovery, the threads consumption was immediately detected and recovered.

The automatic recovery avoided that the response time increases up and that the service throughput was severally affected. *SHõWA* took about 20 seconds to detect the anomaly and the recovery procedure took about 90 seconds, covering all checks implicit therein.

In Table 18, we present the number of errors experienced by the end-users.

Table 18: Threads Consumption: Errors observed considering non-recovery, *SHõWA* recovery and cold-recovery scenario

	HTTP 4XX	HTTP 5XX	Sessions Lost
Non-Recovery	0	220	0
SHõWA Recovery	0	0	0
Cold-Recovery	0	4	0

The results presented in Table 18 show that without recovery there were 220 HTTP server side errors - status code 5XX. The high-availability service was not able to prevent failures while performing a cold-recovery. At least 4 HTTP server errors were experienced by the end-users. *SHõWA* allowed to recover from the anomaly, before any error has been experienced by the end-users.

7.4.4 Application change scenario

The application change simulates a scenario in which an upgraded version of the application introduces an unexpected performance degradation. It affects the application that is deployed on both of the application server instances. The recovery procedure consists of the rollback of the application version.

A performance anomaly motivated by an application change is detected and pinpointed by the *SHõWA* framework when: the impact of dissociation, computed by the *Performance Anomaly Analysis* module, reveals no association between the response time and the amount of user-transactions processed; the *Root-cause Failure Analysis* module highlights the application calls associated with the performance anomaly.

Table 19 contains the results obtained on the response time and server throughput. Are contemplated two scenarios: in the non-recovery scenario is not made any recovery attempt, in the recovery scenario the anomaly is detected and recovered by the SHõWA framework. Like in the previous analysis, the results are presented for both of the application server instances and are separated in three periods of analysis: [***S*** : ***I***] corresponds to the response time and server throughput observed between the start of the experiment and the period when the anomaly was injected; the]***I*** : ***R***] comprehends the periods between the anomaly injection and the detection and recovery provided by the *SHõWA* framework; the]***I*** : ***E***] points out the periods between the anomaly injection and the end of the experiment.

The anomaly manifested itself first in the application server instance B.

Table 19: Application Change - Response time and Throughput per instance: *SHõWA* recovery versus without recovery

	Avg resp_tm (ms)			Avg throughput (req/sec)		
	[S:I]	]I:R]	]I:E]	[S:I]	]I:R]	]I:E]
AppServer-A-Non-Recovery	12	–	*201*	1114	–	*937*
AppServer-B-Non-Recovery	12	–	*194*	1113	–	*935*
AppServer-A-*SHõWA*	11	20	–	1115	1056	–
AppServer-B-*SHõWA*	11	25	–	1115	927	–

The results presented in Table 19 shows that, without recovery, the response time has increased up to 201 milliseconds per request (with a standard deviation of 280 milliseconds). With *SHõWA*, the response time has increased up to 25 milliseconds (with 10 milliseconds of standard deviation). In comparison with the recovery provided by the *SHõWA* framework, without recovery the server throughput has dropped approximately 5.6%. *SHõWA* took about 55 seconds to recover from the anomaly.

From the analysis about different types of recovery used (*SHõWA* recovery, cold-recovery or without recovery), the cold-recovery was the only type which has originated HTTP errors: 55 client side errors (status code 4XX) and 670 server side errors (status code 5XX).

7.4.5 *SHõWA*: time-to-repair

Considering the results presented above, the advantage of using the *SHõWA* framework to detect and recover from anomalies, due to non-execution of recovery is clear. However, *SHõWA* is not the only way that exists to detect and recover from anomalies.

Taking this into consideration, we installed a system-level monitoring tool and prepare it to detect and repair different types of anomalies. The system-level monitoring tool was the Zabbix [Zabbix]. Zabbix is used to monitor different system parameters. It also includes a Web monitoring service that acts like a client accessing a predefined page, or sequence of pages, and verifies if the response time exceeds an established threshold. Zabbix was configured to collect the parameters with a 5 seconds polling interval, and execute the recovery procedures when: the heap memory usage exceeds 95%; the processor load exceeds 8 processes (2x the number of system processors); the user-transactions response time latency exceeds 2 seconds. There are different perceptions about the maximum response time of a Web application, however, most studies (e.g., [Gomez 2010], [Forrester Consulting]) agree that an increase in response time higher than 2 seconds reduces the users satisfaction and the revenue/user (conversion rate).

In Table 20 we present the time (in seconds) spent, by the *SHõWA* framework and by Zabbix, to detect and repair from the anomalies. The time-to-repair goes from the period when the anomaly was injected until the recovery procedure ends.

Table 20: Time-To-Repair (sec): *SHõWA* framework versus system-level monitoring (Zabbix)

	SHõWA	Zabbix
Memory consumption	788	848
CPU consumption	421	432
Threads consumption	127	1168
Application change	73	–

The results presented in Table 20 are similar to those obtained in the study, presented in Chapter 4 about different monitoring techniques used in Web-based applications. *SHõWA* has detected the four anomalies used while Zabbix has only detected three of them. It was also the first to detect and recover from all the anomalies tested.

It is noted that the results obtained by using Zabbix can be different if the monitoring items and corresponding triggers are re-parameterized. In practice we witness the constant adjustment of these parameters by system administrators to adjust the monitoring parameters according to the application and system where it runs. In this context, *SHõWA* facilitates the monitoring process. It uses a single threshold value to detect anomalies for all transactions, independently from the system, avoiding defining a threshold value for each type of transaction/system.

The *SHõWA* framework was also efficient in detecting, pinpointing and recovering from anomalies, including those that are driven by changes in the application and that are hard to be detected and pinpointed by monitoring systems, such as Zabbix.

7.5 Conclusions

In this chapter we evaluated the recovery mechanism included in the *SHõWA* framework. The recovery mechanism comprises three modules - *Recovery Planner*, *Executor* and *Effector* - and it is invoked whenever a performance anomaly is detected and pinpointed through the data analysis modules included in the *SHõWA* framework.

The *SHõWA* recovery mechanism was tested through an experimental study. In the study it was used a test environment similar to a production system. The TPC-W was adopted as the benchmarking application. It was running on an environment provided with high-availability mechanisms and scalability services. We injected four anomaly scenarios that disrupt the proper functioning of the application or system. For each anomaly scenario it was defined a recovery procedure. This procedure ensures the controlled recovery of the service or system. The experimental analysis allowed us to make a comparison between a controlled recovery and a recovery that simply takes advantage of the high-availability mechanisms provided by the clustering services. We also made a comparison about the time-to-repair achieved by a system-level monitoring tool and *SHõWA*.

According to the results obtained it can be said that:

- *SHõWA* was able to autonomously detect and heal the application, under different anomaly scenarios, avoiding the negative impacts on the response time and server throughput, and without causing any visible error or loss of the work-in-progress;

- The results observed with cold-recovery show that mechanisms for high-availability does not prevent the occurrence end-user failures. In addition to not being able to cope with performance anomalies, we observed failures when we restarted one of the cluster instances to remove the accumulated error conditions and freeing up the system resources. As can be seen from the results, the adjustment of the load balancing included in the recovery procedures is one type of controlled actions that prevents the occurrence of such failures;

- When compared to a system-level monitoring tool, *SHõWA* was able to detect, pinpoint and recover from all the anomaly scenarios tested with lower recovery time.

The achievements of this study allows us to conclude that *SHõWA* assumes a critical role to reduce the complexity and the time required to detect, pinpoint and recover from performance anomalies in Web-based applications.

Chapter 8
Self-adaptation for Cloud-hosted Web-based Applications

KEY POINTS

- ◇ Cloud computing is the fastest growing field in Information Technology. The *elasticity* of resources, the pay-per-use cost model and the possibility, at least theoretically, to achieve unlimited resources prove to be very advantageous for this growth to occur.
- ◇ The underlying benefits of cloud computing are shared resources and the flexibility to adapt the infrastructure. Thus, service level agreements (SLAs) span across the cloud and are offered by service providers as a service based agreement.
- ◇ Due to the complex nature of the environment, the downside of cloud computing, relative to SLAs, is the difficultly in determining the root cause for service interruptions and know how end-users are experiencing the service.
- ◇ Techniques able to monitor and report on performance of the cloud-hosted applications, from the consumer perspective, and perform self-adaption actions very quickly, are very important to address problems that may arise and which are difficult to be detected by the cloud service providers.
- ◇ In this chapter we evaluate the usefulness of *SHõWA* to detect service degradations and to quickly adapt the cloud infrastructure, in order to mitigate the impacts of those degradations.

The number of software applications, of a variety of domains, hosted in the cloud continues to grow. The *elasticity* of resources and the reduction of costs are factors that underpin this growth. Although, since cloud computing and consumer applications are in constant growth and adaptation, there are some issues that must be considered. Performance and the ability to provision more computational resources in a timely manner are two of these issues.

Several studies [Ueda 2010, Bjorkqvist 2012, Calcavecchia 2012, Tsakalozos 2011] reported that the performance variation of the cloud computing resources is high. These performance variations raise serious concerns regarding the manner in which the quality of service of the hosted applications is guaranteed.

The existence and management of service level agreements (SLAs) is a critical factor to be considered by both cloud providers and consumers. Although, it is widely recognized that most of cloud providers do not provide adequate SLAs for their cloud infrastructure [Durkee 2010, Suleiman 2012, Armbrust 2009]. This concern, along with the limitations of existing SLAs frameworks to express and enforce SLA requirements in an autonomic and fast manner, creates the need for techniques able to monitor the cloud-hosted applications from the consumer perspective and perform self-adaption actions very quickly.

In this chapter we present and evaluate the usefulness of *SHõWA* considering cloud-hosted Web-based applications. *SHõWA* provides an insight into the performance of the service, from the perspective of the end-user and, in this chapter, we show how it can be used to report service degradations to the cloud service provider and initiate the cloud provisioning process to mitigate scenarios of service degradation.

8.1 Elasticity and recovery in the cloud

Cloud computing presents several advantages that make it the fastest growing field in Information Technology. One feature which enhances all the others is *elasticity*. *Elasticity* concerns the ability to adapt the computational resources to workload changes by provisioning and deprovisioning resources, such that at each point in time the available resources match the current demand as closely as possible [Herbst 2013]. Taking advantage of *elasticity*, cloud consumers can replace up-front capital infrastructure expenses with low variable costs that scale with the business demands, achieve low rates of system administration labor and reduce hardware/software costs, as well licensing fees.

The services provided from a cloud must be seen from different perspectives: cloud service provider, cloud consumer and end-users. The cloud service provider perspective focuses on the satisfaction of the service levels agreements (SLAs) defined with the customers (e.g., server performance, network speed, availability) and in the reduction of operating costs (e.g., collocation, energy savings). From the perspective of a cloud customer, it is important to ensure the quality of service for end-users (e.g., response time, availability, security) while minimizing the operational costs. End-users represent the revenue return for the cloud customer and cloud provider. This return depends directly on the quality of service experienced by the end-users.

Depending on the perspective, there are several challenges for which the scientific and industry communities seek to answer. Thus, and more oriented to the cloud infrastructure management, in [Mao 2012], [Ueda 2010], [Laszewski 2012] are addressed the VM provisioning, the startup time, the response time and the throughput performance aspects, considering the differences between cloud platforms. Other studies, like [Bjorkqvist 2012], [Calcavecchia 2012] and [Tsakalozos 2011], focus on the importance of optimal VM placement (collocation). Optimal collocation is a fundamental aspect to minimize the operational costs and to minimize performance impacts due to the concurrency that may exist when several VMs are placed in the same physical server. In [Mazzucco 2010] authors address the VM collocation aspects as means to reduce the carbon footprint, save energy and maximize the cloud providers revenue.

Cloud *elasticity* is a very active research topic. In [Weinman 2011] and [Islam 2012] are proposed numerical measures on the *elasticity* of a cloud platform. Both reports, study the monetary penalty related with the waste by paying for resources that are not needed at the time (over-provisioning), and the cost of not providing enough resources (under-provisioning) for surges in workload. In [Weinman 2011], authors use the linear difference between the demand and the over and under allocation to determine the cost. In [Islam 2012], authors use penalties associated with SLAs, thus attributing a higher cost to when a given SLA is violated.

In a more operational approach, in [Ali-Eldin 2012] authors adopt queue modeling to model the cloud infrastructure and create an autonomic *elasticity* controller. On a regular basis, the controller observes the resources heterogeneity, the amount of delayed requests and the VMs service rates. Then, considering the workload and the available capacity, the controller proactively increases or decreases the number of VMs (horizontal scaling). A comparison between the proactive and a reactive controller shows that with the proactive controller the number of SLA violations is significantly lower. As a tradeoff the approach increases the costs of over-provisioning between 20% and 30%.

Software aging and rejuvenation in cloud and virtualized systems is also studied by several authors. In [Rezaei 2010] the authors measure the advantage, in terms of systems availability, by rejuvenating the VMs as well as the virtual machine manager (VMM). In [Bruneo 2013] authors seek to improve the VMM rejuvenation by modeling the aging process, taking into the consideration the number of running VMs and the SLA constraints.

The projects presented above focus essentially the perspective of the cloud service provider. Consumer-centric techniques able to monitor the cloud hosted applications and adapt the cloud infrastructure in an autonomic manner are seen as extremely important to ensure the adequate quality of service for the end-users. With this propose, in [Wang 2012] authors adopt a Kalman filter to dynamically estimate the CPU consumption of the application-level sharing multi-tenancy Web applications. Through experimentation authors show that with their approach it is possible to estimate the CPU consumption, independently from the non-determinism and multicollinearity of the workload. These results are very important to quickly detect performance anomalies and to tune the system accordingly to the workload demands. In [Sakr 2012] authors present a framework targeted for the dynamic provisioning of the database tier hosted in a cloud infrastructure. The framework takes into consideration the consumer perspective. It continuously monitors the database workload, tracks the satisfaction of the application-defined SLA and scales out/in the database tier when necessary. In a more commercial perspective, Amazon provides CloudWatch [Amazon CloudWatch]. CloudWatch is a monitoring service that allows to collect metrics about the cloud resources and applications running on EC2 instances. By using CloudWatch it is possible to set up thresholds, spot trends, and take automated actions over the cloud infrastructure to keep applications and businesses running as smoothly as possible.

Like in [Sakr 2012], *SHõWA* takes into consideration the service from the consumer perspective. It collects application-level data and detects performance anomalies motivated by workload variations, resource consumption or application changes. When an anomaly is detected, it interacts with the cloud service provider, in order to self-adapt the computational resources allocated to the service. Our approach differs from [Sakr 2012], to the extent that we take into consideration workload variations as well performance anomalies. *SHõWA* also promotes the horizontally/vertically scaling of the resources allocated to application server tier in order to mitigate application-level performance anomalies.

8.2 SHõWA and cloud provisioning

The *SHõWA* framework, presented in Chapter 3, is made up from several building blocks that cooperate each other to achieve the self-healing property. The cooperation between the modules corresponds to the MAPE control loop performed by an autonomic element [Ganek 2003]: Monitor, Analyze, Plan and Execute.

In this chapter we show how *SHõWA* can be used to monitor Web-based applications hosted in the cloud and how it can be used to adapt the cloud infrastructure in order to mitigate the impact of workload and performance anomalies.

8.2.1 *SHõWA* : Monitoring

The monitoring is done by the *Sensor* and *Data Preparation* modules.

The *Sensor* module gathers system, application and application server parameters at runtime. It makes use of Aspect-Oriented Programming (AOP) [Kiczales 1997], and do not requires manual changes to the application source code. As referred before, it collects data with two different levels of granularity: *user-transaction* level monitoring and *profiling* level monitoring. In the *user-transaction* level it intercepts and measures the server response time of the user-transactions and gathers different system and application server parameters (e.g., CPU load, JVM heap memory, number of open files, number of running threads). In the *profiling* level, it intercepts and records the execution time of the calls involved in the user-transactions call-path. To minimize the performance impact induced with the *profiling*, the *Sensor* includes adaptive and selective algorithms. These algorithms take into consideration the state of the system to determine, at runtime, the sampling frequency and the list of calls to be monitored. The algorithms were presented in Chapter 3 and its advantages, in terms of low performance impact induced on the application, are presented in Chapter 5.

The data gathered by the *Sensor* module is sent to the *Data Preparation* module. The *Data Preparation* module aggregates the data in time intervals, and characterizes the mix of user-transactions in each interval. The mix of transactions is determined by the percentage of transactions existing in the intervals of analysis.

8.2.2 *SHõWA*: Data analysis

The data analysis is performed by the *Workload Variation Analysis*, *Performance Anomaly Analysis*, *Anomaly Detector* and *Root-cause Failure Analysis* modules. These modules kept track of the Spearman's rank correlation coefficient ($\boldsymbol{\rho}$) [Zar 1972] between two variables $\boldsymbol{(X, Y)}$ with the purpose of: (1) verify if a performance variation is due to a workload change or if it is a symptom of performance anomaly; (2) verify if there is a system or application server change associated with the performance anomaly; (3) verify if there is an application or remote service change associated with the performance anomaly.

8.2.2.1 Performance anomaly analysis

The *Performance Anomaly Analysis* module includes the Algorithm 4, presented in Chapter 3. It computes the impact of dissociation ($\boldsymbol{F}$) between the accumulated value of response time per user-transaction and the number of user-transactions processed.

The algorithm takes as input a series of $\boldsymbol{n}$ intervals of data, a vector $\boldsymbol{X_{t,k}}$ and a vector $\boldsymbol{Y_{t,k}}$,written as $\boldsymbol{x_i}$ and $\boldsymbol{y_i}$, where $\boldsymbol{i = 1, 2, ..., n}$, $\boldsymbol{t}$ is a user-transaction and $\boldsymbol{k}$ the workload mix key. It measures the dependence between $\boldsymbol{X_{t,k}}$ and $\boldsymbol{Y_{t,k}}$ using the Spearman's rank correlation coefficient - $\boldsymbol{\rho}$ - and determines its variation by calculating the difference between $\boldsymbol{\rho}$ and the maximum value of $\boldsymbol{\rho}$ previously obtained. The algorithm also measures the user-transactions response time variation of the user-transactions. In this case, it is taken the difference between the response time observed in the current interval of analysis and the average response time observed in the first half of the previous intervals. The impact of dissociation ($\boldsymbol{F}$) returned by the algorithm corresponds to the product between the square roots of the variation in $\boldsymbol{\rho}$ and the variation observed in the user-transaction response time.

Under normal behavior the accumulated value of response time per user-transaction and the number of user-transactions processed are statistically associated. Thus, the impact of dissociation $\boldsymbol{F}$, returned by the algorithm, will be close to zero. If the response time deviates from the number of user-transactions processed, then $\boldsymbol{F}$ increases.

8.2.2.2 Workload variation analysis

The *Workload Variation Analysis* module aims to detect workload contention scenarios. This module takes as input a series of $\boldsymbol{n}$ intervals of data containing the accumulated response time $\boldsymbol{X}$ and total number of requests waiting in the application server queue $\boldsymbol{Y}$. The data is written as $\boldsymbol{x_i}$ and $\boldsymbol{y_i}$, where $\boldsymbol{i = 1, 2, ..., n}$ intervals. The correlation between the data is determined using the Spearman's rank correlation coefficient - $\boldsymbol{\rho}$.

From the analysis point of view it is expected that the $\boldsymbol{\rho}$ value remains close to zero and be stable over time. If $\boldsymbol{\rho}$ grows, then it means that there are requests waiting in the application server queue, suggesting the existence of an extra load. This extra load could be detected together with the occurrence of a performance anomaly, highlighting its impact on the user-transactions response time.

8.2.2.3 Anomaly detector

The *Anomaly Detector* module verifies if there is a system or application server parameter associated with a workload change or a performance anomaly. In this module, $\boldsymbol{X}$ is defined as the total number of user-transactions per interval and $\boldsymbol{Y}$ contains the cumulative value of each parameter (e.g., amount of JVM memory used) per interval of analysis. If the number of requests increases, then the cumulative value of each parameter should also increase. When, for some reason, the cumulative value of a parameter increases, and that is not motivated by an increase in the number of requests, then the $\boldsymbol{\rho}$ degree will decrease, highlighting the parameters associated with the performance anomaly.

8.2.2.4 Root-cause failure analysis

The *Root-cause Failure Analysis* module aims to verify if a performance anomaly is associated with an application or remote service change. In this module, $\boldsymbol{X}$ is defined as the frequency distribution of the transactions response time and $\boldsymbol{Y}$ as the response time frequency distribution of the calls belonging to the transaction call-path. Only the transactions that have reported a performance anomaly are analyzed in this step. The set of calls potentially associated with a performance anomaly are detected when the $\boldsymbol{\rho}$ degree remains constant or increases, while the $\boldsymbol{\rho}$ of the other calls decreases. The variation in $\boldsymbol{\rho}$ occurs when the calls response time becomes highly correlated with the delay observed in the user-transaction response time.

8.2.3 *SHõWA* : Recovery

The recovery service involves three modules: *Recovery Planner*, *Executor* and *Effector*. When a workload change or performance anomaly is detected and pinpointed, the *Recovery Planner* module selects automatically a recovery procedure. The recovery procedure is executed by the *Executor* module. The *Executor* module implements the recovery actions through the *Effector* module, and receives feedback on its implementation.

The recovery rules included in the *Recovery Planner* module are procedure-based and selected by a rule like: `IF (anomaly == <type-of-anomaly>) THEN pickRecProcX`

8.2.4 *SHõWA*: interaction between cloud consumer and cloud provider

The provisioning of resources, requested by the cloud consumers, can be only done through a cloud platform management interface that is typically managed by the cloud provider. So, instead of acting directly on the managed resource, *SHõWA* must interact with the cloud provider services. The interaction between *SHõWA* and the cloud provider is illustrated in Figure 53.

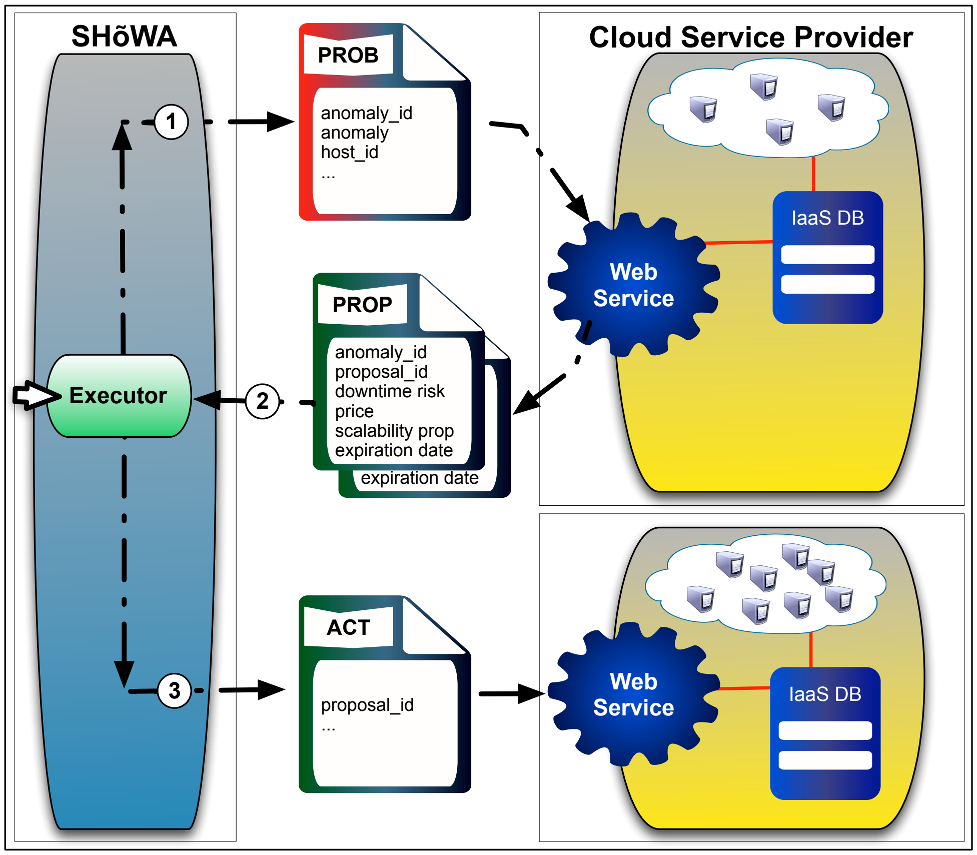


Figure 53: Interaction between *SHõWA* and the cloud service provider

As illustrated in Figure 53, when *SHõWA* detects a workload or performance anomaly it initiates the resource provisioning, indicating, through a Web Service, the problem it has detected. The Web Service is part of the services provided by the cloud provider. Upon receiving a request, the cloud service provider Web Service responds to *SHõWA*, sending provisioning proposals. These proposals include the price, the provisioning type (e.g., horizontally/vertically scaling, live migration), the risk of downtime that might occur during the provisioning process and the expiration date. *SHõWA* selects the proposal that best fits the rules/SLAs defined in the *Recovery Planner* module, and the adaptation is finally performed by the cloud provider.

8.3 Detecting anomalies and self-adapting the cloud

In this section we present and evaluate the usefulness of *SHõWA* considering cloud-hosted Web-based applications. More specifically, we:

- *Investigate the utility of SHõWA to cross the application state to the state of the cloud infrastructure*;
- *Analyze the behavior of SHõWA for the dynamic provisioning of resources, distinguishing workload changes from performance anomalies*;
- *Verify the role of SHõWA to ensure the quality of service of Web-based applications upon the occurrence of performance degradation scenarios*;
- *Analyze the cloud resources provisioning and cost management from the cloud consumer perspective.*

To address these objectives, we prepared a test environment on which we performed several experiments. The test environment consists of a set of physical servers that host multiple virtual machines (VMs). The environment is managed through the OpenNebula [OpenNebula] cloud management tool. Between other applications, the cloud environment runs a Web-based benchmark application composed by one or more application servers, backed by a database server and provided with load balancing and session replication mechanisms. The benchmarking application is continuously monitored by *SHõWA* and it is submitted to workload and resource contention scenarios. As *SHõWA* detects anomalies, it initiates the cloud provisioning process, in order to mitigate the quality of service degradation. The tasks performed by *SHõWA* and the dynamic provisioning of resources are done in an autonomic manner.

8.3.1 Experimental evaluation

In this section we present the experimental evaluation, conducted to access the ability of the *SHõWA* framework to detect and self-adapt the cloud infrastructure. We start by describing the test environment, and then the experimentations conducted. For each experimentation we evaluate the role of *SHõWA* for the dynamic adaptation of the cloud infrastructure, and how it is essential to mitigate the degradation of service.

8.3.2 Testbed

The testbed is illustrated in Figure 54. We adopt the bookshop implementation of TPC-W benchmarking application [Smith 2001] and deploy it in the "cyclone" cloud: a private cloud used for different research projects at the University of Coimbra.

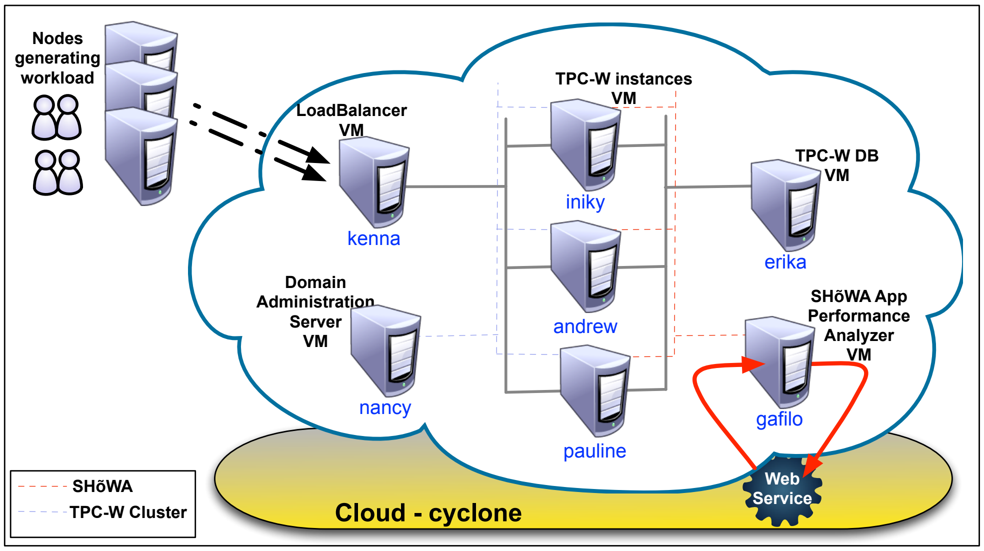


Figure 54: Test environment: Web-based application running in a private cloud

TPC-W uses a MySQL database as a storage backend and contains its own workload generator (RBE) that emulates the behavior of the end-users. The TPC-W application server instances are running in a cluster configuration, controlled by the "Domain Administration Server" VM. The cluster service provides session replication, allowing the user-transactions to continue processing even when one of the TPC-W application server instances becomes unavailable. The client requests are sent to a layer-7 load balancer (HaProxy) that forwards the requests to the active TPC-W application server instances. Each TPC-W application server instance includes a *SHõWA Sensor* module that collects application, system and application server parameters. The collected data is sent to the "*SHõWA* Application Performance Analyzer" VM. When a performance anomaly is detected, the "*SHõWA* Application Performance Analyzer" interacts with the cloud platform through a Web Service. The cloud adaptation proposals are prepared automatically by the cloud provider and applied according to the decisions taken by the "*SHõWA* Application Performance Analyzer".

The "cyclone" cloud is composed by several physical servers. Each physical server holds multiple VMs. "Cyclone" is managed by OpenNebula [OpenNebula]. OpenNebula is an open-source solution for building and managing virtualized enterprise data centers and enterprise private clouds. It offers an unified management console, allowing to provision and manage the cloud resources with just a few clicks. The cloud infrastructure state is stored in a SQLite database.

The specs of the cloud infrastructure are summarized in Table 21.

Table 21: Cloud "Cyclone": physical servers and virtual machine specs

kenna (i7 2600 3.4 Ghz - 4 cores - 8 threads; 16 GB MEM)		
VMs	#VCPU	MEM GB
LoadBalancer Haproxy	2	2
VM176	2	4
VM150	4	1
nancy (i7 2600 3.4 Ghz - 4 cores - 8 threads; 16 GB MEM)		
VMs	#VCPU	MEM GB
Domain Administration Server	4	4
VM149	4	1
inky (i7 2600 3.4 Ghz - 4 cores - 8 threads; 16 GB MEM)		
VMs	#VCPU	MEM GB
TPC-W instance	1	2
VM175	2	4
VM177	2	4
andrew (i7 2600 3.4 Ghz - 4 cores - 8 threads; 16 GB MEM)		
VMs	#VCPU	MEM GB
TPC-W instance	1	2
VM174	4	4
pauline (i7 2600 3.4 Ghz - 4 cores - 8 threads; 16 GB MEM)		
VMs	#VCPU	MEM GB
TPC-W instance	1	2
VM178	2	4
gafilo (i7 2600 3.4 Ghz - 4 cores - 8 threads; 16 GB MEM)		
VMs	#VCPU	MEM GB
SHõWA **App Perf Analyzer instance**	4	4
VM173	2	4
erika (Xeon z5355 2.66 Ghz - 4 cores; 12 GB MEM)		
VMs	#VCPU	MEM GB
TPC-W DB	4	2

The cloud nodes were created using the XEN virtualization technology and organized as an Infrastructure-as-a-Service (IaaS) in OpenNebula. The nodes responsible for generating workload are outside of the cloud. These nodes include a 3GHz Intel Pentium CPU and 1GB of RAM. All the nodes run Linux 2.6 kernel version and are interconnect through a 100Mbps Ethernet LAN.

8.3.2.1 Cloud self-provisioning scenarios

Cloud self-provisioning, or self-service provisioning, refers to the capacity of obtaining and removing cloud services (e.g. applications, infrastructure) without requiring the assistance of IT staff. This capacity is also referred as *elasticity on demand*. It aims to adapt the amount of computational resources in an autonomic manner, such that the available resources match the service demand as closely as possible. Thus, it can be used to avoid the costs of over-provisioning or scenarios of service degradation due to the under-provisioning of system resources. Amazon offers this kind of functionality through a service called *auto scaling*. This service allows the Amazon EC2 customers to define rules to automatically adjust the amount of computational resources/instances.

Cloud scaling can be done either horizontally, by increasing or decreasing the number of VMs allocated, or vertically, by changing the hardware configuration (e.g. CPU or memory) of already running VM. Scalability is undoubtedly a great advantage, but there is an analysis that needs to be considered. The analysis consists in knowing the cost of having an infrastructure over-provisioned, versus the losses that may result from the performance impact caused by the *elasticity on demand*. For example, by using the Amazon EC2 pricing calculator, to date have an environment like the one illustrated in Figure 54, but with only 1 TPC-W application server VM, with 1 virtual CPU and 2 GB of memory, costs $751.26 per month (hosted in the Europe (Ireland) Amazon data center). Have the same environment, with 1 extra application server instance, represents an increase of 8.3% on the total cost.

Depending on the type scalability, the performance impact caused by scalability process might change. From the cloud consumer perspective, scaling horizontally does not mean just having another VM. The process entails the deployment of the application, which when done at runtime can impact the service performance or functionality, leading to financial and reputation losses (e.g. an increase of 500 milliseconds in the access latency to Google results in a loss of 20% of its traffic, and an increase of 100 milliseconds in the access latency to Amazon results in a reduction of 1% of sales [Linden 2006]).

To compare the performance and functionally impact induced by the different scalability processes we conducted an experimental analysis that performs a scalability action when the system load is at 50% and 90% of its maximum capacity. In our experimental evaluation we take advantage of the two types of scalability. The test environment used for this experimentation is the same as illustrated in Figure 54, but it considers only 1 TPC-W application server VM, with 1 virtual CPU and 2 GB of memory. The TPC-W application server VM scales vertically by adding more virtual CPUs. It scales horizontally in two different ways: (1) by creating another VM, with 1 virtual CPU and 2 GB of memory, and deploying the TPC-W instance; (2) by performing a live migration of the TPC-W application server VM to another physical server, followed by an increase in the number of virtual CPUs by 1. To compare the different scalability approaches we measured the percentage of end-users affected by a response time delay superior to 100 milliseconds, the percentage of end-users affected by a response time delay superior to 500 milliseconds and the percentage of end-users affected by HTTP errors. The results are presented in Table 22.

Table 22: Performance and functionality impacts caused by the scalability process

Type	Load	Action	%+100ms	%+500ms	%errors
Baseline	90%	–	0.6	0.0	0.0
Vertical	90%	Add 1 VCPU	0.7	0.0	0.0
Horizontal	90%	Add 1 TPCW inst	3.8	1.6	0.0
Horizontal	90%	Live-migration	3.2	35.8	7.8
Baseline	50%	–	0.0	0.0	0.0
Vertical	50%	Add 1 VCPU	0.0	0.0	0.0
Horizontal	50%	Add 1 TPCW inst	0.2	0.1	0.0
Horizontal	50%	Live-migration	5.7	22.4	4.8

As can be seen from Table 22, the impact caused by the different scalability approaches is lower when the load on the system is at 50% of its maximum capacity. Since the vertical scaling does not require the deployment of the application at runtime, the performance impact caused by the scalability process is lower. The live migration was the technique that caused a higher impact. During the live migration warm-up phase phase, 3.2% of the user-transactions suffered a response time delay of 100 milliseconds and 35.8% suffered from a response time delay higher than 500 milliseconds. During the live migration stop-and-copy phase, 7.8% of the user requests failed due to HTTP errors. Create and deploy another TPC-W application server instance - horizontal scaling - has penalized 3.8% of the user requests with 100 milliseconds of response time delay and 1.6% of the users have experienced a response time delay superior to 500 milliseconds. With regard to this scalability approach is worth noting that the response time penalty comes from the fact that the deployment of the application is done at runtime.

Despite the vertical scaling to be better in terms of performance impact, it should be noted that horizontal scaling is extremely useful to increase the system redundancy enhancing the service availability.

Highlighting the main points mentioned in this section, to obtain the maximum efficiency and minimum cost in a cloud environment: (1) the resource provisioning must be done as close as possible to the time that is required; (2) the scalability approach, if not carefully implemented, may lead to financial loss greater than a over-dimensioning scenario. For us, optimal *elasticity* occurs when the resources allocation equals demand and the allocation allows as many service levels to be met as possible. Next we analyze how *SHõWA* contributes for these points.

8.3.2.2 *SHõWA*: Threshold values adopted

In our experimental evaluation, the *SHõWA Data Preparation* module aggregates data at intervals of 5 seconds. Per interval it determines the workload mix key.

A performance anomaly is detected by the *Performance Anomaly Analysis* module when the impact of dissociation ($\boldsymbol{F}$) between the response time and the number of user-transactions processed is greater than 10. A workload variation is detected when the *Workload Variation Analysis* module returns a variation of $\boldsymbol{\rho}$ (Spearman's rank correlation) higher than 0.1 degrees. The performance anomaly is pinpointed when the $\boldsymbol{\rho}$ variation, returned by the *Anomaly Detector* or *Root-cause Failure Analysis*, is higher than 0.1 degrees. In the basis of these values, as indicated in a previous subsection, are studies that were previously conducted.

Upon the scalability proposals received from the cloud service provider, and considering the results presented in the previous section, the *Recovery Planner* module gives preference to vertical scalability. If it is not possible, it performs horizontally scalability, adding more VMs and deploying the TPC-W application application server instance at runtime.

8.3.2.3 Cloud adaptability with *SHõWA*: System overload

In this experimentation we analyze the adaptability driven by the *SHõWA* framework, taking into account a scenario of system overload. A scenario of system overload affects the user-transactions response time, raising the risk of quality of service degradation. In a cloud environment, it becomes easier to deal with a problem of this kind. Cloud infrastructure is *elastic* by nature, thus allowing to adjust the computational resources to match the workload demand.

From the analysis point of view, a system overload scenario allows to verify the ability of the *SHõWA* framework to detect situations of service degradation and trigger the necessary cloud infrastructure adaptations to mitigate the anomaly.

The TPC-W contains its own workload generator (RBE) that simulates the activities of a business oriented transactional Web server. The workload exercises a breadth of system components associated with such environments and reports some metrics, such as, the response time and the server throughput. Despite its usefulness, and unless a large number of nodes are available, the RBE implementation presents some challenges to determine the server's maximum capacity. An alternative, and widely used tool to find out the application and server limits, is *httperf* [Mosberger 1998]. In its simplest form, it allows to choose a given URI, or a list of URIs, to be accessed by a given number of parallel requests. With some efforts, it also allows to submit a workload that is more representative of the end-users behavior.

Taking this into consideration, we started by performing a workload execution using the TPC-W RBE, followed by a test with *httperf*. This was achieved by extracting the session requests sequence within each session from the log file generated from the TPC-W RBE client and saving that information in a "session_file" to be used by the *httperf*. The "session_file" was replayed by *httperf* varying the number of concurrent requests from 34 to 760 requests per second. Each experiment ran for 90 minutes, beginning with a 10 minutes warm-up period and ending with a 10 minutes ramp-down period.

The average response time obtained with the execution of the workload is illustrated in Figure 55. The results illustrated in the figure reflect different options in terms of infrastructure capacity. The labels *A*, *B*, *C*, *D*, *E* and *F* refer to different infrastructure states, in which scalability is not considered. The *SHõWA* refers to the self-provisioning made using the *SHõWA* framework. The vertical lines indicate the times when *SHõWA* has detected the problem and triggered the scalability process. The tags *E1*, *E2*, *E3*, *E4* and *E5* corresponds to the adaptations *A-to-B*, *B-to-C*, *C-to-D*, *D-to-E* and *E-to-F* illustrated in Figure 57.

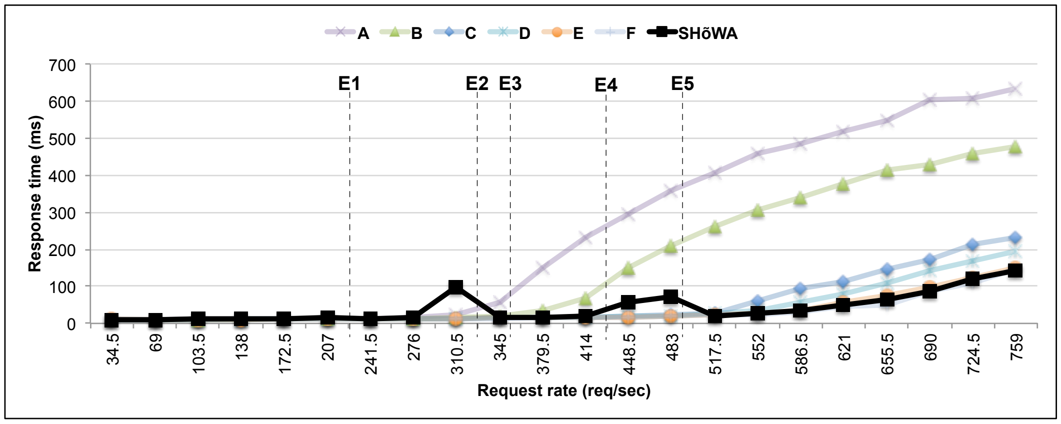


Figure 55: Response time considering different options in terms of infrastructure capacity and self-provisioning (*SHõWA*)

In Figure 56 is illustrated the server throughput, considering different static infrastructure configurations and the self-provisioning achieved with *SHõWA*.

The results, illustrated in the figures above, reflect different options in terms of

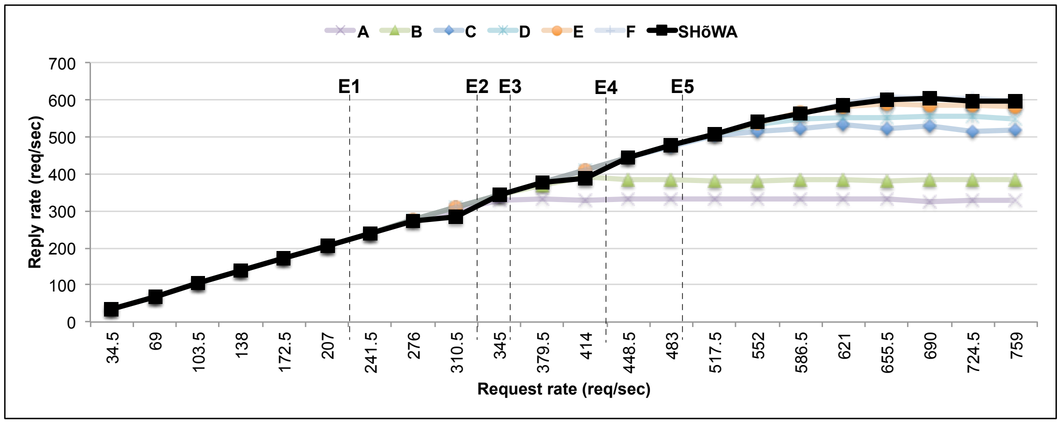


Figure 56: Throughput considering different options in terms of infrastructure capacity and self-provisioning ($SH\tilde{o}WA$)

infrastructure capacity. These options are illustrated in Figure 57.

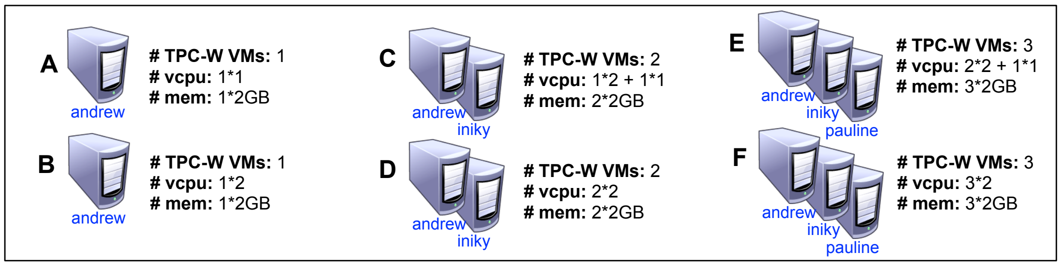


Figure 57: Cloud scalability configurations used for the TPC-W application server instances

As shown in the figures above, with the self-provisioning provided by $SH\tilde{o}WA$, the user-transactions response time and the server throughput is dynamically set to normal levels as the load increases. This is because the computational capacity is properly adjusted, making it close to the scenarios in which the infrastructure is over-provisioned, but with the advantage of saving money due to the over-provisioning of system resources before they are needed.

In Figure 58 and Figure 60 we illustrate the percentage of end-users affected by a response time delay of 100 milliseconds and 500 milliseconds respectively. This analysis allows obtaining, in more detail, the impact on the user-transactions response time due to under-provisioning, and also the impact achieved when the infrastructure is adapted automatically by $SH\tilde{o}WA$.

In terms of response time experienced by the end-users, and when compared to the infrastructure identified by the letter F (the infrastructure over sized used in this study),

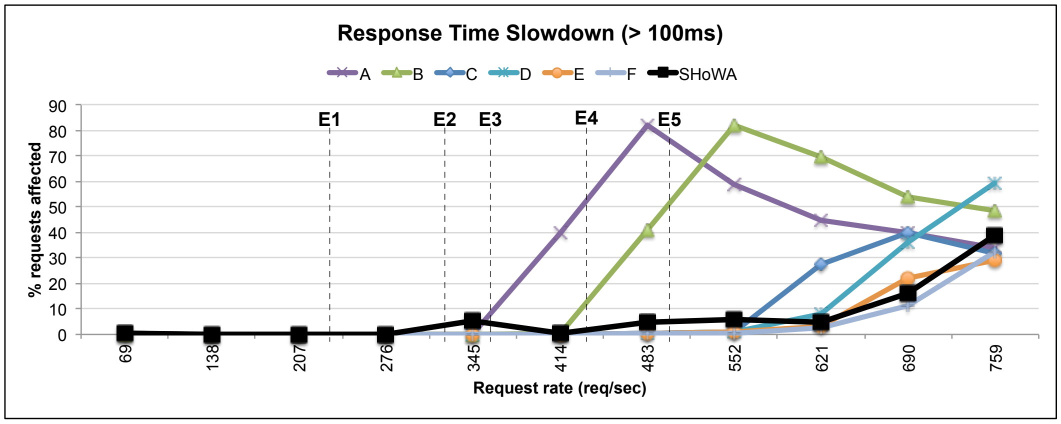


Figure 58: Percentage of end-users affected by a response time delay higher than 100 milliseconds: static provisioning versus *SHõWA* based self-provisioning

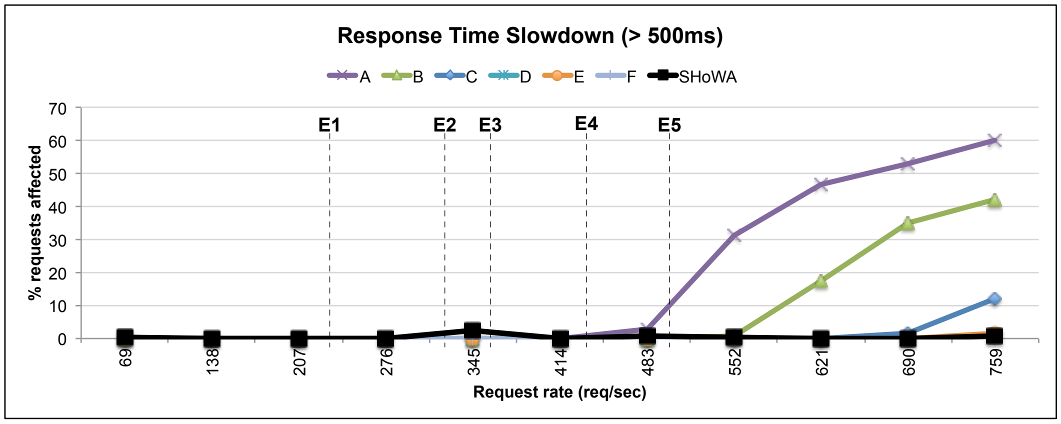


Figure 59: Percentage of end-users affected by a response time delay higher than 500 milliseconds: static provisioning versus *SHõWA* based self-provisioning

with the self-provisioning provided by *SHõWA*, the percentage of end-users affected by a response time delay of 100 milliseconds increased just 2.3% and the percentage of end-users affected by a response time delay of 500 milliseconds was 0.4%. When compared with the other static infrastructure states, the percentage of end-users affected by a response time delay is 2 to 10 times lower.

It should also be noted that the response time impact observed with the self-provisioning is mainly due to the horizontal scaling and in particular to the deployment of TPC-W at runtime. This impact may be reduced if the scaling is vertical or if there are TPC-W instances ready to be activated.

8.3.2.4 Cloud adaptability with *SHõWA*: CPU consumption

In this experimentation we analyze the adaptability achieved with *SHõWA*, when one of the TPC-W VM instances is affected by CPU load. In a cloud infrastructure, the CPU consumption can occur on the physical server, on the VM or on both. Regardless of where it occurs, its occurrence may affect the application behavior. In this context, instead of monitoring the CPU load in isolation, *SHõWA* assesses whether the application is being affected by a performance anomaly and establishes associations with the cause behind the anomaly.

Based-on the analysis provided by *SHõWA* and on the infrastructure state, it becomes easier to identify the set of actions that produce a more effective recovery. For example, faced with a CPU contention on the physical server, horizontal scaling seems a better option than vertically scaling. However, if the CPU load is only on the VM and if it is possible to increase its computational resources, then vertically scaling could be a better option.

For this experimentation we considered the test environment illustrated in Figure 54, but provided with only 1 TPC-W application server VM instance. The TPC-W VM instance runs on the physical server *iniky*. We evaluated different adaptability scenarios. Our first scenario is static and it considers the VM equipped with 1 virtual CPU and 2 GB of memory. The second scenario is also static and it considers the VM equipped with 4 virtual CPU and 2 GB of memory. The self-provisioning scenario starts with a VM equipped with 1 virtual CPU. Since *iniky* has 3 CPUs available the VM can scale vertically up to 4 virtual CPUs. After that is can scale horizontally. The scalability options are returned by the cloud provider, and chosen automatically according to the rules defined in the *Recovery Planner* module. For the reasons presented before, it gives preference to vertical scalability.

In each type of infrastructure we injected a CPU load and evaluated the response time impact caused by the CPU consumption. From the results, we observe how *SHõWA* guides the self-provisioning process, evaluating in how it contributes to mitigate the impact caused by the CPU load.

To impose load on the system we used the `stress` [Stress] tool. This is a simple tool which allows, between several options, to inject a gradual **CPU consumption**. After 35 minutes from the beginning of the experimentation, were launched two processes consuming the total CPU. After eight minutes it was released one more process and after a further 20 minutes another case was launched. In total we have 4 processes in the VM that are competing in parallel with the application server for processing capacity. The experiment ran for 90 minutes, beginning with a 10 minutes warm-up period and ending with a 10 minutes ramp-down period.

The results obtained with each type of environment are illustrated in Figure 60. The results show the percentage of end-users affected by a response time delay exceeding

100 and 500 milliseconds respectively. The setting "1 VCPU" corresponds to a scenario of under-provisioning and the setting "4 VCPU" to a scenario of over-provisioning. The infrastructure is adjusted by increasing 1 VCPU at a time.

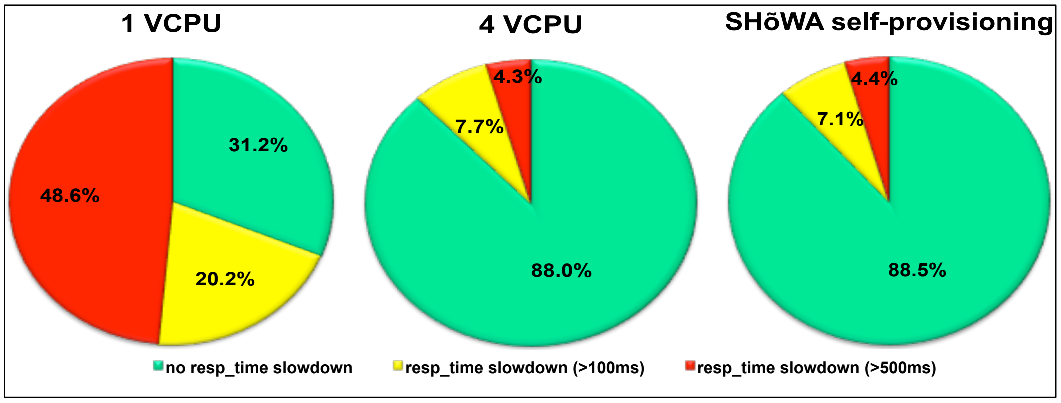


Figure 60: Percentage of end-users affected by a response time delay higher than 100 milliseconds and 500 milliseconds: static provisioning versus *SHõWA* based self-provisioning

As shown in Figure 60, between the scenario in which we have 4 virtual CPUs (over-provisioning) and scenario in which the cloud infrastructure was self-provisioned with more virtual CPUs, the percentage of end-users affected by a response time delay of 100 milliseconds and 500 milliseconds is very similar. The setting in which we have 1 virtual CPU and the infrastructure does not scale-up, makes about 50% of users affected by delays exceeding 500 milliseconds in response time. The infrastructure has been re-dimensioned on average 25 seconds after the injection of more CPU load. In terms of server throughput, we observed that in the setting in which we have 1 virtual CPU, the number of requests processed by the TPC-W application server instances dropped 3.8%. In the settings we have 4 virtual CPU and self-provisioning, the throughput impact was inferior to 1%.

The results achieved with this experimentation reveals the advantage of the self-provisioning of resources.

8.4 Conclusions

In this chapter, we presented the usefulness of self-healing framework (*SHõWA*) for the dynamic adaptation of the cloud infrastructure, as means to mitigate performance anomalies in Web-based applications hosted in a cloud platform.

SHõWA allows to monitor (by means Aspect-Oriented Programming) Web-based applications, without require knowledge about its implementation details or changes to the source code. The collected data is analyzed by means of statistical correlation. The analysis distinguishes performance anomalies from workload variations, and verifies if there is a change in the system, application server, or application that is statistically correlated with the anomaly. When *SHõWA* detects a workload change or performance anomaly, it executes, automatically, a recovery procedure. Considering cloud-hosted Web-based applications, when *SHõWA* detects an anomaly it interacts with the cloud service provider to know which adaptations are possible and decide, according to the rules/customer SLAs defined, which adaptation procedure should be performed.

The experimental work considered a cloud platform, provided with *elasticity* services. We have injected different types of anomalies to disrupt the proper functioning of the application. For each anomaly we compared the performance impact, taking into consideration different infrastructure setups: under-provisioning, over-provisioning and self-provisioning controlled by the SHõWA framework.

From the results we observed:

- With *SHõWA* it becomes possible to combine the SLAs of both cloud consumer and cloud provider;
- *SHõWA* is able to autonomously detect workload and resources contention scenarios in cloud-hosted Web-based applications from the consumer perspective, and interact with the cloud service provider to self-adapt the infrastructure when needed;
- With *SHõWA* the costs of over-provisioning and the revenue and corporate reputation losses due the lack of computational resources, is more safeguarded;
- From the tests we performed, the cloud provisioning is made by *SHõWA* while the number of end-users affected by a response time delay is very low. According to the results:
  - When compared with the scenario of over-provisioning, the percentage of requests affected by a response time delay of 500 milliseconds was inferior to 0.5%;
  - the server throughput penalty observed was inferior to 1%.

The contribution of the approach presented in this chapter, substantiated by the experimental results seems very interesting. *SHõWA* provides to cloud customers more control over the service, allowing them to react more quickly and autonomously to the anomalies that may arise at runtime. This benefits the adoption of cloud environments and maximizes the business success for the different parties involved.

As future work, we will make a comparative study about the performance impact caused by monitoring solutions used by different cloud providers and check the time-to-provisioning provided by the cloud service providers. Extend the number of adaptation strategies and inject more anomalies is also in our plans.

Chapter 9
Prediction of performance anomalies

KEY POINTS

- ◇ The mean time to repair (MTTR) is considered a fundamental aspect for the availability of a system. The scientific community and industry are making every efforts to find solutions to achieve a low MTTR.
- ◇ Despite the progress made in reducing the MTTR, the ideal situation is to predict the occurrence of anomalies before they compromise the availability or performance of the service.
- ◇ Although, like any forecast, predict the occurrence of anomalies in dynamic systems such as the Web-based applications is not a simple task.
- ◇ The use of techniques such as regression analysis, time series analysis and machine learning algorithms are being explored by several authors. These techniques are used to model the application behavior, estimate the resources state and analyze patterns or trends to anticipate the future system state.
- ◇ In this chapter we present an extension of the *SHõWA* framework that aims to examine the feasibility of implementing a system for predicting performance anomalies in Web-based applications.

In the previous chapters we presented the *SHõWA* framework and how it does the system monitoring, detects and pinpoints anomalies and performs the necessary recovery actions to mitigate performance anomalies. The results obtained with *SHõWA* show that it can detect different types of anomalies with a low detection time. However, anomalies are detected only after they occur, which inevitably means that there are users affected. Whenever there are users affected, the revenue and image of the company is also affected.

Predict in advance the occurrence of anomalies is a significant contribution to avoid these constraints. In chapter we present an extension to the *SHõWA* framework - *SHõWA+*. The extension considers the use of machine learning algorithms and time series estimation models to predict in advance the occurrence of anomalies in Web-based applications. Our approach consists in modeling, by means of machine learning, the historical knowledge obtained through *SHõWA*. Then, based-on the current system state, we use time series models to estimate the future state of the resources and request to the machine learning algorithms the classification of these estimates. This way we intend to predict performance anomalies before they occur and avoid their impacts on the normal functioning of the service. We will, through experimental analysis, to present a proof of concept for our approach.

9.1 Predicting anomalies

There are several methods and approaches in the state of the art on failure prediction.

[Salfner 2010] presents a survey about online failure prediction methods. The survey is divided into four categories: failure tracking, symptom monitoring, detected error reporting and undetected error auditing. The idea behind the failure tracking is to draw conclusions about possible failures from failures that happened previously. Estimate the probability distribution of the time to the next failure from the previous occurrence of failures and the notion of co-occurrence, i.e., the fact that system failures can occur close together, either in time or space, can be explored to infer about failures that might occur in the near future. The symptom monitoring regards the assessment of system parameters, such as the amount of free memory, CPU usage or disk I/O, in order to predict service failures. The key notion is that the system parameters state can provide symptoms for predicting failures. For example, when memory is getting scarce, the system may first slow down (e.g., due to memory swapping or due to the garbage collector activity) and only if there is no memory left an *"OutOfMemory"* error is throw and a failure might result. Symptom-based online failure prediction methods frequently address non-failstop failures, which are usually more difficult to detect. Function approximation, classifiers, system models and time series analysis are some of the approaches used for symptom-based failure prediction. In the detected error reporting category, authors include approaches that use error reports as input to predict failures. Typically when an error is detected, the detection event is reported using some logging facility. Events occurring in a time window are analyzed in order to decide whether there will be a failure at some point in the future. Rule-based systems, co-occurrence, pattern recognition, statistical tests and classifiers are some of the prediction methods used in this category. The undetected error auditing category consists on actively searching for incorrect states within a system. The main difference between the detected error reporting and this method, is that rather than checking on data that is actually used or produced, the checking is active regardless whether the data is used at the moment or not. Perform a file system consistency check is an example that fits the undetected error auditing category.

From the list of categories presented above, the symptom monitoring is the one that stands out in terms of adoption for failure prediction in computer systems.

We found some remarkable projects performing symptom monitoring and analyzing data using function approximation, classifiers, system models or time series analysis lie in the area of software aging and rejuvenation. In [Garg 1998] authors present a methodology for detecting and estimating aging in operational software. Authors make use of statistical techniques to detect trends in the data and to validate the existence of aging. The estimated time to the parameters exhaustion is determined by means of slope estimation and linear approximation analysis. Thus, in presence of aging, it is possible to predict how long will take until one parameter reach a saturation point, putting at risk the normal operation of the system. In [Li 2002] and in [Vaidyanathan 1999] are presented extensions of this approach. In [Vaidyanathan 1999] authors adopt a semi-Markov reward model based-on the workload and the resources usage to predict the resource exhaustion over time. In [Li 2002] is adopted the autoregressive with auxiliary input model (ARX) to estimate the time to exhaustion of resources considering the workload received by the Apache Web server.

In [Alonso 2010] authors pointed out some important aspects to be taken into consideration when analyzing the effects of software aging. Considering the workload dynamics and different types of resources demands, authors propose a software aging prediction model, based-on machine learning (ML). The prediction model takes into account that software aging may change in a non-deterministic way and may affect multiple parameters. The machine learning M5P decision tree algorithm is used to model the software aging phenomena and predict the time to resource exhaustion. From the experimental results, the authors show that their approach predicts the time to resource exhaustion more accurately, than the time to exhaustion obtained through a simple linear regression model.

Another evaluation using of well know machine learning (ML) algorithms is presented in [Andrzejak 2008]. Authors evaluated the effectiveness of Naive Bayes, decision trees and support vector machines algorithms to model and predict software aging in presence of partial non-determinism and transient failures. Using these algorithms the authors show that it is possible to predict the system performance with good accuracy, even when the dataset used for training is small.

In [Kelly 2006], authors present an approach to predict the response time, as a function of workload mix. Authors consider three performance models which evolve from a simple model in which the volume of transactions is modeled as a function of the response time, until models that consider the queuing times and the resources utilization. Different datasets were calibrated using the least square residuals and the ordinary least square regression models, in order to compute the parameters that maximize the model accuracy. The results show that the response time for a real production application, calibrated under light load, can be predicted to within 10% of estimation errors. Under heavy and non-stationary workload the approach was able to predict the response time with the estimation errors varying between 13 and 33%.

Another method used to predict the system state consists on the analysis of business parameters. In [Poggi 2011], authors present a methodology to determine the thresholds of user satisfactions as the quality of service degrades, and to estimate its effects on actual sales. Authors adopt machine learning (ML) techniques to predict the expected sales volume over time and look for deviations over the expected values during overload periods that introduce performance degradation. Using a three years sales log, the authors estimated the impact that a higher response time can have on the business activities of an service. A portion of the sales dataset was trained using different classification algorithms found in WEKA (a popular suite of machine learning software) [WEKA]. Using the classifier with higher accuracy - M5P decision tree - the remainder of the dataset was used to predict the sales based-on the response time. The results achieved with their approach allow to detect inflection points where sales start to be affected by the application's response time.

The *SHõWA* framework extension, that we present in this chapter, fits the symptom analysis category. Our approach combines time series analysis and classifiers. Like in [Garg 1998, Vaidyanathan 1999] and [Li 2002] we use time series analysis to estimate the number of requests for each user-transaction and the state of system resources. Like in [Andrzejak 2008, Alonso 2010] and [Poggi 2011] we use machine learning algorithms to classify the data. Our aim is to anticipate the system state and predict performance anomalies that may lead to failures before they are experienced by end-users.

9.2 Predicting performance anomalies with *SHõWA*

Throughout this Thesis we present a self-healing framework targeted for detection, localization and recovery from performance anomalies in Web applications. The *SHõWA* framework is able to distinguish a performance anomaly from a workload variation. It also pinpoints anomalies by observing system, application server or application changes that are associated with a performance anomaly and executes recovery actions autonomously to mitigate the effects of an anomaly.

Regardless the advantages obtained with the detection and pinpointing of anomalies, the *SHõWA* framework is only capable of detecting performance anomalies after users experience some kind of service degradation (e.g., slow response time). In this context, devise a system able to anticipate the occurrence of anomalies is of great importance. To this end, we extended the *SHõWA* framework. As in [Alonso 2010] and [Andrzejak 2008], the extension makes use of machine learning (ML) classification algorithms and considers that the occurrence of anomalies is not deterministic. Our approach consists in using a dataset, trained by the ML algorithms. The dataset contains historical data about system. At runtime, the parameters collected by the monitoring system, are estimated using time series models. The estimates obtained are classified by ML using the algorithms previously trained. As result, we obtain an indication of the future state of the service with regard to the occurrence of possible performance anomalies.

9.2.1 Framework extension

The *SHõWA* framework was redesigned to accommodate the prediction of anomalies modules. For presentation purposes, the new framework will be designated by *SHõWA+*.

The *SHõWA+* framework is illustrated in Figure 61. It contains four extra modules: *Build Dataset*, *Train Classifier*, *Parameter Estimation* and *Classify*. The dataset is created by the *Build Dataset* module and it is then submitted for training using different machine learning algorithms. The dataset training is made in offline. At runtime, the prediction block uses the data gathered by the *Sensor* module to estimate the values for the different parameters. The estimations are then classified by the *Classify* module.

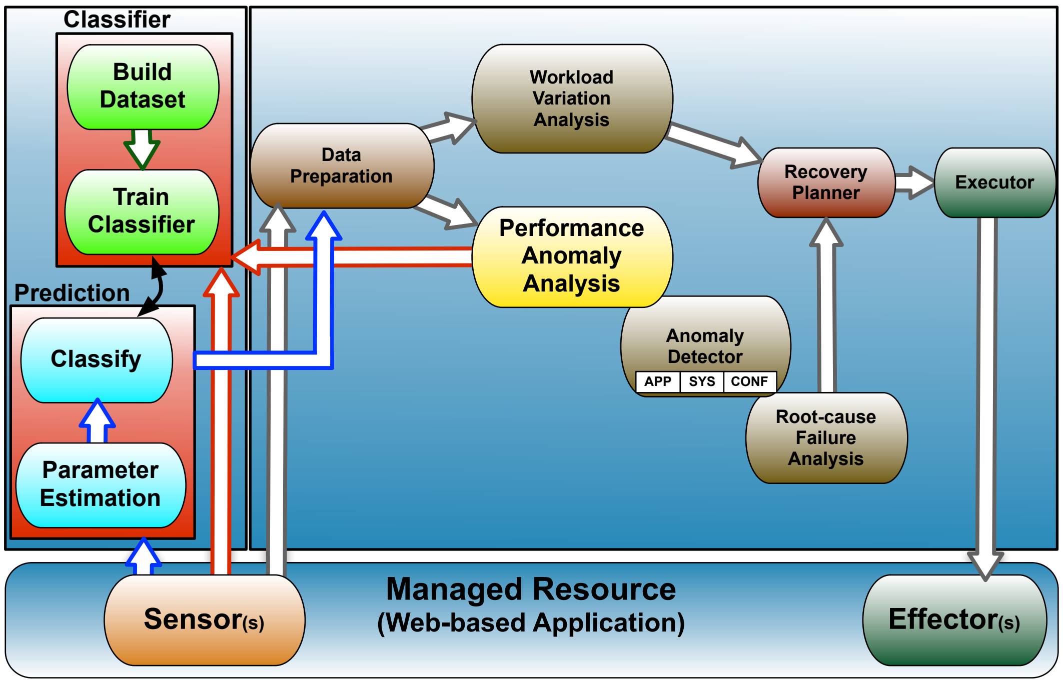


Figure 61: *SHõWA+* framework: extension of the *SHõWA* framework to support the prediction of performance anomalies

The *Build Dataset*, *Train Classifier*, *Parameter Estimation* and *Classify* modules are separated in two different blocks: Classifier and Prediction. This is a logic separation between the modules that run in offline (Classifier) from those running online (Prediction). The offline mode relates to the preparation and training dataset using historical data. Their actions are periodic and aims to determine the ML algorithm which maximizes the data classification accuracy. Thus, the prediction block can take advantage of a dataset trained according to a recent state of the system, as well, know the algorithm that best fits the classification of estimates. It combines the historical knowledge provided by the classifier block with runtime data to estimate and predict the future behavior of the system. This is done by the *Parameter Estimation* module, through time series estimation. The *Classify* module interacts with the *Train Classifier* module, to classify the estimated data. The result can be analyzed by the *SHõWA* framework in order to detect and pinpoint the causes behind the potential performance anomaly.

It should be noted that the prediction of performance anomalies is particularly useful for those cases in which the degradation happens slowly. In cases where there is a sudden degradation, the prediction mechanism may not be able to anticipate the performance anomaly.

9.2.2 Preparing the dataset

The dataset is prepared by the *Build Dataset* module using the data collected by the *Sensor* module and analyzed by the *Performance Anomaly Analysis* module.

As described previously, the *Sensor* module is implemented using a Java implementation of AOP - AspectJ [Laddad 2009]. It intercepts all the user-transactions as they arrive to the application server, measures their execution time and collects several system and application server parameters. The collected data is prepared and sent for analysis. The *Performance Anomaly Analyzer* module measures the correlation between the user-transactions response time and the number of transactions processed. It uses the Spearman's rank correlation coefficient, also known as $\boldsymbol{\rho}$ (Eq. 9.2.1), and the historical average of the response time for, according to Algorithm 4, quantify the impact of the dissociation between the response time and the number of user-transactions processed. The impact of dissociation is mapped to a value representative of a performance anomaly.

$$\boldsymbol{\rho = \frac{\sum_{i=1}^{n}(X_i - \bar{X})(Y_i - \bar{Y})}{\sqrt{\sum_{i=1}^{n}(X_i - \bar{X})^2}\sqrt{\sum_{i=1}^{n}(Y_i - \bar{Y})^2}}} \tag{9.2.1}$$

Thus, using the output provided by these modules is created a dataset with a structure equal to that presented in Table 23.

The class is achieved through a simple transformation. According to the output presented in Table 2, the impact of the dissociation between the response time and the number of user-transactions processed is transformed according to the logic presented in the Table 24.

As said before, the training dataset is regularly updated. Therefore, and as more historical information exists, the better is expected to be the result of the classification.

Table 23: Example of the attributes present in the dataset

Attributes	Description
`AvailableMem`	amount of available JVM memory in bytes
`RunnThread`	number of threads running on the application server
`AvailableFD`	number of file descriptors available on the system
`RunnClasses`	number of classes running on the application server
`NetRecv`	amount of network bytes recv by the Web server
`NetSend`	amount of network bytes send to the Web server
`DskRead`	amount of bytes read from the hard disk
`DskWrite`	amount of bytes written to the hard disk)
`CPU_IDLE`	percentage of CPU idle
`ProcRun`	number of processes running in the system run-queue
`ProcBlk`	number of processes blocked in the system run-queue
`ProcNew`	number of new processes entering to the system run-queue
`OpenConn`	number of open connections
`CurrQueue`	number of processes waiting in the application-server queue
Transaction_#	number of requests per user-transaction
Class	{GREEN, YELLOW, RED}

Table 24: Dataset - Class definition: GREEN suggests absence of performance anomalies; YELLOW is a symptom of performance anomaly; RED is a strong indication of performance anomaly

	[0.0, 5.0]	**]5.0, 10.0]**	**> 10.0**
Class	**GREEN**	**YELLOW**	**RED**

9.2.3 Classification algorithms

Among many ML algorithms available, we have chosen to use the Naive Bayes classifier, two decision trees (J48 and LMT) and a neural network model (MLP). These algorithms are included in the WEKA software package [WEKA]. WEKA is provided with tools for data pre-processing, classification, regression, clustering, association rules, visualization and simple output analysis.

The Naive Bayes (NB) is a simple probabilistic classifier. It assumes that the presence of a particular feature of a class is unrelated to the presence of any other feature. The J48 [Quinlan 1993], also known as C4.5, is a decision tree algorithm. It builds decision trees from a set of the dataset previously classified. Each node of the tree is chosen according to the normalized information gain criteria. The attribute with the highest normalized information gain is chosen to make the decision.

LMT stands for Logistic Model Tree [Landwehr 2005] and results from the idea of combining two complementary classification schemes: linear logistic regression and tree induction. The main differences between J48 and LMT rely on the adoption of the LogitBoost algorithm to build a logistic regression model for the root node and to build logistic regression models at all nodes in the tree. LMT also adopts recursive partitioning (CART) to prune the tree, producing usually smaller and accurate trees. A clear disadvantaged of LMT comparatively to J48 is the time required to build the model.

The MLP [Haykin 1994] is a feed-forward artificial neural network model that establishes the relationship between sets of input data into a set of appropriate output. It is composed by several layers of nodes, logically organized like a fully connected direct graph, except for the input nodes. Each node/neuron performs a comparison to decide which neuron should be called after. The MLP is trained using a supervised technique called back-propagation. This technique is a generalization of the least mean squares algorithm and consists on changing the connection weights after each piece of data is processed to minimize the amount of error between the observation and the expected predictions.

9.2.4 Estimation and classification of performance anomalies

The estimation of parameters to predict the resource exhaustion is explored in different research works. Techniques like the detection of trends combined with slope estimation and autoregressive moving average (ARMA) models were used by some authors [Garg 1998, Li 2002]. Both, techniques (slope-based estimations and pure ARMA models) assumes the data linearity, i.e., that data has a constant fluctuation over time.

Considering that Web applications are characterized by highly dynamic and non-deterministic workloads, we adopted two non-stationary models: ARIMA (Autoregressive Integrated Moving Average) [Box 2008] and Holt-Winters (Triple Exponential Smoothing) [Box 2008]. ARIMA and Holt-Winters are both adaptive and can model trends and seasonality. The main reason to choose these algorithms relies in how they consider the previous data: Holt-Winters gives more weight to recent rather than to distant observations. ARIMA tends to be more conservative, considering that all data have the same weight in the analysis. This combination may provide a suitable balance when estimating effects on the parameters that tends to degenerate with time and load.

Attending that a performance anomaly may result from different combinations of parameters, each parameter is estimated by up to N epochs ahead. The set of estimated parameters is then submitted to the classifier, that verifies if such combination may result in absence ("GREEN"), symptom ("YELLOW") or strong indication ("RED") of a performance anomaly.

9.3 Experimental study

In order to evaluate the potential of the *SHõWA+* framework to predict anomalies, we decided to conduct an experimental study. To this end, we proceeded with the *SHõWA+* framework extension, by implementing the *Build Dataset* and *Train Classifier* modules, as well as, the *Parameter Estimation* and *Classify* modules. The implementation was done using JAVA and it integrates time series estimation algorithms and the machine learning algorithms included in the WEKA tool.

These modules integrate with the existing ones in the *SHõWA* framework. The *Sensor* module provides fine-grain monitoring of key application/system parameters. These parameters are collected and analyzed by the *Performance Anomaly Analyzer* module in order to evaluate the correlation between the application response time and the input workload. Through the correlation coefficient the *Performance Anomaly Analyzer* module quantifies the impact of the dissociation between the response time and the number of transactions processed, i.e., the impact of a performance anomaly. With the resulting data, the *Build Dataset* module prepares the dataset and submits it for training using different ML algorithms (e.g., decision trees, neural networks). Then, at runtime, the parameters collected by the *Sensor* module are estimated by up to *N* epochs ahead using the ARIMA and Holt-Winters time series algorithms. Those estimations are then classified by the ML algorithms previously trained to determine in advance if the application may incur in some performance anomaly.

The experimental work conducted aims to answer the following questions:

- *Which machine learning (ML) classification algorithms are the best to model the behavior of the application?*

- *What is the behavior of the estimates made using time series models for the parameters?*

- *How much confidence do we obtain from the estimation-classification technique to predict performance anomalies?*

We understand that this study is still an exploratory work on the potential of the *SHõWA* framework for the prediction of anomalies. In this context, the answer to the questions above define the basis for further work in the area of anomaly prediction through the *SHõWA* framework.

We also want to refer that, the ability of the dataset to be updated and trained as new data become available is contemplated in the *SHõWA* roadmap, but it has not been implemented yet and therefore this functionality it will not be used in this experimental study.

9.3.1 Test environment

The test environment used is similar to that presented in other chapters. It is illustrated in Figure 62. The environment consists of an application server (Glassfish) running a benchmark application (TPC-W [Smith 2001]). The user requests are simulated using several remote browser emulators (RBEs). TPC-W simulates the activities of a bookstore Web application. It defines 14 user-transactions which are classified as browsing or ordering types. These interactions interact with a MySQL database which is used as a storage backend.

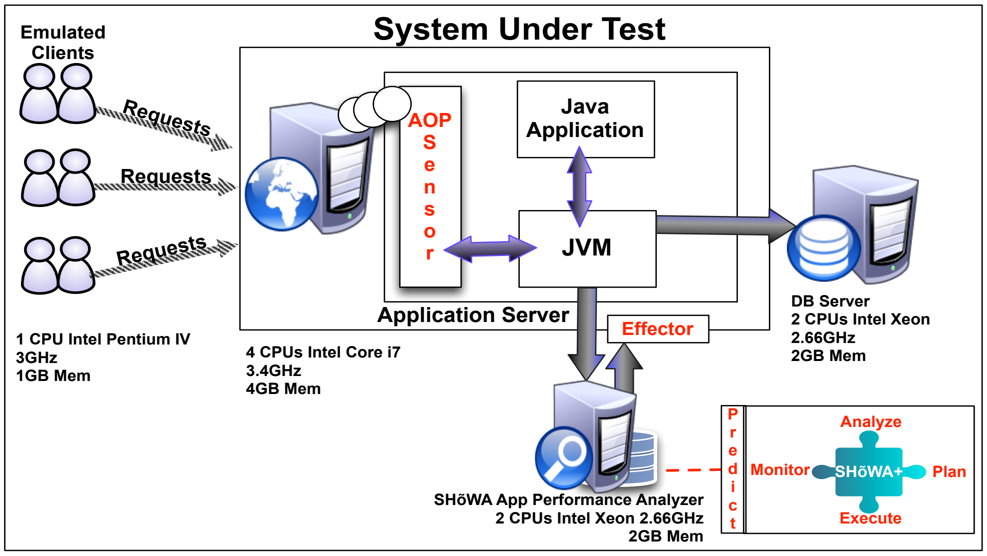

System Under Test
Emulated Clients
Requests
Requests
Requests
1 CPU Intel Pentium IV 3GHz 1GB Mem
AOP Sensor
Java Application
JVM
Application Server
4 CPUs Intel Core i7 3.4GHz 4GB Mem
Effector
DB Server 2 CPUs Intel Xeon 2.66GHz 2GB Mem
Predict
Analyze
Monitor
SHõWA+
Plan
Execute
SHõWA App Performance Analyzer 2 CPUs Intel Xeon 2.66GHz 2GB Mem

Figure 62: Test environment (*SHõWA+*)

The *Sensor* module intercepts each one of the 14 TPC-W user-transactions, gathers several system and application parameters and aggregates data into intervals of 5 seconds. The dataset preparation, the training of the ML classification algorithms and the prediction of performance anomalies is performed in the "*SHõWA* Application Performance Analyzer" server.

To prepare the initial dataset we used data from previous runs. The data was obtained using a test environment equal to that shown in Figure 62. To access the ability of the *SHõWA+* framework to predict anomalies we have executed new test-runs and considered different types of anomaly.

The test environment is composed by several machines. It includes four workload generator nodes with 3GHz Intel Pentium CPU and 1GB of RAM each. The "*SHõWA* Application Performance Analyzer" machine includes two 2.66GHz Intel Xeon CPUs and 2GB of RAM. The application server machine includes four 3.4GHz Intel Core i7 CPUs and 4GB of RAM and the database server machine includes two 2.66GHz Intel

Xeon CPUs and 2GB of RAM.

All the nodes run Linux with 2.6 kernel version and are interconnect through a 100Mbps Ethernet LAN.

9.3.2 Preparing and training the datasets

For experimentation purposes we prepared two datasets using data previously collected by the *Sensor* module and analyzed by the *Performance Anomaly Analyzer* module. The datasets resulted from dynamic workload executions and are characterized by periods of normal behavior and periods containing performance anomalies motivated by different types of anomaly.

We present the dataset attributes in Table 23. The class corresponds to the data transformation presented in Table 24. This transformation is applied to the impact of the dissociation between the response time and the number of user-transactions processed.

The `datasetA` corresponds to the `TPCW_customer_registration` user-transaction. It contains 5898 instances, corresponding to approximately 8 hours of execution. 69% of the instances are classified as “Green”, 16% are classified as “Yellow” and 15% as “Red”. The `datasetB` refers to the `TPCW_home_interaction` user-transaction. It holds 6790 instances and counts with 72% instances classified as “Green”, 25% as “Yellow” and 3% as “Red”. The datasets were submitted for training in WEKA using four ML classification algorithms: NB, J48, LMT and MLP.

In Table 25 we present the accuracy, the precision, the recall, the F-Measure and the time in seconds required to train the algorithms. The accuracy corresponds to the percentage of correctly classified instances. It results from the sum of correct classifications divided by the total number of classifications.

A major problem with the accuracy is that it can be biased when the number of items in each class is very divergent. Thus, to allow a more accurate analysis it is also presented the precision and recall achieved by each class. The precision for a class is the number of items correctly labeled as belonging to the positive class (true positives) divided by the total number of elements labeled as belonging to the positive class. Recall is defined as the number of true positives divided by the total number of elements that actually belong to the positive class (the sum of true positives and false negatives). In simple terms, high recall means that an algorithm returned most of the relevant results and high precision means that an algorithm returned more relevant results than irrelevant.

The F-measure is a measure that considers the precision and the recall of the test to compute the weighted average of the precision and recall. The F-measure score reaches its best value at 1 and worst score at 0.

The best algorithm is the one that cumulatively:

1. Has a higher F-Measure;
2. Has a higher accuracy;
3. Requires a lower time to build the model.

To train the dataset we used the percentage split test option. With this option the dataset is randomly reordered and it is then split into training and test sets with a specific proportion of the examples. We used a percentage split of 66%, i.e., two-thirds of the data in the dataset were used for the training process. The remaining data was used to evaluate the prediction results.

Table 25: Results achieved during the datasets training phase

Dataset			Naive-Bayes	J48	LMT	MLP
A	Precision	Green	0.86	0.97	0.96	0.95
		Yellow	0.27	0.92	0.90	0.78
		Red	0.96	0.92	0.99	0.98
	Recall	Green	0.57	0.98	0.97	0.94
		Yellow	0.62	0.90	0.85	0.82
		Red	0.95	0.99	0.99	0.99
	Accuracy		0.67	0.96	0.95	0.93
	F-Measure		0.70	0.95	0.94	0.91
	Time (sec)		0.10	0.38	29.12	34.10
B	Precision	Green	0.99	1.00	0.99	0.99
		Yellow	0.99	1.00	0.99	0.99
		Red	0.82	0.97	0.97	0.95
	Recall	Green	0.36	0.94	0.90	0.96
		Yellow	0.91	1.00	0.99	0.99
		Red	0.74	0.86	0.91	0.81
	Accuracy		0.91	0.99	0.98	0.98
	F-Measure		0.78	0.96	0.96	0.95
	Time (sec)		0.11	0.20	29.27	33.42

From the results presented in Table 25 we observe that J48, LMT and MLP have a high F-Measure and accuracy value. The precision and recall per class is also very high, which means that the algorithms have classified the data according to the appropriate class. The time required for the dataset training has varied between 0.38 seconds (J48) and 34 seconds (MLP). Since the difference between the three algorithms is small, we considered all of them for the estimation-classification analysis presented in the next section.

9.3.3 Estimation-and-classification of performance anomalies

To evaluate the results obtained with the *SHõWA+*, we simulated the presence of a memory leak and a CPU contention. We executed dynamic workloads: the workload lasts for 8400 seconds; the workload mix varies in a range between 5 and 15 minutes; the number of emulated-users varies between 30 and 400. The data is aggregated in time intervals of 5 seconds.

To simulate a memory leak we used JAFL [Rodrigues 2008]. JAFL is implemented in Java and allows the simulation of some of the most common types of failures in Web applications. The memory consumption module was used to consume a random amount of JVM memory (between 64KB and 128KB).

The CPU contention scenario was simulated using some O.S. processes that gradually consume CPU cycles reducing the percentage of `CPU_idle` and increasing the number of processes in the `run-queue`. The `stress` [Stress] tool was used for this purpose. It allows to spawn multiple processes that spine on the ***sqrt()*** function and generate load on CPU.

For each experiment we:

1. Use ARIMA and Holt-Winters models to estimate the values for each of the parameters collected by the *Sensor* module;

2. Conduct a classification through the previous trained ML algorithms;

3. Compare the results provided by the *Performance Anomaly Analyzer* module with the predictions achieved by the estimation-classification approach.

As the *Sensor* module collects data from the system, the parameters are estimated up to 24 periods ahead, i.e., 2 minutes ahead. Currently each parameter is estimated independently, but in future work we will do a joint estimation of the parameters to better reflect the dependency between them. Once the estimates are obtained, the *Classify* module is invoked. This module predicts a class based-on the data. The class gives an indication of a potential performance anomaly. Depending on the result, the *SHõWA* framework can be used to pinpoint the potential performance anomaly and trigger recovery actions to prevent the occurrence of failures.

We present below the results for the estimation of the parameters and classification of the estimated values, for both of the anomaly scenarios used in this study. The parameters estimation is presented graphically and complemented with the mean absolute error (MAE) analysis. The MAE value corresponds to the average of the absolute difference between the true values and the estimated values. In our case, it corresponds to the average of the MAE values obtained by the successive estimates. A MAE value of 2 means that for each of the 24 periods estimated there are 2 estimates in average that were not estimated correctly.

To give a more concrete idea on how the estimated data was classified by the ML algorithms, we present the precision and recall results per class. We also present the F-Measure and the percentage of correctly predicted classes (accuracy).

9.3.3.1 Memory leak analysis

With the memory leak analysis we aim to verify if the estimation approach followed by machine learning classification, is able to predict accurately the performance anomaly, i.e., before that the availability or performance of the application becomes compromised by the lack of JVM memory.

In Figure 63 we illustrate the estimations achieved through the ARIMA and Holt-Winters time series estimators. The *SHõWA+* framework makes the estimation of all parameters collected by the *Sensor* module, including the number of user-transactions up to 24 periods ahead. For presentation purposes we only present the estimates for the `availableMEM` parameter.

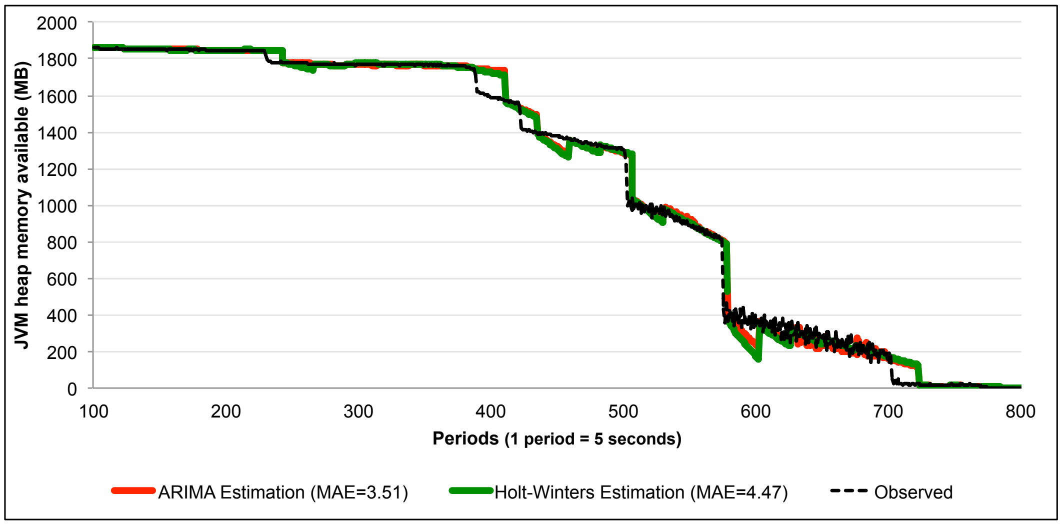


Figure 63: JVM Available Memory - Estimated versus Observed

From Figure 63, one can see that both estimates were able to adapt to the changes observed. In addition to the visual representation, we also determined the mean absolute error (MAE) of each of the estimation technique. According to the results the ARIMA had a MAE of 3.51 and Holt-Winters a MAE of 4.47. With a small MAE difference, ARIMA was able to provide close forecasts to the outcomes.

After the parameter estimation, the analysis continues with the classification of estimates. It is through the classification that we aim to predict in advance the occurrence of performance anomalies.

In Table 26, we present the results obtained by the classification algorithms, for the 24 periods ahead. The results include the precision, recall, accuracy and F-measure for each the machine learning algorithm used (J48, LMT and MLP) and for the estimates achieved by the ARIMA and Holt-Winters estimators. The best prediction is one that has a higher F-Measure value.

Table 26: Estimation+Classification - Memory leak - 24 periods ahead

		ARIMA			Holt-Winters		
		J48	LMT	MLP	J48	LMT	MLP
Precision	Green	0.98	0.90	1.00	0.97	0.81	0.82
	Yellow	0.31	1.00	1.00	0.30	0.78	0.52
	Red	1.00	1.00	0.99	1.00	0.94	0.46
Recall	Green	1.00	1.00	0.99	1.00	0.99	0.94
	Yellow	0.73	0.86	0.40	0.67	0.40	0.28
	Red	0.52	1.00	0.74	0.51	0.74	0.48
Accuracy		0.90	0.98	0.97	0.89	0.82	0.75
F-Measure		0.76	0.97	0.94	0.74	0.77	0.58

The results presented in Table 26 show that the estimates provided by ARIMA and the subsequent classification made by LMT algorithm was the combination that achieved the best results: a F-Measure of 97% and a good precision and recall. This result indicate that the estimation-classification technique included in the *SHõWA+* framework was able to predict the anomaly in advance. By predicting the anomalies in advance, it is possible to activate the recovery mechanisms earlier, preventing the degradation of service and the impact on the end-users.

9.3.3.2 CPU contention analysis

In this experimentation we evaluate if *SHõWA+* is able to predict the occurrence of performance anomalies driven by a CPU consumption.

The training dataset includes situations where the application performance was affected by CPU issues. This dataset is used to train different machine learning algorithms. These algorithms are then used to classify estimates, up to 24 periods ahead, anticipating the occurrence of anomalies at runtime. The estimates are made using the time series models (ARIMA and Holt-Winters) using the data collected by the *Sensor* module.

In Figure 64 and Figure 65 we illustrate the estimations provided by the ARIMA and Holt-Winters models for the `CPU idle` and `Run Queue` parameters.

From the figures we see that the estimates follow the changes observed in the parameters. The proximity between the prediction and the observed value is also verified

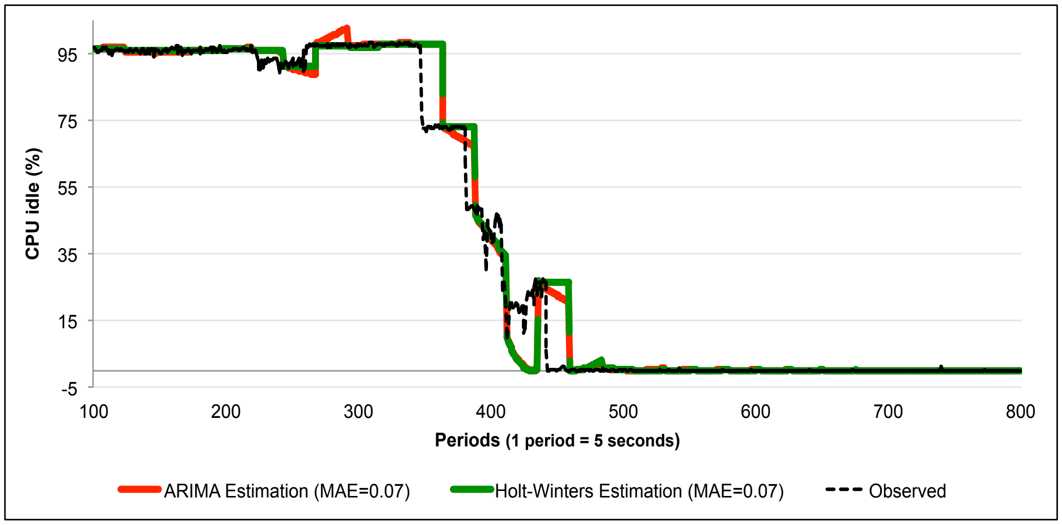


Figure 64: `CPU idle` - Estimated versus Observed

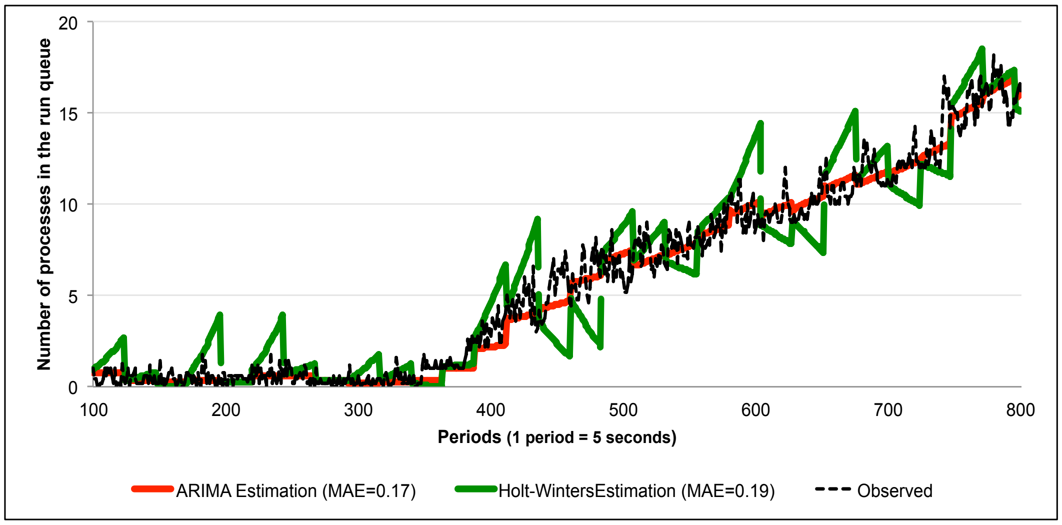


Figure 65: `Run Queue` - Estimated versus Observed

by the low value of MAE obtained:

- Estimate of `CPU idle`: ARIMA and Hold-Winters achieved a MAE of 0.07;
- Estimate of number of processes in the `Run Queue`: ARIMA achieved a MAE of 0.17 and Hold-Winters a MAE of 0.19.

The fact of the MAE values are close and low, indicates that both estimates obtained a good hit rate. This is undoubtedly an important step for the analysis that follows: classify the data, to predict in advance the occurrence of performance anomalies.

In Table 27, we present the classification results for the estimations achieved with the ARIMA and Holt-Winters estimators. The precision, the recall, the F-Measure and the percentage of correctly classified classes is used to compare the results achieved by the different classification algorithms. Like in the previous analysis, since the F-measure considers both the precision and the recall of the test, we can say that the best classification algorithm is the one that presents a value of F-measure higher.

Table 27: Estimation+Classification - CPU contention - 24 periods ahead

		ARIMA			Holt-Winters		
		J48	LMT	MLP	J48	LMT	MLP
Precision	Green	1.00	1.00	1.00	1.00	0.97	1.00
	Yellow	0.44	0.70	0.57	0.45	0.84	0.66
	Red	0.54	0.38	0.54	0.55	0.58	0.56
Recall	Green	0.60	0.70	0.83	0.60	0.86	0.85
	Yellow	1.00	0.61	0.45	1.00	0.64	0.52
	Red	1.00	1.00	0.93	1.00	0.94	0.93
Accuracy		0.72	0.72	0.74	0.72	0.81	0.77
F-Measure		0.75	0.73	0.72	0.75	0.81	0.75

As shown in Table 27, the estimate results provided by Holt-Winters and the subsequent classification provided by LMT algorithm was the combination that yielded the better prediction results - F-Measure of 81%. This result is significantly inferior to the accuracy achieved in the previous scenario, when the approach has predicted performance anomalies motivated by the memory consumption scenario with a F-Measure of 97%. This difference arises from the low precision achieved by the algorithms while predicting the "Yellow" and "Red" classes. This is justified, in large part, because the CPU consumption was more abrupt than the memory consumption.

It should be noted that these results are based-on a relatively small dataset. As the training dataset include more data and anomaly scenarios, the better will be the forecasting accuracy.

9.4 Conclusions

In this chapter we presented an approach for the prediction of performance anomalies in Web-based applications.

The *SHõWA* framework has been extended with four new modules. Two of these modules are responsible for preparing a dataset and train the classifier using machine learning algorithms. The other modules are used to estimate up to ***N*** time periods ahead the parameter values and to predict, in advance, the occurrence of performance anomalies considering the parameters estimations.

To check the operation and the results obtained with the training, estimation and classification approach we have conducted an experimental analysis. The experimental analysis aims to verify the ability of the *SHõWA+* framework to predict anomalies in Web-based applications. Initially the experimental work focused on the preparation and training, by means of machine learning algorithms, of two datasets containing data previously collected and analyzed by the *SHõWA* framework. In a second step we simulated the occurrence of performance anomalies to verify how the *SHõWA+* framework is able to predict the occurrences of such anomalies. This second stage occurred at runtime. As the data was collected by the *Sensor* module, the system and application level parameters were estimated by means of time series models (ARIMA and Holt-Winters) up to 24 time periods ahead. The estimates were then classified (using the algorithms previously trained) in order to verify if within 24 periods ahead, there will occur some performance anomaly that may impact the normal functioning of the application.

The results allow us to take two types of conclusion. A first conclusion, and more related with results obtained, reveals that:

- The parameter estimates made by ARIMA and Holt-Winters and the observed values were close (low mean absolute error (MAE));
- Even with a small training dataset, the combination between the estimation of parameters and classification has proved to be useful to anticipate the occurrence of anomalies:
  - The memory consumption was predicted with 97% of certainty of its occurrence;
  - The CPU consumption was predicted with 81% of certainty of its occurrence.

More generally and about the approach used, we conclude that:

- There is no single ML classification algorithm that can be used to predict the user-transactions response time;
- A combination between different time series models should be considered when doing parameters estimation, to cope with the changes that may occur in both time and load;
- It is essential to continually train the classification algorithms in order to know which one best fits the recent state of the system.

We are aware that the results obtained so far are preliminary, but nevertheless they are important to understand and explore the potential of our approach. Given the advantages that comes from predicting anomalies in Web-based applications, this is definitely an area that we will develop in future work.

Chapter 10
Conclusion

This final chapter summarizes the self-healing framework developed in my research to detect, pinpoint and recover from anomalies in Web-based applications and describes the main contributions obtained through the experimental studies performed. This chapter also points toward future avenues of research.

10.1 Overview

In this Thesis we presented a self-healing system for Web applications. Self-healing is one of the four attributes included in the Autonomic Computing paradigm [Ganek 2003]. It is considered a fundamental aspect to address the complexity of detecting, diagnosing and recover from anomalies in computational systems.

The motivation for the development of Autonomic Computing systems is explained by the increasing complexity of computing systems. One type of computing system, that is complex to manage, and in which one anomalous behavior has serious consequences are Web-based applications. These applications are made up from multiple components, developed by different parties, and they are increasingly dependent of external services. Failure scenarios or performance anomalies undermine the normal functioning of these systems, causing financial losses and affecting the corporate reputation. Given the complexity of such systems and the impacts due to failures, it is unquestionable the necessity to devise mechanisms able to prevent scenarios of downtime or performance slowdown.

The self-healing framework (*SHõWA*), presented in this Thesis, is targeted for the detection and recovery of performance anomalies in Web-based applications. It can be seen as a fault tolerance system, intended to recover from anomalies that fit the **fail-stutter** failure model [Arpaci-Dusseau 2001]. It takes into account performance-faulty scenarios, i.e., situations in which a component provides unexpectedly low performance, but continues to function correctly with regard to its output.

The *SHõWA* framework is composed by several building blocks that cooperate each other to achieve the self-healing property. The cooperation between the modules reflects the MAPE control loop performed by an autonomic element: Monitor, Analyze, Plan and Execute. Monitoring is done at the application-level through a small program implemented according to the Aspect-Oriented Programming (AOP) paradigm. It does not require changes in the application source code and collects data on the application and execution environment (system parameters and application server) at runtime. The performance impact induced by the monitoring system, is reduced by means of adaptive and selective algorithms. These algorithms regulate, in an automatic and dynamic manner, the frequency of monitoring and the list of application calls to intercept. The data analysis is made by means of statistical correlation. It detects response time variations and identifies if the variations are due to workload changes or due to a performance anomaly. When it detects a performance anomaly, it verifies if there is any system or application server parameter change that is associated with the slowdown. It also examines the response time of the application calls, allowing to pinpoint which parts of the application are associated to a delay in response time. As soon as a performance failure or workload contention is detected, *SHõWA* selects automatically a recovery plan and executes it, in order to mitigate the impacts of the anomaly.

In addition to the above features, in this Thesis we also present an extension of the *SHõWA* framework that is intended to predict in advance the occurrence of anomalies. This extension combines two modes of operation: background and runtime. In the background, it models the historical knowledge and trains different machine learning classification algorithms. At runtime, it estimates the state of parameters up to $\boldsymbol{N}$ time intervals ahead and classifies this estimates (using the classification algorithms trained in background) about possible performance anomalies.

10.1.1 Experimentation and main results

We implemented each of the modules included in the *SHõWA* framework and conducted several experimental studies. The studies are divided into four areas: Monitoring; Detection and localization of anomalies; Recover from performance anomalies; Prediction of performance anomalies.

10.1.1.1 Monitoring

The experiments on monitoring are divided into two phases. In a first phase we compared different monitoring systems, commonly used to monitor Web applications: system-level monitoring; end-to-end monitoring; log analyzer; application-level monitoring. The application-level monitoring was done using *SHõWA*. This study aimed to evaluate:

- The coverage provided by the different monitoring systems;
- The detection latency provided by different monitoring systems;
- The effectiveness, i.e., determine which monitoring system minimizes the impact of the anomaly on the end-users.

To carry out this study we prepared a test environment similar to that found in production systems, and we injected 11 types of anomalies. The anomalies were chosen taking into account the common failures in Web applications, reported in the literature. The results provided by the *SHõWA* monitoring were very encouraging:

- **Coverage:** *SHõWA* was the only monitoring system to detect 100% of the anomalies injected;
- **Detection latency:** *SHõWA* was the first technique to detect 9 of the 11 anomalies injected;
- **Effectiveness:** The early detection achieved by *SHõWA* proved to be very important to reduce the impact of the anomalies on the end-user. It detected anomalies when the number of users affected by a delay in response time over 500 milliseconds was still low.

In the second phase of the experimental study about monitoring Web-based applications, we analyzed the performance overhead induced by application-level monitoring. Through experimental analysis we checked:

- The performance impact induced by different application-level monitoring tools;
- The source code coverage (profiling level) provided by different application-level monitoring tools;
- The performance penalty induced by the *SHõWA Sensor* module without using adaptive and selective monitoring;
- The performance penalty induced by the *SHõWA Sensor* module while using adaptive and selective monitoring.

For this analysis we considered the use of different application-level monitoring tools, and the use of adaptive and selective monitoring algorithms included in the *SHõWA* monitoring modules. The application-level monitoring tools used were chosen based-on their availability and status of the project. The system was submitted to dynamic workloads, varying both the number and the mix of user-transactions. The analysis takes into consideration the performance impact induced with application-level monitoring, according to the load of the system. From the results we observed that:

- **Performance impact induced by different application-level monitoring tools:** The performance impact induced by the application-level monitoring tools used is very low. According to the results, the response time is affected only in 2.7 milliseconds per request, and the server throughput is practically not affected (it has processed just less 0.2% requests per second);
- **Profiling level:** The application-level monitoring tools used in the analysis, can only cover 3.7% of the methods invoked by the application workload. Since the *SHõWA Sensor* module includes a generic pointcut (`call(*.*(..))`), it can intercept 100% of the methods executed;

- **Performance impact induced by** ***SHõWA*****:** Using the adaptive and selective algorithms for monitoring, the performance overhead induced by the *SHõWA Sensor* module is low:
  - The server throughput was affected in less than 1%;
  - When the server load is below 75% of its maximum capacity, the response time delay observed was just 0.5 milliseconds per request;
  - When the server load is above 75% of its maximum capacity, the response time delay observed was just 2 milliseconds per request.

It is noteworthy that the coverage results achieved by the *SHõWA* framework, already contemplates the use of adaptive and selective monitoring mechanisms. According to the results, we can say the adaptability does not affect the ability to detect and pinpoint anomalies.

10.1.1.2 Detection and localization of anomalies

The data analysis performed by the *SHõWA Workload Variation Analysis*, *Performance Anomaly Analysis*, *Anomaly Detector* and *Root-cause Failure Analysis* modules, takes a central part in our work. It is responsible for detecting and pinpointing performance anomalies in Web-based applications, independently from the variations verified in the workload. In this Thesis, in addition to describing the data analysis approach, we included a chapter evaluating the detection and pinpointing of anomalies achieved with *SHõWA*. The evaluation had the following objectives:

- Evaluate the ability of *SHõWA* to detect performance anomalies independently from the variations verified in the workload;
- Evaluate the ability of *SHõWA* to distinguish between performance anomalies and workload contention scenarios;
- Evaluate the ability of *SHõWA* to pinpoint performance anomalies at different levels of detail;
- Evaluate how timely and accurate is the detection and pinpointing of anomalies provided by *SHõWA*.

For the evaluation, we considered two benchmark applications (TPC-W and RUBiS), running in an environment similar to that found in production environment. We injected different types of anomalies, that are common in Web applications. The benchmark applications were submitted to dynamic workloads, in which the number and mix of user-transactions varies periodically. From the results we observed that:

- **Distinguish performance anomalies independently from workload variations/Accurary:** When there is a very dynamic workload but there are no anomalies injected, the tool was able to detect this situation and did not raise any false alarms;

- **Detecting performance anomalies/Accurary:** *SHõWA* was able to detect all the anomalies we injected and to distinguish them from the workload variations;

- **Pinpointing performance anomalies:** *SHõWA* was able to spot if a performance anomaly is motivated by the server resources contention or an application server contention;

- **Pinpointing performance anomalies:** *SHõWA* was able to spot application changes, highlighting the application calls associated with a response time slowdown;

- **Timely detection/Accuracy:** *SHõWA* has detected and pinpointed the anomalies, while the number of end-users affected is low: we found only 0.3% of user-requests, on average, affected by a delay in response time greater than 500 milliseconds. This allows to say that, for the set of anomalies injected, that with *SHõWA*, the time required to detect anomalies is low.

10.1.1.3 Recover from performance anomalies and workload contention scenarios

The analysis about the recovery provided by *SHõWA* was divided in two phases. In the first phase, we analyzed the behavior of the recovery modules, considering a high availability production environment, composed by a high-availability cluster and load balancing services. The experimental study in this phase had as main objectives:

- Evaluate the *SHõWA* framework ability to recover, autonomously, from different types of anomalies;

- Evaluate the time-to-repair achieved by the *SHõWA* framework;

- Analyze the advantages obtained by performing controlled recovery actions, even though there are high-availability mechanisms implemented.

In the first phase we injected different types of anomalies and executed dynamic workloads. The *Recovery Planner* recovery module includes different recovery procedures, which were selected and executed automatically upon the detection of anomalies. From the results we observed that:

- **Ability to recover from different anomalies:** *SHõWA* was able to autonomously detect and heal the application, under different anomaly scenarios, avoiding the negative impacts on the latency and throughput and without causing any visible error or loss of the work-in-progress;
- **Time-to-repair:** When compared to a system-level monitoring tool, *SHõWA* was able to recover from all the anomaly scenarios tested with a lower recovery time;
- **Controlled recovery actions:** Even considering a cluster-configuration provided with high-availability services, the number of errors observed when a cold-recovery strategy is adopted, has turned clear the importance of executing controlled recovery actions, like the ones included in the *SHõWA Recovery Planner* module, to avoid loosing the work-in-progress.

In the second phase, we considered a cloud-hosted environment, in which the computing resources can be adapted dynamically. In this phase the *SHõWA* framework is able to interact with the cloud service provider, reporting performance anomalies and workload contention scenarios and initiate the necessary resource provisioning to cope with these scenarios. Using a cloud-hosted Web-based application environment, we conducted an experimental study to:

- Check how *SHõWA* can contribute to expose the state of the application, from the customer perspective, and to accelerate the provisioning of system resources, together with the cloud service provider;
- Analyze the behavior of *SHõWA* for the dynamic provisioning of resources in a cloud platform, distinguishing workload changes from performance anomalies;
- Verify the role of *SHõWA* to ensure the quality of service of cloud-hosted Web-based applications upon the occurrence of performance degradation scenarios.
- Analyze the cloud resources provisioning and cost management from the cloud consumer perspective.

As in previous studies, we injected anomalies at the level of the resource and workload contention, and measured the ability of *SHõWA* to detect and recover from these failures. As metric of analysis, we considered the percentage of user requests affected by a delay of more than 500 milliseconds. According to the results, we found that:

- **Expose the state of the application to the cloud service providers:** With *SHõWA* it becomes possible to combine the SLAs of both cloud consumer and cloud provider;

- **Dynamic provisioning of resources, distinguishing workload changes from performance anomalies:** *SHõWA* was able to autonomously detect workload and resource contention scenarios from the consumer perspective, and it has interacted autonomously with the cloud service provider to self-adapt the infrastructure;

- **Ensure the quality of service of cloud-hosted Web-based applications:** From the tests we performed, the cloud provisioning was done while the number of end-users affected by the anomaly was still low. According to the results, between the injection of the anomaly and the provisioning of computing resources:

  - the percentage of requests affected by a response time slowdown of 500 milliseconds was inferior to 0.5%;
  - the server throughput penalty observed was inferior to 1%.

- **Resources provisioning and cost management:** With *SHõWA* the cloud consumer has more control over the infrastructure that supports its business. So, the costs of over-provisioning and the revenue and corporate reputation losses due the lack of computational resources, is more safeguarded.

10.1.1.4 Prediction of performance anomalies

The prediction of anomalies is a very interesting topic, but it carries several challenges. In this Thesis we proposed the combination of parameter estimation, up to $\boldsymbol{N}$ time intervals ahead (time series analysis) and the classification of these estimates (machine learning) to predict the occurrence of performance anomalies. The experimental work done so far, focused on the following points:

- Determine which time series models allow to estimate the parameters with the lowest error;

- Know which machine learning (ML) classification algorithms are the best to model the historical behavior of the application;

- Evaluate the accuracy achieved with the estimation-classification technique we propose.

For analysis purposes, we considered two time series models (ARIMA and Holt-Winters) and four machine learning classification algorithms (Naive Bayes, J48 decision tree, LMT decision tree and MLP neural network). The historical data was trained through the machine learning algorithms. The data collected at runtime was estimated using the time series models. The estimates were classified using the machine learning algorithms already trained. In each step of the analysis we measured the accuracy obtained with the approach. The results showed that:

- **Estimation errors:** The mean absolute error between the parameter estimates made by ARIMA and Holt-Winters and the observed values was low, which means the estimations and the observed values were close;
- **Machine learning classification algorithms:** There is no single machine learning classification algorithm that can be used to predict the user-transactions response time. In some cases some were better than others. The classification algorithms should be continuously trained in order to know which one best fits the recent state of the system;
- **Prediction accuracy:** Even with a small training dataset, the combination between the estimation of parameters and classification has proved to be useful to anticipate the occurrence of anomalies:
  - A scenario of performance anomaly, motivated by a memory consumption, was predicted in advance with 97% of certainty of its occurrence;
  - A scenario of performance anomaly, motivated by a CPU consumption, was predicted in advance with 81% of certainty of its occurrence.

10.2 Main contributions

The main contribution of this work is ability provided by the *SHõWA* framework. It allows to detect, identify and recover from anomalies that may occur at runtime and affect the performance of production Web applications. These tasks are performed autonomously, and allow mitigating the impact that these anomalies have on the revenue and corporate reputation.

This work includes several areas: monitoring of Web-based applications; detection and localization of performance anomalies; recovery from performance anomalies; prediction of performance anomalies. Along with the design and implementation of the *SHõWA* framework, we proposed approaches in each of the areas. These approaches are, to the best of our knowledge, not used before in the context of self-healing for Web-based applications. The approaches and results obtained through the experimental analysis, yielded some specific contributions.

10.2.1 Monitoring

The main contributions, with regard to the monitoring of Web-based applications are:

- The ability to perform application-level monitoring, at runtime and without require changes in the application source code, or previous knowledge about its implementation details;

- The availability of algorithms that adapt the monitoring at runtime, according to the state of the system;
- The ability to measure the execution time of all invocations of the application, with low performance impact;
- The possibility of using the monitoring module already implemented, to monitor different Web applications hosted across different J2EE application servers.

10.2.2 Detecting and localization of anomalies

The main contributions, with regard to the data analysis approach used to detect and pinpoint performance anomalies in Web-based applications, are:

- With the analysis provided by the Spearman's rank correlation, we just need to set a single threshold value to detect performance anomalies in all user-transactions: agnostic in respect to Web-based applications. This way we avoid setting specific thresholds, that usually vary from application to application and between running environments;
- The data analysis allows to detect performance anomalies independently from the workload;
- The detection of anomalies is made from the point of view of application, but focuses on the end-users experience:
  - The data analysis correlates changes in response time of the application, with changes in the state of system or application server resources. Thus it is possible to determine the cause behind the performance anomaly in more detail;
  - The data analysis correlates changes in response time of the application, with changes in the processing time of application calls involved in the processing of the transactions. Thus it is possible to identify which parts of the application are related to the delays observed in the response time and verify if such is related to any recent application upgrade or if it is due to a change in the behavior of a remote service (e.g., Web Service, database interaction);
  - It allows to detect anomalies in external components without requiring the installation of any monitoring agent on those components (black-box analysis). This is particularly important for systems composed by multiple components and managed by different parties (e.g., Web-based applications, SOA-based applications).

- According to the coverage and detection latency results, obtained through experimental analysis, the data analysis approach included in *SHõWA* is able to detect more anomalies, when compared to other monitoring systems, and it is also able to detect the anomalies more quickly. These results are very important to reduce the mean-time-to-repair, thereby increasing the dependability of these systems.

10.2.3 Recovering from anomalies

The recovery functionality is directly influenced by the ability to detect and pinpoint anomalies. Timely detection allows a faster recovery, reducing the number of users affected by the anomaly. The pinpointing of anomalies facilitates the selection of the recovery procedure, maximizing the efficiency of the recovery process.

The results obtained through the experimental analysis show that *SHõWA* is able to detect different types of anomalies and pinpoint the parameters responsible for the anomaly. The detection latency achieved by the *SHõWA* framework was also significantly lower when compared to other tools. By the set of detection, localization and recovery results, we highlight the following contributions:

- *SHõWA* provides a measurement-based recovery mechanism. As opposed to time-based recovery, with our approach the recovery is only done when there are anomalies in the system, thus maximizing the service availability and performance;

- The recovery procedures are selected autonomously, based-on high level policies defined by an operator;

- The recovery process allows the definition of hierarchies, allowing a new procedure to start if the previous does not get the desired success;

- The recovery is done automatically, facilitating the communication between the parties involved. For example, in the context of a cloud-hosted environment, the process of resource provisioning can be triggered by *SHõWA*, avoiding the involvement of human operators in the process;

- The recovery process was designed to collect information about its own operations. This helps to make the process more intelligent, allowing it to choose automatically the recovery plans that had more success in the past (e.g., low recovery time, less impact on system performance).

10.2.4 Predicting anomalies

In the context of dependable systems, the ability to predict in advance the occurrence of anomalies is of utmost importance. We propose an approach for predicting performance anomalies in Web-based application, that allows to:

- Model the behavior of the application, using historical data;
- Estimate the future state of the application and the system resources usage;
- Check out what these estimates predicts in terms of normal system operation.

The methodology and the results require further study. However, based-on the prediction accuracy, the estimation-classification approach reveals potential to predict in advance the occurrence of anomalies, avoiding the impact caused by them on the revenue and corporate reputation.

10.3 Future work

Much work can still be done in the area of self-healing for Web-based applications. Throughout the execution of this work we were faced with various challenges and ideas. The time spent and depth of this work has forced us to focus on some of these. In this section we briefly summarize some of the open issues that we will address in future work.

10.3.1 Monitoring

Despite the low performance impact achieved by means of adaptive and selective monitoring, there is still room for improving in this area. The adoption of in-memory databases and new adaptive and selective algorithms can be considered to reduce, even more, the performance impact induced with the application-level profiling.

10.3.2 Detection and localization of anomalies

Throughout the study we evaluated the ability of the *SHõWA* framework to detect and pinpoint one anomaly at a time. Since the simultaneous occurrence of anomalies can exist, it becomes important to verify if *SHõWA* is able to detect the occurrence of simultaneous anomalies and to establish a kind of hierarchy among them.

10.3.3 Recovering from anomalies

In our experimental analysis we adopted a procedure-based recovery mechanism to restore the service when an anomaly is detected.

The approach has proved to be useful, both to mitigate anomalies in traditional Web environments, either to self-adapt the infrastructure when the application is hosted on a cloud platform. However, we consider that the approach can be improved, becoming more intelligent and autonomous. Given the existence of problem management databases and a signature extracted from the anomaly, it becomes possible to determine, automatically, the set of recovery actions that should be applied to recover the system. With the analysis and reemployment of a past solution, the incident management process becomes more autonomous, reducing the human dependency in the recovery process. These techniques can be further improved with the creation of algorithms able to self evaluate the result of implementation of recovery, improving future decisions.

10.3.4 Predicting anomalies

The approach used to predict performance anomalies requires some additional study. Improved data selection mechanisms, parameter estimation techniques and classification algorithms are required in favor of more assertive predictions. This is definitely one of the topics that we will address in the short term.

10.3.5 Domains of applicability

In this work we used the *SHõWA* framework to detect and recover from performance anomalies in Web-based applications. Once *SHõWA* allows monitoring from the application point of view, there are other type of applications that can take advantage of the tool. Given the increasing adoption of SOA-based applications, we intend to use *SHõWA* to detect and recover from anomalies in service composition and service orchestration based applications.

Also, given the increasing adoption of cloud platforms, we intend to extend the functionality of the *SHõWA* framework with respect to the scaling up/down of the infrastructure, considering always the cloud adaptation from the consumer perspective.

Another aspect we wish to proceed, relates to the use of *SHõWA* in a real production environment.

10.4 Future of Autonomic Computing and self-healing: our vision

We could not finish this Thesis without saying that the reasons which lead to the Autonomic Computing concept, presented more than a decade ago, are still very relevant.

The management and operation of computing systems remains complex. The computing systems integrate numerous components developed by various entities (e.g., SOA-based applications) and run in increasingly heterogeneous and dynamic environments (e.g., public cloud platforms). The number of users, as well as, the requirements on the features and operation of these systems continues to grow. This complexity imposes severe difficulties to the configuration, maintenance, detection and recovery from anomalies or security events, and aggravates the risk of occurring unexpected problems. As in a vicious cycle, the occurrence of problems, causes serious damages for the companies or organizations that use these systems. So, regardless of its name, the development of systems able to self-configure, self-protect, self-heal and self-optimize, continues to be fundamental to cope with the complexity and mitigate the effects of the problems that might occur.

For the reasons presented, and taking into account the market trends presented by reference entities, such as Gartner, Forrester and IDC, the areas of self-healing, fault tolerance, resilience and security of the computing systems, will remain relevant for a long time. The market will require the scientific, academic and industry communities, that continue to work to find the best solutions to cope with unavailability, performance and security issued that may arise. As much as possible, we expect to continue contributing actively to achieving such solutions.

Bibliography

[Aberdeen Group] Aberdeen Group. *Web Performance Impacts.* http://www.webperformancetoday.com/2010/06/15/, 2010. (Cited on pages 2 and 29.)

[Aguilera 2003] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds and Athicha Muthitacharoen. *Performance debugging for distributed systems of black boxes.* In Proceedings of the nineteenth ACM symposium on Operating Systems Principles, pages 74–89, New York, NY, USA, 2003. ACM. (Cited on pages 68 and 88.)

[Ali-Eldin 2012] Ahmed Ali-Eldin, Maria Kihl, Johan Tordsson and Erik Elmroth. *Efficient provisioning of bursty scientific workloads on the cloud using adaptive elasticity control.* In Proceedings of the 3rd workshop on Scientific Cloud Computing Date, pages 31–40, Delft, Netherlands, June 2012. (Cited on page 163.)

[Alonso 2008] Javier Alonso, Jordi Torres, Rean Griffith, Gail E. Kaiser and Luís Moura Silva. *Towards Self-adaptable Monitoring Framework for Self-healing.* In CoreGRID Workshop on Grid Middleware, pages 1–9, 2008. (Cited on page 33.)

[Alonso 2010] Javier Alonso, Josep Ll. Berral, Ricard Gavalda and Jordi Torres. *Adaptive on-line software aging prediction based on Machine Learning.* In Proceedings of the International Conference on Dependable Systems and Networks, pages 507–516, Chicago, Illinois, USA, 2010. (Cited on pages 183 and 184.)

[Amazon CloudWatch] Amazon CloudWatch. *Monitoring for AWS cloud resources.* http://aws.amazon.com/cloudwatch/, 2013. (Cited on page 164.)

[Andrzejak 2008] Artur Andrzejak and Luis Silva. *Using machine learning for non-intrusive modeling and prediction of software aging.* In Proceedings of the Network Operations & Management Symposium, pages 7–11, 2008. (Cited on pages 183 and 184.)

[Apache JMeter] Apache JMeter. *Load test functional behavior and measure performance tool.* http://jakarta.apache.org/jmeter/, 2012. (Cited on page 70.)

[AppInternals Xpert] AppInternals Xpert. *Riverbed performance management solution.* http://www.opnet.com/solutions/application_performance/, 2011. (Cited on page 66.)

[Appleby 2001] Karen Appleby, Sameh A. Fakhouri, Liana Fong, Germán S. Goldszmidt, Michael H. Kalantar, Srirama M. Krishnakumar, Donald P. Pazel, John A. Pershing and Benny Rochwerger. *Océano - SLA Based Management of a Computing Utility.* In Proceedings of the International Symposium on Integrated Network Management, pages 855–868, 2001. (Cited on page 21.)

[Applications Manager] Applications Manager. *Application performance monitoring tool.* http://www.manageengine.com/products/applications_manager/, 2011. (Cited on page 66.)

[Armbrust 2009] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica and Matei Zaharia. *Above the Clouds: A Berkeley View of Cloud Computing.* Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009. (Cited on page 162.)

[Arnold 2001] Matthew Arnold and Barbara G. Ryder. *A framework for reducing the cost of instrumented code.* Proceedings of the Conference on Programming Language Design and Implementation / ACM SIGPLAN Notices, vol. 36, pages 168–179, May 2001. (Cited on page 90.)

[Arpaci-Dusseau 2001] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Fail-Stutter Fault Tolerance.* In Proceedings of the Eighth Workshop on Hot Topics in Operating Systems, pages 33–38, Washington, DC, USA, 2001. (Cited on pages 3, 27, 28 and 201.)

[Avizienis 2004] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell and Carl Landwehr. *Basic Concepts and Taxonomy of Dependable and Secure Computing.* IEEE Transactions on Dependable and Secure Computing, vol. 1, no. 1, pages 11–33, 2004. (Cited on page 26.)

[AWStats] AWStats. *Real-time logfile analyzer.* http://awstats.sourceforge.net/, 2009. (Cited on page 67.)

[Baresi 2004] Luciano Baresi, Carlo Ghezzi and Sam Guinea. *Towards Self-Healing Service Compositions.* In Proceedings of the First Conference on the Principles of Software Engineering, 2004. (Cited on page 34.)

[Barham 2003] Paul Barham, Rebecca Isaacs, Richard Mortier and Dushyanth Narayanan. *Magpie: Online Modelling and Performance-aware Systems.* In Proceedings of the 9th Conference on Hot Topics in Operating Systems, page 15, Berkeley, CA, USA, 2003. USENIX Association. (Cited on page 114.)

[Barham 2004] Paul Barham, Austin Donnelly, Rebecca Isaacs and Richard Mortier. *Using Magpie for Request Extraction and Workload Modelling.* In Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation, pages 18–18, Berkeley, CA, USA, 2004. USENIX Association. (Cited on page 88.)

[Bellur 2007] Umesh Bellur and Amar Agrawal. *Root Cause Isolation for Self Healing in J2EE Environments.* In Proceedings of the First International Conference on Self-Adaptive and Self-Organizing Systems, pages 324–327, Washington, DC, USA, 2007. IEEE Computer Society. (Cited on page 113.)

[Bjorkqvist 2012] Mathias Bjorkqvist, Lydia Y. Chen and Walter Binder. *Opportunistic Service Provisioning in the Cloud.* In Proceedings of the IEEE Fifth International Conference on Cloud Computing, pages 237–244, Honolulu, Hawaii, USA, 2012. (Cited on pages 161 and 163.)

[Bodíc 2005] Peter Bodíc, Greg Friedman, Lukas Biewald, Helen Levine, George Candea, Kayur Patel, Gilman Tolle, Jonathan Hui, Armando Fox, Michael I. Jordan and David A. Patterson. *Combining Visualization and Statistical Analysis to Improve Operator Confidence and Efficiency for Failure Detection and Localization.* In Proceedings of the International Conference on Autonomic Computing, pages 89–100. IEEE Computer Society, 2005. (Cited on pages 67 and 111.)

[Bohra 2004] Aniruddha Bohra, Iulian Neamtiu and Florin Sultan. *Remote Repair of Operating System State Using Backdoors.* In Proceedings of the First International Conference on Autonomic Computing, ICAC '04, pages 256–263, Washington, DC, USA, 2004. IEEE Computer Society. (Cited on page 143.)

[Bouchenak 2011] Sara Bouchenak, Fabienne Boyer, Benoit Claudel, Noel De Palma, Olivier Gruber and Sylvain Sicard. *From Autonomic to Self-Self Behaviors: The JADE Experience.* ACM Transactions on Autonomous and Adaptive Systems, vol. 6, no. 4, pages 28:1–28:22, Oct 2011. (Cited on pages 22 and 23.)

[Box 2008] George Box, Gwilym Jenkins and Gregory Reinsel. Time series analysis, forecasting and control. ISBN: 978-0-470-27284-8. Wiley-Interscience, 4th edition, 2008. (Cited on page 188.)

[Brown 2003] A. Brown. *A Recovery-oriented Approach to Dependable Services: Repairing Past Errors with System-wide Undo*. PhD thesis, Computer Science Division-University of California, Berkeley., 2003. (Cited on page 144.)

[Bruneo 2013] Dario Bruneo, Salvatore Distefano, Francesco Longo, Antonio Puliafito and Marco Scarpa. *Workload-Based Software Rejuvenation in Cloud Systems*. IEEE Transactions on Computers, vol. 62, no. 6, pages 1072–1085, 2013. (Cited on page 163.)

[Calcavecchia 2012] Nicolò Maria Calcavecchia, Ofer Biran, Erez Hadad and Yosef Moatti. *VM Placement Strategies for Cloud Scenarios*. In Proceedings of the IEEE Fifth International Conference on Cloud Computing, pages 852–859, Honolulu, Hawaii, USA, 2012. (Cited on pages 161 and 163.)

[Candea 2001] George Candea and Armando Fox. *Designing for High Availability and Measurability*. In Proceedings of the 1st Workshop on Evaluating and Architecting System Dependability, Göteborg, Sweden, July 2001. IEEE Computer Society. (Cited on page 142.)

[Candea 2002] George Candea, James Cutler, Armando Fox, Rushabh Doshi, Priyank Garg and Rakesh Gowda. *Reducing Recovery Time in a Small Recursively Restartable System*. In Proceedings of the International Conference on Dependable Systems and Networks, pages 605–614, Washington, DC, USA, 2002. (Cited on page 142.)

[Candea 2003a] G. Candea, P. Keyani, E. Kiciman, S. Zhang and A. Fox. *JAGR: An Autonomous Self-recovering Application Server*. In Proceedings of the 5th Annual International Workshop on Active Middleware Services / Autonomic Computing Workshop, pages 168–178, Seattle, WA, USA, June 2003. (Cited on pages 33 and 112.)

[Candea 2003b] George Candea, Mauricio Delgado, Michael Chen and Armando Fox. *Automatic Failure-Path Inference: A Generic Introspection Technique for Internet Applications*. In Proceedings of the third IEEE Workshop on Internet Applications, page 132, Washington, DC, USA, 2003. (Cited on page 113.)

[Candea 2004a] George Candea, James Cutler and Armando Fox. *Improving availability with recursive microreboots: a soft-state system case study*. Performance Evaluation, vol. 56, no. 1-4, pages 213–248, Mar 2004. (Cited on page 32.)

[Candea 2004b] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman and Armando Fox. *Microreboot - A technique for cheap recovery*. In Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation, pages 31–44, San Francisco, CA, USA, 2004. (Cited on pages 34 and 143.)

[Candea 2006] George Candea, Emre Kiciman, Shinichi Kawamoto and Armando Fox. *Autonomous Recovery in Componentized Internet Applications.* Cluster Computing, vol. 9, no. 2, pages 175–190, April 2006. (Cited on pages 33, 111 and 143.)

[Castelli 2001] Vittorio Castelli, Richard E. Harper, Philip Heidelberger, Steven W. Hunter, Kishor S. Trivedi, Kalyanaraman Vaidyanathan and William P. Zeggert. *Proactive Management of Software Aging.* IBM Journal of Research and Development, vol. 45, no. 2, pages 311–332, March 2001. (Cited on page 142.)

[Cecchet 2002] Emmanuel Cecchet, Julie Marguerite and Willy Zwaenepoel. *Performance and scalability of EJB applications.* In Proceedings of the 17th Conference on Object-oriented Programming, Systems, Languages, and Applications / ACM SIGPLAN Notices, volume 37, pages 246–261, New York, NY, USA, Nov 2002. (Cited on page 117.)

[Chang 2008] Hervé Chang, Leonardo Mariani and Mauro Pezzè. *Self-healing Strategies for Component Integration Faults.* In Proceedings of the Automated Software Engineering Workshops, pages 25–32, Italy, Fev 2008. (Cited on pages 33 and 143.)

[Chaudhuri 1998] Surajit Chaudhuri and Vivek R. Narasayya. *AutoAdmin 'What-if' Index Analysis Utility.* In Proceedings of the International Conference on Management of Data, pages 367–378, Seattle, Washington, USA, June 1998. (Cited on page 21.)

[Chen 2002] Mike Y. Chen, Emre Kiciman, Eugene Fratkin, Armando Fox and Eric Brewer. *Pinpoint: Problem Determination in Large, Dynamic Internet Services.* In Proceedings of the International Conference on Dependable Systems and Networks, pages 595–604, Bethesda, MD, USA, June 2002. (Cited on pages 33, 67, 88, 112 and 143.)

[Chen 2004] Mike Y. Chen, Anthony Accardi, Emre Kiciman, Jim Lloyd, Dave Patterson, Armando Fox and Eric Brewer. *Path-Based Failure and Evolution Management.* In Proceedings of the International Symposium on Networked Systems Design and Implementation, pages 309–322, San Francisco, California, March 2004. (Cited on pages 111 and 112.)

[Cherkasova 2008] Ludmila Cherkasova, Kivanc M. Ozonat, Ningfang Mi, Julie Symons and Evgenia Smirni. *Anomaly? Application Change? or Workload Change? Towards Automated Detection of Application Performance Anomaly and Change.* In Proceedings of the International Conference on Dependable Systems and Networks, pages 452–461, Anchorage, Alaska, USA, June 2008. (Cited on pages 68, 113 and 114.)

[Cohen 1988] Jacob Cohen. Statistical power analysis for the behavioral sciences. Lawrence Erlbaum, ISBN 0-8058-0283-5, 2nd edition, Jan 1988. (Cited on page 49.)

[Corsava 2003] Sophia Corsava and Vladimir Getov. *Intelligent Architecture for Automatic Resource Allocation in Computer Clusters.* In Proceedings of the 17th International Symposium on Parallel and Distributed Processing, page 201.1, Washington, DC, USA, 2003. (Cited on page 32.)

[DJProf] DJProf. *Java profiling tool based-on AspectJ.* http://homepages.mcs.vuw.ac.nz/-djp/djprof/, 2011. (Cited on page 89.)

[Dong 2003] Xiangdong Dong, Salim Hariri, Lizhi Xue, Huoping Chen, Ming Zhang, Sathija Pavuluri and Soujanya Rao. *AUTONOMIA: An Autonomic Computing Environment.* In Proceedings of the International Conference on Performance, Computing and Communications, pages 61–68, Phoenix, Arizona, USA, April 2003. (Cited on page 22.)

[Durkee 2010] Dave Durkee. *Why cloud computing will never be free.* Commun. ACM, vol. 53, no. 5, pages 62–69, May 2010. (Cited on page 162.)

[e-Valid] e-Valid. *Client-Side Testing and Performance Analysis of Web Applications Web Application Testing Systems.* http://www.soft.com/eValid/, 2010. (Cited on page 66.)

[EMMA] EMMA. *Java code coverage tool.* http://emma.sourceforge.net/, 2005. (Cited on page 100.)

[eXternalTest] eXternalTest. *Monitoring servers and services online.* http://www.externaltest.com/, 2010. (Cited on page 66.)

[Forrester Consulting] Forrester Consulting. *eCommerce Web Site Performance Today: An Updated Look At Consumer Reaction To A Poor Online Shopping Experience.* Technical report, Akamai Technologies, Inc., 2009. (Cited on pages 2 and 157.)

[Fuad 2006] M.M. Fuad, D. Deb and M.J. Oudshoorn. *Adding Self-Healing Capabilities into Legacy Object Oriented Application.* In Proceedings of the International Conference on Autonomic and Autonomous Systems, page 51, Silicon Valley, California, USA, July 2006. (Cited on page 33.)

[Ganek 2003] A. G. Ganek and T. A. Corbi. *The Dawning of the Autonomic Computing Era.* IBM Systems Journal, vol. 42, no. 1, pages 5–18, 2003. (Cited on pages 2, 16, 24, 34, 164 and 201.)

[Garg 1998] S. Garg, A. Van Moorsel, K. Vaidyanathan and K. S. Trivedi. *A Methodology for Detection and Estimation of Software Aging.* In Proceedings of the ninth International Symposium on Software Reliability Engineering, pages 283–292, Paderborn, Germany, Nov 1998. (Cited on pages 120, 183, 184 and 188.)

[Gomez] Gomez. *Web performance division of Compuware Inc.* http://www.compuware.com/application-performance-management/, 2010. (Cited on page 66.)

[Gomez 2010] Gomez. *Why Web Performance Matters: Is Your Site Driving Customers Away?* Technical report, Compuware, 2010. (Cited on page 157.)

[Gray 1986] Jim Gray. *Why Do Computers Stop and What Can Be Done About It?* In Proceedings of the Fifth Symposium on Reliability in Distributed Software and Database Systems, pages 3–12, Los Angeles, California, USA, Jan 1986. (Cited on page 26.)

[Griffith 2005] Rean Griffith and Gail Kaiser. *Manipulating managed execution runtimes to support self-healing systems.* ACM SIGSOFT Software Engineering Notes, vol. 30, no. 4, pages 1–7, May 2005. (Cited on page 32.)

[H2 Database] H2 Database. *Java SQL database.* http://www.h2database.com/, 2012. (Cited on page 96.)

[HAProxy] HAProxy. *Reliable, High Performance TCP/HTTP Load Balancer.* http://haproxy.1wt.eu/, 2012. (Cited on page 146.)

[Haydarlou 2005] A. Reza Haydarlou, Benno J. Overeinder and Frances M. T. Brazier. *A Self-Healing Approach for Object-Oriented Applications.* In Proceedings of the 16th International Workshop on Database and Expert Systems Applications, pages 191–195, Copenhagen, Denmark, Aug 2005. (Cited on page 33.)

[Haykin 1994] S. Haykin. Neural networks: A comprehensive foundation. MacMillan Publishing Company, New York, 1994. (Cited on page 188.)

[Hellerstein 1999] Joseph L. Hellerstein, Mark M. Maccabee, W. Nathaniel Mills Iii and John J. Turek. *ETE: A Customizable Approach to Measuring End-to-End Response Times and Their Components in Distributed Systems.* In Proceedings of the International Conference on Distributed Computing Systems, pages 152–162, Austin, TX, USA, May 1999. (Cited on page 66.)

[Herbst 2013] Nikolas Roman Herbst, Samuel Kounev and Ralf Reussner. *Elasticity in Cloud Computing: What it is, and What it is Not.* In Proceedings of the 10th International Conference on Autonomic Computing, pages 24–28, San Jose, CA, June 2013. (Cited on page 162.)

[Herder 2006] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg and Andrew S. Tanenbaum. *MINIX 3: a highly reliable, self-repairing operating system.* SIGOPS - Operating Systems Review, vol. 40, no. 3, pages 80–89, Jul 2006. (Cited on page 32.)

[Horn 2001] Paul Horn. *Autonomic Computing: IBM's perspective on the State of Information Technology.* http://www.research.ibm.com/autonomic/, October 2001. (Cited on page 21.)

[HP Operations Manager] HP Operations Manager. *Fault and performance monitoring.* http://www8.hp.com/us/en/software-solutions/software.html?compURI=1170678, 2011. (Cited on page 65.)

[Huang 1995] Y. Huang, Nick Kolettis and N. Dudley Fulton. *Software Rejuvenation: Analysis, Module and Applications.* In Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing, pages 381–390, Pasadena, CA, USA, Jun 1995. (Cited on page 142.)

[Huhns 2003] Michael N. Huhns, Vance T. Holderfield and Rosa Laura Zavala Gutierrez. *Robust software via agent-based redundancy.* In Proceedings of the 2nd international joint conference on Autonomous Agents and Multiagent Systems, pages 1018–1019, Melbourne, Australia, Jul 2003. (Cited on page 32.)

[IBM Tivoli Monitoring] IBM Tivoli Monitoring. *Proactive system monitoring.* http://www-03.ibm.com/software/products/us/en/tivomoni/, 2012. (Cited on page 65.)

[InfraRED] InfraRED. *Opensource J2EE Performance Monitoring Tool.* http://infrared.sourceforge.net/, 2006. (Cited on page 89.)

[Islam 2012] Sadeka Islam, Kevin Lee, Alan Fekete and Anna Liu. *How a consumer can measure elasticity for cloud platforms.* In Proceedings of the 3rd ACM/SPEC

International Conference on Performance Engineering, pages 85–96, Boston, MA, USA, Apr 2012. (Cited on page 163.)

[JavaMelody] JavaMelody. *Monitoring of JavaEE applications.* http://code.google.com/p/javamelody/, 2011. (Cited on page 89.)

[jeeObserver] jeeObserver. *J2EE performance monitoring tool.* http://www.jeeobserver.com, 2011. (Cited on page 89.)

[Jiang 2006] Guofei Jiang, Haifeng Chen and Kenji Yoshihira. *Discovering Likely Invariants of Distributed Transaction Systems for Autonomic System Management.* Cluster Computing, vol. 9, no. 4, pages 385–399, Oct 2006. (Cited on pages 68 and 112.)

[JIP] JIP. *Java Interactive Profiler.* http://jiprof.sourceforge.net/, 2010. (Cited on page 89.)

[JRat] JRat. *Java Runtime Analysis Toolkit.* http://jrat.sourceforge.net/, 2007. (Cited on page 89.)

[Kaiser 2002] Gail Kaiser, Phil Gross, Gaurav Kc, Janak Parekh and Giuseppe Valetto. *An Approach to Autonomizing Legacy Systems.* In Proceedings of the first Workshop on Self-Healing, Adaptive and Self-Managed Systems, New York City, NY, June 2002. (Cited on page 22.)

[Kelly 2005] Terence Kelly. *Detecting performance anomalies in global applications.* In Proceedings of the 2nd conference on Real, Large Distributed Systems, volume 2, pages 42–47, San Francisco, CA, USA, Dec 2005. (Cited on page 49.)

[Kelly 2006] Terence Kelly and Alex Zhang. *Predicting performance in distributed enterprise applications.* Technical Report HPL-2006-76, HP Laboratories Palo Alto, Palo Alto, CA, USA, May 2006. (Cited on pages 49 and 183.)

[Kiciman 2005] E. Kiciman and A. Fox. *Detecting Application-level Failures in Component-based Internet Services.* IEEE Transactions on Neural Networks, vol. 16, no. 5, pages 1027–1041, 2005. (Cited on pages 3, 24, 30, 69 and 111.)

[Kiczales 1997] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier and John Irwin. *Aspect-Oriented Programming.* In Proceedings of the 11th European Conference on Object-Oriented Programming, pages 220–242, Jyväskylä, Finland, June 1997. (Cited on pages 40, 41, 68, 88, 89 and 164.)

[Kubiatowicz 2000] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishan Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells and Ben Zhao. *OceanStore: an architecture for global-scale persistent storage*. ACM SIGPLAN Notices, vol. 35, no. 11, pages 190–201, Nov 2000. (Cited on page 22.)

[Kumar 2006] Naveen Kumar, Bruce R. Childers and Mary Lou Soffa. *Low overhead program monitoring and profiling*. ACM SIGSOFT Software Engineering Notes, vol. 31, pages 28–34, Jan 2006. (Cited on page 90.)

[Laddad 2009] Ramnivas Laddad. AspectJ in Action: Practical Aspect-Oriented Programming. ISBN: 978-1933988054. Manning Publications, Shelter Island, NY, USA, 2nd edition, Oct 2009. (Cited on pages 70, 89 and 186.)

[Lamport 1982] Leslie Lamport, Robert Shostak and Marshall Pease. *The Byzantine Generals Problem*. ACM Transactions on Programming Languages and Systems, vol. 4, no. 3, pages 382–401, Jul 1982. (Cited on pages 3 and 27.)

[Landwehr 2005] Niels Landwehr, Mark Hall and Eibe Frank. *Logistic Model Trees*. Journal Machine Learning, vol. 59, no. 1-2, pages 161–205, May 2005. (Cited on page 188.)

[Laster 2007] Sharee Laster and Ayodeji Olatunji. *Autonomic computing: Towards a self-healing system*. In Proceedings of the American Society for Engineering Education Illinois-Indiana Section Conference, pages 62–78, Indianapolis, IN, USA, Mar 2007. (Cited on page 25.)

[Laszewski 2012] Gregor von Laszewski, Javier Diaz, Fugang Wang and Geoffrey C. Fox. *Comparison of Multiple Cloud Frameworks*. In Proceedings of the IEEE Fifth International Conference on Cloud Computing, pages 734–741, Honolulu, Hawaii, USA, Jun 2012. (Cited on page 162.)

[Li 2002] Lei Li, Kalyanaraman Vaidyanathan and Kishor S. Trivedi. *An Approach for Estimation of Software Aging in a Web Server*. In Proceedings of the International Symposium on Empirical Software Engineering, pages 91–100, Nara, Japan, Aug 2002. (Cited on pages 110, 142, 183, 184 and 188.)

[Li 2007] Junguo Li, Gang Huang, Jian Zou and Hong Mei. *Failure Analysis of Open Source J2EE Application Servers*. In Proceedings of the Seventh International Conference on Quality Software, pages 198–208, Portland, OR, USA, Oct 2007. (Cited on page 112.)

[Liang 2007] Yun-Chia Liang and AliceE. Smith. *The Ant Colony Paradigm for Reliable Systems Design*. In Gregory Levitin, editor, Intelligence in Reliability

Engineering: New Metaheuristics, Neural and Fuzzy Techniques in Reliability, volume 40 of *Studies in Computational Intelligence*, pages 1–20. Springer, 2007. (Cited on page 22.)

[Linden 2006] Greg Linden. *Make Data Useful*. http://www.scribd.com/doc/4970486/, 2006. (Cited on pages 29, 78, 137 and 171.)

[Lohman 2002] Guy M. Lohman and Sam Lightstone. *SMART: Making DB2 (More) Autonomic*. In Proceedings of the 28th International Conference on Very Large Data Bases, pages 877–879, Hong Kong, China, Aug 2002. (Cited on page 21.)

[LTW] LTW. *Load Time Weaving*. http://eclipse.org/aspectj/doc/released/devguide, 2011. (Cited on page 89.)

[Mao 2012] Ming Mao and Marty Humphrey. *A Performance Study on the VM Startup Time in the Cloud*. In Proceedings of the IEEE fifth International Conference on Cloud Computing, pages 423–430, Honolulu, Hawaii, USA, Jun 2012. (Cited on page 162.)

[Mazzucco 2010] Michele Mazzucco, Dmytro Dyachuk and Ralph Deters. *Maximizing Cloud Providers' Revenues via Energy Aware Allocation Policies*. In Proceedings of the International Conference on Cloud Computing, pages 131–138, Miami, FL, USA, Jul 2010. (Cited on page 163.)

[Menon 2003] Jai Menon, David A. Pease, Robert M. Rees, Linda Duyanovich and Bruce Light Hillsberg. *IBM Storage Tank - A heterogeneous scalable SAN file system*. IBM Systems Journal, vol. 42, no. 2, pages 250–267, Apr 2003. (Cited on page 21.)

[Mi 2008] Ningfang Mi, Ludmila Cherkasova, Kivanc M. Ozonat, Julie Symons and Evgenia Smirni. *Analysis of Application Performance and Its Change via Representative Application Signatures*. In Proceedings of the Network Operations and Management Symposium, pages 216–223, Salvador, Bahia, Brazil, Apr 2008. (Cited on pages 68, 113 and 114.)

[Modafferi 2006] Stefano Modafferi, Enrico Mussi and Barbara Pernici. *SH-BPEL: a self-healing plug-in for Ws-BPEL engines*. In Proceedings of the 1st workshop on Middleware for Service Oriented Computing, pages 48–53, Melbourne, Australia, Nov 2006. (Cited on page 34.)

[Montresor 2001] Alberto Montresor. *The Anthill Project Part II: The Anthill Framework*. http://www.cs.unibo.it/projects/anthill/papers/anthill-4p.pdf, 2001. (Cited on page 22.)

[Mosberger 1998] David Mosberger and Tai Jin. *httperf - A Tool for Measuring Web Server Performance.* In Proceedings of the First Workshop on Internet Server Performance, pages 59–67, Madison, WI, USA, Jun 1998. (Cited on pages 75, 82, 96 and 174.)

[Munawar 2006] Mohammad Ahmad Munawar and Paul A. S. Ward. *Adaptive Monitoring in Enterprise Software Systems.* In Proceedings of the First Workshop on Tackling Computer Systems Problems with Machine Learning Techniques, Saint-Malo, France, Jun 2006. (Cited on page 90.)

[N1 Grid Technology] N1 Grid Technology. *Just In Time Computing.* http://www.sun.com/software/solutions/n1/wp-n1.pdf, 2003. (Cited on pages 21 and 22.)

[Nagios] Nagios. *IT Infrastructure Monitoring.* http://www.nagios.org/, 2009. (Cited on page 65.)

[OpenNebula] OpenNebula. *Building and managing virtualized enterprise data centers and enterprise private clouds.* http://opennebula.org, 2013. (Cited on pages 168 and 170.)

[Oppenheimer 2003] David Oppenheimer, Archana Ganapathi and David A. Patterson. *Why do internet services fail, and what can be done about it?* In Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems, volume 4, page 1, Seattle, WA, USA, Mar 2003. (Cited on page 69.)

[Oracle Glassfish Server] Oracle Glassfish Server. *Java EE 6 application server.* http://www.oracle.com/technetwork/middleware/glassfish/, 2012. (Cited on pages 71, 95 and 119.)

[Parashar 2005] Manish Parashar and Salim Hariri. Autonomic computing: An overview. Springer Verlag, 2005. (Cited on page 23.)

[Parashar 2006] M. Parashar, H. Liu, Z. Li, V. Matossian, C. Schmidt, G. Zhang and S. Hariri. *AutoMate: Enabling Autonomic Applications on the Grid.* Journal Cluster Computing, vol. 9, no. 2, pages 161–174, Apr 2006. (Cited on page 22.)

[Patterson 2002] David Patterson, Aaron Brown, Pete Broadwell, George Candea, Mike Chen, James Cutler, Patricia Enriquez, Armando Fox, Emere Kiciman, Matthew Merzbacher, David Oppenheimer, Naveen Sastry, William Tetzlaff, Jonathan Traupamn and Noah Treuhaft. *Recovery Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies.* Technical report, UC Berkeley, Mar 2002. (Cited on pages 22, 23, 24, 143 and 144.)

[Pernici 2007] Barbara Pernici and Anna Maria Rosati. *Automatic Learning of Repair Strategies for Web Services*. In Proceedings of the Fifth European Conference on Web Services, pages 119–128, Halle, Germany, Nov 2007. (Cited on page 144.)

[Pertet 2005] Soila Pertet and Priya Narasimhan. *Causes of Failure in Web Applications*. Technical report - Parallel Data Lab, Carnegie Mellon University, December 2005. (Cited on pages 2, 24, 29, 74, 76, 120 and 121.)

[Poggi 2011] Nicolas Poggi, David Carrera, Ricard Gavalda and Eduard Ayguade. *Non-intrusive Estimation of QoS Degradation Impact on E-Commerce User Satisfaction*. In Proceedings of the 10th International Symposium on Network Computing and Applications, pages 179–186, Cambridge, MA, USA, Aug 2011. (Cited on pages 29 and 184.)

[Power 2010] Sean Power. *Metrics 101: What to Measure on Your Website*. http://www.slideshare.net/bitcurrent/metrics-101, June 2010. (Cited on pages 2 and 29.)

[Profiler4j] Profiler4j. *CPU profiler for Java*. http://profiler4j.sourceforge.net/, 2006. (Cited on page 89.)

[Psaier 2011] Harald Psaier and Schahram Dustdar. *A survey on self-healing systems: approaches and systems*. Journal Computing - Cloud Computing, vol. 91, no. 1, pages 43–73, Jan 2011. (Cited on pages 25 and 30.)

[Quinlan 1993] J. Ross Quinlan. C4.5: Programs for machine learning. Morgan Kaufmann, January 1993. (Cited on page 187.)

[R Development Core Team 2010] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2010. ISBN: 3-900051-07-0. (Cited on page 72.)

[Rezaei 2010] Arash Rezaei and Mohsen Sharifi. *Rejuvenating High Available Virtualized Systems*. In Proceedings of the International Conference on Availability, Reliability and Security, pages 289–294, Krakow, Poland, Feb 2010. (Cited on page 163.)

[Rheem 2010] Carroll Rheem. *Consumer Response to Travel Site Performance*. In A PhoCusWright and Akamai WHITEPAPER, Abr 2010. (Cited on pages 2 and 29.)

[Rish 2005] I. Rish, M. Brodie, Sheng Ma, N. Odintsova, A. Beygelzimer, G. Grabarnik and K. Hernandez. *Adaptive diagnosis in distributed systems*. IEEE Transactions on Neural Networks, vol. 16, no. 5, pages 1088–1109, Sep 2005. (Cited on page 90.)

[Rodrigues 2008] Nuno Rodrigues, Décio Sousa and Luis Silva. *A Fault-Injector Tool to Evaluate Failure Detectors in Grid-Services.* In Making Grids Work, ISBN: 978-0-387-78447-2, pages 261–271. Springer US, 2008. (Cited on pages 147 and 193.)

[Ryan 2007] Thomas P. Ryan. Modern engineering statistics. ISBN: 978-0470081877. Wiley-Interscience, 1st edition, Sep 2007. (Cited on page 51.)

[Sakr 2012] Sherif Sakr and Anna Liu. *SLA-Based and Consumer-centric Dynamic Provisioning for Cloud Databases.* In Proceedings of the IEEE fifth International Conference on Cloud Computing, pages 360–367, Honolulu, Hawaii, USA, Jun 2012. (Cited on pages 163 and 164.)

[Salfner 2010] Felix Salfner, Maren Lenk and Miroslaw Malek. *A survey of online failure prediction methods.* ACM Computing Surveys, vol. 42, no. 3, pages 1–42, part 10, Mar 2010. (Cited on page 182.)

[Schneider 1990] Fred B. Schneider. *Implementing fault-tolerant services using the state machine approach: a tutorial.* ACM Computing Surveys, vol. 22, no. 4, pages 299–319, Dec 1990. (Cited on pages 3 and 27.)

[Schroeder 1995] Beth A. Schroeder. *On-Line Monitoring: A Tutorial.* Journal Computer, vol. 28, no. 6, pages 72–78, Jun 1995. (Cited on page 62.)

[Schurman 2009] Eric Schurman and Jake Brutlag. *Performance Related Changes and their User Impact.* http://www.scribd.com/doc/16877297/, June 2009. Presented at Velocity Web Performance and Operations Conference. (Cited on pages 78 and 137.)

[Shapiro 2004] Michael W. Shapiro. *Self-Healing in Modern Operating Systems.* Magazine Queue - Programming Languages, vol. 2, no. 9, pages 66–75, Dec 2004. (Cited on page 32.)

[Silva 2006] Luís Silva, Henrique Madeira and Joao Gabriel Silva. *Software Aging and Rejuvenation in a SOAP-based Server.* In Proceedings of the fifth IEEE International Symposium on Network Computing and Applications, pages 56–65, Cambridge, MA, USA, Jul 2006. (Cited on page 143.)

[Silva 2008] Luís Moura Silva. *Comparing Error Detection Techniques for Web Applications: An Experimental Study.* In Proceedings of the seventh IEEE International Symposium on Network Computing and Applications, pages 144–151, Cambridge, MA, USA, Jul 2008. (Cited on page 74.)

[Silva 2009] Luís Moura Silva, Javier Alonso and Jordi Torres. *Using Virtualization to Improve Software Rejuvenation*. IEEE Transactions on Computers, vol. 58, no. 11, pages 1525–1538, Nov 2009. (Cited on page 143.)

[Simic 2010] Bojan Simic. *Ten Areas that are Changing Market Dynamic in Web Performance Management*. http://www.trac-research.com/web-performance/, December 2010. (Cited on pages 2 and 29.)

[Site 24x7] Site 24x7. *Website Monitoring*. http://www.site24x7.com, 2010. (Cited on page 66.)

[Smith 2001] Wayne D. Smith. *TPC-W: Benchmarking an Ecommerce Solution*. http://www.tpc.org/tpcw/, 2001. (Cited on pages 70, 72, 95, 117, 145, 169 and 190.)

[Souders 2007] Steve Souders. High performance web sites. ISBN: 978-0-596-52930-7. O'Reilly Media, Sep 2007. (Cited on pages 30 and 37.)

[Sterritt 2002] Roy Sterritt. *Towards Autonomic Computing: Effective Event Management*. In Proceedings of the 27th Annual NASA Goddard Software Engineering Workshop, pages 40–47, Greenbelt, MD, USA, Dec 2002. (Cited on page 20.)

[Sterritt 2005a] Roy Sterritt. *Autonomic computing*. Innovations in Systems and Software Engineering, vol. 1, no. 1, pages 79–88, Abr 2005. (Cited on page 17.)

[Sterritt 2005b] Roy Sterritt, Manish Parashar, Huaglory Tianfield and Rainer Unland. *A Concise Introduction to Autonomic Computing*. Advanced Engineering Informatics, vol. 19, no. 3, pages 181–187, Jul 2005. (Cited on page 16.)

[Stress] Stress. *Load and stress test tool*. http://linux.die.net/man/1/stress, 2010. (Cited on pages 75, 122, 147, 152, 177 and 193.)

[Suleiman 2012] Basem Suleiman, Sherif Sakr, D. Ross Jeffery and Anna Liu. *On understanding the economics and elasticity challenges of deploying business applications on public cloud infrastructure*. Journal Internet Services and Applications, vol. 3, no. 2, pages 173–193, Sep 2012. (Cited on page 162.)

[Sultan 2003] Florin Sultan, Aniruddha Bohra, Pascal Gallard, Iulian Neamtiu, Steve Smaldone, Yufei Pan and Liviu Iftode. *Nonintrusive Failure Detection and Recovery for Internet Services using Backdoors*. Technical Report DCS-TR-524, Rutgers University, December 2003. (Cited on page 143.)

[Sultan 2005] Florin Sultan, Aniruddha Bohra, Stephen Smaldone, Yufei Pan, Pascal Gallard, Iulian Neamtiu and Liviu Iftode. *Recovering Internet Service Sessions* *from Operating System Failures.* Journal IEEE Internet Computing, vol. 9, no. 2, pages 17–27, Mar 2005. (Cited on page 143.)

[Swatch] Swatch. *Active log file monitoring tool.* http://swatch.sourceforge.net/, 2008. (Cited on pages 67 and 70.)

[Tanenbaum 2006] Andrew S. Tanenbaum, Jorrit N. Herder and Herbert Bos. *Can We Make Operating Systems Reliable and Secure?* Journal Computer, vol. 39, no. 5, pages 44–51, May 2006. (Cited on page 31.)

[Tesauro 2004] Gerald Tesauro, David M. Chess, William E. Walsh, Rajarshi Das, Alla Segal, Ian Whalley, Jeffrey O. Kephart and Steve R. White. *A Multi-Agent Systems Approach to Autonomic Computing.* In Proceedings of the third International Joint Conference on Autonomous Agents and Multiagent Systems, volume 1, pages 464–471, New York, NY, USA, Jul 2004. (Cited on page 32.)

[Trivedi 2000] K. S. Trivedi, K. Vaidyanathan and K. Goseva-Popstojanova. *Modeling and Analysis of Software Aging and Rejuvenation.* In Proceedings of the 33rd Annual Simulation Symposium, pages 270–279, Washington, DC, USA, Apr 2000. (Cited on pages 22 and 23.)

[Trivedi 2004] KishorS. Trivedi and Kalyanaraman Vaidyanathan. *Software Rejuvenation - Modeling and Analysis.* In Information Technology, volume 157 of *IFIP International Federation for Information Processing*, pages 151–182. Springer US, 2004. (Cited on page 142.)

[Tsakalozos 2011] Konstantinos Tsakalozos, Mema Roussopoulos and Alex Delis. *VM Placement in non-Homogeneous IaaS-Clouds.* In Proceedings of the International Conference on Service-Oriented Computing, pages 172–187, Paphos, Cyprus, Dec 2011. (Cited on pages 161 and 163.)

[Ueda 2010] Yohei Ueda and Toshio Nakatani. *Performance variations of two open-source cloud platforms.* In Proceedings of the IEEE International Symposium on Workload Characterization, pages 1–10, Atlanta, GA, USA, Dec 2010. (Cited on pages 161 and 162.)

[Vaidyanathan 1999] Kalyanaraman Vaidyanathan and Kishor S. Trivedi. *A Measurement-Based Model for Estimation of Resource Exhaustion in Operational Software Systems.* In Proceedings of the 10th International Symposium on Software Reliability Engineering, pages 84–93, Boca Raton, FL, USA, Nov 1999. (Cited on pages 183 and 184.)

[Wang 2012] Wei Wang, Xiang Huang, Xiulei Qin, Wenbo Zhang, Jun Wei and Hua Zhong. *Application-Level CPU Consumption Estimation: Towards Performance Isolation of Multi-tenancy Web Applications.* In Proceedings of the IEEE fifth International Conference on Cloud Computing, pages 439–446, Honolulu, Hawaii, USA, Jun 2012. (Cited on page 163.)

[Weinman 2011] Joe Weinman. *Time is Money: The Value of "On-Demand".* http://www.joeweinman.com/Resources/Joe_Weinman_Time_Is_Money.pdf, 2011. (Cited on page 163.)

[WEKA] WEKA. *Data Mining Software in Java.* http://www.cs.waikato.ac.nz/ml/weka/, 2012. (Cited on pages 184 and 187.)

[WPWG] WPWG. *Web Performance Working Group.* http://www.w3.org/2010/webperf/, 2010. (Cited on page 66.)

[Zabbix] Zabbix. *Enterprise Monitoring Solution.* http://www.zabbix.com/, 2010. (Cited on pages 65, 70 and 157.)

[Zar 1972] Jerrold H. Zar. *Significance Testing of the Spearman Rank Correlation Coefficient.* Journal of the American Statistical Association, vol. 67, no. 339, pages 578–580, 1972. (Cited on pages 49, 116 and 165.)