UNIVERSITA' DEGLI STUDI DI NAPOLI FEDERICO II

Dottorato di Ricerca in Ingegneria Informatica ed Automatica

ON-LINE DETECTION OF ANOMALIES IN MISSION-CRITICAL SOFTWARE SYSTEMS

ANTONIO BOVENZI

Tesi di Dottorato di Ricerca
(XXV Ciclo)
Aprile 2013

Il Tutore
Prof. Stefano Russo

Il Coordinatore del Dottorato
Prof. Francesco Garofalo

DIPARTIMENTO DI INGEGNERIA ELETTRICA E TECNOLOGIE DELL'INFORMAZIONE

Via Claudio, 21 – I-80125 Napoli - ☎ *[#39]* (0)81 768 3813 - *[#39]* (0)81 768 3816

On-line Detection of Anomalies in Mission-Critical Software Systems

By

Antonio Bovenzi

SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
AT
UNIVERSITY OF NAPLES FEDERICO II
VIA CLAUDIO 21, 80125 – NAPOLI, ITALY
APRIL 2013

© Copyright by Antonio Bovenzi, 2013

Table of Contents

Table of Contents iii

List of Tables v

List of Figures vi

Introduction 1

1 The Role of Anomaly Detection for Dependable Systems 10
- 1.1 Basic Concepts and Definitions 10
- 1.2 The Genesis of Software Anomalies 18
- 1.3 The Anomaly Detection Problem 32
- 1.4 Anomaly Detection Challenges in Mission-Critical Software Systems 33

2 Monitoring and Detection: Approaches and Frameworks 36
- 2.1 Background 36
- 2.2 Approaches and Frameworks 43
  - 2.2.1 Monitoring 45
  - 2.2.2 Detection 63
- 2.3 Metrics for Quantitative Evaluation 77

3 OS-level Detection of Anomalies 83
- 3.1 Requirements and Assumptions 83
- 3.2 High-level Architecture of the Framework 90
- 3.3 Internals of the Framework 94
  - 3.3.1 OS Monitoring Infrastructure 94
  - 3.3.2 The Statistical Predictor and Safety Margin Algorithm 103
  - 3.3.3 The α-counter 107
- 3.4 Parameters Tuning and Computational Cost 110

4 **Experimental Methodology** **112**
4.1 Motivations . . . 112
4.2 The Adopted Methodology . . . 113
4.2.1 Overview . . . 113
4.2.2 Definition . . . 116
4.2.3 Planning . . . 120
4.2.4 Execution . . . 125
4.2.5 Analysis . . . 126

5 **Experimental Results** **128**
5.1 The SWIM-BOX Case Study . . . 128
5.1.1 Definition phase . . . 129
5.1.2 Planning phase . . . 135
5.1.3 Execution phase . . . 139
5.1.4 Analysis Phase . . . 141

Conclusion **153**

Bibliography **156**

List of Tables

1.1 Some examples of ODC defect types 24
1.3 Proportion of ODC defect types in field studies 26
1.5 Some examples of ODC defect triggers 26

2.1 Most common indirect monitoring approaches 50
2.2 Differences among LTT, Systemtap and DTrace tracing mechanisms 58
2.3 Most relevant monitored indicators of the surveyed frameworks 62
2.4 Perspectives of analysis for model-based detection approaches 64
2.6 Surveyed detection approaches and techniques 77
2.7 Basic metrics for characterizing detector performance 78
2.8 Performance metrics to evaluate anomaly detectors 82

3.1 Monitored variables for Linux and Windows 100

5.1 Considered fault types 135
5.2 Source-code faults injected in the case study 136
5.3 Distribution of failures observed in faulty runs 140
5.5 Coverage and accuracy of the detectors 144

List of Figures

1.1 A high-level overview of system and its operational environment 11
1.2 Scope of the dissertation 18
1.3 Origin and detection of anomalies during development and operational phases 19
1.4 The pathology of failures 21
1.5 Proportions of Bohr-Mandelbugs across a timeline 31
1.6 Different fault tolerance strategies that can be used with Bohr-Mandelbugs 31

2.1 A general high-level overview of monitoring and detection infrastructures . 37
2.2 The perspectives of analysis for the monitoring and detection approaches . 40
2.3 An example of contextual anomaly 40
2.4 An example of collective anomaly in electrocardiogram (medical domain) . 41
2.5 A high-level overview of distributed detectors 44
2.6 Push- vs pull-style monitoring approach 47
2.7 The architecture of Ganglia 53
2.8 Example of probes at multiple levels 55
2.9 LTT architecture 56
2.10 DTrace architecture 57
2.11 Conceptual view of Magpie 60
2.12 Pinpoint architecture 76
2.13 Time line showing true positives according to the parameter *Time to Detect* 80

3.1 A simplified architecture of ATC systems 84
3.2 A high-level overview of the sosmon framework 87
3.3 The high-level architecture of the framework 90

3.4 An example of trace collected by sosmon . . . 92
3.5 The SystemTap tool overview . . . 97
3.6 The monitoring infrastructure for Windows . . . 101
3.7 An example of adaptive vs static anomaly detection thresholds . . . 105

4.1 The experimental methodology . . . 114
4.2 An example of star schema . . . 124

5.1 SWIM-BOX high-level architecture . . . 130
5.2 Interaction scenario for the case-study . . . 132
5.3 Data repository designed for OLAP analysis . . . 138
5.4 Best experimental results in Linux using coverage $c = 0.9999$, memory $m = 20$, combination window $w = 5$ and $G = 0.2$ . . . 142
5.5 Best experimental results in Windows using coverage $c = 0.99$, memory $m = 40$, combination window $w = 20$ and $G = 0.4$ . . . 143
5.6 Experimental results for Linux using coverage $c = 0.9999$, memory $m = 20$, combination window $w = 5$ and varying global threshold G . . . 146
5.7 Experimental results for Linux using coverage $c = 0.9$, memory $m = 20$, combination window $w = 5$ and varying global threshold G . . . 147
5.8 Experimental results for Linux using coverage $c = 0.9999$, combination window $w = 5$, global threshold $G = 0.18$ and varying memory m . . . 148
5.9 Experimental results for Linux using coverage $c = 0.9999$, memory $m = 20$, global threshold $G = 0.2$ and varying w . . . 149
5.10 Overhead for the SWIM-BOX varying the invocation period of the operations 150
5.11 Results of sensitivity analysis of Coverage and Accuracy to the number of monitored indicators in Linux . . . 152
5.12 Results of sensitivity analysis of Coverage and Accuracy to the number of monitored indicators in Windows . . . 152

Introduction

"The integrating potential of software has allowed designers to contemplate more ambitious systems encompassing a broader and more multidisciplinary scope", stated Michael Lyu in 1996 [1]. After twenty years this is completely perceived by our society. Almost every daily activity is related to the software: the simplest actions, such as turning on the light, and the more complex activities, e.g., organizing a business trip with your boss, depend, directly or indirectly, upon the underlying software. Hence, everyone needs dependable software, i.e., software that can be justifiably trusted.

There is a category of software systems for which failures cannot be admitted since they may be critical for the success of their mission. These are called **mission-critical software systems**. Examples are the SCADA systems in power grids, the software infrastructure in banking and transportation systems, and the control software in spacecraft. Hence, depending on the domain of systems, software failures may be business-critical, i.e., when they affect essential operations that dramatically impair company affairs, or even safety-critical, i.e., when failures may hamper human life. *Revealing anomalies in such scenarios is fundamental to avoid unexpected failures that may lead to loss of business or even may endanger our lives.*

The IEEE Standard 1044-2009 defines anomalies as abnormality, irregularity, inconsistency, or variance from expectations. Expectations represent nominal or desired behaviors that may be derived from requirements, design documents, standards and on-field experience.

Anomalies can be observed at each stage of the software development life cycle. This dissertation focuses on *runtime software anomalies*, hereafter anomalies, namely the deviations from the expected behavior that occur during operation.

In business- and life-critical infrastructures –e.g., power grids, transportation systems, financial services –anomalies need to be revealed timely to support diagnosis procedures for the identification and the activation of proper countermeasures before they lead to irremediable failures. For example, anomalies in system call traces of privileged processes and then anomalies in the network traffic in a server may be related to malicious actions of intruders that have compromised the machine and are communicating sensitive information to unauthorized destinations [2]. An anomalous pattern in entering or leaving critical sections may be related to process hang, (i.e., the process is indefinitely blocked or is endlessly wasting CPU cycles) [3]. Anomalies in memory consumption, when not properly treated, can lead to performance degradation and to crash [4].

New industry trends in designing and developing mission-critical software systems lead towards *(i)* the use of commercial off-the-shelf (OTS) for minimizing costs and time to market and *(ii)* larger and more complex systems to push performance to the limit. Revealing anomalies in such a scenario is an intricate task for a bunch of reasons that are discussed as follows.

The notion of expectations encompassing every possible normal behavior is very difficult to apply. This is especially true for OTS items since expectations typically depend on the type of component, the application domain, and the *operating environment*, and very often a thorough assessment of OTS failure modes and theirs vulnerabilities is not unavailable [5]. Furthermore, the operating environment, i.e., the combination of network, hardware, OS, virtualization and required components may force the conditions to trigger the activation of residual software faults leading to failure modes, which are not foreseen at design time. Indeed, the operating environment in which the OTS item is deployed may have been not

well-tested because of time constraints and technical limitations –indeed exhaustive testing is typically unfeasible. The difference between production and operating environments can be so large that there have been movements in the software warehouses to encouraging early release of software so that customers can do the debug [6]. Thus, for all these reasons *the boundaries between normal and anomalous behavior are often not precise*.

The integration of several interconnected and interdependent components emphasizes another aspect of OTS-based mission-critical systems: the *domino effect*, i.e., the propagation of undesirable effects from an entity to another that depends on it.

A further problem in architecting anomaly detection framework for mission-critical systems is due to the evolution they undergo during the lifetime; indeed, they may be integrated with other systems, and/or extended to fulfill new demanding requirements. This may force the integrated (sub)systems to operate beyond the original design conditions [7]. Performance anomalies, e.g., in overload conditions due to changes in the workload with respect to what the system was originally designed for, may arise in such scenarios. For this reason, *the adopted detection mechanism shall adapt to the evolving situation to be effective*.

Mission-critical software systems are also exposed to intentional malicious faults [8], such as worms, and cyber-attacks. Attackers can target the border of the system (e.g. Distributed Denial of Service) or maliciously sneak within its boundaries and act internally, such as the recent Stuxnet worm in 2010 [9]. The U.S. Department of Transportation report [10] determined that the Air Traffic Control's Web-based applications are far to be secure against attacks or unauthorized access, and that many intrusion-detection systems cannot support remediation in a timely manner. In such a scenario *the attackers may mask anomalies to make their actions appearing like normal*.

Last but not the least, the type and the amount of load imposed to the systems may also contribute to jeopardize its dependability. Indeed, many studies have observed how the the type of failures and the failure rates may vary dramatically depending on the workload

being executed [11, 12, 13]. For instance, in [12, 14] and in our work [15, 16] the workload is observed to be an important *software aging* factor. It is noteworthy that the term software aging in this dissertation refers to the accumulation of errors in the software components and/or in their operating environment that causes performance degradation and increasing failure rate of software systems and eventually lead to system hang or crash [17].

Revealing anomalies by monitoring the operating environment in which OTS items are deployed is a promising approach when traditional detection mechanisms have poor performance or cannot be applied [3, 18]. The driving idea is to shift the observation perspective to the operating environment by monitoring the communication and the resource usage patters between the component(s) and the OS. Indeed, operating systems, especially the most recent, offer facilities to collect a wealth of data, such as system call errors, scheduling delays and waiting time on shared resources, useful to reveal when the application components deviate from expectations.

The approach is particularly suited for OTS-based systems, since it does not require to modify the components. Furthermore, a framework monitoring the components from the OS perspective can be of general use and applied to a variety of circumstances and applications in a much more efficient and cheap way than instrumenting the application components themselves; in fact, instead of re-instrumenting every OTS item and application each time, it will be just necessary to tune a ready-to-use framework.

Understanding how to exploit OS-level tracing facilities for anomaly detection is an intricate task. First, the selection of the probe points, i.e., the (kernel- and user-level) OS functions that need to be monitored, has to be carefully addressed. Indeed, a fine-grained tracing enables to accurately reveal and diagnose component anomalies [19]; however, the prize to pay for such level of detail is a high overhead for the target application which is unaffordable for production environments.

Second, OS-level anomaly detection requires to collect large amount of data, e.g. coming from probes spread over the system, that needs to be analyzed to reveal that something is not working as expected. The amount of data is usually so huge that the detection and diagnosis processes are still performed off-line [18, 19] and even with human support that guides the process of filtering and correlating the relevant information [18].

Third, the effectiveness of OS-level detection approaches to timely reveal software anomalies is still somehow unknown. Different operating environments and the monitored OS indicators may influence the possibility to reveal relevant anomalies and/or the accuracy of the detection. Very few studies have exploited such approaches to perform on-line anomaly detection and, furthermore, only specific classes of anomalies, such as application hangs and system crashes, or one specific environment (e.g., Linux or Windows) have been addressed [3, 20].

Finally, *existing OS-level anomaly detection techniques are based on worst-case static thresholds, or on lengthy profiling phases, or on training*, e.g.,[3, 18, 20] and are not suited for systems subject to highly variable and non-stationary operating conditions.

Contribution

This dissertation investigates the suitability of the OS-level anomaly detection for OTS-based mission-critical software systems. In particular, the effort has been devoted to address the following fundamental issues: *(i)* evaluate the effectiveness of the approach under different worklaod, faultload (which lead to anomalies) and under variable and non-stationary operating conditions; *(ii)* assess the applicability of the approach under different operating systems; *(iii)* investigate the intrusiveness of the approach and its overhead.

The work has been conducted by following a bottom-up approach. First, literature

addressing on-line detection of software anomalies has been reviewed. Then, current OS-level detection approaches and their shortcomings have been explored. The results of this preliminary phase have driven the definition of the requirements and the design a novel framework for OS-level anomaly detection. The proposed framework, called *sosmon*, hinges on kernel-level tracing facilities and on on-line statistical analysis. The framework is configurable in the types and number of monitored indicators; the detection algorithm in turn has configuration parameters that allow tuning the way indicators are combined to reveal anomalies.

The proposed OS-level detection framework effectiveness has been evaluated by means of extensive experiments based on fault injection conducted on an OTS-based mission-critical system for Air Traffic Management (ATM). The considered industrial case study is the SWIMBOX, a middleware for the interoperability of future ATM systems, developed in the context of the SESAR European research project [1]. The injected faultload represents a share of representative faults that can be commonly found in software systems [21]. Furthermore, the experimental methodology is also meant to support practitioners to the framework configuration.

The suitability of the approach under different OSs has been investigated by implementing the framework for two operating systems used in production environments, namely Red Hat EL 5 and Windows Server 2008, and in which the SWIMBOX is deployed. The results corroborate the thesis that the OS-level anomaly detection is valuable approach when traditional detection mechanisms have poor performance or cannot be applied since the sosmon framework is indisputably useful to detect anomalies. However, the framework, and in particular the monitored indicators, need to be carefully selected to achieve comparable performance in both environments.

Finally, **the work has explored the framework performance varying the level**

[1] http://www.sesarju.eu/about

of intrusiveness of the monitoring infrastructure. Indeed, when the detection is on-line and timely decisions ought to be taken, the framework has to limit the impact on the mission of the system, and the overhead needs to be minimized. For this reason, the performance of the framework has been analyzed by varying the number of instrumentation points. Results are encouraging since the anomalies due to the activation of the injected faults can still be revealed at the prize of a slightly lower accuracy.

The work includes material from the following research papers, already published in peer-reviewed conferences and journals or submitted for review:

- A. Bovenzi, F. Brancati, A. Bondavalli, S. Russo, An OS-level Framework for Anomaly Detection in Complex Software Systems. IEEE Transactions on Dependable and Secure Computing, under revision.

- A. Bovenzi, F. Brancati, A. Bondavalli, S. Russo, A statistical anomaly-based algorithm for on-line fault detection in complex and critical system. The 30th International Conference on Computer Safety, Reliability and Security (SAFECOMP 2011). Napoli, Italy. September 19-21, 2011. Lecture Notes in Computer Science Volume 6894/2011, pp. 128-142. DOI: 10.1007/978-3-642-24270-0_10

- A. Bovenzi, F. Brancati, A. Bondavalli, S. Russo, Towards Identifying OS-level Anomalies to Detect Application Software Failures. The IEEE International Workshop on Measurements and Networking (M&N 2011). Anacapri (NA), Italy. October 10-11, 2011. DOI: 10.1109/IWMN.2011.6088494

- A. Bovenzi, M. Cinque, D. Cotroneo, R. Natella, G. Carrozza, OS-level hang detection in complex software systems. International Journal Critical Computer-Based Systems, Vol. 2, Nos. 3/4, pp.352-377.DOI: 10.1504/IJCCBS.2011.042333

- A. Bovenzi, G. Carrozza, D. Cotroneo, R. Pietrantuono, Error Detection framework for Complex Software Systems. Proceedings of 15th European Workshop on Dependable Computing (EWDC 2011). Pisa, Italy. May 11-12, 2011.ACM, New York, NY, USA, pp. 61-66. DOI: 10.1145/1978582.1978596

- A. Bovenzi, G. Carrozza, Monitoring Infrastructure for Diagnosing Complex Software. Innovative Technologies for Dependable OTS-based Critical Systems: Challenges and Achievements of the Critical Step Project. D. Cotroneo Ed., 2012. Springer-Verlag New York.

- A. Bovenzi, D. Cotroneo, R. Pietrantuono, S. Russo, On the Aging Effects due to Concurrency Bugs: a Case Study on MySQL. The 23nd International Symposium on Software Reliability Engineering (ISSRE 2012). Dallas, Tx, USA. November 27-30, 2012. DOI: 10.1109/ISSRE.2012.50

- A. Bovenzi, D. Cotroneo, R. Pietrantuono, S. Russo, Workload Characterization for Software Aging Analysis. The 22nd International Symposium on Software Reliability Engineering (ISSRE 2011). Hiroshima, Japan. November 29-Dicember 2, 2011. DOI: 10.1109/ISSRE.2011.18

The following material is related to the design and the implementation of a fault injection tool that can be used for assessing the dependability of OTS items, in particular for assessing their robustness. The material is not included in this dissertation since it is more focused on the dependability assessment during the testing phase and not during operation:

- A. Bovenzi, A. Napolitano, G. Carrozza, C. Esposito, JFIT: an Automatic Tool for Assessing Robustness of DDS-Compliant Middleware. Innovative Technologies for Dependable OTS-based Critical Systems: Challenges and Achievements of the Critical Step Project. D. Cotroneo Ed., 2012. Springer-Verlag New York Incorporated.

- A. Napolitano, A. Bovenzi, G. Carrozza, C. Esposito, Automatic Robustness Assessment of DDS-Compliant Middleware. The 17th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC 2011), Industry Track. Pasadena, CA, USA. December 12-14, 2011. DOI: 10.1109/PRDC.2011.51

- A. Bovenzi, G. Carrozza, S. Celentano, C. Esposito, A. Napolitano Automatic Robustness Testing of a DDS-compliant middleware. Workshop on Real-time, Embedded and Enterprise-Scale Time-Critical Systems. Washington, DC, USA. March 22-24,2011.

The dissertation is organized as follows:

Chapter 1 provides the basic concepts of dependability and the adopted terminology and motivates the importance of anomaly detection in dependable systems. Furthermore, the anomaly detection problem addressed in this dissertation and its challenges are discussed.

Chapter 2 discusses the state-of-the-art concerning monitoring and detection approaches and techniques, their benefits and their limitations for the target systems. A synthetic description of the contribution provided by this work, which is detailed in successive chapters, is also presented.

Chapter 3 details the proposed OS-level anomaly detection framework, the requirements, the design and the implementation.

Chapter 4 presents the experimental methodology used to assess the performance of the framework.

Chapter 5 discusses the results of the experiments, conducted using the methodology explained in Chapter 4.

Chapter 1

The Role of Anomaly Detection for Dependable Systems

This chapter provides the basic definitions and notions that will be used in the rest of the dissertation. Indeed, during the past decades many terms have entered into use to address anomaly detection issues since software systems have permeated through different domains (e.g., avionics, space, automotive, railway, power grids, medical). First, dependability, i.e., the ability to deliver a service that can be justifiably trusted, and the fundamental aspects as defined by the IFIP WG 10.4 are discussed. This is because, by using a top-down approach, they provide a comprehensive concept of dependability that encompasses many fundamental aspects, including dependability threats and means, that need to be introduced to understand the anomaly detection problem. Furthermore, this lays the ground for the subsequent analysis regarding the genesis of anomalies, their role in the system failure behavior and the actual challenges to anomaly detection in mission-critical software systems.

1.1 Basic Concepts and Definitions

Dependability issues have been addressed since the beginning of computer systems. Indeed, the first attempt to provide the principles of dependable computing were already discussed by Von Neumann in 1957 [22] –even if the paper only discusses general tendencies– and by Avizienis in 1967 [23] where detection, diagnosis and recovery concepts were integrated into *fault-tolerant* systems. Despite many efforts during successive decades, the seminal paper that defines the basic concepts of dependability (and security) and related terminology has been published only in 2004 [8] – even if the ideas were around since 1980 when the joint

committee on “Fundamental Concepts and Terminology” was founded. Hence, the concepts and the definitions provided in the following are taken from [8].

First, the dependability definition provided in the abstract of the chapter –*the ability to deliver a service that can be justifiably trusted*– has been further refined in *the ability of the system to avoid service failures that are more frequent and more sever than is acceptable*. With respect to the previous definition, which was given in 1985 by Laprie [24], three fundamental concepts need to be explained in order to understand this alternative definition: *system*, *service* and *failures*.

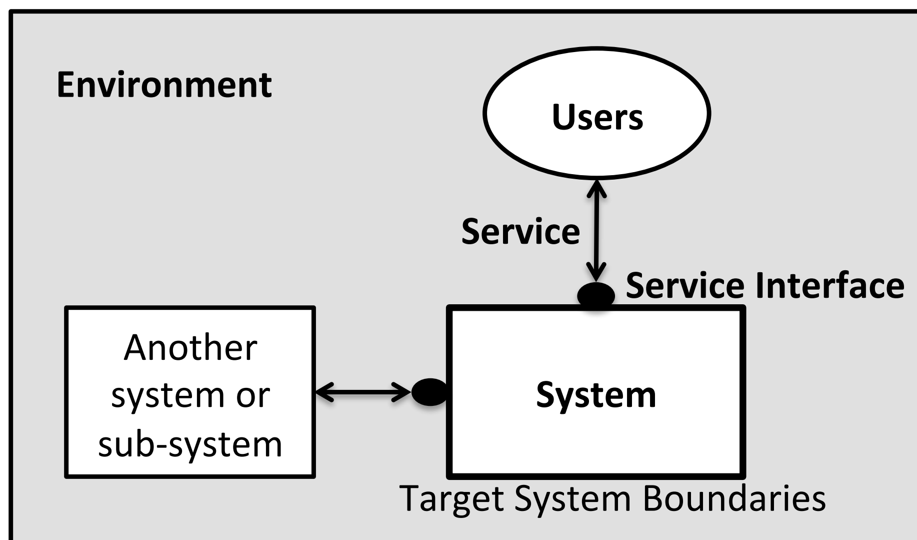


Figure 1.1: A high-level overview of system and its operational environment

In order to explain these concepts Figure 1.1 is introduced. The system is defined as the entity that interacts with other entities, i.e., other systems, which may include hardware, software, human operator and the physical world. The system boundary is the common frontier that separates the given system from other entities, which constitute its

environment.

The other basic concept, i.e., the service, can be explained introducing two other definitions: the *function* and the *behavior*. The former, as described by the functional and non functional specification, is what the system is intended to do. The latter is what the system does to implement its function and it is described by a sequence of states. The **service** is thus defined as *the behavior of the system as perceived by its user(s)*; the portion of the boundary of the system where the service is delivered is called *service interface*.

A **service failure**, hereafter failure, is an event that occurs when the delivered service is no more implementing the system function; in other words, a transition from correct service to incorrect service according to the specification.

Finally, in order to quantify the dependability, the specification of a system needs to include the requirements in terms of *(i)* the “acceptable frequency and severity” of service failures and *(ii)* the given operational environment.

It worth to mention that the system is usually the result of –and this is the case of actual mission-critical systems– the integration and cooperation of other (sub)systems, or **components**. For dependability analysis, the recursion stops when a further internal decomposition is not of interest, or impossible to discern, so that the component is considered *atomic*. In OTS-based mission-critical systems most of atomic entities can be OTS components or legacy components. It is worth to note that the term legacy system used in this dissertation refers to a software system for which maintenance actions (e.g., modifications to the source code) are prohibitively costly because *(i)* the component is written in a programming language which has become obsolete as compared to the rest of the technologies

used by the enterprise and/or *(ii)* the component is not well-documented [25].

The behavior of a given system is thus the result of the sequence of its total state that is a set of computation, communication, storing, interaction and physical condition states. The total state can be partitioned into external and internal state. The former is the part of the system state that is perceivable at its interface, i.e., by the users; while, everything else belongs to the internal state. The internal state can also be defined as the combination of the external states of the atomic components of the system.

The delivered service is perceived by the user(s) as the sequence of the external states of the systems. Hence, a service failure turns to be a deviation of at least one external state from the correct state. This deviation is called **error**. The root cause of an error that when activated may lead to the incorrect external state is called **fault**.

Faults have been classified from different perspectives[8]. The most important classes, which need to be introduced to narrow the scope of this work, are: *system boundaries*, *dimension*, *phase*, *objective*.

With respect to system boundaries, a *fault* can be *internal* or *external*. Then, according to the phase of creation or occurrence, a fault can be a *development fault* or a *operational fault*. According the *dimension*, i.e., to where they originate or what they affect, faults are classified into *hardware* and *software faults*. Another important distinction arises from the maliciousness of faults, i.e., the property of being introduced with the precise objective of causing harm to the system. A concrete example of a malicious fault is represented by an external fault introduced by an attacker that, by exploiting the presence of a prior internal fault (also called vulnerability [8]), enables to gain unauthorized access to the

system. Finally, according to its persistence, a fault can be permanent or transient, i.e., bounded in time; of course development faults are always permanent. even if they may be not systematically reproducible, in such case they are called *elusive faults*.

According to the definition of anomalies provided in the Introduction of this dissertation, each error is anomaly but the vice-versa is not valid. The motivations that have led to detail such difference will be clearer in a while, i.e., when the genesis of anomalies will be further discussed. However, first the dependability means are briefly introduced in order to point out the contribution of this work towards dependable mission-critical systems.

Dependability Means

Over the course of the past fifty years many means have been developed to attain dependability. These means can be grouped into four major categories:

- **fault prevention**: the techniques belonging to this class aim to prevent the occurrence or the introduction of faults in the system. Examples are design review, component screening, testing, quality of control methods, formal methods and software engineering methods and best practice in general.

- **fault tolerance**: it consists of using proper techniques to allow the continued delivery of services at an acceptable dependability level, after a fault is activated. Fault tolerance is carried out by error processing and fault treatment: the first aims at removing errors from the computational state, possibly before the occurrence of a failure, while the second aims at preventing faults from being activated again.

- **fault removal**: these techniques aim to reduce the presence (number, seriousness) of faults, and they are obtained by means of a set of techniques used after that the system has been built. They are verification (checking whether the system fulfills its specifications), diagnosis (identify the fault(s) which prevented the verification conditions from being fulfilled), and correction.

- **fault forecasting**: the purpose of the fault forecasting techniques is to estimate the present number, the future incidence and consequences of faults activation. Indeed, no existing fault tolerant technique is capable to avoid a failure scenario, then fault forecasting represents a suitable mean to verify the adequacy of a system design with respect to the requirements given in its specification.

The first two classes of means aim to provide the ability to deliver a service that can be trusted, while fault removal and fault forecasting aim to reach confidence in that ability by justifying that the functional and the dependability & security specifications are adequate and that the system is likely to meet them.

This dissertation addresses methodologies and techniques that attend to the improvement of fault tolerance mechanisms. For this reason, the fault tolerance mechanisms are discussed in detail in the following.

The main phases that are typically involved in fault tolerance strategies [8] can be summarized as follows:

1. **Error Detection**, i.e., the ability to identify that an error occurred in the system. Its goal is to trigger a warning in order that an error does not remain latent and prior that the error leads to a failure for other system components, or for the overall system. It can be performed on-line (i.e., concurrent detection), i.e., while the system delivers the service, or offline (i.e., pre-emptive detection), where the system is checked while the service is not delivered.

2. **System Recovery**, i.e., the operations that lead the system in an error-free and fault-free state. Recovery consists of: *(a) Error Handling*, which aims at removing the error condition, by bringing the system in a correct state. This can be achieved through Rollback, Rollforward and Compensation. *(b) Fault Handling*, which aim is to prevent fault re-activation. This can be accomplished through different stages, i.e., Diagnosis, Isolation, Reconfiguration and Re-initialization. Typically, fault handling may be followed by a corrective maintenance activities to definitely remove the diagnosed faults.

The most basic strategy for implementing fault tolerance is based on error detection and recovery. Upon error detection, *rollback*, *rollforward* and *compensation* may be invoked on demand. With rollback the system is brought back to a saved state that existed prior the occurrence of the error; hence, this require to periodically save the system state, via checkpointing techniques. On the other hand, rollforward brings the system to a new but pre-defined, error-free state. Finally, compensation conceals such errors.

It is worth to note that also proactive techniques can be used during operation to bring the system in an error-free state. For instance, software rejuvenation [26] is a widely used technique that allow to remove, or at least mitigate, software aging effects. It consists in periodically operations, among which components restart, and the reboot of the OS, that restore a clean state of the system, i.e., without aging effects.

An alternative fault tolerance strategy, the *fault masking*, is based on the systematic usage of compensation –i.e., independently on detected errors– by means of redundancy. Redundancy may be temporal or spatial. The former assumes that failures are transient since it consists in the re-executioning the operation that caused the failure. The latter consists in many identical components, i.e., replicas, performing the same task and it is effective under the assumption that the replicas fail independently. When this assumption does not hold an alternative, and by far more expensive, fault tolerance technique is *design diversity* [27]. This consists in different systems that implement the same function via separate designs and implementations.

Fault masking does not necessarily require error detection; however, practical implementations exploit these techniques since if not handled over time errors can lead to a progressive loss of redundancy because of accumulation and propagation phenomena [8].

The measure of effectiveness of any given fault tolerance technique is called coverage. Imperfect coverage, is strictly depended upon the ability to accurately detect and recover from faults activation during system operation. Error detection is thus a crucial aspect in fault tolerance strategies. Indeed, non-detected or non-timely detected errors may hamper the possibility to successfully handle such errors or even to mask them.

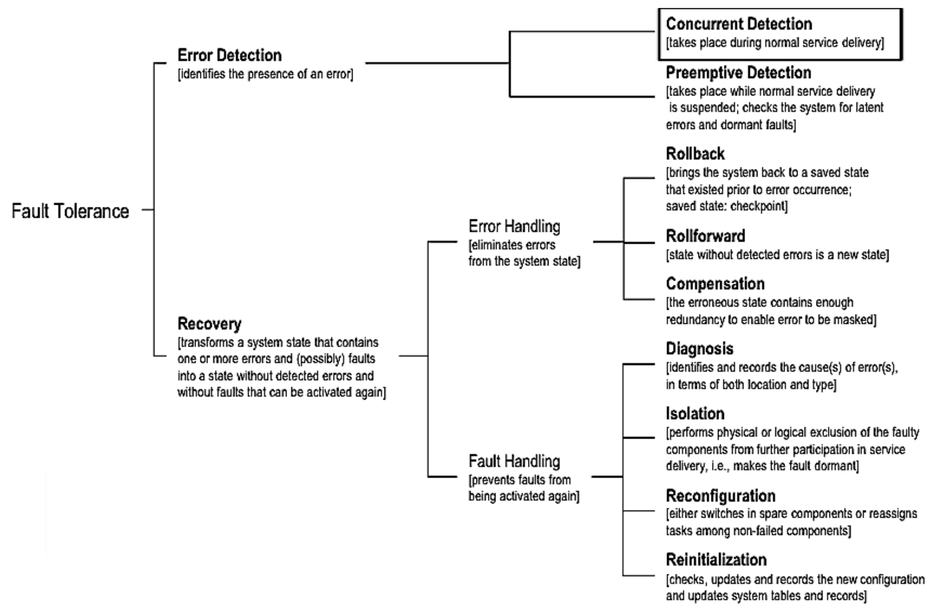


Figure 1.2: Scope of the dissertation

The focus of this dissertation (Figure 1.2) is on the improvement of error detection mechanisms of mission-critical software systems made of OTS components. The purpose is to achieve better fault tolerance coverage, which in turn leads to an overall improvement of the dependability.

1.2 The Genesis of Software Anomalies

Anomaly has been defined as abnormality, irregularity, inconsistency, or variance from expectations by borrowing the term used in the IEEE Standard 1044-2009. The standard also defines a uniform approach for the classification of anomalies regardless of when they

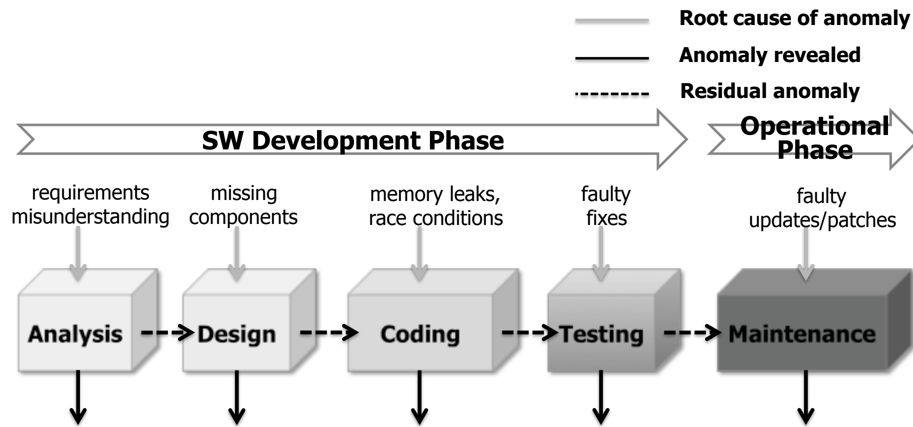


Figure 1.3: Origin and detection of anomalies during development and operational phases

are introduced, i.e., during development phase or during use phase, or when they are revealed (see Figure 1.2). This can be used for defect causal analysis, project management, and software process improvement. This dissertation focuses on the detection of anomalies occurring during the operational phase of the systems.

Although the standard is indisputable useful to classify software anomalies with respect to the whole software development life cycle, it does not provide any characterization of what are the expectations not met when anomalies occur.

In a previous version of the IEEE Standard 1044, published in 1993, it is specified that the expectations derive from documentation (requirements specification, design documents, user documents, standards) or from someone's perceptions or experiences. An attempt was made to classify *(i)* the types of symptoms that can be used to reveal anomalies, and also *(ii)* the type anomalies based on the semantic of their root cause location, e.g., flow control logic, computational code, data.

This dissertation shares with the IEEE Standard 1044 both the definition of anomaly and expectations, but the aforementioned classification of symptom and anomaly types is misleading. In fact, the term anomaly was confounded with the term fault. Anomalies may be of course caused, but are not limited to, the activation of software faults. However, this classification was either too general, e.g., to encompass anomaly leading to performance issues, or too narrowed and applicable only to specific software systems (e.g., the OS for the interruption handling type).

The pathology of software anomalies, i.e., the conditions that lead to deviations from expectations, is far from being simple. A normality region that encompasses every possible expected behavior is recognized as an intricate task [2]. Furthermore, the boundaries of the normality region are often not precise. Thus, some observations that, according to actual expectations, results in anomalies may really belong to the normality region, and vice-versa.

Different perspectives of analysis of software anomalies are introduced in the following to clarify the concept of deviations from expectations and to identify which kinds of "traps" are needed to capture anomalies.

The Fault-Error-Failure propagation chain

Figure 1.4 is introduced to characterize the relationship between the failure-behavior of the systems and anomalies. The figure shows that, with respect to the system, the **faults** can be **internal** or **external**. In the former case these are harmless –and not leading to anomalies– till the *triggers*, i.e. conditions that lead to fault activation, do not occur. In both cases, the fault turns into error and becomes potentially danger. The term potential is

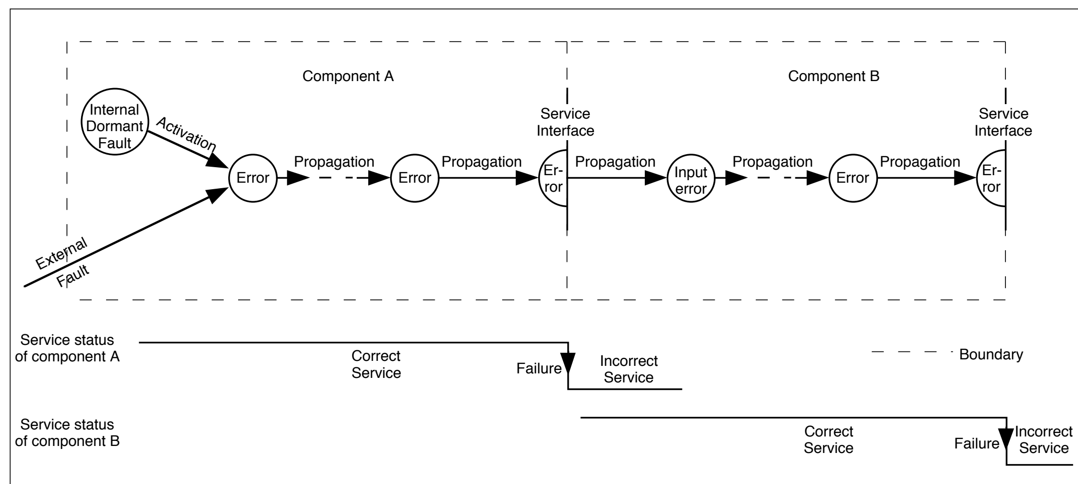


Figure 1.4: The pathology of failures

used since, as shown in Figure 1.4, the error needs to be propagated to the system interface to cause a failure, i.e., incorrect service. An error is for definition an anomaly since the state of the system deviates from the correct one. A first difficult with this definition arises because which all possible states of the systems cannot usually be included in the so called expectations.

Figure 1.4 also highlights that, with respect to the components of the system, error propagation is of two types. *Internal error propagation* occurs if the errors are confined into the component. *External error propagation* occurs when errors are propagated among at least two dependent components (component B receives the service from component A in Figure 1.4). Different types of anomalies may occur in case of internal or external propagation, which in turn may lead to different information that can be used to reveal anomalies.

However, Figure 1.4 does not show the occurrence of even more complex activation and propagation phenomena that have been ascertained in recent studies [28, 29] and that lead to different anomalies – which may of course occur in mission-critical software systems made of many interacting and interdependent OTS- and legacy-based components. For instance, the complex activation/propagation consists in the influence of indirect factors, such as the state of internal components (e.g., hardware and operating system), on failure occurrence. Concurrency faults, for instance, are becoming increasingly important with the prevalence of multi-core architectures. These are caused by wrong management of shared resources access and lead to notorious non-deterministic failures, among which atomicity violation, data race, deadlock, that are very difficult to reproduce [30] and only manifest when some specific combination of conditions occurs. Moreover, the repeated activation of the same fault may lead to errors accumulated overtime, which then lead to a failure (e.g., memory leak activation may wastes all available memory). This last situation is typical of software aging phenomena in which a huge time interval passes between the fault activation and the occurrence of a failure.

The aspects of fault activation and failure-reproducibility and their relation with anomaly detection will be further discussed in the following sections addressing the ODC methodology [31] and the Borh-Mandel bug theory [17] will be introduced.

ODC Concept

Orthogonal Defect Classification (ODC) is a method, first presented in 1992 [31], that allows the quantitative analyses of the software development process with practical insights to developers team. ODC tries to gather the benefits of defects/process analysis techniques (e.g., root cause analysis) and of quantitative measures (e.g., reliability growth analysis).

Although ODC is more focused on the development process and not on the fault classification, the ODC system captures some important attributes, i.e., *defect type* and *defect triggers*, that are useful to understand the sources of most of anomalies and the triggering conditions [1].

The *defect type* captures the meaning of the fix, expressed in a value set describing design or programming terms. The *defect trigger* maps into the test process and expresses conditions leading to a defect to surface. It is worth to underline that these definitions have been designed for software development process improvement; however, they still hold for use phase. Indeed, for instance, certain triggers represent a complex operation when this is experienced in the field [6]. The different fault types have been defined by looking at the type of fixes a programmer can make. Some of these classes can be useful to establish which kind of indicators need to be monitored to reveal anomalies. The following basic fault types, defined in [31] and summarized in Table 1.5, should be taken into account:

- **Function** - The fault affects significant capability, end-user interfaces, interface with

[1] It is worth to note that in this dissertation the terms fault and defect are interchangeable; however, in the classification provided by IEEE Standard 1044-2009 a defect is a fault if it is encountered during software execution (thus causing a failure).

Table 1.1: Some examples of ODC defect types

Defect type	**Examples**
Function	missing implementation of functionality
Interface	missing/wrong return value; missing/wrong parameter in function call.
Assignment	missing/wrong initialization of variable; missing/wrong parts of logical expression.
Checking	missing if statement; missing/wrong and/or condition or data in conditional statements.
Timing/Serialization	missing/wrong resource serialized; wrong serialization technique used.
Algorithm	wrong branch/flow construct; missing "if (cond) else statement(s)" around statement(s) requesting a design change.
Build/package/merge	problems due to library systems, management of changes, or version control.

hardware architecture or global data structures and should require a formal design change. Usually these faults affect a considerable amount of code and refer to capabilities either implemented incorrectly or not implemented at all.

- **Interface** - This fault type corresponds to errors in interacting with other components, modules or device drivers, via services, macros, call statements, control blocks or parameters list.

- **Assignment** - The fault involves a few lines of code, such as the initialization of control blocks or data structures.

- **Checking** - This fault addresses program logic that has failed to properly validate

data and values before they are used. Examples are incorrect validation of parameters or data in conditional statements.

- **Timing/Serialization** - The management of shared resources access (e.g., wrong resources serialized) is not done properly. Examples are deadlocks or missed deadline in hard real time systems.

- **Algorithm** - This fault includes efficiency and correctness problems that affect the task and can be fixed by (re)implementing an algorithm or local data structure without the need for requesting a design change.

- **Build/package/merge** - Describe faults that occur because of library systems, management of changes, or version control.

In a further extension of ODC fault types, made by Duraes et al. [21], this classification has been further refined by taking into account other factors, such as those related to language programming constructs being used. Indeed, different fixes can occur for the same fault. With such modifications each fault type is also distinguished in the subtypes missing, wrong and extraneous. For instance, it is possible to have Missing Function Calls (MFC) type. This is a particular kind of algorithm fault in which a required function call is missing.

Developers may come to this detailed classification by adopting the ODC as a first step, and then, in a second step, by grouping faults according to their nature, defined from the mentioned "programming constructs" perspective. Finally, in the third and last step, faults are further refined and classified in specific types.

Table 1.3: Proportion of ODC defect types in field studies

ODC Fault Type	**Distribution (min-max)**
Checking/Assignment	38-56%
Interface	2-10%
Algorithm	20-43%
Function	2-12%
Timing/Serialization	6-14%
Build/Package/Merge	2-4%

Table 1.3 shows the minimum and maximum (rounded) distribution of defect types in different field studies [21, 32, 31]. These studies have been conducted on mature software such as IBM operating systems and databases [31, 32] but also on open source software of different size [21] (small such as vim to larger such as the Linux kernel). These results suggest that a share of faults, i.e., due to function, algorithm or assignment, is very difficult to reveal using black-box approaches since a deeper knowledge of the component at hand is required. However, anomalies due to bad shared resources management, library systems, checking and interactions with other components can be revealed by monitoring the usage patterns of the component, e.g., by means of kernel-level tracing facilities. In Chapter 3 the OS-level indicators exploited to monitor such behaviors will be discussed.

Table 1.5: Some examples of ODC defect triggers

Trigger type	**Examples**
Boundary Conditions / Workload	low memory, heavy network traffic, many users
Concurrency	simultaneous use of the same resources
Timing	particular sequence of operations performed
Logic Flow	the delivered service is inconsistent or incorrect
Recovery	activation of standby spare, backup procedure
Exception Handling	execution of a “catch” clause

A trigger as previously discussed is the condition that allows a fault to be activated. Particularly of interest for this work are the following trigger categories.

- **Concurrency** - it applies when shared resources are accessed simultaneously. It has implications of security, locking mechanisms, and performance.
- **Logic Flow** - it relates to those situation when the operational semantics are in question.
- **Side Effects** - it applies to faults that are characterized by seemingly unrelated conditions, often difficult to diagnose.
- **Recovery** - the fault is activated when the system is recovering from a previous failure.
- **Exception Handling** - the fault is activated after an unforeseen exception-handling path is executed.
- **Timing** - the fault is activated when particular timing conditions occur.
- **Workload** - the fault is activated only when a particular workload condition occur.
- **Normal Mode** - is a trigger which says, in effect, that no special conditions need to exist in order for the fault to be activated. Of course, a real trigger exists, maybe associated with earlier operations.

We argue that these triggering conditions can be valuable guidelines to design black-box anomaly detection approaches. In particular, we believe that these can be useful in limiting false positives (see Section 2.3). Indeed, when the knowledge on the monitored component is limited and the boundaries of the normality region are not precise, it is very common

to trigger false positives when in effect, no anomaly occurs. By monitoring most common defect triggering specified by ODC and the indicators related to specific defect type, the accuracy of the anomaly detector could be higher. For instance, let assume the workload trigger and the indicators related to timing/serialization are monitored. The former are for instance the number of concurrent requests, the amount of byte write/read, and theÊlatter the time to acquire/release locks. It is more likely that an anomaly caused by bad synchronization occur when the combination of two events occur: the number of concurrent requests is over the average and the time to release locks increases.

Bohr-Mandelbug Theory

The ODC classifies software faults according to the semantic of their fix. However, in [8] it has been recognized that the faults can be elusive, i.e., the failure are not systematically reproducible. A crucial concern for software engineers is thus to understand what are the conditions that makes failures reproducible. The reproducibility problem was first discussed in 1986 by Jim Gray [33]. In that work, Gray first distinguished faults whose activation is easily to achieve, i.e., solid or hard faults, from faults whose activation conditions are not systematically reproducible, i.e, elusive or soft faults. Thus, solid faults manifest consistently under a well-defined set of conditions and that can easily be isolated, since their activations and error propagations lack complexity.

Gray's paper used the term **Bohrbug** for solid faults. This is because their activation conditions recall the rather simple atomic model of the physicist Niels Bohr: *Bohrbugs, like*

the Bohr atom, are solid, easily detected by standard techniques, and hence boring [33]. Using again ODC concepts, these include the easy fixes (e.g., deleting an extraneous checking), but also more complex fixes (e.g., algorithm change).

As for **Mandelbugs**, there is no common and widely accepted definition –and, it will be probably true for a long time. In this dissertation, by using ODC concepts, we provide the following definition: *software faults whose activation conditions are intricate enough that they can only be met when a complex combination of triggers occur.* According to Grottke et al. [28], the "complexity" may arise from *(i)* a time interval between the fault activation and the occurrence of a failure, and/or *(ii)* the influence of one or more of the following indirect factors: interactions with the operating environment (hardware, operating system, other components); the influence of the timing/sequence of inputs and operations (relative to each other, or in terms of the system runtime or calendar time).

According to this view, Mandelbugs also include those faults which activation is non-deterministic. This kind of faults were initially named **Heisenbugs** (conied by Lindsay when working with Gray [28]) referring to the *uncertainty principle* investigated by the physicist Werner Heisenberg.

Heisenbugs were envisioned as *bugs in which clearly the system behaviour is incorrect, and when you try to look to see why it's incorrect, the problem goes away.* The term recalls the uncertainty principle since the measurement process (in this case the fault probing) alters the observed phenomenon. One typical example is a fault that only occurs during testing but not when diagnosed under debug-mode. The common reason for this non-deterministic behaviour is that executing a program in debug mode often cleans memory

before the program starts, and forces variables onto stack locations, instead of keeping them in registers. Indeed, many Heisenbugs are caused by uninitialized variables and when variables are non initialized the activation of the fault depends on previous state of the memory.

For this reasons, Heisenbugs are very difficult to reproduce, due to their non-deterministic activation, and are considered as special case of Mandelbugs. Indeed, the fault activation is influenced by the system-internal environment.

Another, subclass of Mandelbugs that devotes further explanations is the one of **aging-related bugs** that are the underlying cause of software aging phenomena [17]. These are faults whose activation and/or error propagation is influenced by the total time the system has been running and the type and amount of workload served. Their repeated activation causes the accumulation of internal error states (hence, not propagated to the interface) that leads to increasing failure rate and/or degraded performance, and eventually to system crash or hang. As a concrete example of aging-related bug let consider a memory-leak in program's heap area, which is memory allocated and never released. Under repeated memory-leak activation over time, free memory will be no more available.

Figure 1.5 taken from [6] shows the cumulative growth of Bohr-Mandelbugs against time. Even if the percentage of Mandelbugs is less than half of the percentage of Bohrbugs it is in any case increasing. This is worrisome since due to their different nature, detecting anomalies due to Bohr- or Mandel bugs activation (and also the recovery procedure) requires that distinct techniques are employed [34]. For instance, Figure 1.6 show that different fault tolerance strategies need to be adopted when dealing with Bohr-Mandelbugs. When dealing

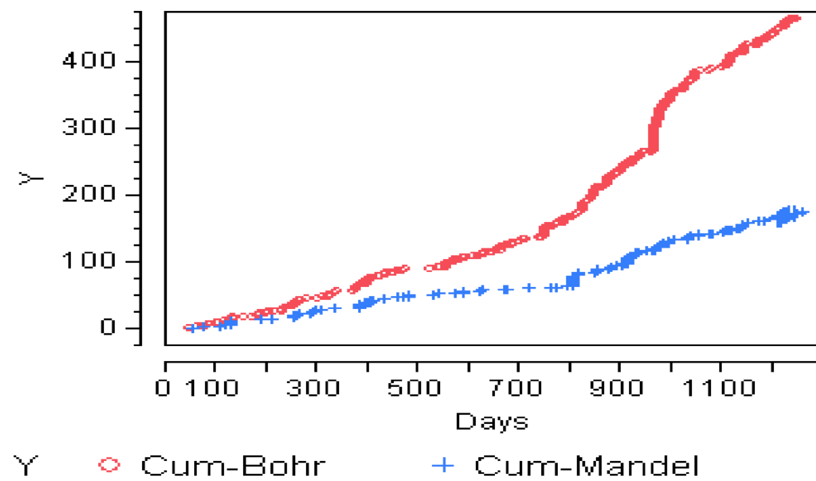


Figure 1.5: Proportions of Bohr-Mandelbugs across a timeline

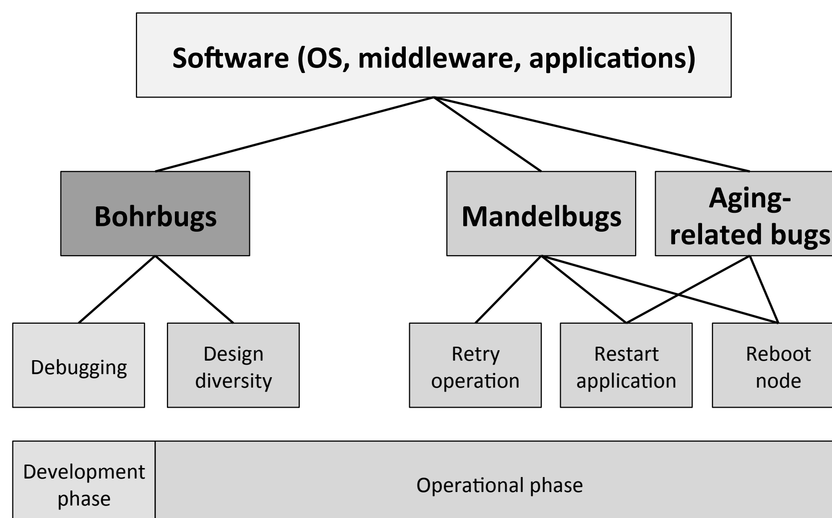


Figure 1.6: Different fault tolerance strategies that can be used with Bohr-Mandelbugs

with the detection of software aging related anomalies it is required that statistical trend analysis techniques are employed.

1.3 The Anomaly Detection Problem

It is widely accepted that the anomaly detection problem, in its most general form, is not easy to solve –indeed, most of existing anomaly detection techniques solve a specific formulation of the problem [2].

This dissertation addresses anomalies that need to be detected on-line and that originate from software faults created during development but that are activated during the *use phase*, i.e., when the system is released and delivering its service. Errors due such fault activations are both transient, i.e., with a presence that is bounded in time, or permanent, i.e., continuous overtime. Furthermore, this work focuses on anomalies due to non-malicious faults activation. Indeed, even if the proposed anomaly detection approach is general can be applied to detect different classes of anomalies, among which the one caused by vulnerability exploitation, the experimental evaluation of the framework has been conducted by means of a fault injection tool that do not allow to emulate malicious activities; thus, the framework effectiveness against malicious faults has not been assessed.

If all the correct states of the system were enumerable, a straightforward way to detect anomalies would be to compare current system state with such well-known "normal states". Of course, such information is not available, but even if it were, the number of possible system states (combination of communication, computation, storing, interaction and physical condition states for each component) would be so big to make unfeasible to solve the search

problem.

To make the detection of incorrect system behavior feasible, it is typically performed by *(i)* monitoring the service delivered by the internal components of the systems at the component interface level, so only their external states, or *(ii)* by checking the warning/errors messaging it is able to store into the event logs –and these situations are especially true when dealing with OTS-based components. In the former option, the components are treated as black-boxes since nothing is known about their internal states and thus, detection is possible only when an error reaches the component interface. The latter relies on the component internal built-in detection mechanism that signal anomalous conditions. Of course, mixed approaches are also possible.

However, the available information to reveal anomalies is often limited (e.g., no failure modes and effect analysis performed), imprecise (e.g., inconsistent data in logs) and not always well-suited for to automatic on-line analysis. These and other challenges to the on-line anomaly detection are discussed in the following section.

1.4 Anomaly Detection Challenges in Mission-Critical Software Systems

Thirty years ago the predominant cause of failures in mission-critical systems was the hardware components. On the contrary, most of the runtime anomalies occurring in modern computing industry systems are due to software [5]. This is true also for mission-critical systems. For instance, some studies of anomalies of NASA space missions Voyager and

Galileo, which are collected in an institutional reporting system, has revealed that anomalies of the most recent mission, i.e., Galileo that was launched in 1989, are mostly due to software and are fixed by changing in-flight or ground-software systems [35, 36]. This imply that the monitoring, the detection and the mitigation means need to be more sensitive to software anomalies than before.

As discussed previously, the Borhbug vs Mandelbug theory classifies software faults based on their intrinsics difficult to reproduce the failure (i.e., independently from the skills of the reproducer). In fact, different indicators need to be collected and different detection mechanisms are required to timely reveal anomalies depending on the nature of software fault. For instance, the difficult of reproducing aging-related bugs is due to the complex error propagation phenomena that lead to component(s) or even OS failures. The problem in revealing software aging-related anomalies rely on the detection of an aging trend, namely the occurrence of progressive resource and performance degradation not due to load peaks or other transitory phenomena. However, software aging trends are intricate to detect. In fact, in [16] we have shown that according to the applied workload and to the type of aging-related bug, software aging can lead to quite smoothly and linear trends or to extremely non-linear performance degradation.

Another important aspect of modern mission-critical systems is that they increasingly rely on the integration of OTS and third party components, such as OS, communication middleware and database. This is indispensable to reduce time to market but leads to systems that show unexpected behaviors due to unforeseen components interdependency [5]. These anomalies often arise in the production environments under specific conditions

due to the occurrence of special states of hardware/software and/or workload (e.g., load peak, processes schedule, allocated memory over a threshold). Moreover, OTS components often do not have a specification, and when this is present, it is not precise and complete, as the hardware counterpart [5]. So it is very difficult to have a clear distinction between failing and non-failing behaviors of the components.

The quality of information sources is often not adeguate. For instance, logs, which represent one of the most common source of information to analyze and to diagnose systems behavior [37, 38, 39], have been discovered to be inadequate for the assessment of system reliability [40]. This is basically due to the lack of a systematic approach for the production of logs that is currently dependent on skills and competencies of developers [38].

Finally, the operating environments may be variable and not stationary. Sources of variability are *(i)* the alternation between periods with heavy load, which may imply high demand of cpu, memory and network bandwidth, and periods having low resource usage, *(ii)* the occurrence of background activities (e.g., logs rotations, backup procedures, clock synchronization), *(iii)* the reconfiguration of components and communication elements, *(iv)* the recovery procedures. In such cases, the monitoring infrastructure should "adapt" its internal logic to the changing situation. For instance, detectors that typically use worst case bounds to reveal anomalies have little efficacy in case of structural changes to the operating environments, such as after the reconfiguration of the network. The recent cascade failures of the Amazon Elastic Block Storage have been ascertained to be in part caused by timeouts that were too conservative for timely detecting the failing components [41].

Chapter 2

Monitoring and Detection: Approaches and Frameworks

Monitoring computer-based systems consists in collecting information about their activities, with the aim of verifying (i.e., detecting) if these are compliant with the expected or specified behaviors. This thesis focuses on approaches and techniques for on-line monitoring, i.e., the output of the verification activity is meant to be produced at runtime. On-line monitoring is crucial for timely activate mitigation means in face of anomalies that may lead to irremediable or even catastrophic failures. This chapter surveys monitoring and detection approaches and frameworks proposed in the field of dependable computing. A classification of monitoring and detection approaches and techniques is provided with the purpose of better identifying and analyzing the suitability and/or the limitations of actual solutions. Finally, the chapter is concluded by discussing the set of metrics that allow to characterize the performance of on-line anomaly detectors.

2.1 Background

The academic work on on-line monitoring and detection of software anomalies is surely endless. In fact, since computers have started to replace human or mechanical devices the monitoring and detection problems have arisen.

A typical infrastructure for monitoring computer systems, which is depicted in Figure 2.1, is composed of the following logical layers: *(i) sensing*, *(ii) transforming*, *(iii) communication* and *(iv) detection*. The sensing layer separates the concrete elements, hereafter *probes*, that gather relevant system information from the rest of the infrastructure. The

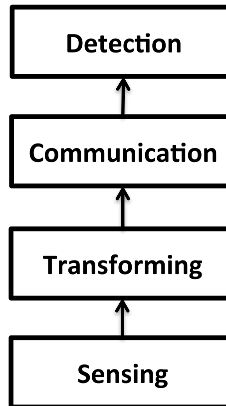


Figure 2.1: A general high-level overview of monitoring and detection infrastructures

probes collect at predefined time, on-demand, or when an event occurs data of interest about system behavior. The probes can be realized in hardware or software and with respect to the system, these can be placed internally or externally. In the former case, the monitored system needs to be modified accordingly.

The transforming layer is in charge to prepare the collected data in order to be best processed in the upper layer. The operations typically performed are: changing format, filtering, correction and aggregation. The result of these operations are called hereafter *processed data*.

The communication layer consists in protocols and communication means used to deliver data to the detection layer. If the detection logic is placed in a different node with respect to the sensing and transforming modules, this layer needs to guarantee that the monitored information is correctly delivered to the detection module, i.e., the monitored processed data are delivered timely, in order and unmodified. Usually, a dedicated network can be

used to send the monitored data by limiting the impact on the bandwidth used by the applications.

The detection layer encompasses the logic needed to reveal anomalies. For large scale distributed systems the detector cannot consist in a central and unique point, but the detection logic is usually spread across several modules, often called agents. Each agent can have only a partial view of the system state. Hence, the decision about the health of the system as whole, can only be obtained by aggregating the views of agents. Usually, an agent, which monitors only a part of the system, can also request some information to other agents to make the decision about the part of the system state whose is responsible.

It is worth to note that depending on the target system, the separation between these layers could not be perfectly clear.

In this chapter the focus has been devoted to the analysis of the first and the last layer, i.e., the sensing and detection, by discussing in detail the monitoring and detection approaches and frameworks proposed in the field of dependable computing.

The type of target system, its dependability requirements and the available knowledge about the behavior of system and its components determine the constraints that engineers need to take into account when architecting the monitoring and detection infrastructure. These constraints impact on:

- the *detection performance*, i.e., the requirements of the verification activity, such as timeliness, coverage and accuracy, that also drives the design of other layers. More

details about the metrics commonly used to evaluate detection performance are discussed in Section 2.3.

- the *overhead* of the monitoring infrastructure, i.e., the extra work imposed to the target system because of the integration of the probes, the transforming, the communication and the detector elements.

- the *intrusiveness* of the sensing elements, i.e., the degree of modifications that can be performed to the components and the operating environment to add the monitoring probes.

- the *type of sensing*, i.e., the possibility to use *hardware*, *software* or *hybrid* probes to collect data. In this dissertation only software probes are discussed.

Furthermore, beside the aforementioned aspects, to architect efficient and effective monitoring and detection strategies engineers also need to take into account the issues discussed in Section 1.4. In particular, the characteristics that the framework should have are: *(i)* be suited for OTS-based systems, *(ii)* adapt to changing operating environments, *(iii)* be of general use, i.e., useful to reveal different kinds of anomalies.

The literature review has been performed by taking into account the reference monitoring and detection architecture (see Figure 2.1), the discussed monitoring issues and, in particular, different perspectives of analysis, which are summarized in Figure 2.2 and discussed as follows:

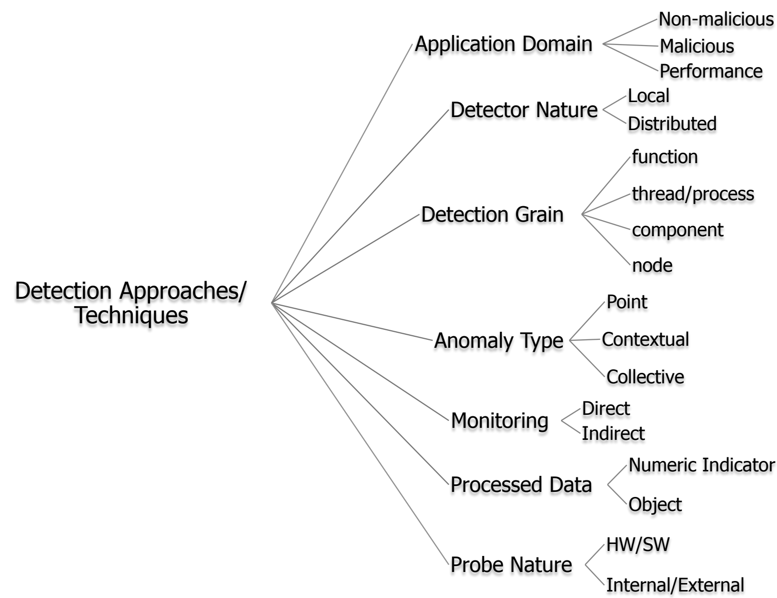


Figure 2.2: The perspectives of analysis for the monitoring and detection approaches

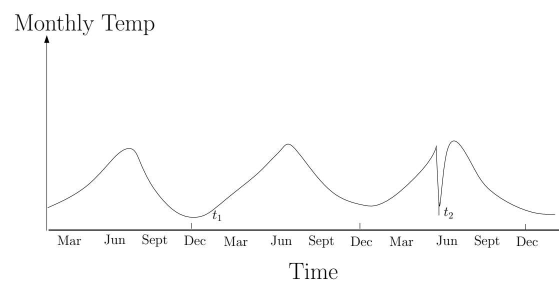


Figure 2.3: An example of contextual anomaly

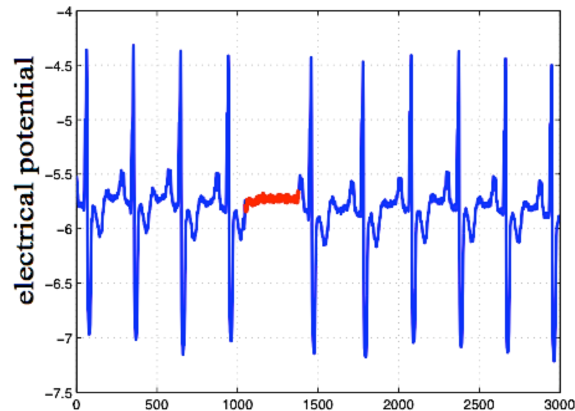


Figure 2.4: An example of collective anomaly in electrocardiogram (medical domain)

- ***application domain***, i.e., the scope in which the detector has been used. These are classified into: *non-malicious* activities (e.g., due to infinite loops, memory bloating, crash), *malicious* activities (e.g., intrusions, worms, viruses) and *performance* issues (e.g., overload, bottleneck identification, which are not caused by fault activation or malicious attacks).

- ***detector nature***, i.e., the way the decision process is made. It can be *local* if the decision about the processed data is independent of other detectors (the hexagons in white boxes in Figure 2.5), i.e., the agents; or *distributed* if the decision is dependent upon the data processed in more than one node of the system and on the decisions of other agents (the hexagons in the grey boxes in Figure 2.5).

- ***detection grain***, i.e., the granularity of the revealed anomaly. Usually, the finer the grain, the greater the overhead of the monitoring and detection framework. The following levels can be generally found for on-line detection: *function*, *thread*, *process*,

component, *node*.

- ***anomaly nature***, according to [2] there are three kinds of anomalies: *point anomaly*, which is a processed data that is always considered anomalous independently of the rest the information; *contextual anomaly*, i.e., processed data that can be judged to be anomalous only in a specific context (e.g., spacial or time), which may depend on the application domain (see Figure 2.3 taken from [2]); *collective anomaly*, namely a collection of data in a given context (e.g., in a ordered sequence), which can only be considered anomalous with respect to the whole data set (see Figure 2.4 taken from [2]).

- ***processed data***, the type of data that are managed by the detector to reveal an anomaly. These can be generally divided into the following classes: *numeric indicator*, hereafter simply indicator, for which exists a mapping from the monitored entity to a measurement value; *object*, which has a particular structure and properties (e.g., the vertexes of a graph, the event logs, the packets sent/received). The numeric indicators can be further classified according to the *scale*, i.e., the type of mapping, used to obtain the measure. The most common scale types are the following [42]: *categorical*, *ordinal*, *interval*, *ratio*, *absolute*. An indicator with categorical scale takes values from a set of exclusive, unordered values (e.g., male/female). An indicator with ordinal scale takes a value from a set of exclusive, ordered values (e.g., low/medium/high) such that the difference between any two values is undefined but the relative difference is meaningful. An indicator with interval scale takes values for which differences can be

computed. However, the values start from an arbitrary point (i.e., there is no notion of a zero value, such as in temperature measured in Fahrenheit or Celsius). If there exists a meaningful zero value and the ratio between two indicators is meaningful, then the ratio scale can be used (e.g., the Kelvin temperature scale). The absolute scale is used when the value itself is the only meaningful transformation (e.g., a counter). The proposed detector focuses on indicators having a ratio scale and absolute scales.

- ***monitoring approach***, i.e., the way the processed data are obtained: *directly*, when these are derived straight from the monitored components, or *indirectly*, if they are derived from intermediate data.

- ***probe nature***, the way the sensing elements are implemented, i.e., *hardware*, *software* or *hybrid*. A further classification for probes can be made by looking at their placement with the respect to the target system. Thus the probe can be *internal* or *external*.

2.2 Approaches and Frameworks

Detection approaches and techniques have been proposed in different research areas among which statistics, machine learning, data mining and information theory. The suitability of the approaches and the techniques that can be exploited to reveal anomalies depends on the application domain and, of course, on the nature of the anomalies and the processed data. For instance, in case of statistical detection approaches, different techniques need to be used for ordinal or ratio numerical indicators.

Complex software systems such as databases, application servers, web servers and data

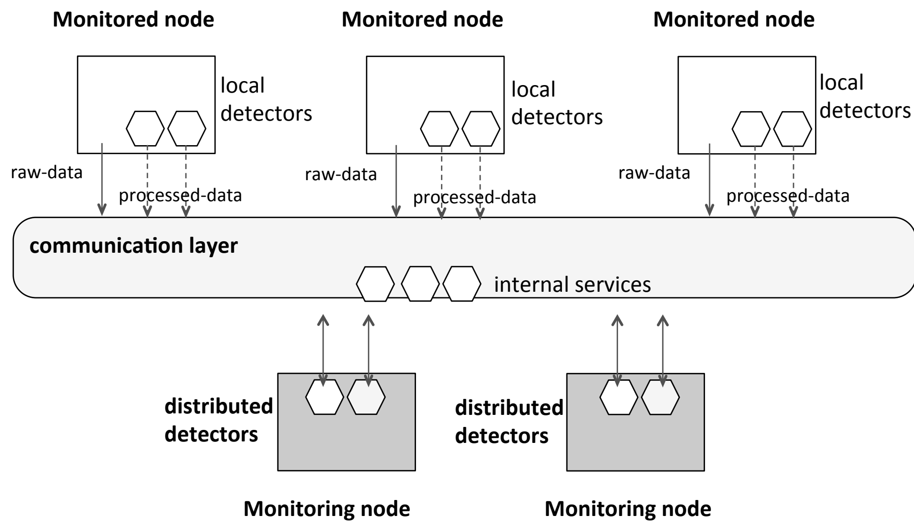


Figure 2.5: A high-level overview of distributed detectors

distribution middleware are enhanced with monitoring tools that can be used independently, without the need for any pre-existing infrastructure. Since they exploit the knowledge of the internals of the components such tools feature advanced detection facilities at lowest grain possible. However, despite their potential, the manual configuration and tuning of such built-in detector have an adverse effect on their real application and exploitation. Indeed, since they lack a general management infrastructure, often system administrators prefer general purpose monitoring framework, such as the IBM Tivoli Monitoring [43].

In the following sections the scientific work in the field of dependable and secure computing addressing anomaly detection problem and proposing general-purpose monitoring framework is reviewed. In particular, the conducted analysis is divided in two sections. *monitoring* and *detection*. In the former, the literature is analyzed from the perspective of the monitoring approach (i.e., directly or indirectly), the probe nature, and the process

data, without addressing the detection process. In particular, the architecture of the most interesting monitoring frameworks are also described.

In the second section, the work is surveyed from the detection perspectives, i.e., the application domain, the detector nature, the detection grain and the anomaly type. It is worth to note that some studies also address the diagnosis problem, namely, other than monitoring and detection issues, the identification of the faulty component(s) and the type of activated fault is also investigated. Indeed, a timely detection is crucial for efficient diagnosis (and recovery) procedures [34].

It is worth to recall that our target applications are mission-critical systems consisting in many interconnected nodes in which OTS components are deployed. The proposed monitoring and detection framework, which is described in Chapter 3, detects anomalies at single node by means of indirect monitoring approach and software probes placed at OS-level. For this reason the literature review is more focused on local rather than global detectors with particular interest on indirect monitoring approaches.

2.2.1 Monitoring

Many detection techniques exploit *direct monitoring* approaches to collect useful system health information. For instance, a component may be queried by means of external probes [44, 45] or it can be internally prepared to periodically send heartbeats [46, 47] or to store relevant event logs [48, 49] when some conditions occur.

Log-based mechanisms typically exploit errors and warning in the logs for classical post-portem analysis [37, 38, 39, 48, 50], i.e., off-line analysis with the aim of diagnosing the root

cause of system failures. This is especially true when dealing with large, complex systems, consisting of heterogeneous software components for which logs are often the only source of information about the health status of the monitored system [51]. Several studies also exploit event logs for on-line anomaly detection activities [52, 49, 53]. For instance, in [49] logs are collected to detect anomalies that lead to system failures, such as crashes; while, in [52] these are used along with system activity reports to predict the performance of a clustered system and take proactive management actions. In [53] event logs are correlated by means of a bayesian network approach in order to enhance intrusion detection systems (IDS) and reveal compromised users.

However, several studies have highlighted the inadequacy of the logs for the assessment of reliability. Hence, the suitability for on-line anomaly detection is also doubtful. The first issue of current logs are related to their heterogeneity and impreciseness [50]; indeed, they may provide ambiguous information [38] and often accurate filtering techniques are needed. As discussed in Section 1.4, this arises from the lack of a systematic approach for the production of logs that is currently dependent on skills and competencies of developers [54]. Crucial decisions regarding the production and collection of logs are taken only in the latter stages of the life cycle of the software (e.g., during the development of the code). For these reasons, it is reasonable to state that the current logging systems are not designed to fully support automated on-line anomaly detection.

Other direct probing approaches, especially studied in the work related to fault tolerant distributed systems, are based external probes, also called *pull-based*; while, the approaches relying on heartbeats are also known as *push-based* monitoring [46]. The differences between

these two approaches are shown in 2.6

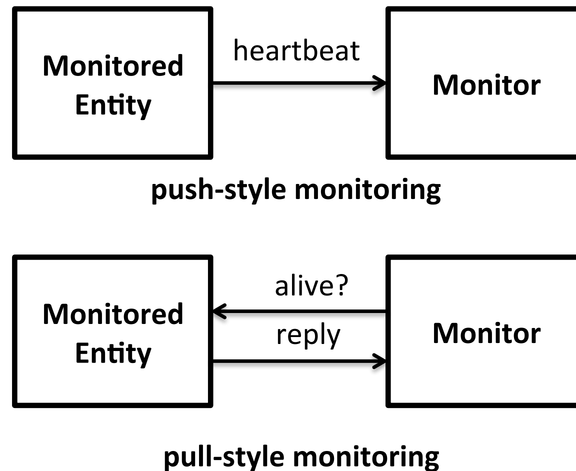


Figure 2.6: Push- vs pull-style monitoring approach

Direct external probes, i.e., the pull-based approach, are widely used for anomaly detection. This, indeed, can be performed by appropriately selecting the probes and analyzing the obtained results. In [55], for instance, the detection of software aging anomalies in application servers is accomplished by means of external probes. The probes request an amount of load that exceed the server capacity and the throughput of the application is measured. Machine learning techniques such as *Naive Bayes*, *J48* and *Support Vector Machine* are used to detect the failed responses under heavy workload that are due aging problems. This probing approach has a non-negligeble overhead because of the excessive load requested, which, above all, could potentially lead to robustness problems.

In [56] different implementations of unreliable detectors are compared by using both external and internal direct probing. In [46] a modular detector based on heartbeat and

on the on-line prediction of the timeout is proposed and experimentally evaluated. In [57] the heartbeat detection mechanism is integrated into the OS, i.e., Minix, to reveal the failures of unreliable drivers, i.e., those OS components that are expected to fail more frequently. However, the choice of the timeout is crucial since short values could lead to an excessive amount of false alarms, which can trigger unnecessary recovery actions. For instance, in Minix the reincarnation server, which supervises the drivers, can kill and recreate the processes that implies unnecessary delay of driver operations.

The push/pull monitoring approaches are also envisioned by the Fault Tolerant CORBA specification [58] that defines an architecture and a framework for resilient, highly-available, distributed software systems. This infrastructure provides means for transparent replication and recovery. An object can be monitored by implementing the *is_alive*() function, in the pull-style mode, or by sending periodically "alive" reports onto the Fault Notifier channels, in the push-style counterpart.

Designing monitoring approach based on push/pull style depends on the system requirements and are usually well-suited when the status of components has to be known only in some predefined periods. In the case of push-style monitoring the same detection performance can be obtained by the half of messages exchanged. Hence, push-style detectors are generally preferred in mission-critical systems that need to be monitored for the entire system lifetime [46].

However, the problem with external and internal direct probing lies behind their design. In particular, the placing of the probe stations, the monitored interval, the probe points, and the specific probe implementation have to be carefully optimized for the target system.

This task is often accomplished in an ad-hoc manner and it is based on the knowledge of the system, on engineers experience and on rules of-thumb. Probes are usually selected off-line and run periodically using a fixed schedule, which depends on the detection latency requirements. For detection and diagnosis purposes, this approach may be quite inefficient since it needs to run repeatedly an unnecessarily large set of probes capable of revealing all possible problems, many of which might in fact never occur [45]. Moreover, existing direct probing approaches typically assume a static model of the system, i.e., the system state does not change during the detection process. This contrasts with the nature of software failures that, as discussed in Section 1, may be also transient (e.g., concurrency bugs, performance degradation).

The optimization of the probe set selection has been addressed in [44, 59] where the authors propose some approaches based on simple heuristic search that yield close-to-optimal solutions. In [45] an adaptive incremental probing scheme is proposed.

When the monitored components are black-boxes designed without heartbeat mechanisms and cannot be modified, an alternative to the pull-based scheme is the **indirect monitoring approach**. This consists in obtaining the information without directly relying on the monitored components. Indeed, useful information can be collected at different probe-points of the system, such as the network, the virtual machine and the operating system, by inserting internal or external probes without the components-awareness. Commonly used techniques use network sniffing, hardware counters, OS-level tracing mechanisms, and virtualization monitoring facilities. In Table 2.1 some examples of indirect monitoring are

described.

Table 2.1: Most common indirect monitoring approaches

Probe Level	**Examples of monitored entities**	**Examples of monitored collected information**
Network	LDAP servers, Web server, Database, Application Server	#Packet, Avg packet size, #Errors, Source/Destination addresses
HW	Machine (pipeline, bus, cache), Virtualized Env., Applications	TLB misses, stall cycles, hang, memory access latency
OS (user/kernel)	Process and Threads	CPU usage, memory, disk/network IO syscall, scheduling events, #Threads
JVM / .NET	Java and.NET Applications	#thread, memory, classes gc statistcs, cpu time

Several frameworks detect anomalies by sniffing packets exchanged between the components on the network. In [60, 61] the authors collect packages to deduce the global behavior of networked systems and to detect faulty entities. In [62] wavelet analysis on the packet traces are used to reveal network performance anomalies. In [63] packet size and timing are used to detect malicious stepping stones, i.e., non authorized opened connections among machines.

Other detectors use OS tracing facilities to collect the low-level information such as syscall, interrupts, scheduling times and IPC. One of first studies in this direction has addressed the intrusion detection problem [64]. The authors propose to detect the intrusions by identifying anomalous system call traces of privileged processes. This approach has been extended in many directions. In [65] network connections are considered along with system call traces. Wagner et al. [66] add several models of the nominal behaviors, which are not based on statistical inference but on application source code static analysis. However, in

case of OTS- and legacy-based systems the source code may be not available; hence, the application cannot be analyzed. In [67] the sequence of system call as well as their parameters are considered and cluster analyses are used to characterize the normal behavior. The overhead experienced to collect all system call traces makes difficult to use these techniques during operation.

Wang et al. [68] reveal hangs by means of OS signals and hardware performance counters. In [3] the tracing facilities provided by Linux are exploited to detect application hangs or crashes. The detection framework is composed of a set of monitors in charge of generating alarms and a detector that decides about the state of the system by correlating alarms of the monitors. In [20] the authors use some indicators representing the state of the Windows OS, e.g. the number of context switches, of semaphores and mutexes, to predict imminent crashes or hangs.

Among indirect probing strategies, function boundary tracing techniques that monitor entry/exit events of functions can also be adopted to reveal anomalies in application and OTS behavior [54] by using instrumentation tools, such as the DTrace [69]. However, these techniques are more suited for performance bottleneck analysis (e.g., average service time of functions) or reverse engineering (e.g., by building call graphs). Indeed, they are useful to monitor the execution of a software component at the binary level and to perform manual analysis of the observed behavior. Hence, it is difficult to reveal complex anomalies, such as software aging-related, solely looking at times of function entry/exit.

In [70, 71] the authors use both direct and indirect monitoring approaches. Agarwal et al. [70] exploit application- and system-level indicators such as number of served requests, CPU

usage, response time, number of queued requests, which are collected at regular instants of time at the nodes of a clustered multi-tier enterprise application. The aim of the framework is to identify the root node(s) of anomalies, e.g., application hangs or crashes, overloads and configuration errors. In [71] the authors identify performance degradation due to software aging anomalies in a large telecommunication system by means of system-level indicators, such as the available memory, and user-level indicators, such as number of packet loss in a time interval, are collected. The degradation is detected when the packet loss and capacity are over predefined thresholds.

External indirect probes are used by Monitor [61]. This is a hierarchical, application neutral framework to reveal anomalies at protocol level in large-scale distributed systems running legacy code. The Monitor is based on indirect probes that observe exchanged messages between the protocol participants. The messages are used to reconstruct a runtime state transition diagram of the participants. The Monitor has also been extended to accomplish diagnosis tasks by means of a set of test procedures [72] chosen starting from the inferred state of the participants.

Monitoring frameworks

A lot of research effort has been devoted to the enhancement of data representation and communication protocol to limit the overhead of the monitoring infrastructure. Probably, the most popular standard addressing such issues is the Simple Network Management Protocol (SNMP) [73], which is also part of the Internet Protocol Suite. Other popular

standards for specific technology such as Java, .Net, and Web Service, are JMX [74], WMI [75], and WSDM [76], respectively.

Many tools focus on reducing communication and storage overheads in a distributed environment. When monitoring hundreds or even thousand of nodes in cluster and/or grid systems scalability becomes the main requirement. In [77] the authors present Ganglia, a distributed infrastructure for monitoring clusters and grids resources (e.g. load, CPU, memory, disk usage, response time) at different time-scale. The architecture of Ganglia is depicted in Figure 2.7. To meet scalability requirement, the infrastructure uses a hierarchical

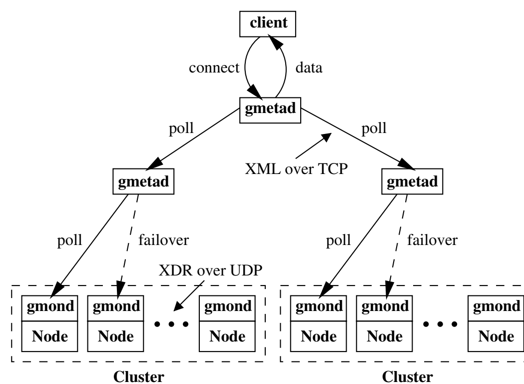


Figure 2.7: The architecture of Ganglia

architecture and the aggregation of collected data.

Another approach to monitoring large-scale systems that also organizes nodes using hierarchy has been proposed in Astrolabe [78]. Furthermore, a peer-to-peer gossip-based protocol is used to share the collected data.

CoMon, presented in [79], also monitors resources of distributed systems, such as the nodes of PlanetLab. The main purpose is to understand their interactions remaining largely

agnostic about the applications running on the nodes.

Other tools designed to be high scalable and flexible are Nagios, widely recognized by IT industries as the most used monitoring infrastructure (http://www.nagios.com/users/), Zabbix (www.zabbix.com) and the Zenoss (www. Zenoss.com) platforms.

IBM Tivoli Monitoring [43] also allows to manage operating systems, databases and servers in distributed and host environments. The approach used is a centralized monitoring solutions providing system operators a control centre from which they can oversee the target system and control what monitoring data gets collected. However, such a solution allows to monitor a limited number of servers (about 500 [1]). Furthermore, Tivoli provides a set of facilities to automate the detection such as the creation of queries for filtering and correlating events and the setting of thresholds to trigger alarms. However, configuring and effectively using such tool is the totally responsibility of human operators.

Many monitoring frameworks for networked systems are based on probing technology. In this case, direct external probes are predefined tests whose outcomes depend on the health of the monitored components. For instance, in the context of networked systems, probes can be implemented as programs executing on a particular machine (which is usually called the probe station) that sends requests to servers or to network elements and evaluates the responses. The ping and traceroute [80] [81] utilities are probably the most popular external probing mechanism to detect network availability. At application-level probing tools, such as IBM's EPP technology [82], provide more sophisticated tests with respect to network-level probes since the application logic needs to be known.

[1] See Section 2.2 at http://www.redbooks.ibm.com/redbooks/SG247217/wwhelp/wwhimpl/js/html/wwhelp.htm

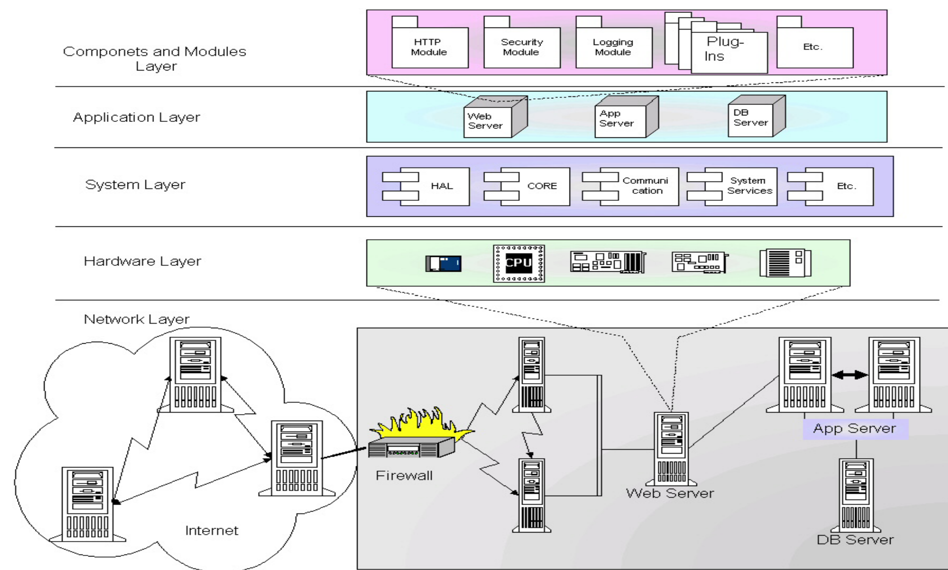


Figure 2.8: Example of probes at multiple levels

Figure 2.8, taken from [45], illustrates the core ideas of direct external probing technology. The bottom left of the picture represents an external cloud (e.g. the Internet), while the greyed box in the bottom middle and right represents an example intranet, e.g. a web site hosting system containing a firewall, routers, web server, application server (running on a couple of load balanced boxes) and database server. Each of these contains further substructure. Probing can be performed at different levels of granularity that depend upon what is the objective of the monitoring. For example, in case of Service Level Agreement (SLA) monitoring, just the response time at the external cloud and the intranet (depicted in Figure 2.8) may be sufficient. To perform a finer-grain detection all system components, such as the web server, the application server, the database, should be probed.

SysStat [83] is a monitoring framework for Linux OS that enables the collection of IO,

CPU, memory and interrupts statistics at at node-level. This is simply accomplished by integrating the most used monitoring utilities, such as *sar*, *sadf*, *mpstat*, *iostat*, *nfsiostat*, *cifsiostat* and *pidstat*.

Other monitoring tools use indirect probes to collect raw events, called *trace*, about the target system are LTT (Linux Trace Toolkit) [84], Systemtap [85], DTrace [69], Chopstix [18] and SysProf [86] and WRPM (Windows Reliability and Performance Monitor). These tools exploit the kernel-level tracing facilities to monitor raw level events such as system calls, process creation and termination, file open/close/read/write and disk/network I/O. LTT has been the first attempt to provide a finer-grain monitoring tool capable of tracing per-process raw level events such as syscall entry/exit, Inter Process Communication (IPC), page allocation/deallocation, scheduling, interrupts. The architecture of LTT is shown in 2.9. The *trace facility* module is the unique entry point for all the kernel-level tracing points. The *module trace* takes the collected events and then delivers them to the trace daemon that definitely allows to store the desired events in files.

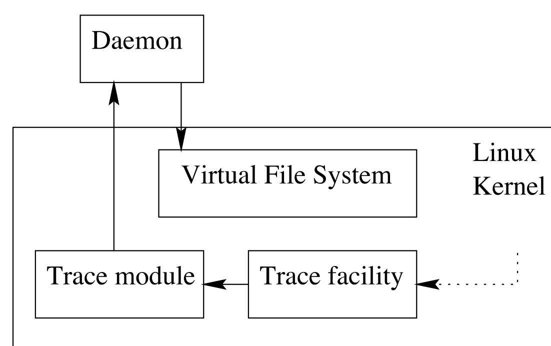


Figure 2.9: LTT architecture

LTT is useful for off-line analysis even if the overhead is kept low, i.e., about 2% when monitoring kernel-level events. One limitation of LTT is that the tracing facility has no timer-based probing (i.e., an event issued each time a timeout expires). However, the focus of the work is on the design and on the implementation of the tool rather than providing useful insights on anomaly detection mechanisms (this is left to the community since the project is open source).

DTrace has the ability to dynamically instrument both user-level and kernel-level processes and has zero probe effect if not enabled. The Dtrace architecture is shown in 2.10.

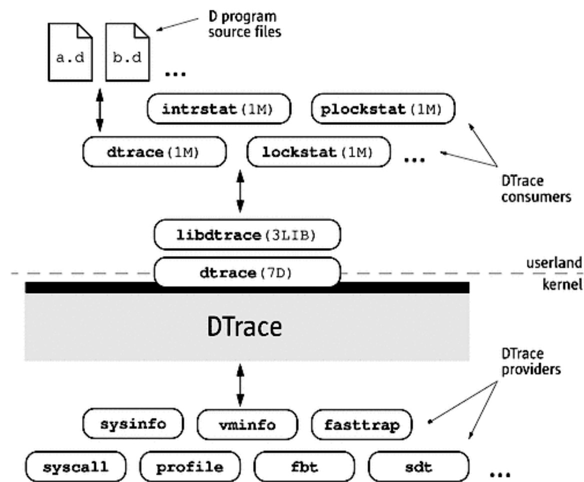


Figure 2.10: DTrace architecture

The core elements are a set of providers (e.g., syscall, profile, vminfo) and the consumers (e.g., intrstat, plockstat) that allow to the users to filter relevant information and to get some statistics. As LTT, the focus in [69] is on the design and on the implementation. However, some useful DTrace applications in diagnosing performance problems in production environments are also presented. Unfortunately, only manual analysis are discussed and no

automatic detector based on DTrace is proposed.

SystemTap (stap) [85] is a dynamically loadable instrumentation tool that allows to run user-written script in C-like language to extract, filter and summarize data without to recompile the kernel, install it, and reboot. This facilitates the diagnosis of a wide types of complex issues (such as performance problems and functional problems). Compared to LTT and DTrace, which have a limited number of probes (about a thousand) it can potentially monitor millions of events. Since version 1.2 stap also allows to monitor Java Virtual Machine (JVM). For these reasons, stap has been chosen as ready-to-go tool to implement the proposed framework in Linux platforms.

Table 2.2: Differences among LTT, Systemtap and DTrace tracing mechanisms

	LTT	**Systemtap**	**DTrace**
OS support	Linux	Linux	Solaris, OS X, BSD, QNX
processor support	x86-32/64, SPARC/64 ppc/64 sh/64, ia64, s390 MIPS32/64, ARM (kernel); x86-32/64 (user)	x86-32/64, ppc64, ia64, s390, arm, sparc?	x86-32/64, SPARC, ppc/64
overhead	low	high	high
target usage	debugging, tracing, profiling, monitoring	debugging, tracing, profiling	debugging, tracing, profiling
language style	C	scripting	scripting
speculative tracing	work in progress	yes	yes
binary tracing	yes	yes	?
probe execution	native code	native code	interpreted
#probes	thousands	millions	thousands
java tracing	yes	soon	yes
timer-based probe	no	yes	yes
built-in	in progress	no	yes
analysis tool	yes (lttv)	no	no
type of analysis	offline	online	online

Table 2.2 summarizes the differences of the LTT, Systemtap and DTrace kernel-level tracing mechanisms.

The windows counterpart of Systemtap is the Event Tracing for Windows (ETW) [87]. ETW is part of the Windows Reliability and Performance Monitor (WRPM) a monitoring infrastructure, available on both Windows desktop and Server releases, which provides several functionalities to: *(i)* monitor applications and hardware performance counters in real time; *(ii)* track the performance impact of applications and services; *(iii)* generate alerts and reports; *(iv)* take actions when user-defined thresholds are exceeded. ETW collects data from trace providers that report actions or events related to components of the OS (kernel-mode) or of individual applications (user-mode). Events monitored by this tool include: Process (Thread) creations or terminations, system call, disk I/O, TCP/UDP network I/O, context switches. Output from multiple trace providers can be combined into a trace session. Then, the trace may be analyzed by one or more consumers by allowing large-scale server applications to write events with a minimum overhead. We have exploited WRPM to implement our framework for Windows platforms because of the possibility of collecting raw level events and the low overhead. More details on the WRPM and ETW are given in Chapter 3.

In case of Chopstix, the authors adopts a non-deterministic monitoring approach that, with respect to periodic sampling, increases the probability to sample rare events and also addresses the overhead issue. This work is closer to ours since, even if the authors do not propose an on-line detection, the collected events are periodically sent to a remote node that build some "vital sign" of the monitored application. These, along with the stack traces of monitored applications, can be exploited for off-line diagnosis to identify the activation of non-malicious faults. On the contrary our proposal is to build this vital sign on-line and

timely detect anomalies.

As for SysProf [86], this tool uses the collected events to evaluate the performance of client requests, such as response time, average time spent by requests in user-level and kernel-level, to multi-tier enterprise server. The derived performance can be exploited to identify the bottleneck component in such systems.

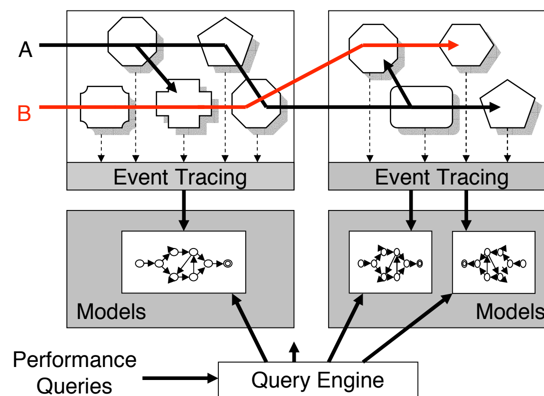


Figure 2.11: Conceptual view of Magpie

Another monitoring tool proposed to diagnose performance problems in multi-tier-based and distributed enterprise systems is Magpie [19]. Unlike previous tools, the goal of the work is to enrich the OS with a service that provides on-line performance models of the monitored components. A conceptual view of Magpie architecture is shown in 2.11. The work is close to our since the authors recognize that, despite several tracing tools collecting a wealth of data have been already proposed, the real problem lies behind the on-line data consuming and the reporting of the most useful data to detect anomalies.

PerfMon [88] is a tool proposed by Hewlett Packard (www.hp.com) which exploits hardware performance counters of modern CPU chipset (e.g., ITANIUM, PENTIUM, AMD) to monitor micro-architectural level events from, for example, pipeline, system bus, caches. From these events useful performance indicators, such as TLB misses, stall cycles, memory access latency, are extracted. The goal of PerfMon is to provide a portable and extensible tool with a standard interface that can be used to monitor a wide set of hardware; hence, detection issues are not the main focus.

Another interesting tool that can be used to monitor and collect data is Daikon [89]. This is a tool that collect traces of program execution and automatically builds likely invariants, i.e., properties that holds at a given time in a program in execution. For examples, these include constants, non-null/zero values, ranges, linear relationships, ordering, sort, and containment. To build the good likely invariants the traces need to be collected by performing several experiments with different workload. Then the tool also implements the dynamic detection of likely invariants during program execution since it checks these invariants at runtime and, when these are violated it signals anomalies. Despite Daikon has been used for many applications among which predicting incompatibilities in component integration, automating theorem proving, it does not scale well for extremely complex systems that consist of several KLoc such as middleware, application servers, databases.

Tables 2.3 summarizes most relevant information collected by the discussed monitoring frameworks.

Table 2.3: Most relevant monitored indicators of the surveyed frameworks

Frameworks	**Collected Information**
Ganglia [77] and	load, CPU, memory,
Comon [79]	swap, disk usage, response time
SysStat [83]	IO, CPU, memory,
	and interrupts statistics
LTT [84]	syscall, trap, interrup,
	scheduling, process, file system, timer,
	memory, swap,page in/out, socket, IPC
DTrace [69]	syscall, lock, timer, stack trace, CPU,
	kernel functions
Chopstix [18]	disk/net IO, CPU, lock, cache misses,
	memory, socket, scheduler, syscall
Systemtap [85]	syscall, trap, interrup,
	disk/net IO, CPU, timer, file system
WRPM [87]	disk/net IO, CPU, timer,
	file system,mutex, object, CPU, syscall, memory
SysProf [86]	syscall, network, scheduler,
	driver, process, file system
Magpie [19]	uses WRPM
PerfMon [88]	TLB misses, stall cycles, memory access
	latency from pipeline, system bus, caches

2.2.2 Detection

As previously discussed, the detection of anomalies requires a notion of the system normal behavior. This can be *(i)* derived by building models, *(ii)* inferred from data collected during system operation (e.g., analyzing likely invariants, event logs, response times), *(iii)* intuitively expressed and then validated with experiments, or even *(iv)* obtained during the testing phase.

Models are often used to define a synthetic representation of the system, which can be useful to abstract its principal proprieties and characterizing its function and behavior, to correlate relevant events, and to choose the proper remediation in case of failures. Models can be built exploiting the knowledge of the structure, the behavioral and the functional principles of the target system [90, 91] that can derive, for example, from documentation, design artifact and source code. This knowledge can be expressed in mathematical terms, i.e., relations between the inputs and outputs (quantitative models) or by qualitative features (qualitative models).

When the knowledge of the system is limited or even absent *data-driven* approaches can be used. These models are usually built by applying techniques coming from machine learning and statistics to data collected on the field or by means of experiments. Many studies applies data-driven approaches to detect anomalies, such as [92, 52].

Another perspective of analysis of model-based detection concerns what kind of behavior is captured from the model. Since failure-related data are usually limited – indeed failures are rare events –, most of model-based detection approaches reveal anomalies by modeling

the normal behavior of the systems. However, some studies also propose to reveal anomalies by modeling the anomalous behaviors (e.g., [93, 94]).

Finally, models can be generally distinguished based on what is their specific target of the model. For instance, many studies [90, 91] propose performance models (e.g., to predict resource utilization, throughput, and response time) and detect anomalies when the current predicted performance are different from the actual. Instead of performance, other studies exploit behavioral or functional models to detect anomalies, such as [95].

Table 2.4: Perspectives of analysis for model-based detection approaches

System Knowledge	Present
	Absent
What is modeled	Normal Behavior
	Anomalous Behavior
Type of model	Performance
	Functional
	Behavioral

Table 2.4 summarizes the different perspectives of analysis that are considered to briefly introduce model-based detection.

Many studies propose models that predict some attributes, such as resource utilization, throughput, and response time, which are related to the performance of the target system or its components. The resulting models are used for capacity planning, provisioning and to tune configuration parameters to maximize the performance; however, they can be also used for detecting and diagnosing anomalies [90].

A widely used modeling approach for systems serving multiple users is queuing theory. A queuing model is an abstraction of the system consisting of a set of interconnected queues. A queue is associated with a service or a resource, which holds requests that are waiting to be served. For example queuing theory has been used to model a web server. In [91] the authors use a single queue to model Apache HTTP web server; while, [96] with several queues are used to model different resources of the system such as disk, network and CPU.

Performance models have a number of limitations to be applied for on-line anomaly detection. First, they only allow to reveal anomalies in the performance metrics of interest (e.g., response time and throughput); hence, when anomalies have already propagated to the system or component interface. Second, they typically need a lengthy profiling phase to estimate model parameters such as service times, which may require unaffordable manual effort if the system is made of many components and no historical data is available. Third, the level of detail that the model can capture arise from a trade-off between accuracy of the model and computational resources required to solve the model efficiently. When the timeliness requirements of the detection are very strict the level of detail of the model (and so its accuracy) may be very low.

Data-driven approaches are often exploited when dealing with complex software systems. For instance, Li et al. [92] exploit resource usage data collected from a node hosting a web server and then use auto-regressive moving average (ARMA) models to predict resource exhaustion (e.g., memory and swap space) due to software aging anomalies. In [52] the author discuss the use of statistical and machine learning to predict rare event (e.g., failure) in large clusters. In particular, they analyze the time-series models, rule-based models,

bayesian network. The study reveals that, among the three techniques analyzed, time-series models are the most accurate for their target system. The model proposed in [97] allows to predict CPU consumption of the transactions in web applications. This model is constructed by means of statistical linear regression using measurements on a real system. The model is then used to detect performance anomalies, which are defined as those changes in CPU usage not explained by the actual workload.

Rule-based approaches are particularly used for detection and diagnosis purposes. They were developed in the mid-1970s and have formed the basis for a large number of expert systems in medicine and other diagnosis-related areas [98]. They are based on a set of rules that can be represented in the form "IF condition THEN action" and formalized by the use of the first order logic or more powerful expressive languages, e.g., OWL (http://www.w3.org/TR/owl-ref/) the language to publish ontology on the web, which can be used to reveal anomalies. The condition refers to the information about the functions of the system and the collected events; while, the action usually contains the use of parameters for choosing the next rule or the countermeasure (e.g., reconfigurations and recovery).

Li and Malony apply rule-based reasoning, by using rules in the CLIPS expert system, based on architectural patterns [99] for the detection of the performance bottleneck. Benoit et al. apply the same approach for database systems [100].

In [61] temporal and combinatorial rules, obtained from protocol specifications and system administrators, are used to reveal anomalies in distributed applications. Processed data are collected using the aforementioned Monitor. This has also been extended to accomplish diagnosis tasks by means of a set of test procedures [72] chosen starting from the inferred

state of the participants. However, the rulebase approach is very difficult to manage in dynamic environments where the frequency of rules update may be excessive.

Another rule-based approach is proposed in [101]. The authors describe a system exploiting event logs and identifies episode rules, i.e., a temporal ordering of events. They correlate the errors in the logs by counting the number of similar sequences. For example, an episode rule is in the form "if errors A and B occur within five seconds, then error C occurs within 30 s with probability 0.8". Several parameters such as the maximum length of the data window, types of error messages, and ordering requirements have to be a-priori specified. However, the algorithm returns too many rules such that they need to be presented to human operators having system knowledge to filter out the most relevant. In the field of power system process an interesting rule-based anomaly detection has been proposed in [102]. The authors exploited fault tree analysis. However, since fault tree analysis offers a statical view of the system, i.e., without taking into account the current state of the system, they combine the leaf nodes with on-line detectors, and logical expressions are transformed into a set of rules so that they can be used as an on-line detector [103].

The advantage of rule-bases approaches is that the rules are easily readable by the user, and their effect is intuitive. However, the this approach is suitable for cases in which the relations between events are well known and can be clearly formulated. Furthermore, there are intrinsic limitations in the management of the rules due to the frequency of changes in the operating environment of mission-critical systems. In other words, if the system changes rapidly, it is hard to maintain an accurate set of rules. Finally, these approaches do not deal with incomplete, incoherent and uncertain information that can be collected in

complex software systems.

Another model-based approach is the Hidden Markov Model (HMM), which is used for both anomaly detection and diagnosis tasks [104, 49]. Indeed, HMMs allow both the definition of a framework through which the detection and diagnosis problem can be formalized, and the adoption of a detection and diagnostic mechanism. HMM is basically a Markov chain whose state is not observable (indeed, the state is "hidden"); however, when the system is in a given state, observable symbols are emitted depending upon a probabilistic function of the state. HMMs can be used to represent probability distributions over sequences of observations. A good tutorial about HMMs and their use can be found in [105]. In [104] the authors propose to use HMM to infer whether the state of a monitored component is anomalous or not by means of a sequence of probing results and the use of the forward algorithm, which is an efficient dynamic programming algorithm, to compute the sequence likelihood of hidden Markov models.

In [106, 49] the authors propose to use hidden semi-Markov models (HSMM) that encode restrictions on properties of errors in the event logs and on the given sequences. They associate the detection with the observations that are generated by the states of the HSMM; errors may be mapped to groups of hidden states of the chain. With HSMMs, similarity of error sequences can be obtained by the use of the forward algorithm; these are then used to detect anomalies leading to system failures. The HMM mechanism is flexible enough to take into account the uncertainty of both observed events and observers, so that it can be also used to detect malicious attacks. For instance, in [107] HMMs are used to implement an anomaly-based intrusion detection system.

Bayesian approaches are also widely adopted. In [108] the authors exploit a set of a monitors with limited coverage, i.e., that cannot obtain all relevant information, whose outputs are correlated to detect the faulty component. The reasoner exploits a Bayesian inference engine to update the probabilities of having a faulty component. Nickelsen et al. [109] apply Bayesian Networks (BNs) for probabilistic fault detection and diagnosis for TCP end-to-end communication both in the wireless and wired domains. BNs are used to infer system insights and a-priori system knowledge, by using the probabilistic reasoning systems. The structure and the initial parameters of the BNs depend on the prior knowledge of the domain expert. However, if the system is properly modeled, these techniques lead to high performance in terms of accuracy and detection latency.

In [110] clustering techniques are used to identify normal behavior. Then, the anomalies are detected observing the entropy of the clusters of similar metrics, which are grouped by means of Normalized Mutual Information.

In [111] the authors exploits invariants between indicators, defined as stable correlations in monitored variables. These are obtained without the knowledge of the system structure and are exploited to build a behavioral model of the system. The detection of the activation of a fault, e.g., memory leak, missing file, hang, is performed by revealing missing invariants. Another detection approach, which also uses invariants is proposed in[112]. In [112] the authors combine both models and on-line analysis by means of an on-line reliability monitoring that periodically evaluates system reliability (during operation). The approach combines static reliability modeling and dynamic analysis. In particular, the reliability model is obtained by means of Discrete Time Markov Chain (DTMC) and takes

into account the architectural model of the system, the single component reliability and the interactions among components. Then, the operational reliability is estimated by taking into account invariant violations, where the likely invariants are found at testing time by means of the Daikon tool [89]. Even if the approach is promising to reveal anomalies in system level reliability, it has critical limitations because of the use of a static threshold for discriminating reliability level violations and the high number of false positives (i.e., an alarm is triggered but no anomaly occurs), which is about 69%; hence, untolerable for mission-critical systems.

A similar approach has been proposed in our previous work by combining static analysis performed during the testing phase and dynamic analysis performed during operation [113]. Likely invariants between data exchanged among components, interactions pattern among components, OS-level indicators (e.g., number of syscall errors, task scheduling times, I/O throughput and waiting/holding times in critical sections) are collected during the testing phase to build known normal and failing behaviors. These models are exploited on-line to detect anomalies in application behaviors. However, this approach has non-negligible overhead since many data need to be recorded (e.g., system call parameters are recorded or IO invariants). Moreover, it is not well-suited to build a reliable failing models to detect the activation of faults that cannot be easily reproduced during testing, such as Mandelbugs.

Instead of focusing on the normal behavior the system, approaches that model the anomalous behavior have also been exploited to reveal anomalies. In [93, 94] the authors assume that exist a unique combination of information which correlates with performance

violations due failure occurrence. The signatures are hence used to detect anomalies. In [94] Agarwal et al. define the *problem signature* as a unique combination of changes (or absence of changes) in one or more monitored indicators. By means of change point detection algorithm alarm are triggered when such indicators show a sudden variation from their mean values. These alarms are then filtered by means of user specified policies to reduce false positives. The final alarms from the metrics that are part of the problem signature are then correlated according to the problem signature to detect the occurrence of the given anomaly. In [114] a bayesian belief network, which encodes both healthy statuses and known problems, and a set of of models, including generic statistical and resource-specific models, are exploited to identify specific fault activations. However, this approach requires that experts encode system components and their specific dependencies in the Bayesian network.

Simple mechanisms suggested by intuitive reasoning and then validated by experiments or modeling are the so-called *heuristic approaches*. An example of a heuristic mechanism for the discrimination between transient and permanent faults is the $\alpha - count$ mechanism [115]. This heuristic implements the "count-and-threshold" criteria to discriminate between transient, intermittent and permanent faults. The following assumptions are made: failures of the system components can be permanent, intermittent and transient; many repeated error conditions within a little time window are easily evidence for permanent or intermittent failures; isolated errors collected over-time could be evidence for transient failures. Hence, the $\alpha - count$ increases a counter when error messages and signals are collected over-time and decreasing the counter when no errors are collected. An anomaly is detected when the

counter passes a predefined threshold.

The first attempt to detect transient failures has been proposed for mainframes (e.g., IBM 3081): in [116] they count the number of errors accumulated in a time window. When the errors are over a predefined threshold, a permanent failure of the component is detect and a maintenance intervention is triggered. In the architecture proposed in [117] for the TMR MODIAC railway interlocking system by Ansaldo, a detector operates in the following way: two failures experienced by the same hardware component, which is part of a redundant structure, in two consecutive operating cycles make the other redundant components to consider it as definitively faulty. In [118], it has been implemented in the GUARDS architecture for assessing the extend of the damage in the individual channels of a redundant architecture. Romano et al. [119] use the $\alpha - count$ for detecting replica failures in OTS-based replicated systems.

Several studies also investigate the tuning of the $\alpha - count$ mechanisms. For instance, in [115], the trade-off between promptness and accuracy is evaluated. A discussion about the dynamic selection of thresholds for QoS adaptation can be found in [120]. Finally, other variants (e.g. the double-thresholds) are described in [25, 121].

The detector proposed in [3] is the most similar to our framework since it exploits heuristic detection approach and the tracing facilities provided by the Linux kernel to reveal application hangs or crashes. The detector is composed of a set of monitors in charge of generating alarms and a module that decides about the state of the system. Monitors collect raw events, such as the times to acquire/release semaphores and mutexes, the syscall errors, OS signals, processes/threads creation/termination and disk and network I/O throughput.

These signal anomalous conditions by means of counting and threshold approach. For instance, the semaphore monitor triggers an alarm if the number of expirations of the timeout to acquire a semaphore is out of bounds in a given time window. Finally, the detector reveals the occurrence of an anomaly if, by summing all the received alarms, these are over the anomaly detection threshold. Thresholds, for both the detector and the monitors, are determined after a preliminary profiling phase. More details about the detector are provided in Chapter 3 where also a comparative analysis is discussed.

Another OS-level approach is proposed in [20]. Indeed, quantities representing the state of the OS, e.g. the number of context switches, of semaphores and mutexes, are used to predict imminent crashes or hangs.

The techniques proposed in [3, 20, 68] all detect anomalies by using static thresholds that are tuned by administrators [68] or are set after a profiling phase [3, 20]. However, often boundaries between normal and anomalous behaviors cannot be exactly known. The static definition of a normality region including every nominal behavior can be inadequate for long-running systems. Indeed, the normal behavior may considerably change according to the actual load, system updates, reconfigurations. Hence, static thresholds, invariants or learned clusters appear not suited for systems with variable and non-stationary behavior if not well adapted to the operational environment. The need of adaptive solutions is well highlighted also in literature as shown in the following.

Agarwal et al. [70] exploit a monitoring framework, that as previously discussed, collet application- and system-level indicators with the purpose of identifying the root node(s) of

anomalies (i.e., hangs, crashes, overloads and configuration errors). The detection technique is based on eigenvalues analysis and change point detection. In particular, the principal eigenvector is extracted from the covariance-matrics of the collected events and the anomaly is revealed by comparing recent means of the principal eigenvalue, where recent refers to the last D seconds. However, the detection is not triggered timely since the proposed detection approach needs $2 \times D$ to reveal an anomaly.

In [45] an incremental approach is proposed that continuously updates the current diagnosis as more observations become available and a more sophisticated model, which accounts for system dynamics, can provide a more accurate diagnosis in such cases.

In [122] it is presented Stardust, i.e., general purpose framework for adaptive and efficient (in terms of time and space complexity) window-based feature extraction and for indexing multiple time-series. This can be useful for burst and trend detection and for pattern matching in time-series that may occur at different time scale.

In [123] the authors investigate detection mechanisms for hang OS components in microkernel-based architecture such as Minix (www.minix3.com). The detection is based on OS-level probes that periodically ping monitored processes. The work is close to our since an adaptive timeout mechanism is used. Indeed, an anomaly is detected if no response is received from the process before a timeout expires. In particular, the authors explore several heuristics, among which exponential moving average and weighted sum, to compute on-line the adaptive timeouts starting from previous response times. The study proves the efficacy of adaptive timeouts with respect to the static timeout adopted in Minix using both simulation and experiments. Unfortunately, the proposed heuristics are difficult to be

tuned since the parameters need to be found empirically and it is only suited for infinite loops or deadlock detection.

Adaptive methods are also implemented in the solutions offered by the industry such as BMC ProactiveNet [124] and Netuitive [125]. These solutions are based on periodical evaluation of the application workload and the thresholds are properly adjusted. The work in [126], like our, copes with the variability issue; it addresses service-level anomalies of multi-tier Web-based systems. The on-line service dependencies are tracked through a dependency matrix. Then, the principal left singular vector is used to capture the normal dependencies over time. Finally, anomalies are detected using a windowing approach to extract the principal eigenvector. The method is designed specifically to detect faults in Web-based systems with redundancy; it is computational heavy and it is not appropriate for anomaly detection of rarely invoked services.

Among the frameworks that also address diagnosis process it is worth to mention Pinpoint. In [95] the Pinpoint diagnosis framework is proposed to recognize the most likely faulty component in large networked systems. The conceptual architecture of Pinpoint is depicted in Figure 2.12 (taken from [95]). It employs statistical learning techniques to “learn from system behavior” and to diagnose the root cause of failures in a Web farm environment. The approach relies on collecting client traces and clustering techniques to identify the components most relevant to a failure. The authors modify the middleware to insert the probe for tracing requests and components used along with a set of accessory informations (e.g., component version) to find dissimilarity that explain the observed behavior. They use built-in detector at middleware level, such as exception handling and assertion,

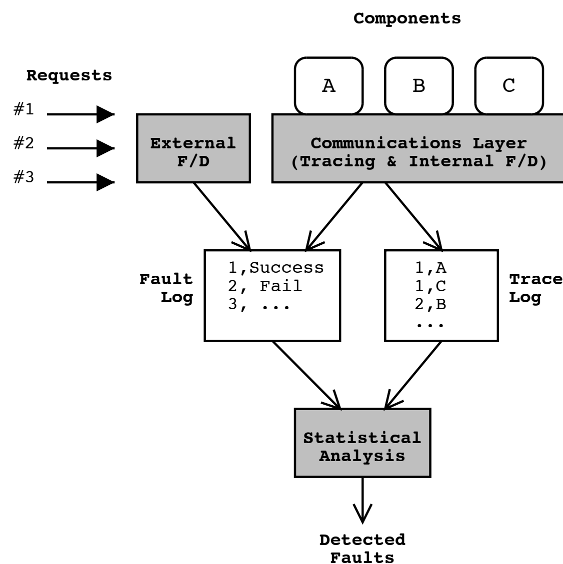


Figure 2.12: Pinpoint architecture

and implement a packet sniffer to detect error in HTTP packet or TCP datagram. The major limitations of this approach are that *(i)* the tracing mechanisms need to modify the middleware *(ii)* it exhibits a significant false positive rate of $40 - 50\%$.

Another framework design to diagnose performance or unexpected behavior in networked systems is Pip [127]. This tool, like Pinpoint, constructs causal paths and reveals anomaly by means of expectations violation. The approach presumes that the expected behavior is encoded by programmers by means of annotations.

The Rainbow framework [128] also uses monitoring techniques to detect Quality of Service (QoS) violations by comparing the considered indicators with the predefined threshold. However, the goal of the framework is to close the loop when changed environment or/and specification occur by adapting the running software system.

Table 2.6 summarizes the surveyed detection approaches and frameworks.

Table 2.6: Surveyed detection approaches and techniques

Reference	**Approach/ Technique**	**Application Domain**	**Remark**
[111]	Model-based	Non-malicious	Invariants are used to build the models
[112]	Model-based	Non-malicious	Reliability Model (DTMC) in combination with on-line invariant violations detection
[97]	Model-based	Non-malicious, Performance	Model is built from measurements on a real system
[19, 90]	Model-based	Performance	Off-line analysis
[104, 49]	HMM	Non-Malicious	Uncertainty (observations and obervers) taken into account
[107]	HMM	Malicious	Anomaly-based intrusion detection
[61] [72]	Rule-based	Non-Malicious, Malicious	Protocol specific; Difficult to manage in dynamic environments
[101, 127]	Rule-based	Malicious	Rules filtered by Human Operators
[52]	Rule-based, Time-series, Bayesian	Non-Malicious	To predict rare event (e.g., failure) in large clusters
[92]	Time-series analysis	Non-Malicious	To predict resource exhaustion due to software aging anomalies
[94]	Time-series	Non-malicious	Assume linear correlation among monitored indicators
[108]	Bayesian	Non-malicious	The main focus is on automated recovery
[109]	Bayesian	Non-malicious	The focus is on network-level faults
[114]	Bayesian	Non-malicious	Require experts knowledge of system dependencies
[110]	Clustering and Information Theory	Non-malicious	Identify also the faulty component(s)
[70]	Clustering and PCA	Non-malicious	Assume linear correlation among monitored indicators
[95, 93]	Clustering	Non-malicious	Main focus is on diagnosis
[115, 116, 117] [3, 119, 128]	Heuristics	Non-malicious	To discriminate transient, intermittent and permanent faults
[45]	Active-Probing	Non-malicious	Account for system dynamics

2.3 Metrics for Quantitative Evaluation

In this Section criteria that should be used to characterize on-line detectors performance for the target systems are analyzed. First, some of the most used metrics in literature are surveyed; then, the most representative metrics in the considered scenario are discussed.

Roughly speaking the aim of an on-line anomaly detector is to generate some detection events that allows to timely reveal all the occurring anomalies, by not generating a detection event when no anomaly occurs. In other words, the detector need to reveal anomalies caused by the activation of non-malicious faults, malicious faults, and by performance issues (e.g., overload) from the normal behavior of the system.

To ease the description of metrics and to precisely understand their meaning some basic definitions are introduced, summarized in Table 2.7:

Table 2.7: Basic metrics for characterizing detector performance

	Anomaly Occurs	**No Anomaly**
Anomaly Detected	True Positive (**TP**)	False Positive (**FP**)
Anomaly Not Detected	False Negative (**FN**)	True Negative (**TN**)

- *True Positive* (TP): an anomaly correctly detected;
- *False Positive* (FP): a detection event that does not correspond to an actual anomaly;
- *False Negative* (FN): an anomaly that is not detected;
- *True Negative* (TN): no anomaly occurs and no detection event is triggered.

Metrics coming from the diagnosis literature are usually used to compare the performance of detectors [129, 130]. For instance, the *coverage* measures the detector ability to reveal all anomalies; *accuracy* is related to mistakes that a detector can make. Coverage can be measured as the number of detected anomalies divided by the overall number of anomalies; while for accuracy there are different metrics.

Chen, Toueg and Aguilera [130] propose three primary metrics to evaluate detectors quality, in particular their accuracy. The first one is *Detection Time* (DT), which informally accounts for the promptness of the detector –basically, its coverage. The second one is the *Mistake Recurrence Time* (TMR), which accounts for time elapsed between two consecutive erroneous transitions from *Trust* to *Suspect*. Finally, they define *Mistake Duration* (TM), which is related to the time that the detector takes to correct the mistake. Other metrics can be simply derived from the previous ones. For instance, *Average Mistake Rate* (λ_M), represents the number of erroneous decisions in the time unit; *Good period duration* (TG) measures the length of period during which the detector event is not a false positive; *Query accuracy probability* (PA) is the probability that, given no anomaly occurs, the detector's output at a random time is correct, (i.e., TN).

Basseville et al. [131] consider the mean delay for detection (MDD) and the mean time between false alarms ($MTBFA$) as the two key criteria for on-line detection algorithms. It is worth to note that such metrics can be evaluated by using Chen and Tueg primary metrics. For instance, MDD, can be evaluated by averaging DT; while, $MTBFA$ can be computed from TMR and TM. The best performance can be usually found by minimizing the mean delay for a given $MTBFA$ and on other indexes derived from these criteria.

In [103] metrics borrowed from information retrieval research are used, namely *precision* and *recall*. In their context recall (R) measures the ratio of the failures that is correctly detected, i.e., $TP/(TP + FN)$, while precision (P) measures the portion of the anomalies that lead to real failures, i.e., $TP/(TP + FP)$. Thus perfect recall ($R = 1$) means that all failures are detected and perfect precision ($P = 1$) means that there are no false positives.

A convenient way of taking into account precision and recall at the same time is by using $F - measure$, which is the harmonic mean of the two quantities. It is worth to note that, since in the field of detection and diagnosis literature the recall is also called coverage, in the following we use only the term coverage.

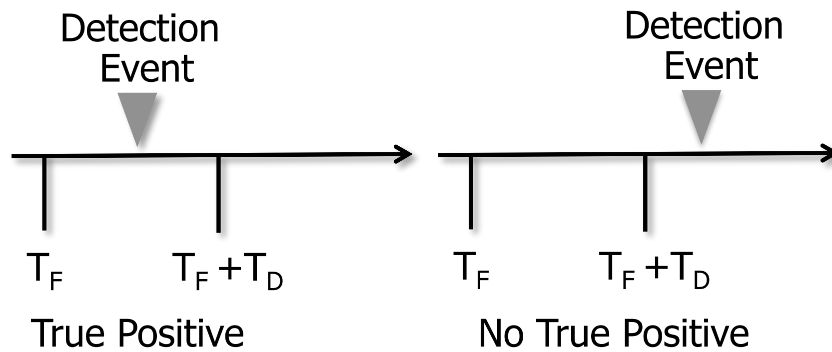


Figure 2.13: Time line showing true positives according to the parameter *Time to Detect*

A useful parameter for coverage evaluation is the *Time to Detect*, T_D. This parameter can be used to distinguish TP detection events. It represents the maximum delay to detect an anomaly. For instance, let consider an anomaly caused by the activation of a fault. Let be T_F the time of fault activation. If a detection event is raised after $T_F + T_D$ (see right side of Figure 2.13), it needs not to be accounted as a TP. Hence, this parameter can be used to take into account the detector promptness, since only TPs that are effectively useful into the diagnosis and recovery processes are considered to evaluate the coverage.

However, precision and coverage alone are not enough to fully evaluate the performance of the detector. Indeed, they do not account for true negatives. Since anomalies usually

occurs rarely, it is mandatory to evaluate the detector mistake rate when no anomaly occurs. Hence, *False Positive Rate* (FPR) needs to be used in combination with precision and coverage. FPR can be defined as *the ratio of incorrectly detected anomalies to the number of all non-anomalies*, thus $FP/(FP+TN)$. Fixing P and C, the smaller the FPR, the better. Another metric accounting for TN is *Accuracy* [103], which is defined as *the ratio of all correct decisions to the total number of decisions that have been performed*, i.e., $(TP+TN)/(TP+TN+FP+FN)$.

Bearing in mind that our target applications are long running mission-critical systems in which just the detection of one anomaly may be sufficient to trigger the needed actions (e.g., the activation of a stand-by spare component), we believe that mistake duration and, thus, TG are less appropriate than coverage, accuracy and mistake rate. Indeed, coverage is essential because if the detector does not reveal an anomaly, then more severe (and potentially catastrophic) consequences may happen. Accuracy and (λ_M) are useful to take into account false positives because each detector mistake may result in costly actions (such as reconfiguration, roll-back, shut down, reboot). Furthermore, the query accuracy probability is not sufficient to fully describe the accuracy of a failure detector. In fact, for application domains in which every mistake causes a costly interrupt, the mistake rate is an important accuracy metric [130].

In Table 2.8 we summarize the discussed metrics and the way to compute them starting from basic metrics, i.e., TPs, FPs, TNs and FNs that are the total number of true positives, false positives, true negatives, and false negatives, respectively.

Table 2.8: Performance metrics to evaluate anomaly detectors

Metric	**Formula**
C	$\frac{TPs}{TPs+FNs}$
A	$\frac{TPs+TNs}{TPs+FPs+TNs+FNs}$
T_D	Interval for fault activation and detection
λ_M	$\frac{1}{E(T_{MR})}$
T_M	Time to correct a wrong decision
PA	$\frac{E(T_{MR}-T_M)}{E(T_{MR})}$

Chapter 3

OS-level Detection of Anomalies

The detection of anomalies in mission-critical systems is a challenging task due the use of black-box OTS-components, unreliable information sources (e.g., the event log) and variable and non-stationary operating conditions. In Chapter 2 we have observed that most of detection mechanisms can reveal anomalies when errors are propagated to the component interfaces. Indeed, the use of external direct probes cannot allow to understand the internal behavior of the component since only its output is analyzed. On the other hand, detection mechanisms based on tracing mechanisms (such as function boundary tracing or kernel-level tracing) observe the monitored components from a grey-box perspective since allow (i) to collect finer-grain information about the components, (ii) to infer their internal behavior and (iii) detect any deviation from the healthy status. However, despite this wealth of data has been indisputably useful to find performance bottleneck and root cause of anomalies, the analysis is mostly done manually or off-line [69, 18]. In this Chapter, we describe a monitoring and detection framework, called ***sosmon****, that exploits the low-level information collected by means of kernel-level tracing mechanisms (e.g., ETW and stap) and statistical analysis, performed by means of a novel algorithm (i.e., Statical Prediction and Safety margin, SPS), for on-line automatically detection of relevant anomalies. First, the requirements that drive the design of the framework and the applicability assumptions are introduced; then, the high-level architecture and the internal components of the framework are described.*

3.1 Requirements and Assumptions

Our target applications are dedicated (but possibly also open) distributed mission-critical software systems; examples are air traffic control systems, command and control and surveillance systems. Such systems are in general the result of the integration of many strongly interacting heterogeneous subsystems, including legacy components and, often because of time and budget constraints, commercial OTS components. These are typically deployed

on many remote nodes communicating through a network.

In order to guarantee high levels of reliability and performance, important components of the system are typically replicated by using active or stand-by schema. For instance,

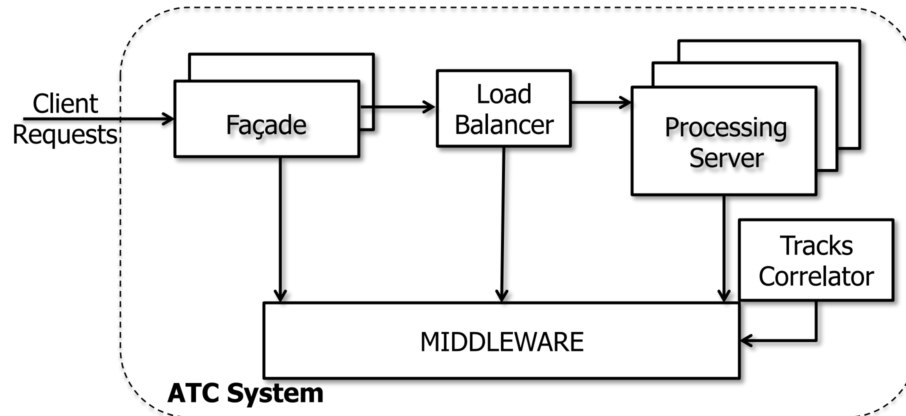


Figure 3.1: A simplified architecture of ATC systems

Figure 3.1 shows a typical (simplified) architecture of air traffic control systems. This is a complex distributed application for Flight Data Plan (FDP) processing; in particular, the system is in charge of processing flight plans and radar tracks by updating the contents of Flight Data Object and distributing them to flight controllers (the clients). The system consists of domain-specific and general purpose components to guarantee high reliability and performance. Domain specific components are the *track correlator*, which collects flight tracks generated by radars and associates them to FDPs, and the *processing server*, which performs other calculations on the route flight information by taking into account the data coming from the correlator, and, finally, the *facade* component, which implements the interface between the clients (e.g., the flight controller) and the rest of the system, and provides

the remote management API for addition, removal, and update of FDPs.

Reliability and performance features are offered by the middleware fault tolerance mechanisms, such as replication, heartbeat and automated transparent recovery of faulty components. For instance, the facade is replicated using to the warm-passive replication schema. As for the processing server, it has several active replicas on different nodes (usually three) and a *load balancer* routes the operations to the servers.

Because of the criticality of the domain, human operators may also track the system's behavior and performance (e.g., by continuously visualizing summaries of system performance). It is also common that, along with internal detectors, system operators exploit some external monitors that trigger warnings when conditions of the system require a manual check. These monitors are typically based on rules of thumb that are not always effective and require huge manual effort to be tuned for each monitored component.

The operating environment in which these components are deployed can alternate stable periods, during which the required resources (e.g., cpu, memory, IO, and network bandwidth) and the monitored indicators have some stability properties (i.e., they are under control) [111, 132], with transient periods (smaller compared with the stable periods), during which a variation occurs because of the workload, new configurations, recovery procedures, or fault activation.

Despite the use of fault tolerance and human monitoring, due to the interactions and component interdependences, errors can propagate among components, resulting in other errors and eventually in a system failure. Thus, the timely, and automatically, detection of anomalies is vital for fast root cause analysis and isolation of the issue, and to avoid any

non-remediable system failures.

These characteristics of such class of systems pose the following requirements on the detection framework:

- to limit modifications to the monitored systems to deal with OTS-based and legacy components;
- to be independent of components and protocols, and to work with different OSes, in order to cope with the heterogeneity of the integrated subsystems and their operating environments;
- to deal with non-stationary and variable operating conditions;
- to reveal anomalies timely, in order to cope with propagations and domino effects;
- to reveal different kinds of anomaly related to unintentional and malicious faults, and to performance issues;
- to be easy configurable for the specific system and detection needs;

To fulfill the requirements, we propose ***sosmon***, i.e, a detection framework that relies on OS-level probes and on on-line statistical analysis to timely reveal anomalies. The kernel-level monitoring infrastructure allows to collect several information about the running components (among which cpu usage, syscall errors, scheduling time, triggered OS signals, acquired semaphores/mutexes); however, it remains independent of the type and the number

of deployed components. Statistical analysis allows to exploit the framework even when the operating conditions are variable and non-stationary.

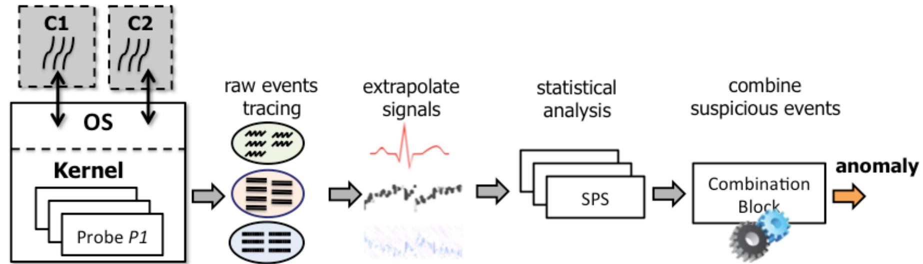


Figure 3.2: A high-level overview of the sosmon framework

Figure 3.2 provides a conceptual high-level overview of sosmon and allows to explain the basic idea that has driven the design of the framework. The detection is based on revealing through the kernel-level tracing probes suspicious behaviors related to user-level or privileged processes, such as middleware daemons and services, that are responsible of important system operations.

The sosmon framework can reveal the anomaly by means of on-line statistical analysis performed on crutial monitored indicators. As depicted in 3.2 raw events are captured by the probes; such events are then transformed into indicators, i.e., the vital system signals, that need to be in control, i.e., have to show some stability properties during a certain interval of time. Then, statistical analysis is performed on the extrapolated signals to reveal when they are out of control.

The detection approach is based on combining adaptive thresholds computed by means of the on-line statistical analysis and α-count methods to group together suspicious vital

signals that may be the symptoms of relevant anomalies. The details of the framework will be provided in the following Sections.

To emphasize the basic differences with traditional detection mechanisms let assume the detector needs to reveal an anomaly of the load balancer (LB), see Figure 3.1. The LB is a critical component since any issue (e.g., overload) may be propagated the to processing servers. Using sosmon this can be accomplished by deploying the framework in the same node of LB. Let assume the LB becomes unresponsive due to an active hang, i.e., the process is incorrectly using cpu cycles. Traditional built-in detection mechanisms of the middleware can detect this anomaly when no heartbeat is received within the timeout T_{LB}. Hence, before instantiating another LB at least T_{LB} needs to pass. The critical problem is that, since the LB is a very busy component that has to dispatch each request received from the clients, such timeout is typically very large to avoid false positive, i.e., the LB does not send the heartbeat just because it is busy dispatching the requests. It is worth to note that a similar problem has in part caused the tremendous, in term of business lost, Amazon outage of the 2010 [41].

An anomaly caused by an active hang of the LB-related processes may be detected by sosmon when, for instance, the cpu usage of the processes becomes “suspiciously” high and the bytes sent on the network become “suspiciously” low. The central point of the detection is how to determine such suspiciousness of the monitored vital signals, i.e., cpu usage, network IO. Indeed, the dynamics described above are rarely known *a priori* and it is difficult to profile each component and each node of the system to derive descriptive statistics of

the nominal behavior. For this reason, on-line statistical analyses and the $\alpha - count$ methods turn to be an cost-effective way to to reveal suspicious behaviors avoiding preliminary training and instrumentation.

The assumptions needed by sosmon are on the data used for statistical analysis. These need to be numerical indicators received at regular time intervals by the monitoring infrastructure. Let x_1,...,x_K be the K monitored indicators. We define:

Def. 1. Time-series $X_i[t]$*: the sequence of* j *samples of* x_i *up to time* t*,* $x_i[t_1]$*,* $x_i[t_2]$*, ...,* $x_i[t_j]$*,* $t_1 < t_2 < ... < t_j \leq t$*,* $t_{j+1} > t$*.*

The time series correspond to the behaviors of the system which need to be analyzed for anomaly detection. The stream of samples $\mathbf{X}[t] = (X_1[t], X_2[t], .., X_K[t]) \in R^K$ constitutes the collection of measures at time t.

We model X_i dynamics as a random walk, and we consider suspicious the changes in the features of X_i caused by non-random factors, such as: the activation of a residual fault; an overload condition, e.g. due to a burst of requests; a malicious activity.

The definition of anomaly provided by the IEEE standard[133] is thus specialized as follows:

Def. 2. Anomaly*: a change in one or more indicators characterizing the behavior of the system caused by specific and non-random factors (e.g., the activation of faults and malicious attacks).*

This means that for each monitored indicator the admitted variability is the result of the

cumulative effects of constant and casual factors. On the other hand, changes due to non-random factors (such as overload condition, the activation of a residual fault and a malicious activity) produce a variability that violated the properties of the underline random-walk process, which is not admitted and needs to be detected.

It is worth noting that no assumptions are made about the stationarity of the monitored OS variables. This means that if their statistical properties (such as the mean and the variance) change over time, the detection of anomalies is still possible.

3.2 High-level Architecture of the Framework

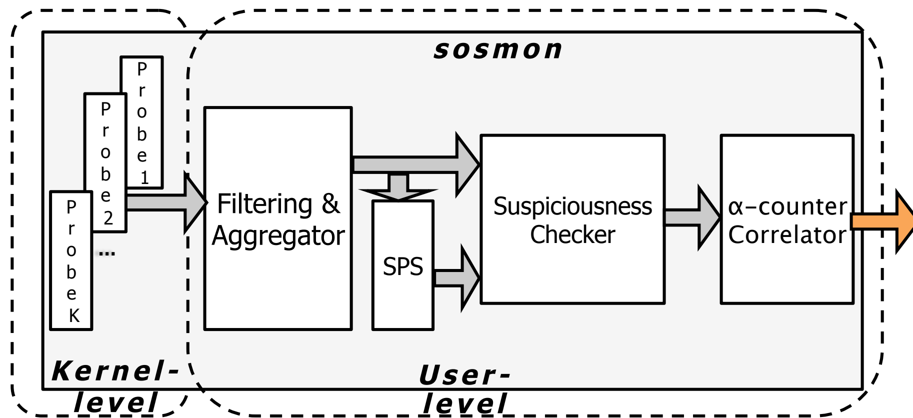


Figure 3.3: The high-level architecture of the framework

The high-level architecture of the framework is depicted in Figure 3.3 and the pseudo-code of the sosmon detection daemon is in Algorithm 1. It worth to note that the core function are provided by the SPS algorithm and the α-counter (namely the function *combineData* in Algorithm 1), which are described in the following sections.

The core of the monitoring infrastructure that allows to detect anomalies is constituted

```
sosmonDaemon

initConfiguration(); /* Initialize the configuration parameters      */
   /* t is the current time and Δt is the monitoring interval
   A is a global two-dimensional array storing history of the
   suspiciousness of the indicators                                  */
while true do
   t ← getCurrentTime();
   for i ← 1 to K do
      X_i ← getIndicator(i, t) /* Store the i − th indicator at time t   */
      T_i ← getSPSThresholds(X_i) ;      /* Array containing the upper and
      lower thresholds */
      /* Check suspiciousness using the thresholds computed by SPS   */
      if X ∉ [T_i[0], T_i[1]] then
         A[i][t] ← 1 ;                  /* Update state variable at time t  */
      end
   end
   if combineData(t) then
      anomalyFlag ← 1 ;       /* Signal to the upper level the anomaly */
   else
      anomalyFlag ← 0;
   end
   wait until getCurrentTime() ≥ (t + Δt)
end
```

Algorithm 1: *sosmon* daemon running at user-space

by a set of kernel-level probes. The monitored indicators x_i $(i = 1,\ 2, ...,\ K)$ are obtained through the kernel-level *probes* and the *filtering and aggregator*.

```
1298975456472#SYSCALL_ERROR_CODE#2 ENOENT#java#12435#12435
1298975457041#SYSCALL_ERROR_CODE#2 ENOENT#java#12435#12435
1298975457041#SYSCALL_ERROR_CODE#2 ENOENT#java#12435#12435
1298975457331#SIGNAL#11#java#12435#12435#java#12435#12435#0
1298975457520#CLONE#java#12435#12548#TASK_ALLOC
1298975457741#MUTEX_HOLD_TIMEOUT#-#1298975451924#0x200e2308#spliced#12290#12295#
1298975457736#PROCESS_EXIT#0#java#12563#12565
1298975456882#THROUGHPUT_SAMPLES#0#0#net-output#68#0#0
1298975457744#SYSCALL_ERROR_CODE#sys_open->2 ENOENT#java#12614#12614
1298975456882#THROUGHPUT_SAMPLES#0#0#disk-input#99192#0#0
1298975457744#SYSCALL_ERROR_CODE#sys_open->2 ENOENT#java#12614#12614
1298975456882#THROUGHPUT_SAMPLES#0#0#disk-output#315#0#0
1298975457744#SYSCALL_ERROR_CODE#sys_open->2 ENOENT#java#12614#12614
1298975457882#THROUGHPUT_SAMPLES#0#0#net-input#1300#0#0
1298975459326#SCHEDULING#12614#12705#java
1298975458882#DISK_READ_TIMEOUT#1298975457524#spliced#12549#12549
1298975461882#DISK_WRITE_TIMEOUT#1298975457561#java#12435#12435
1298975477327#SOCKET#UDP-RCV_TIMEOUT#localhost-45577#12614#17273
1298975477327#SOCKET#UDP-RCV_TIMEOUT#localhost-45567#13425#17448
```

Figure 3.4: An example of trace collected by sosmon

The probes p_i, with $i = 1,\ 2, ...,\ P$, are the kernel instrumentation points that allow to trace raw-level events such as the syscall error, the scheduling times, IPC events, socket operations. However, the collected traces are not yet suitable to perform statistical analysis since each entry just consists in a timestamp with an associated message in a predefined format. For instance, Figure 3.4 shows a trace produced by the probes. The first event collected is related the syscall error and contains the following information (separated by the # separator): timestamp, the type of collected event (i.e., the syscall error), the return code, the process name, the process ID, the thread ID.

The collected trace is analyzed by the *filtering and aggregator*. This module accomplishes a set of modifications to the raw data in order to derive the monitored indicators x_i $(i = 1,\ 2, ...,\ K)$. First, the collected events are filtered by process id. Then, the filtered events

are then aggregated by the timestamp and event type, and, when applicable, by the thread id. Hence, the value of the indicator x_i is the result of the such filtering and aggregation operations.

The processes to monitor, the probe points to enable (and so the collectable event type), the *time unit*, i.e., the granularity of the timestamp for derived indicators, and other options are set by the user by means of sosmon configuration file.

The history of the indicator x_i thus represents the time-series X_i that can be analyzed by means of the Statistical Predictor and Safety margin (SPS) module SPS_i. The SPS module performs some statistical analysis for each indicator x_i and returns lower and upper adaptive thresholds, i.e., $T_i^l[t]$ and $T_i^u[t]$. These adaptive thresholds are then used by the *suspiciousness checker* to verify that the random walk process associated to the indicator x_i is not out of control. Indeed, when the indicator x_i at time t does not fall within the estimated range, it is judged to be suspicious and the global state variable $A[i][t]$, which stores this information, is set to 1.

Finally, the α*-counter* allows to detect an anomaly. This module implements a classical "count-and-threshold" criteria (see Section 2). These heuristics have been proposed to discriminate between relevant from non relevant anomalies [115]. In the proposed framework the α-counter Correlator counts the suspicious variables ($A[i][t]$) signaled over a time window w, raising an alarm when the counter overcomes a predefined threshold. The following assumption is made: suspicious behaviors within a little time window are easily evidence of anomalies for the monitored components. This assumption is supported by many recent studies that recognize some stability properties of system indicators when no anomaly occurs

[132, 111].

The details about the design and the implementation of these modules are describe as follows.

3.3 Internals of the Framework

3.3.1 OS Monitoring Infrastructure

This section describes the instrumentation infrastructure to collect kernel-level events through which OS-level indicators are obtained. In particular, two possible implementations for Linux and Windows environments are detailed.

As discussed in Chapter 1, this work focus on anomalies that do not require the knowledge of the components internals to be revealed. Each ODC defect type can lead to different anomalies and hence different indicators need to be used. The indicators that could be used are discussed as follows.

Timing/Serialization defects may lead anomalies in the usage patterns of OS resources which are typically exploited to manage shared resources is monitored: i.e., semaphores, mutexes, and shared memory. For these reasons, we argue that useful indicators to monitor are the time to acquire shared resources, the holding time of shared resources, the number of resources used and the time for scheduling processes. Indeed, an indefinite wait condition (passive hang) may lead to increasing holding times; an infinite busy loop can lead to an increase in time to schedule processes.

Assignment or Checking defects, such as bad initialization or parameter not validated before used, usually lead to bad memory accesses; hence, other IPC events, such as OS

signals, should also be monitored. For instance, in Linux a bad memory access is reported by the $SIGSEGV$ signal.

Anomalies between the monitored components and the OS, which may be caused by Interface defects, e.g., a system call is called with wrong parameters, could be revealed by monitoring system call (syscall).

To catch anomalies due to Build/Package/Merge defects, the usage pattern of most important system libraries (e.g., *pthread* the multi-thread library) needs to be monitored.

The ODC triggers should also be taken into account. For instance, the Workload and Boundary triggers can be taken into account by counting the number of active processes/threads, the amount of bytes sent(received) to(from) network/disk, the CPU activity and the memory. As for the Concurrency trigger, the number of simultaneous processes/threads acquiring shared resources can be monitored. To take into account the Exception Handling, the errors code returned from syscall could be monitored. Indeed, if the component gets an error from a system call, then it is likely that the error needs to be handled (e.g., by an exception handling routine). By exploiting some knowledge about the system architecture other triggers could also be monitored, e.g., Recovery and Timing.

In order to simplify the sosmon implementation, the monitoring infrastructure has not been developed from scratch but it exploits existing tracing tools for Linux and Windows, i.e., SystemTap and WRPM, respectively. Despite this choice has allowed to reduce the development time, when the framework needs to be deployed on production environments ad-hoc monitoring infrastructure should be implemented to reduce runtime overhead and

the performance penalties. However, the measured overhead of the current implementation of sosmon is still acceptable (less than 5% in the worst case); the details of such an analysis is provided in Chapter 5.

Linux Monitors

The collection of raw OS-level events is accomplished by means of probes dynamically inserted into the kernel, without the modification of the application and the components source code and the recompilation of the OS.

A probe consists of two basic elements: the *breakpoint* and the *handler routine*. The former is a special CPU instruction that suspends the execution of the specific kernel function. The latter consists of a set of predefined commands that execute at the breakpoint to provide the desired information, such as the input parameters and the return values of a called functions.

The events collected in the trace are then filtered and aggregated in order to compute the indicators.

As discussed above, this monitoring infrastructure is implemented by means of the SystemTap, an open source tool downloadable at http://sourceware.org/systemtap/. It has been specifically designed to simplify the development of systems instrumentation [85].

An overview of SystemTap is provided in Figure 3.5.

SystemTap allows to instrument the system by means of different steps: scripting, elaboration, translation and execution. These steps are briefly described as follows; for deeper

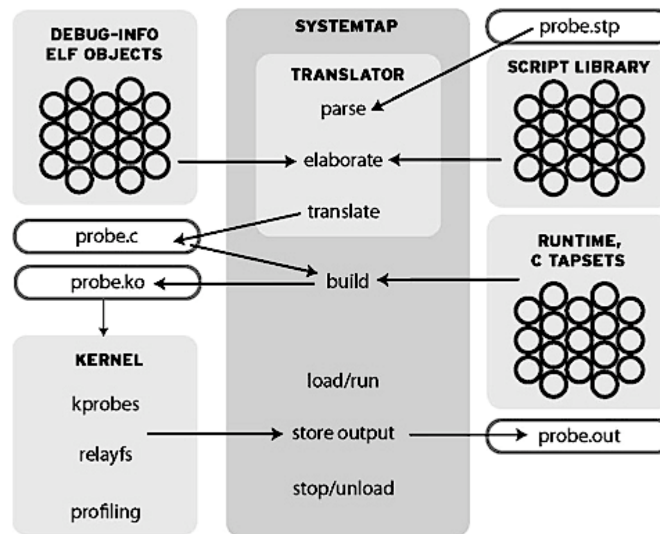


Figure 3.5: The SystemTap tool overview

details see [85].

The first step to instrument the system consists of writing an instrumentation script, which is written in a simple language similar to *awk* scripting (www.gnu.org/software/gawk). The script describes associations of probe points, i.e. the breakpoints, and the handler routines. The user can write its own scripts exploiting already implemented probe points, which are defined into the tapset library. Typical control flow features, such as conditional statements, can be used to perform the so called speculative tracing, i.e., limit the collected events by filtering by PID, and by type of event.

Once the script is written, it can be given in input to SystemTap. The first step to convert the script into an executable that contains the desired instrumentation code is the *elaboration*. Elaboration is the preliminary processing phase, analogous to the linking phase in C/C++ programs. It is needed to resolve references to kernel/user symbols and tapsets

by preparing the successive phase, i.e, the *translation*. The translation allows to insert for each script subroutine a block of C code that also includes necessary locking and safety checks (e.g., infinite loops prevention). Furthermore, it is provided a common runtime with routines for the management of associative arrays, memory, shutdown, startup, I/O.

Then, SystemTap uses a compilation approach to generate the executable from the C code generated by the translation. This contrasts the interpreter approach that has been usually adopted by other similar instrumentation tool, e.g., [69]. The C file is compiled and linked with the SystemTap runtime into a stand-alone kernel module.

The monitoring takes place when the kernel module is loaded. The output that is extracted from the module is sent to userspace through the *relayfs* or *netlink* modules, which are reliable and high performance transport means. In particular, in the current implementation these data are sent to a pipe and the event are consumed on-ilne to limit the amount of memory that can be consumed by the infrastructure. The monitoring ends when the module is unloaded from the kernel.

The sosmon *filtering and aggregation* module has been implemented by means of (i) the basic SystemTap PID and event type filtering facility, and (ii) a simple perl script that aggregates the events based on the chosen time unit (e.g, the second).

The described monitoring infrastructure produces the following indicators in each time unit (Table 3.1):

- **system call errors**: number of errors returned from system calls invocation;
- **signals**: number of OS IPC signals used for coordination or information purposes

(e.g., invalid memory access, process crash, loss of a socket connection, I/O data available);

- **task scheduling timeouts**: number of timeouts expired since a process released the CPU;
- **waiting time for critical section timeouts**: number of timeouts expired since a process (thread) started waiting for entering a critical section;
- **holding time in critical section timeouts**: number of timeouts expired since a process (thread) entered a critical section;
- **process (thread) creation/termination**: number of processes (threads) starting/terminating their execution;
- **disk I/O timeouts**: number of timeouts expired since the last read/write operation on disks;
- **socket I/O timeouts**: number of timeouts expired since the last read/write operation on a network socket;
- **disk I/O throughput**: the aggregate number of bytes transferred per time unit in operations on disks;
- **network I/O throughput**: the aggregate number of bytes transferred per time unit in operations on network devices.

Table 3.1: Monitored variables for Linux and Windows

Indicators	Linux	Windows
System call errors	YES	YES
OS Signals	YES	NO
Task scheduling timeouts	YES	YES
Waiting time for critical section timeouts	YES	NO
Holding time in critical section timeouts	YES	NO
Mutex/semaphore objects	NO	YES
Process (thread) creations/terminations	YES	YES
Disk I/O timeouts	YES	NO
Socket I/O timeouts	YES	NO
Disk I/O Throughput	YES	YES
Socket I/O Throughput	YES	YES

Windows Monitors

The monitoring infrastructure for Windows has been implemented to monitor most of the same OS-level indicators collected for Linux. As discussed above the Windows Reliability and Performance Monitor (WRPM) [87] tool has been used. WRPM is a monitoring tool available on both Windows desktop and Server releases. It provides several functionalities to system administrators: *(i)* the monitoring of the application and hardware performance in real time; *(ii)* the tracing of the performance-related events of both applications and services; *(iii)* the generation of alerts and reports; *(iv)* the triggering of predefined actions when thresholds are exceeded.

A high level view of the WRPM architecture is depicted in Figure 3.6. WRPM consists of three basic monitoring tools: Resource View, Reliability Monitor, and Performance Monitor.

The Resource View and Reliability Monitor tools (consumers in Figure 3.6) use data coming

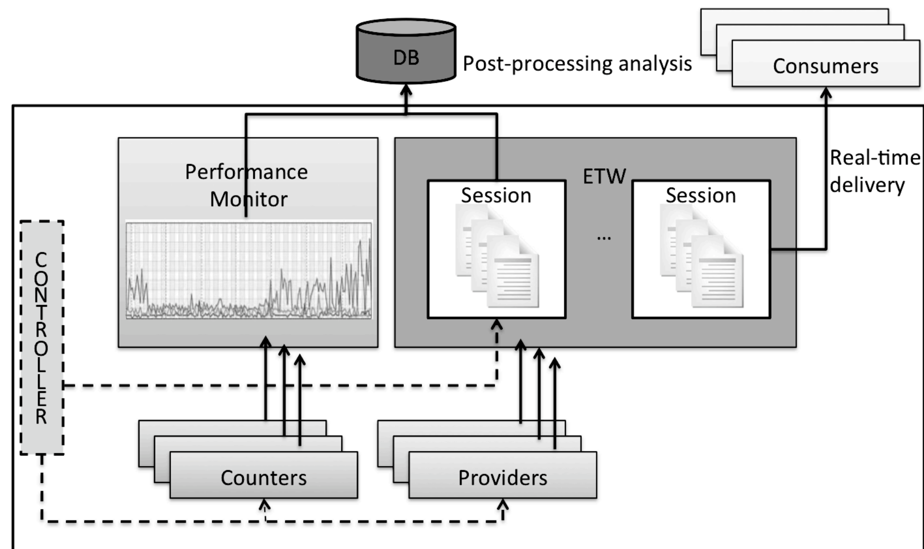


Figure 3.6: The monitoring infrastructure for Windows

from low level monitors (CPU, disk, network, memory) to give a run-time graphical view of the system status. The former provides information about system resources usage such as processes' memory consumption. The latter evaluates an index that help identifying "configuration failures", i.e. issues related to bad systems configurations.

The Performance Monitor tool allows viewing performance data (both at run time and from log files). It is based on a set of Performance counters (such as mutexes, semaphores, bytes read from devices, bytes write to devices, process elapsed time), which collect measures related to the system state or its activities. Counters can be embedded in the operating system or can be part of individual applications.

Another important tool, which can be exploited to observe OS-level indicators, is the

Event Tracing for Windows (ETW). ETW collects data from trace providers that report actions or events related to components of the OS (kernel level) or of individual applications (user-mode level). Events monitored by this tool include: Process (Thread) creations or terminations, system call, disk I/O, TCP/UDP network I/O, context switches. Output from multiple trace providers can be combined into a trace session. Then, the trace may be analyzed by one or more consumers by allowing large-scale server applications to write events with a minimum overhead.

Finally, data collection and logging is performed using Data Collector, the component that also groups data into reusable elements. Once a group of data collectors is stored as a Data Collector Set, operations such as alerts or scheduling can be applied to the entire set through a single property change. By means of the Performance counters and ETW (i.e., the Data Collector Set we built) OS-level events can be monitored as for Linux (see Table 3.1).

It is noteworthy that WRPM allows to collect a huge amount of data, but some events have not the same meaning that the corresponding ones in Linux OS. For instance, Windows signals are exploited from the OS only to send an interrupt to a process to notify events such as abnormal termination, floating-point error, illegal instruction, CTRL+C signal, illegal storage access, termination request; while, in Linux signals are widely used for Inter-Process Communication. Moreover, in Windows we could not monitor the time needed to acquire/release mutex/semaphore because it had implied to modified the underlying applications. Therefore, to satisfy the requirement that the monitoring infrastructure is application-independent, just the number of mutex/semaphore objects is monitored.

Finally, WRPM only provides the overall amount for disk I/O, network I/O and system call errors events, which includes all processes currently running. Therefore, using WRPM to monitor such events it is not possible to filter by PID and so it has been not possible to isolate the contribution of processes the target application.

3.3.2 The Statistical Predictor and Safety Margin Algorithm

Overview

The detection of suspicious indicators is based on the SPS algorithm [134]. SPS was originally designed to estimate the synchronization uncertainty interval of a software clock, namely, the interval of time that contains the real clock offset. In [134] the goal of SPS is to provide an uncertainty interval that most of the times contains the real clock offset.

In this work SPS is used in the reverse way. The nominal behavior of any monitored indicator is modeled with a random walk. An anomalous behavior is detected if the interval estimated by SPS with coverage c does not contain the actual value of the indicator. The coverage c represents the probability that, given no anomaly occurs, the adaptive thresholds contains the real value of the indicator.

At time t , when the function $getSPSThresholds$ is invoked (see Algorithm 1), SPS is requested to compute adaptive threshold for the specific indicator x_i. The adaptive upper and lower thresholds ($T_i^u[t+k]$ and $T_i^l[t+k]$, respectively) computed by the SPS algorithm at time t for the indicator x_i consist in a combination of left and right bounds:

$$T_i^u[t]{=}x_i[t-1] + P[t] + SM[t-1] \tag{3.1}$$

$$T_i^l[t]{=}x_i[t-1] - P[t] - SM[t-1] \tag{3.2}$$

These bounds are computed from three quantities:

1. the last value of the time-series ($x_i[t-1]$);
2. the output of a predictor function ($P[t]$);
3. the value of a the safety margin function at previous step ($SM[t-1]$).

The predictor and the safety margin functions are adapted from [134]. The predictor function provides an estimation of the behavior of the random walk process. Clearly the parameters of the random walk are unknown and may depend on *(i)* the specific monitored variable, *(ii)* the target components and *(iii)* the environment in which it operates (such as the operating system and other active processes).

The parameters to consider for the computation of the adaptive thresholds are estimated using the last m samples of the series, where m is the *memory* of the SPS module. The memory and the coverage of the SPS module can be set in the configuration file and can be specified for each monitored indicator.

As for the safety margin function, it aims at compensating possible errors in the prediction and/or in the collected measures. The safety margin is computed only when new measures arrive.

Figure 3.7 shows an example of the adaptive thresholds computed by means of SPS for the to reveal suspicious behavior in the *number of timeouts expired for the scheduling of processes*, which is one of the monitored indicators for the case-study presented in Section 5. Furthermore, the figure shows for reference the static thresholds evaluated by averaging the maximum and the minimum extracted in preliminary profiling experiments, which have

been performed *under a normal workload*. The dots in Figure 3.7 are the values measured under a high stressful workload; the two constant lines are the static thresholds, while the two continuous lines represent the adaptive thresholds. A fault is manually injected in the source code of a component and it is activated at second 160 when the component stops serving requests.

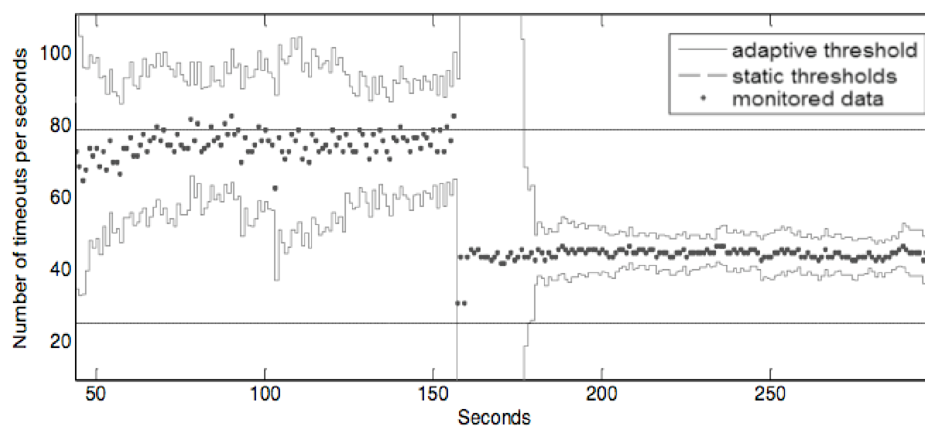


Figure 3.7: An example of adaptive vs static anomaly detection thresholds

Due to the stressful workload, some indicator values in the left part of the figures are higher than the static upper bound. Moreover, the worst-case thresholds do not allow the timely detection of the suspicious behavior caused by the activation of the injected fault. On the other hand, SPS provides thresholds useful to correctly signal the anomaly without producing a false positive before the activation of the fault.

It is noteworthy that after the detection of the anomaly the bounds computed by SPS are much higher. This is due to the way the SPS algorithm is designed and it is explained in the following. The adaptive interval becomes initially larger, but after few samples the

bounds are again closer to the indicator.

In the following subsections the details of the predictor and safety margin function are provided.

The Predictor Function

The behavior of the variation of the indicators (i.e. Δx_i) is modeled with the random walk; however, the parameters of such random walk are unknown and depend on specific factors among which the operating environments and the target system. To take into account evolving situations these parameters are computed by using the last m samples. Let s_d^2 be the sample variance of the indicator variation. From s_d^2 a safe upper bound σ_{sd}^2 to the population variance σ_d^2 is computed with probability c, which is the SPS coverage. Namely, $\sigma_{sd}^2 = s_d^2 \cdot (m-1)/\chi^2_{m-1,1-c}$ is the $1-c$ percentile of the χ^2 distribution using the sample variance of the m samples [135]. Then, the safe bound σ_{sd}^2 is used instead of σ_d^2 to obtain a Gaussian distribution with mean and variance being 0 and σ_{sd}^2, respectively, and to compute the diffusion coefficient D, which describes the random walk at hand [136]. D and the inverse of the Gauss error function $erf^-1(c)$, which indicates the interval underlying a cumulative probability c [135], are finally used to compute an upper bound, $V_i(t)$, to Δx_i in the time interval $[t-1,\ t]$.

$$V_i(t) = erf^{-1}(c)\sqrt{D(t-(t-1))}. \tag{3.3}$$

While $V_i(t)$ represents the upper bound to the variation of the indicator, $v_i(t-1)$ is the absolute value of the real variation of the indicator at time $t-1$. In case of stable indicators

this is supposed to be very small. However, this may be not the case because of several reasons among which non-stationary operating conditions. Hence, an additional term has been taken into account to compute the prediction, which is quantity $max(0, v_i(t-1))$.

The following expression is thus achieved to compute $P_i(t)$ that represents the desired prediction for the variation of the indicator x_i at time t:

$$
\begin{aligned}
P_i(t) = \int_{t-1}^{t} (V_i(x) + max(0, v_i(t-1))\mathrm{d}x \ = \\
= erf^{-1}(c)\frac{2}{3} \cdot \sqrt{D} \cdot (t-(t-1))^{2/3} + \\
+ \ max(0, v_i(t-1)) \cdot (t-(t-1))
\end{aligned}
\tag{3.4}
$$

The Safety Margin Function

The safety margin function is computed to compensate possible errors in the prediction and/or in the collected measures. As for the Δx_i case, a safe bound $\sigma^2_{sx_i}$ to the population variance $\sigma^2_{x_i}$ of the indicator x_i is computed with probability c starting from the last m samples of the indicator x_i. Similarly to eq. 3.5, $SM(t-1)$ is computed as:

$$SM_i(t-1) = \sqrt{2} \cdot erf^{-1}(c) \cdot \sigma^2_{sx_i} \tag{3.5}$$

3.3.3 The α-counter

The α-counter implements the heuristic that allows to distinguish between relevant and not relevant anomalies. As discussed above, this is based on the assumption that many suspicious indicators within a time window w, hereinafter the *combination window*, are easily evidence of anomalies for the monitored components, while isolated suspicious indicators

triggered over-time could be evidence for transient and not relevant situations such as, workload change.

However, the aforementioned assumption can be violated in the case a bunch of monitored indicators are more "critical" than others. Let consider the example of systems that suffer from software aging. In this case just one *out of control indicator*, e.g., the available free memory, may be sufficient to trigger the proper countermeasures (e.g., process restart or OS reboot).

For this reason the α-count heuristic has been implemented using several weights ω_i $(i = 1,\ 2, ...,\ K)$, which can be tuned to reveal different kinds of anomalies. The weight ω_i refers to the relevance of the indicator x_i in the detection process. An anomaly is detected if the combination exceeds a threshold G. When the weights are all equal, G accounts for the number of suspicious indicators that must be revealed during the combination window for detecting an anomaly. In such case, G is expressed as the number of suspicious indicators over the total number of indicators.

Depending on the type of components, the target system and the administrator knowledge it is possible to configure more heuristics by setting different sets of weights, combination windows and thresholds in the configuration file. In such a case, an anomaly is detected if at least one combination exceeds the corresponding threshold.

The pseudo-code of the α-count combinator is shown the Algorithm 2.

```
combineData (t)
input :
t, the current time
output:
flag, a boolean indicating if an anomaly has been detected
/* Initialize the configuration parameters needed by the α counter,
   i.e., the weights, the combination window and the threshold; these
   are then stored in the structure conf                             */
conf ← initConfiguration();
flag ← false;
foreach configuration c in conf do
    counter ← 0;
    weights ← c.weights;
    window ← c.window;
    threshold ← c.threshold;
    for i ← 1 to K do
        for j ← (t − window + 1) to t do
            if A[i][j] == 1 then
                counter ← counter + weights[i];
                break;
            end
        end
    end
    if counter > threshold then
        flag ← true;
    end
end
return flag;
```

Algorithm 2: The α-counter function invoked by sosmon daemon

3.4 Parameters Tuning and Computational Cost

It is worth to recall that the configurable parameters of the framework are *(i)* the global threshold G, *(ii)* the combination window w and *(iii)* the weights ω for the α-counter combinator; while, *(iv)* the memory m and *(v)* the coverage c for the SPS module. Different configurations can be defined for each indicator and the corresponding SPS module; furthermore, different settings (i.e., weights, combination window and global threshold) can be defined for the α-counter module in order to reveal different kinds of anomalies. The performance achieved by the detector depends on these parameters.

The global threshold G may be expressed as the number of suspicious indicators over the total number of indicators. The threshold needs to be tuned for the considered system and the type of anomaly. For instance, safety critical systems should prefer lower values for G because every suspicious indicator may be related to a non-tolerable misbehavior.

As for the combination window w, a small window decreases the probability to combine suspicious indicators that are related to the same non-random factor (e.g., the activation of a fault). A large window increases the probability to combine unrelated suspicious events.

The weights account for the relevance of indicators; when these have the same relevance, all weights are equal to $\omega_i = 1/K$. More in general weights are different, with $\sum_{i=1}^{K} \omega_i = 1$.

The memory of the algorithm m should be sufficiently large, e.g., $m = 30$, so that the assumptions used to estimate the random walk parameters are reasonably satisfied [134]. Small values for m make the computed thresholds more reactive to small changes, which may cause less accurate detection in term of false positives; the vice-versa, larger values

of m lead to high conservative thresholds that may decrease the detector ability to reveal anomalies.

As for the coverage parameter, decreasing c implies that SPS algorithm computes more unreliable thresholds. Hence, the probability that the estimated bounds contain the monitored indicator is smaller. On the other hand, if the coverage is increased, larger bounds for the monitored indicator are computed and the occurrence of suspicious behaviors may not be detected. The requirements of the system in which the anomaly detector is deployed should guide the tuning of this parameter.

In Section 5, a sensitivity analysis for the discussed parameters is provided using experimental data.

The computational cost of the detector depends on the computation of a population-weighted variance. Since the variance is computed using sums of the elements, the computational cost of the detector is linear with the number of samples. If we use accumulators to store the value of the sums in memory, the computational cost becomes constant. This last solution is obviously preferred when the detection framework has to be used run time with a large set of samples.

Chapter 4

Experimental Methodology

This chapter details the experimental methodology used to evaluate the effectiveness of the framework. First, the motivations that lead to the use of such a methodology are discussed. These encompass (i) the reduction of the cost of the analysis, in terms of time and resources, (ii) the sharing, the comparison and the reuse of the obtained results and (iii), from practitioners perspective, the simplification of the configuration and tuning of the framework.
The methodology consists of several steps: (i) Definition, where the objectives of the study, the system under test, the quantities to assess, the workload and the faultload are established; (ii) Planning, in which the experiments and the type of outcomes are defined; (iii) Execution, which encompasses the identification of the tools needed to perform the tests (e.g., the workload generator), and the carrying out of all the experiments; (iv) Analysis, which consists of the computation of the quantities to assess and the results analysis.

4.1 Motivations

Experiments are crucial for engineers who are involved in evaluating and choosing among different methods, techniques, languages and tools [137]. When dealing with heterogeneous, OTS-based and distributed mission-critical systems many challenges may jeopardize experimental campaigns, among which issues in measurement process (e.g., as discussed in Chapter 2, incomplete observation, semantic gap, intrusiveness), the analysis of huge amount of data and non-determinism [138].

It is well-known that the dependability assessment is an intricate task, especially when the experiments produce huge amounts of data, that may be also difficult to interpret. Many

tools can ease the data analysis. However, to assure the comparability and the consistency of the results, which are collected in different experiments and by different subjects, ad-hoc approaches need to be used.

The need of the methodology also arises from the requirement of providing a sound and thorough assessment of the proposed framework and to enable the comparison of its performance with other detectors of the same class (e.g., the ones proposed in [3]).

For these reasons, we have adopted a general methodology that is based on *(i)* the principles of design of experiments (DoE), *(ii)* dependability basics and *(iii)* data warehouse and OLAP techniques. This allows to perform experiments in heterogeneous systems, to share and cross-exploit raw data from different experiments, to analyze and to compare obtained outcomes in fair way and from different perspectives, and, finally, to increase the experimenter trust on the obtained results.

4.2 The Adopted Methodology

In this Section, the steps, which are summarized in the Figure 4.1), of the adopted experimental methodology are described. First, an overview of the method is provided; then, each step is described in a separate paragraph.

4.2.1 Overview

As discussed above, the methodology is based on *(i)* the principles of design of experiments, *(ii)* dependability basics and *(iii)* data warehouse and OLAP techniques.

Very briefly, DoE helps to minimize the experimental error and to get statistically significant answers in the investigation of systems or processes [139]. DoE principles state that

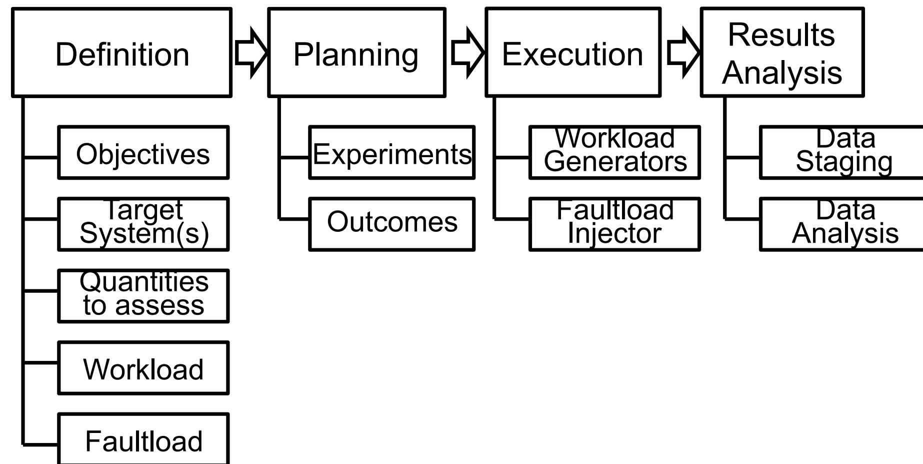


Figure 4.1: The experimental methodology

the first step in planning an experimental campaign is the formulation of a clear statement of the objectives of the investigation. Then, the experimenter needs to identify the *response variables*, also called *quantities to assess*, i.e., important measures that deserve deep analysis, and the *factors* of interest, which are those variables that can potentially affect the response variables but are not the primary objectives of the analysis. The value of the factors chosen in the experiments are called *levels*. A factor is said to be controllable if its level can be set by the experimenter, whereas the levels of an uncontrollable factor cannot be set, but only observed. When response variables, factors, and levels are identified, the test plan is completely determined by defining a list of experiments, called *treatments*.

Previous studies on dependability have highlighted the importance of two factors for experiments: a representative *workload* [140, 13, 141], i.e., the computational load that the target system has to process, and a representative *faultload* [21, 142, 143], i.e. the fault

types, the fault locations, the fault activation times that will be injected to the target system.

Indeed, many studies have observed how the the type of failures and the failure rates may vary dramatically depending on the workload being executed [11, 12, 13]. For instance, in [12, 14] and in our work [15, 16] the workload is observed to be an important software aging factor.

As for the faultload, the injection of representative faults is a valuable mean to test and validate fault tolerance mechanisms [142, 32, 5]. Several approaches and techniques are available both for hardware and software fault injection. In case of software fault injection the ODC classes of faults [31] can be used as staring point for selecting a faultload [21].

Main challenges regarding workload and faultload are *representativeness*, i.e., the capability to closely match the "real" world situation, and *portability*, the property to be not dependent upon any particular system or component, in other words, the capability to be easily implemented in many heterogeneous systems. This last property is more difficult to obtain since, for instance, in case of hardware faults, these could not have any effect using some specific hardware components.

Last but not the least, the need for a common way to store, organize, share and access experimental results is a well-known problem [144]. Centralized repositories that allow to store and to share important data about dependability experiments have emerged and are the natural choice to simplify this task [145].

Summarizing, the adopted methodology allows to perform experiments in heterogeneous systems, to share and cross-exploit raw data from different experiments, to analyze and to compare obtained outcomes in a thorough and fair way and, finally, to increase the experimenter trust on the obtained results.

4.2.2 Definition

This phase defines the *objectives* of the study and the *target system*, which is the entity that is investigated in the experiments, e.g., products, processes, theories. Moreover, it establishes the *quantities to assess*. These represent, according to the objectives of the analysis and to the target system, the most relevant output of the experiments. Then, all the important *factors*, especially those of interest for dependability experiments that may influence the quantities to assess, need to be taken into account. For instance, as previously discussed, important factors when dealing with dependability experiments are the *workload* and, especially when fault tolerance means need to be evaluated, the *faultload*.

Objectives

As suggested by DoE principles the first step for experiments definition is the formulation of a clear statement of the objectives of the investigation. This consists in a synthetic and precise description of the goals of the experiments that answers the following question: what is the issue/problem/theory the experimenter wants to address? This is fundamental to ensure that important aspects of experiments are defined before the planning and, especially, the execution take place [137].

The experiment objectives can be generally formalized by formulating two hypotheses:

H_0 – the null hypothesis – states that there are no real underlying trends or patterns in the treatments and the only reasons for differences in the observations are due to experimental errors.

H_α – the alternative hypothesis – is the one in favor of which the null hypothesis is rejected.

For instance, a possible hypothesis formulation may include the comparison between two different techniques (e.g., two detectors).

Target system

The system under test, or target system, is the entity under investigation for which the experimental campaign need to be designed. The target system may interact with other entities (e.g., other systems). For this reason, the target system has to be unambiguously separated from the environment in which it operates by identifying the system boundaries [8] – as discussed in 1.

It is worth to note that system boundary identification also allows to pinpoint the entities that may influence and alter the system behavior and so the quality of the experiments, namely, the so called *disturbing factors*.

Quantities to assess

The quantities to assess, commonly called response variables, are the most relevant output of the treatments. These are usually defined according to the primary effect under study, i,e, the objectives of the analysis, and to the target system.

The quantities to assess may be directly measurable by means of the instrumentation of the target system or may be indirectly derived from raw or intermediate measures. Quantities to assess should be as intuitive as possible, especially in case of well-known and countable metrics; for instance, in this dissertation, these may be defined using the relevant metrics defined in Section 2.3.

Workload and faultload

The workload is the computational load that the system under test has to process, which influences, as well as the hardware and the software components, the performance and the failure behavior of the target system.

The choice of workload can be considerably simplified by the existence of standard benchmarks that allow to exercise all more crucial functionalities of the target systems assuring a large degree of representativeness. Otherwise, other different types of workload can be considered for experimental purposes: *real workloads*, *realistic workloads*, and *synthetic workloads* [146].

Real workloads may be defined based on the knowledge of the software and the operating environment. It has to be privileged when the target system has to operate in particular

and specific scenarios. *Realistic workload* is an artificial workload composed of a subset of representative operations performed by the target system. This has the advantage to reflect real situations and to be more general than real workloads. *Synthetic workload* can be a set of randomly selected target system functionalities. This is easier to use but, its representativeness is doubtful [146].

Another important aspect that can be addressed at this stage is the workload characterization. This consists in a synthetic model of the essential features of the real load imposed to the system during operation. The characterization is indisputable useful to design synthetic workload generators that mimic the real workload and allow to design accurate capacity planning [147]. Moreover, as observed by recent study, workload characterization is also useful to accelerate the surface of dependability problems. Indeed, in recent studies [14] and in our work [15, 16] some workload factors, such as the amount byte sent or the type of request, have been demonstrated to influence the failure rate of software systems when considered into the experiments.

The faultload represents the residual faults that typically affect the target system during operation. A complete definition of the faultload consists in the description of the fault types, the fault locations, the fault activation times and their distributions for the target system [142].

The approaches and the techniques that can be used for inject repesentative faults generally depend upon answering three important questions: *what*, *where* and *when* to inject? Answering the first question allows to choose between techniques based on fault or

errors (i.e., the effect of fault activation) injection.

Answering the *where question* is important especially when dealing with OTS-based systems. For instance, the injection of faults at component interfaces or into the component internal modules has demonstrated to be not equivalent [21].

Choosing the triggers that activate the injection, i.e., the *when question*, is the last aspect to be addressed. The approaches typically adopted for the injection use the *first occurrence*, i.e., as soon as the injection code is is executed), the *time-based*, i.e., by means of timeout for injection, or a combination of the two. However, recent studies, e.g., [148], have observed that completely different failures can be obtained changing the injection triggers. A good discussion on how to select the proper workload and faultload can be found in [140].

4.2.3 Planning

This phase consists of *experiments*, in which the design of the experimental campaign is carried out, and *outcomes*, in which the structure and the organization of the outputs of the experiments, also called observations, are established.

It is worth to note the planning activities may also help to refine the key aspects defined at the previous phase of the methodology.

Furthermore, the successive phase, i.e., *analysis*, is closely related to the planning. Indeed, for instance, when statistical analysis techniques are used to draw meaningful conclusion, the capability to apply typical statistical tests strictly depends upon the type of design and upon the nature of the collected outcomes.

Experiments

The experiments to perform are defined when quantities to assess, factors of interest, and factor levels are identified. The design the test plan describes how the experiments are organized and it is formally determined by defining a list of tests, usually called *treatments*.

The goal of a proper experimental design is to obtain the maximum information with the minimum number of experiments [149]. Moreover, the experimenter has to carefully design the test plan in order to minimize the variability of the response variables due to the experimental errors.

The plan can be generated according to the objectives of the study using a statistical tool, like JMP (www.jmp.com), that allows to adhere to the general DoE principles, i.e., *randomization*, *replication* and, when needed, *blocking*.

Randomization is used to meet the requirement that the experiment outcomes are independent random variables –indeed, this is fundamental when statistical techniques are used since it is a crystal requirement.

Replication refers to the repeated execution of the treatments in order to have a more precise estimation of the error in the observations.

Blocking is used to systematically eliminate the undesired effect of a factor that the experimenter knows to influence the response variables but he/she is not interested in studying its effect. The test plan may adopt the blocking design to increase the accuracy of the experiment when such the effect of the factor is known and controllable.

When experiments are too costly, in terms of time and physical resources used, different design plans can be adopted such as the fractional experimental design. More details about the different design strategies can be found in [139, 150].

At this stage some preliminary experiments can be defined too. Usually, preliminary tests are mandatory to acquire some knowledge on the target system. For instance, *capacity tests* may be useful to determine the levels of some workload factors, such as the maximum number of requests in the time unit. Given a chosen system configuration, capacity tests consist in preliminary experiments in which the stress level imposed to the target system is progressively increased and performance, e.g., throughput and response time, are measured. When the performance reach a knee, this point represents the system's capacity. Preliminary tests may be also needed for determining the experiment minimal duration or the best testbed configurations [151, 15].

Outcomes

This stage is useful to define the structure and the organization of the experiment outputs. Indeed, text files or spreadsheet can be used for specific analysis, but they become clearly inadequate when the amount of data is huge, the required analysis is complex, and, especially, when the experimenter deals with heterogeneous data. For these reasons, the methodology exploits the data warehouse and OLAP approaches [145].

OLAP (On-Line Analytical Processing) refers to a set of techniques and systems that

are conceived to perform sophisticate analysis on a huge amount of data with the aim to support the management and the decision making process. As discussed above it has been recently adopted for dependability benchmarking and experiments [145, 140]. Systems that implement the OLAP technology allow to analyze a data set by different points of view in a easily and interacting way, without having to reorganize the data.

The central element of this approach is the repository, called data warehouse (DW), which is populated with the data coming from monitored system or process.

A fundamental aspect of the DW is the *multi-dimensional* abstract model used to represent and to organize the collected data. The multi-dimensional model includes two kinds of data: facts and dimensions. Facts are data that represent the specific activity of interest (e.g, the output of the treatments); facts have some specific properties, e.g., a numeric attribute or are countable, which are the subject of the analysis. As for dimensions, they represent different perspectives of analysis, e.g., factor of interests, blocking factors.

This model in data warehousing is usually implemented in relational database using the *star schema*.

In Figure 4.2, taken from [145], the dimensions are the tables, product, store, and time, while the facts contain the important information (e.g., total sales and profit) for a given product in a given store on a single day. It is worth to note that the facts are just numerical quantities and only becomes meaningful when referenced to the dimensions.

The star schema is the starting point for the implementation of the central repository to store the raw data, that are successively required by the OLAP approach to perform analysis and cross-checks with different perspectives in a general and efficient way. According to

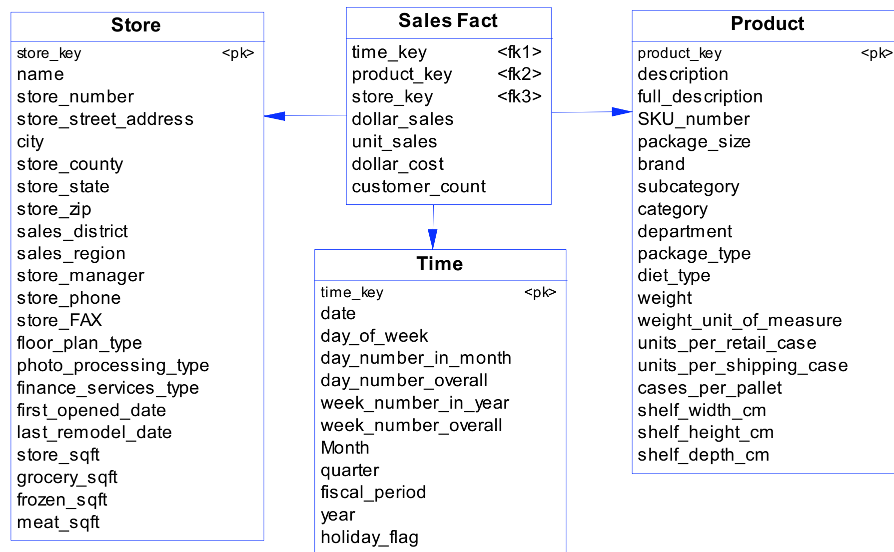


Figure 4.2: An example of star schema

the experimenter needs, the repository may be implemented using a relational database. Depending upon the amount of data that he/she needs to store, the repository may be supported by normal desktop hardware, which can handle up to $20GB$ of data, to more powerful workstations with parallel processing unit in order to manage terabytes of data.

COMPUTER SOCIETY

It is worth to note that the outcomes, defined at this stage, allows to identify in an unambiguous way the purposes (and the contexts) of the analysis. This may also help in refining the objectives, the quantities to assess and the other crucial elements of the experiments, such as the measures to collect.

4.2.4 Execution

In this phase the experiments defined at the previous steps are executed. Of course all the tools needed to execute the defined workload or, for instance, to injection realistic software faults should be identified and developed when not available. In this phase, *intermediate processing software* can also be developed, e.g., log parsers and database loaders.

Once all the tools are available the experimental campaign can be executed.

Monitors

The instrumentation infrastructure needs to be defined to accurately collect the information of interest by minimizing the impact on the performance and the behavior of the target system. Ready-to-use monitoring tools or existing operating systems probes should be preferred.

Workload generator and Fault injector

The execution of the defined workload is accomplished by the workload generator. It may be simply adapted from a driver application used during the testing phase that reproduces the typical load to the system, or it may be designed ad-hoc.

The injection of realistic faults should be performed by means of a tool that supports the fault types, the fault locations, and the fault activation times defined by the faultload.

Intermediate processing

The intermediate processing consists in the developing of tools that simplify the operations needed to evaluate the quantities to assess and to process the experiment outcomes, such as log parsers and filters.

4.2.5 Analysis

This is the last step of the methodology. It consists of the computation of the quantities to be assessed and the analysis of the results in order to draw meaningful conclusion from the experiments. For instance, possible analysis are: *(i)* the evaluation of the detector performance with different configurations, different target systems; *(ii)* the comparison of the performance with other detectors. Further types of analysis, which were not foreseen at the early stage of the definition phase, may be performed for different investigations.

Data staging

Data staging refers to the preliminary operations needed to get results ready for the analysis. Indeed, the central idea of using data warehouse (DW) and OLAP approach is to store the raw data collected during experiments in a multidimensional data structure (i.e., the data warehouse), where the share, the cross-exploitation of the results and the analysis can be done in an efficient and general way. Data staging is thus devised as a temporary processing phase that allows to gather and homogenize information from heterogeneous sources and to purify data by means of the detection (and correction) of corrupted or

inaccurate measures. These operations are usually needed since, as discussed in the previous Sections, the experiments can be conducted by different peoples, in different operating environments (e.g., Windows or Linux) and using several monitoring tools having different format and semantic for the collected data.

A general way of performing such step is by performing the following operations:

1. Extract the data collected by the monitoring infrastructure, such as parsing the application and systems logs;

2. Transform the extracted data so that are consistent with the semantic and the structure of the DW;

3. Develop special programs called *loaders* that populate the DW.

Data analysis

The analysis is performed to draw conclusions and/or recommendations according to the objectives defined in the definition phase. The output the analysis determines if the experiments may terminate or have to be extended.

Different techniques may be used. For instance, descriptive statistics, hypothesis tests, data mining analysis, are practical examples of techniques that can be exploited to uncover hidden patterns in collected data or to evaluate different solutions and theories.

Chapter 5

Experimental Results

In this chapter the performance of the framework are experimentally evaluated by applying the method described in Chapter 4. In particular, sosmon has been assessed against anomalies due to the activation of non-malicious software faults, that may lead to (i) active hang (i.e., CPU cycles are improperly consumed by processes/threads), (ii) passive hang (i.e., a process is indefinitely waiting for acquiring a shared resource), crash (i.e., abnormal termination of processes/threads) and performance degradation (e.g., smoothly degradation of the quality of the delivered service as perceived by the user). Furthermore, the performance of sosmon *are compared with another detector proposed in literature [3], which exploits tracing mechanisms and thresholds approach to detect anomalies.*

To this purpose a thorough experimental campaign has been designed, which has led to the execution of about 100 experiments using the SWIM-BOX as a case study. This is a prototype developed at SESM (a Finmeccanica company) that enables the interoperability of mission-critical systems in the ATM domain.

Results of experiments with two different operating systems, namely Linux Red Hat EL5 and Windows Server 2008, show that the detector is effective for mission-critical systems. Furthermore, the framework can be configured to select the monitored indicators so as to tune the level of intrusiveness. Finally, a sensitivity analysis of sosmon parameters is carried out to show their impact on the performance and to give to practitioners guidelines for its field tuning.

5.1 The SWIM-BOX Case Study

In this section, the methodology described in Chapter 4 is used to evaluate the performance of the framework for detecting the activation of non-malicious software faults. The experimental campaign has been designed to enable the fair comparison of sosmon performance with another detector proposed in literature [3], which exploits kernel-level facilities and training-based algorithm to detect anomalies.

5.1.1 Definition phase

Objectives. The objectives of the experiments are:

(i) to show the suitability of the framework for detecting the activation of software faults;

(ii) to quantitatively assess its performance by demonstrating the effectiveness under different workload, faultload and OSes;

(iii) to perform a sensitivity analisys in order to provide some indications on how the quantities involved in the configuration of the detector affect the detection; and

(iv) to evaluate the extent to which the monitoring intrusiveness impacts on the overall performance.

The performance of the framework needs also be comparable with another detectors proposed in literature. In particular, the detector proposed in [3], hereinafter the *Static algorithm*, has been chosen for comparison. There are two main reasons for such a choise. First, the *Static algorithm* has been designed with similar requirements of sosmon. Indeed, the main objective is to reveal the activation of software faults in mission-critical software systems that are based on OTS components. Second, the implementation of the framework shares with sosmom the using of *(i)* kernel-level tracing and *(ii)* count and thresholds approaches to reveal anomalies. However, differently from sosmon, the algorithm is based on a preliminary profiling and training phases. The purpose of the former is to collect OS-level indicators of the system behaviors and then to derive some descriptive values (i.e., minimum, maximum, mean and the variance). The latter phase is instead performed to derive most suitable thresholds that serve to the detector to reveal anomalies for the target

system. Of course, the Static algorithm needs to be trained for each different target system to work properly.

Target system. The detection activity is performed using an industrial case study, namely the SWIM-BOX®. This is a pilot system realized within the SWIM SUIT FP6 European project[1] for supporting interoperability of the future European ATM systems allowing integration of ATM systems.

SWIM-BOX is designed to offer several facilities: synchronous/asynchronous communication patterns (request/reply, publish/subscribe), security services (e.g., authentication, authorization, encryption) and distributed and transactional data storage.

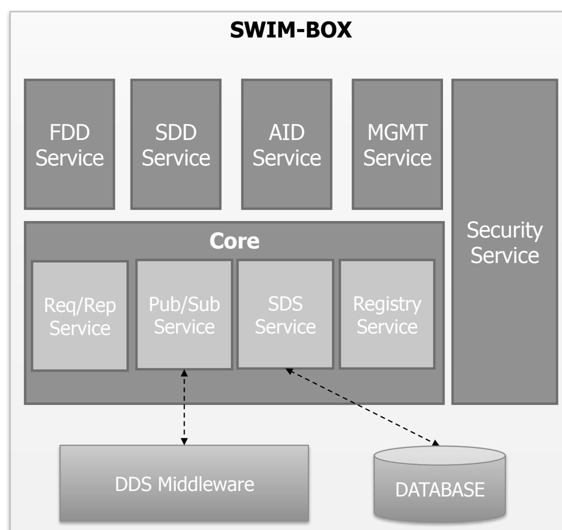


Figure 5.1: SWIM-BOX high-level architecture

[1] For the System Wide Information Management (SWIM) initiative see www.eurocontrol.int/programmes/system-wide-information-management-swim.

SWIM-BOX high-level architecture is depicted in Figure 5.1. SWIM-BOX has been designed by integrating ad-hoc implemented components, which are domain specific, such as Flight Data Domain service (FDD), Surveillance Data Domain service (SDD), Aeronautical Data Domain (ADD), and application-independent OTS components such as the Management (MNG) Service, the JBoss application server, the database and the Data Distribution Service (DSS) middleware, i.e., OpenSplice and RTI.

SWIM-BOX has been deployed on Windows and Linux platforms. The Linux testbed is equipped with Intel Xeon 2.5 GHz (4 cores) CPU, 8GB RAM, running Red Hat Enterprise Linux 5. The Windows testbed consists of an Intel Pentium 4 3.4 GHz (2 cores), with 3GB RAM, running Windows Server 2008.

The case study scenario encompasses two legacy entities, named the *Manager* and the *Contributor*, which cooperate for managing Flight Data Plans. The Manager is in charge of the following operations: creating Flight Objects (FOs), i.e., data including flight trajectory, date of departure, estimated date of arrival; updating FOs if necessary, and managing FOs life cycle. The Manager can publish a FO by sharing it with any interested ATM entity, i.e. a subscriber. Moreover, it can perform a handover operation when the FOs need to be managed by another ATM system.

The Contributor is the entity that receives, asynchronously, FOs information, and periodically reads all available FOs summaries; thus, it acts as subscriber. However, the Contributor may also modify the FOs communicating with the Manager by means of the SWIM-BOX facilities; in this case, it acts as publisher.

A (simplified) interaction scenario between the the Manager and the Contributor is

depicted in Figure 5.2.

By means of the SWIM-BOX facilities the Contributor requests FOs updates by subscribing to the FO update topic. Then, when the Manager publishes a FO, the SWIM-BOX needs to perform a set of checks to verify that the data published is consistent (e.g., it has unique flight object identifier) and the entity publishing the data has the right privileges. If the checks are successful, the SWIM-BOX takes care to distribute the updated data to all entities interested, i.e., the Contributor in such a case.

Quantities to assess. In Section 2.3, the most suitable metrics to evaluate the performance of detector have been discussed. The metrics used to characterize the anomaly detection in the considered scenario are: Coverage (C); Accuracy (A); average Query Accuracy Probability (aPA); average Mistake Rates ($a\lambda_M$); average Mistake Duration (aT_M).

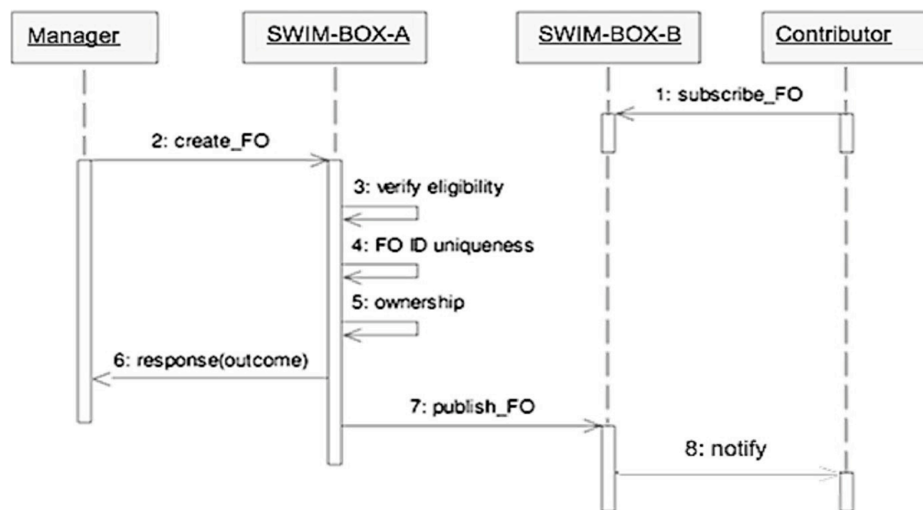


Figure 5.2: Interaction scenario for the case-study

These have been chosen, based on a review of the metrics used in literature described in Section 2.3 and in [152], as the best suited for the case study target system; other metrics could be used in different scenarios.

Workload. The experiments consist in the execution of several performance tests designed during the SWIM-BOX verification and validation phase. In particular, the simple operational scenario depicted in Figure 5.2 has been slightly extended in order to apply different workloads.

By means of the SWIM-BOX facilities the Contributor requests FOs updates. The Manager publishes for a given period of time many FOs at a randomly variable rate. This is useful to represent the arrival of *burst of messages*, i.e., a collection of message received into a single interaction, with respect to the SWIM-BOX, at different rates. Several bursts are then exchanged with varying number of messages per minute, messages per burst and number of bursts. The test ends when all the FOs are received and all accessory operations are completed (e.g., the Contributor and the Manager unsubscribe).

Faultload. To accelerate the collection of failure related data, we inject faults that mimic the activation of residual software faults during operation. In particular faulty experiments emulate realistic failures that occur at the data distribution middleware. These faults are usually activated during the exchange of messages between the Manager and the Contributor. Then, the resulting errors may propagate to the interface of the JBoss application server and may lead to:

- *hang* - the system appears to be running, but its services may be perceived as unresponsive because CPU cycles are improperly consumed (active hang) or because of indefinite waiting for resources that will never be released (passive hang);
- *crash* - a process (thread) ends its execution unexpectedly;
- *content* - FOs are incorrectly delivered;
- *performance degradation* - the quality of the delivered service, as perceived by the user in terms of response time, throughput, progressively degrades overtime.

The tool proposed in [143] has been used to automatically modify the source code and to inject the discussed faultload. When source code is not available, other approaches can be applied, such as injection at binary-level [21]. The goal of the detector is to reveal the activation of those faults before errors propagate to the interface causing more severe consequences.

We inject one fault per experiment using fault classes defined in the empirical study described in Duraes et al.[21]. Authors define the 17 most representative classes of software faults with respect to the Orthogonal Defect Classification [31].

According to them, the fault classes that most frequently occur in real systems are: "missing some small parts of algorithm" (MLPA) and "missing/wrong value assigned to a variable" (MVAV/WVAV). Tables 5.1 and 5.2 list the type and the number of the injected faults, respectively. It is worth to note that as for fault type, it has been used the classification provided in [21].

It should be noted that some faulty experiments may result in more that one type of

Table 5.1: Considered fault types

Fault Type	Acronym
MVAV	Missing Variable Assignment using a Value
MVAE	Missing Variable Assignment using a Value in Expression
WVAE	Wrong Variable Assignment using a Value in Expression
EVAV	Extraneous Variable Assignment using another Variable
MIA	Missing IF construct Around statement
MPFC	Missing Parameter in Function Call
WPFV	Wrong variable used in Parameter of Function Call
MFC	Missing Function Call
MLPA	Missing small and Localized Part of the Algorithm
MIEB	Missing If construct plus statement plus Else Before statement

failures, e.g., crash and content. Moreover, some faulty tests end correctly despite of the injection of a fault. This is usually common for fault injection experiments, since the introduction of a fault in the source code, does not guarantee the its activation. Furthermore, even if the fault is activated some internal redundancy can mask the resulting error(s).

As for the analysis of faulty tests, this work only considers the experiments that fail.

5.1.2 Planning phase

Experiments. Two sets of experiments are planned:

- *Golden runs* are experiments correctly executed, with FOs distributed with no errors returned to the Manager or the Contributor. These runs represent the correct behavior

Table 5.2: Source-code faults injected in the case study

ODC type	Fault Nature	Fault Type	Num. of Faults
Assignment	MISSING	MVAV	8
		MVAE	12
	WRONG	WVAE	8
	EXTRAN.	EVAV	2
Checking	MISSING	MIA	2
Interface	MISSING	MPFC	2
	WRONG	WPFV	2
Algorithm	MISSING	MFC	11
		MLPA	2
		MIEB	1
Total			50

of the system;

- *Faulty runs* consist in tests which fail because of an injected fault. The anomaly detector has to reveal the activation of the injected fault.

The experimental campaign consists of 35 golden runs and 20 faulty runs executed on Linux and Windows OSs, for a total number of 110 experiments. The test duration, expressed in minutes, is chosen randomly in the set $\{5, 20, 45, 90\}$.

As for the publication rate, which is expressed in publications per minute (*ppm*), it is chosen randomly in $\{20, 100, 300\}$ for each run; the time, expressed in seconds, between two bursts of messages is chosen randomly in $\{30, 300\}$. A rate of 200 (20) FOs *ppm* represents a high (low) stressful workload in a typical ATM system. The aforementioned values have been defined with the support of a ATM domain experts.

As for the configurations of the detector, the following parameter values are considered: SPS coverage $c \in \{0.9, 0.99, 0.9999\}$; memory $m \in \{10, 20, 40\}$; combination window $w \in$

$\{2, 5, 10, 20\}$; $G \in [1/14, 2/14, ..., 13/14]$ for Windows OS, and $G \in [1/23, 2/23, ..., 22/23]$ for Linux OS.

As for the Static algorithm, the worst-case (lower and upper) thresholds are configured using the following bounds: $[min, max]$; $[\mu - \sigma, \mu + \sigma]$; $[\mu - 2\sigma, \mu + 2\sigma]$, where: min and max are the minimum and the maximum values in the data set derived from the profiling phase; μ and σ are the mean and the standard deviation, respectively.

Outcomes. Experiment outcomes are organized by means of the star schema illustrated in Figure 5.3. This intuitive model organizes experiment outcomes in facts and dimensions [145]. It is worth to recall that facts generically represent a specific business or process activity; while, dimensions are interesting analysis perspectives.

In the considered scenario facts refer to the events collected by means of the kernel-level facility. Dimensions are instead specific characteristics of the experimental setup and are related to the different perspectives of analysis of the facts. The following dimensions are considered:

- **Target System**. Characteristics of the testbed such as the OS, the number of CPU, the CPU speed, the amount of RAM and the disk speed;
- **Scenario**. The description of the testing scenario, e.g., the number of entities involved in the communication and their role (i.e. Manager or Contributor). This dimension is useful to specify different configurations for the same Target System;
- **Events**. The type of event collected by the tracing facilities, from which indicators

are derived;

- **Workload**. Characteristics of the adopted workload. As discussed in the definition phase, several bursts are exchanged at different number of messages per minute rate, messages per burst and bursts per experiment;

- **Faultload**. This dimension accounts for the type of the injected fault and the software component target of the injection;

- **Run**. Information related to the execution of the test, e.g., start time, end time, sample period.

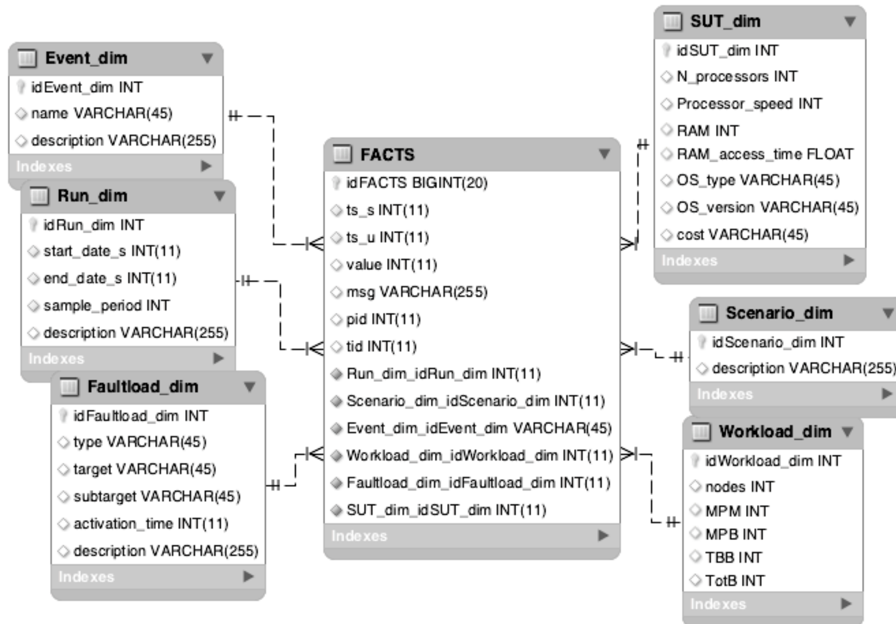


Figure 5.3: Data repository designed for OLAP analysis

5.1.3 Execution phase

The experiments analysis and the result analysis have been performed through two different steps. First, the golden runs and the faulty runs defined in the planning phases of the methodology have been executed and the raw data collected by the monitors inserted in the target system have been temporary stored in a set of csv files. This set of data represent the trace of the system and the input data of the two considered detectors. The algorithms of the detectors have been applied in a post-processing phase rather that on-line. In this way it has been possible to simulate on-line behaviour of both the detectors varying their operating parameters.

The second step consists in post-processing analysis in order (i) to compute the thresholds (adaptive in case of sosmon and fixes in case of Static algorithm) and (ii) to obtain the time in which the anomalies would have been detected.

As for sosmon, the thresholds are computed receiving as input the collected traces and the detectors parameters (i.e., c and m for sosmon). As for the detection time, it has been computed using the indicators, the thresholds and the other detector parameters, i.e., the combination window w and the global threshold G.

Considering the different detector configurations we analyzed 133,650 sets of data, since for each run the anomalies have been evaluated by varying all the detector parameters, i.e., $c \in \{0.9, 0.99, 0.9999\}$, $m \in \{10, 20, 40\}$, $w \in \{3, 5, 10, 20, 40\}$, with $w \leq m$ and $G \in \{1/14, 2/14, ..., 13/14\}$ for Windows OS, and $G \in \{1/23, 2/23, ..., 22/23\}$ for Linux OS.

As for the Static algorithm, a preliminary profiling phase is required to tune its parameters for the target system. We divided the available data in three sets: *i)* the *training set* (we use a subset of about 20% of runs) to evaluate the detector parameters; *ii)* the *validation set*, about 30% of the runs, to select the best configuration evaluated in the training set; and *iii)* the *testing set*, namely the remaining 50% of runs, to evaluate the performance of the algorithm.

For each monitored indicator, three different thresholds are used (which are, as previously discussed, $[min, max]$; $[\mu - \sigma, \mu + \sigma]$; $[\mu - 2\sigma, \mu + 2\sigma]$); then, the configuration that gives the best performance of both training and validation sets is chosen [103]; finally, the Static algorithm is applied by using the chosen configuration. As for the training set, 23,760 sets of data have been analyzed.

As for the outcome of faulty runs, Table 5.3 summarizes the distribution of failures observed by inspecting logs and the data received by the Contributor.

Table 5.3: Distribution of failures observed in faulty runs

Failure Type	**Distribution**
Passive Hang	50%
Active Hang	36%
Crash	12%
Content	2%

5.1.4 Analysis Phase

This section first introduces the preliminary operation, i.e., the data staging, needed for results analysis; then, the results of performance evaluation, for Linux and Windows platforms are shown by carrying out the comparison with the Static algorithm. Finally, a sensitivity and an intrusiveness analysis are discussed to show how much the parameters configuration and the intrusiveness of the monitoring infrastructure influence the performance of sosmon.

Data Staging. The purpose of data staging is to get experiment outcomes ready for the analysis. It is used to gather and homogenize data from heterogeneous sources and to perform operations such as data aggregation and the discovery (and correction) of corrupted or inaccurate data.

An OLAP approach [145] is used *(i)* to store and organize the measurements collected during experiments in a multidimensional data structure, and *(ii)* to analyze the collected data. A single repository is used to store experimental data. The repository is designed according to the output of the planning phase, by using the star schema shown in Figure 5.3 and allows to store data coming from different target systems and experiments.

Performance under Linux and Windows OS

It is useful to remind that the proposed detector has four configuration parameters: coverage c, memory m, combination window w, global threshold G, while the Static algorithm parameters are fixed after the training phase.

In Figures 5.4 and 5.5 the sosmon is compared, for both Linux and Windows, with Static Algorithm in terms of coverage, accuracy, average mistake time, average mistake rate and average query accuracy probability. The mistake time and mistake rate values are normalized with original values on the top of the bars. The best performance of sosmon is achieved by choosing the configuration that achieves the maximum Coverage and then maximum accuracy metrics. These configurations are for Linux and Windows $(c, m, w, G) = (0.9999, 20, 5, 0.2)$ and $(c, m, w, G) = (0.99, 40, 20, 0.4)$, respectively.

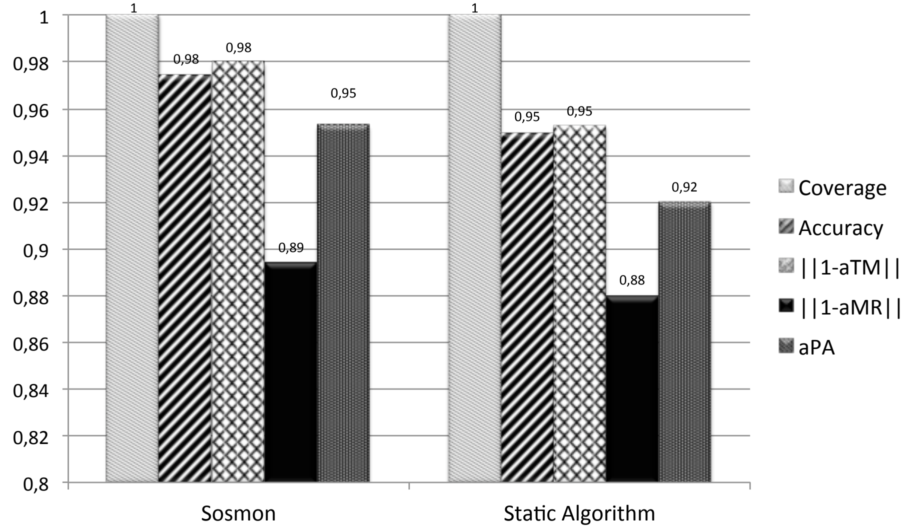


Figure 5.4: Best experimental results in Linux using coverage $c = 0.9999$, memory $m = 20$, combination window $w = 5$ and $G = 0.2$

It is worth noting that for accuracy metrics, i.e., aTM, aMR the lower is the better, vice versa for aPA,A and C.

Table 5.5 summarizes the performance for a subset of configurations, which are selected using the insights provided in previous work comparing the performance of SPS algorithm

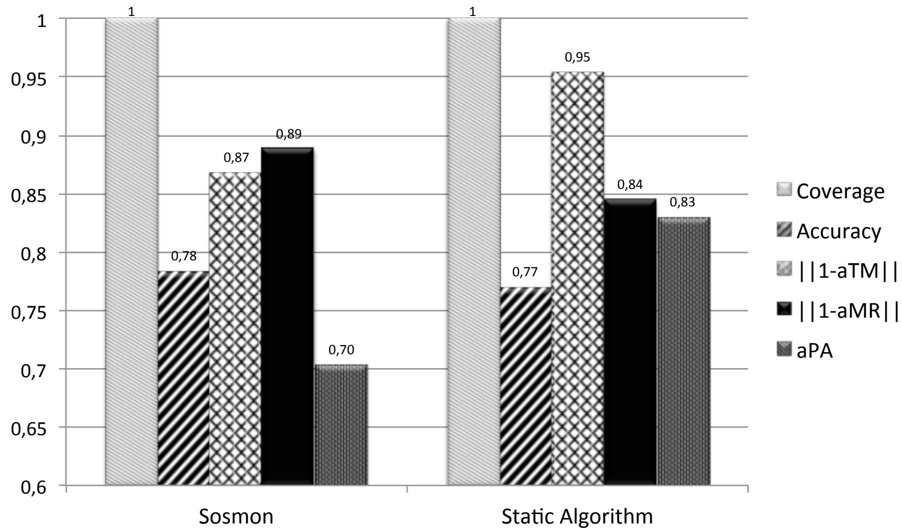


Figure 5.5: Best experimental results in Windows using coverage $c = 0.99$, memory $m = 40$, combination window $w = 20$ and $G = 0.4$

varying the memory and the coverage parameters[134].

The results provided in Table 5.5 reveal that a good trade-off between Coverage and accuracy metrics can be obtained with different configurations. In particular, the configurations with $c = 0.9999$, $m \in \{20, 40\}$, $w \in \{10, 20\}$ and $G \in \{0.1, 0.2\}$ for both Linux and Windows OSs ($4th$ row and the $5th$ row in Table 5.5).

As for the Static algorithm, the results (see the last two rows of Table 5.5) show that the detector can reveal all injected faults. However, the accuracy metrics A and aPA have a large variation from Linux to Windows: 0.95,0.92 and 0.77,0.83, respectively.

Despite, the overall satisfactory results –indeed, the performance are comparable with the Static Algorithm, which needs to be trained for each environments– the Coverage and the accuracy ascertain to be better in Linux than in Windows experiments.

Table 5.5: Coverage and accuracy of the detectors

(c,m,w,G)	OS	C	A	aTM	aMR	aPA
0.9999, 20, 5, 0.1	Lin	1	0.91	3.00	0.035	0.88
	Win	0.7	0.83	3.66	0.047	0.83
0.9999, 20, 5, 0.2	Lin	1	0.97	2	0.021	0.95
	Win	0.6	0.92	3.02	0.027	0.91
0.9999, 20, 10, 0.2	Lin	1	0.94	5	0.014	0.90
	Win	0.9	0.85	6.38	0.025	0.84
0.9999, 40, 20, 0.1	Lin	1	0.82	10.7	0.019	0.74
	Win	0.9	0.69	15	0.024	0.62
0.99, 20, 10, 0.2	Lin	1	0.72	9.27	0.037	0.68
	Win	0.7	0.85	7.00	0.025	0.82
0.99,40, 20, 0.2	Lin	0.91	0.73	14.52	0.016	0.72
	Win	1	0.78	13.18	0.022	0.70
0.99, 20, 20, 0.4	Lin	0.82	0.82	9.11	0.020	0.81
	Win	1	0.75	12.63	0.023	0.70
Static Algorithm	Lin	1	0.95	4.75	0.024	0.92
	Win	1	0.77	4.57	0.031	0.83

The performance under Windows OS can be explained considering that the monitoring infrastructure, i.e., WRPM, does not allow to fully isolate the contribution of the components of interest. Indeed, as explained in Section 3.3.1, it is not possible to filter some collected event traces. For instance, WRPM is not able to filter disk/network I/O throughput for a subset of processes, since it provides only the overall throughput (including all running processes). The same holds for system call errors and semaphore counters. For this reason, an anomaly occurring in some processes may be masked by the behavior of other components. Thus, it is the quality of the monitoring infrastructure that makes more difficult to reveal anomalies in the Windows testbed.

Despite such intrinsic difficulties, Table 5.5 also shows that, as for Windows, sosmon

can reveal relevant anomalies (i.e., those related to the activation of the injected faults) by slightly modifying the detector configuration that achieve full Coverage in Linux experiments. In particular, by decreasing the coverage parameter c and by increasing the memory parameter m the detector achieves full Coverage (see the $3rd$ to last and the $2nd$ to last rows of Table 5.5). The price to pay is a lower accuracy. The configurations giving full Coverage and acceptable accuracy metrics have $c = 0.99$, $m = 40$, $w = 20$ and $G = 0.2$ or 0.4.

In summary, the proposed detector has good performance on both Linux OS and Windows OS and the results are comparable with the Static algorithm, which is separately trained for each environment. Furthermore, the accuracy metrics of sosmon can also be better than the Static Algorithm, e.g., by considering the configurations $0.9999,\ 20,\ 5,\ 0.2$ and $0.99,\ 40,\ 5,\ 0.2$ for Linux and Windows, respectively.

Sensitivity analysis

The sensitivity analysis of sosmon parameters is carried out to show their impact on the performance and to give to practitioners guidelines for its field tuning. First, some observations on the relations between global threshold G and coverage c are provided; then, memory m and combination window w are considered.

It is worth to note that to ease the comparison of performance in graphical results, again normalized values of the aTM and aMR (i.e., $1 - \|aTM\|$ and $1 - \|aMR\|$) are shown. Hence, in the following figures all the considered metrics have a $[0, 1]$ range with

the optimum in 1.

Varying global threshold G and coverage c

Figure 5.6 shows experimental results varying the global threshold G and fixing the parameters $c = 0.9999$, $m = 20$, $w = 5$. The specific values are for Linux, but similar results are achieved in Windows experiments. It is possible to observe a trade-off between C and the other accuracy metrics. Sosmon can reveal the activations of faults with smaller values for G at the price of a lower accuracy. On the contrary, not all activations of injected faults are detected with larger values for G.

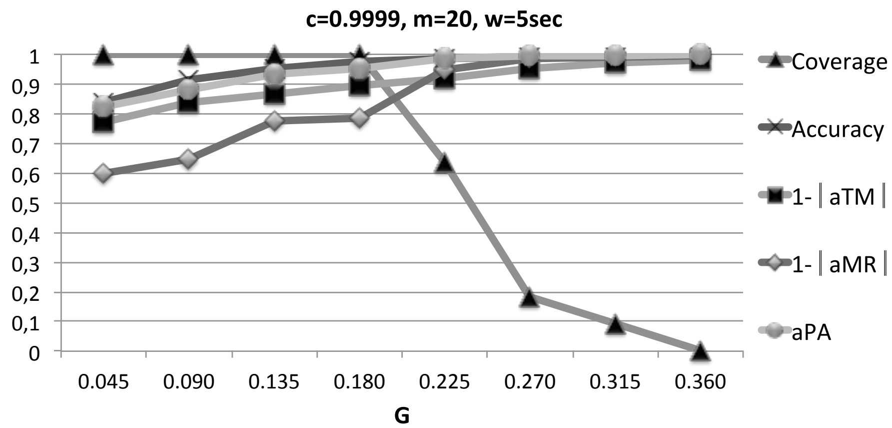


Figure 5.6: Experimental results for Linux using coverage $c = 0.9999$, memory $m = 20$, combination window $w = 5$ and varying global threshold G

The trend of Figure 5.6 is confirmed using $c = 0.9$, as shown in Figure 5.7. However, the thresholds G are much larger. This is due to the decrease in the SPS coverage parameter. In fact, with a lower c the SPS module computes more unreliable bounds; hence, more alarms

are triggered and the global threshold G has to be larger to increase the accuracy.

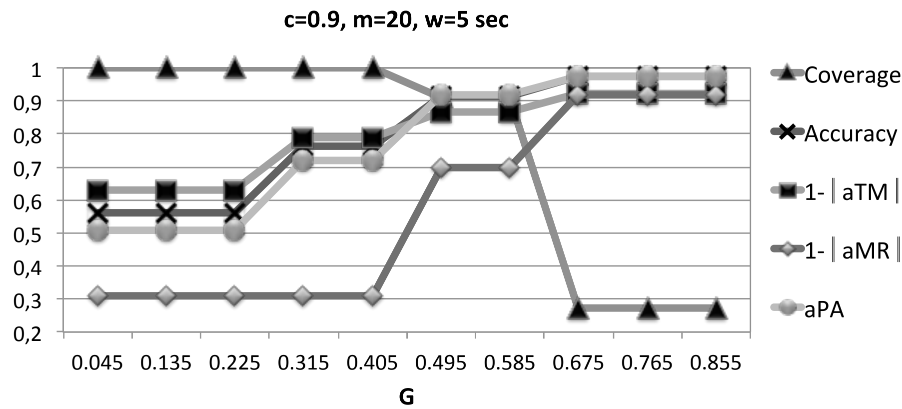


Figure 5.7: Experimental results for Linux using coverage $c = 0.9$, memory $m = 20$, combination window $w = 5$ and varying global threshold G

Varying memory m

Figure 5.8 shows how the memory parameter m affects the detection. The other sosmon parameters are so fixed: $c = 0.9999$, $w = 5$, $G = 0.2$.

Results show that better performance is achieved with $m = 20$. It is useful to recall that small values for m mean that the adaptive thresholds are computed using few data samples. Thus the computed thresholds are less accurate. This explain the higher number of false positive and so the worse accuracy metrics.

Moreover, since so-computed thresholds are very reactive to small changes in the monitored indicators, relevant anomalies could be gone undetected leading to low coverage. The behavior improves when the memory depth is increased since thresholds become both

smooth and sensitive. However, a further increasing in the memory causes the SPS algorithm to compute very large, highly conservative and non reactive thresholds that, for the target system implies a lower Coverage.

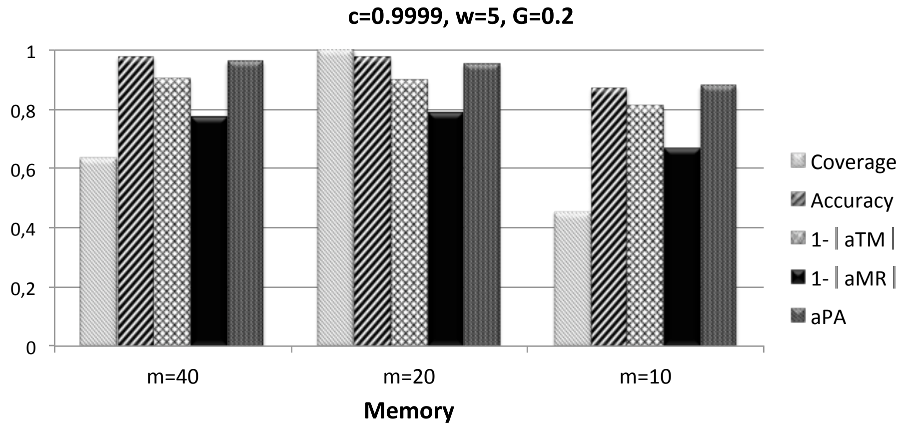


Figure 5.8: Experimental results for Linux using coverage $c = 0.9999$, combination window $w = 5$, global threshold $G = 0.18$ and varying memory m

Varying combination window w

Figure 5.9 reports experimental results varying combination window w, and by fixing other parameters to $c = 0.9999$, $m = 20$, $G = 0.2$.

Results shows that increasing w implies better Coverage. On the contrary, a smaller w increases the accuracy metrics at the price of missing the activation of some faults. A good trade-off for between Coverage and accuracy metrics is achieved with $w = 5, 10$.

Intrusiveness analysis

As discussed in 2, one of the main problems of monitoring infrastructure is its intrusiveness and overhead. Indeed, any modifications to the operating environment (e.g., additional

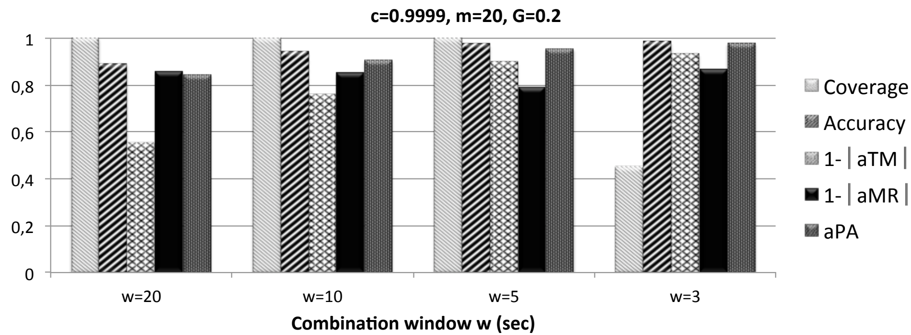


Figure 5.9: Experimental results for Linux using coverage $c = 0.9999$, memory $m = 20$, global threshold $G = 0.2$ and varying w

components, OS patch) may influence the mission of the system. Therefore, reducing the intrusiveness and the overhead of the proposed framework is a key-factor to make it suited for mission-critical systems.

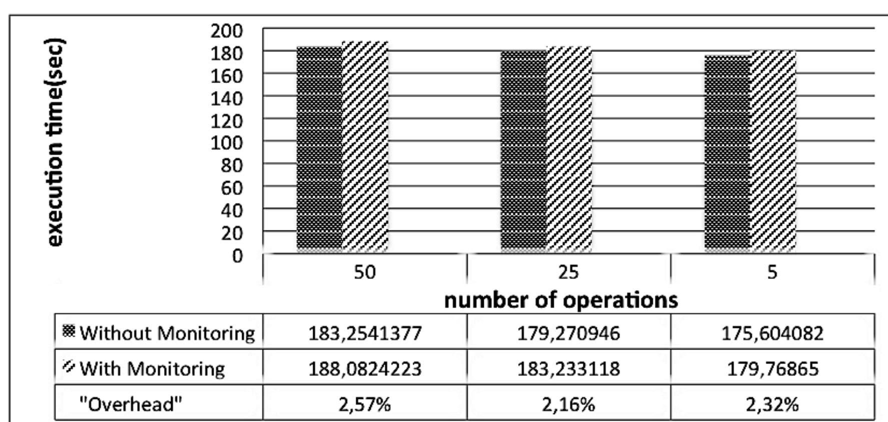
To this aim, since the principal modifications on the target system are done at OS by enabling the kernel-level probes, two further experiments have been performed. First, the we have evaluated the monitoring overhead on the target system when all the probes are enabled. Then the performance of the detector has been compared by using a limited number of probes.

Overhead evaluation Monitoring introduces various overheads, which arise from the measurement, collection, handling, and processing of the monitoring data. The overhead is usually a function of the type of data collected and the collection rate.

The overhead of sosmon framework has been measured by comparing the execution time

of the performance tests, with and without the monitoring infrastructure. In order to be more confident on the results the number of Flight Object operations per seconds has been progressively increased across tests.

Figures 5.10 shows results of this experiments. It should be noted that the overhead has been lower than 2% in every case, even during most intensive workload periods.



	50	25	5
Without Monitoring	183,2541377	179,270946	175,604082
With Monitoring	188,0824223	183,233118	179,76865
"Overhead"	2,57%	2,16%	2,32%

Figure 5.10: Overhead for the SWIM-BOX varying the invocation period of the operations

Performance varying the number of probes

Different sets of indicators have been used: 10 sets of 9 indicators, 20 sets of 4 indicators and 40 sets of 2 indicators. For each set indicators have been chosen randomly using an uniform distribution.

Experimental results for Linux and Windows are summarized in Figure 5.11. For the sake of simplicity, only mean values and variance of Coverage and Accuracy are shown for each considered set size, i.e, 9, 4 and 2. The mean Coverage and Accuracy are on the left y-axis; variance is on the right; the number of monitored indicators is on the x-axis.

The results under Linux show that reducing the number of monitored indicators does not affect Coverage. In particular, even if only two indicators are collected, the framework is able to detect the activation of all the injected faults. The price to pay for the limited instrumentation is a worse accuracy. In fact, the mean values for all the sets of Coverage is 100% but the mean Accuracy is 76%. The smaller variance highlights that all sets of randomly-chosen indicators give similar results.

The results under Windows (dotted lines in Figure 5.12) reveal that when the number of monitored indicators decreases, only specific subsets of indicators allow to detect all the injected faults. Indeed, the mean Coverage decreases is 86% and the mean Accuracy to 72%; while, the variance of the experimental results increases. This means that using a low intrusive monitoring infrastructure in Windows is possible at the prize of monitoring the most appropriate indicators.

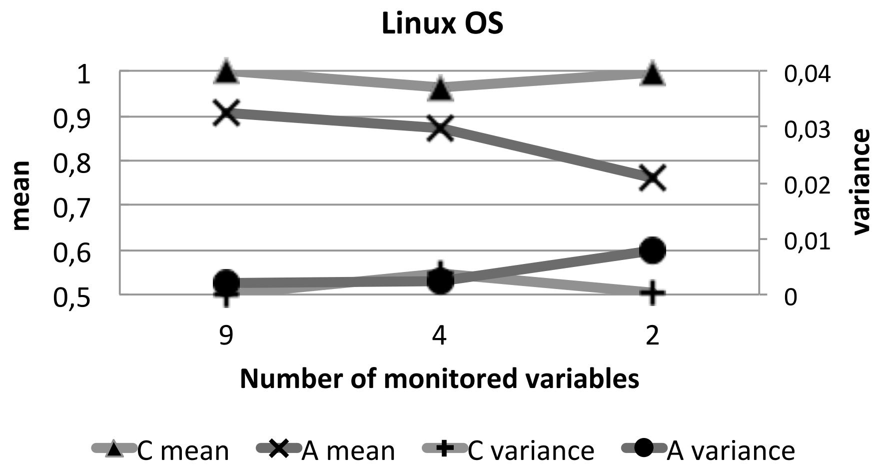


Figure 5.11: Results of sensitivity analysis of Coverage and Accuracy to the number of monitored indicators in Linux

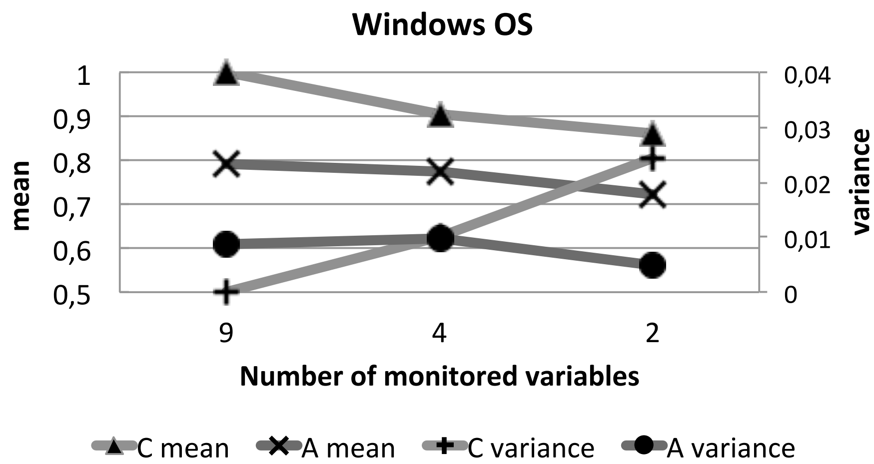


Figure 5.12: Results of sensitivity analysis of Coverage and Accuracy to the number of monitored indicators in Windows

Conclusion

This dissertation addressed the important problem of on-line anomaly detection in mission-critical software systems that are made of several OTS components. The improvement of detection mechanisms is fundamental to achieve better fault tolerance coverage, which in turn leads to an overall improvement of the system dependability. In particular, the work investigated the effectiveness and efficiency of the OS-level anomaly detection approach, which is particularly suited for OTS-based systems since it does not require to modify the application components. The work also contributed to the state-of-the art with a novel detection framework, i.e., *sosmon*, that relies on the following key characteristics: *(i)* the possibilities to deploy this mechanism on different OTS systems, working at the OS-level and, hence, without modifying the monitored application components; *(ii)* exploiting internal algorithms that make use of statistical observations on the monitored indicators to deal with non-stationary and variable operating conditions; *(iii)* the possibilities of tuning the framework according to the type of dependability requirements of the systems.

To investigate the suitability of OS-level detection approach this dissertation analyzed the following important aspects: *(i)* is the approach effective under different worklaod, fault-load (which lead to anomalies) and under variable and non-staionary operating conditions?

(ii) is the approach applicable with different operating systems? *(iii)* how much the intrusiveness of the OS-level monitoring infrastructure influences the detection performance? and its overhead can be limited without hampering detection ability?

The effectiveness of the approach was assessed by quantitatively evaluating the performance of the proposed framework through extensive experiments based on fault injection conducted on an OTS-based mission-critical system for Air Traffic Management (ATM), i.e., the SWIMBOX. The results showed that the proposed framework exhibits good performance (with respect to the detection needs for the target system) – indeed, 100% Coverage and about $95 - 80\%$ of Accuracy can be obtained.

The suitability of the approach under different OSs was shown by implementing the framework for two operating systems, namely Red Hat EL 5 and Windows Server 2008, in which the SWIM-BOX was deployed. The analyses showed that the framework performs better on Linux since in the Windows environment it is more difficult to reveal some indicators' anomalies. Indeed, it was not possible to isolate the contribution of the components of interest with the Windows monitoring infrastructure. Nevertheless, the activation of all the injected faults was revealed.

The performance of the framework was also compared with another OS-level detector proposed in literature [3] that exploits preliminary training phase to reveal anomalies. Results showed that the proposed detector has similar performance of the latter, which needs to be trained in each operating environment.

Finally, the work explored the framework performance varying the level of intrusiveness of the monitoring infrastructure to analyze whether reasonable performance can be obtained

through reducing the number of monitored indicators. Results showed that the performance is acceptable even when the set of OS-level indicators is very limited (e.g., to only 2 or 4). However, while the selection for Linux does not appear critical and could be done almost randomly or selecting those indicators easier to monitor, in the case of the Windows environments a proper selection requires a careful analysis since a bad choice might impair the success of the entire framework.

By summarizing, the ability of the presented anomaly-detector framework to adapt its behavior to different working scenarios and its low intrusiveness, enriched by the encouraging results obtained in the experimental campaign, lays the ground towards practical deployment of sosmon in many real systems (varying from large scale complex and mission-critical OTS-based software systems to smaller and less critical systems), which have to deal with unreliable OTS components. Indeed, the framework can be applied to a variety of circumstances and applications in a much more efficient and cheap way than instrumenting the application components themselves; in fact, instead of re-instrumenting each OTS item and application each time, it will be just necessary to tune a ready-to-use framework.

Bibliography

[1] M. R. Lyu, Ed., *Handbook of Software reliability Engineering.* Hightstown, NJ, USA: McGraw-Hill, Inc., 1996.

[2] V. Chandola, A. Banerjee, and V. Kumar, Anomaly detection: A survey, *ACM Comput. Surv.*, vol. 41, no. 3, pp. 1–58, Jul. 2009.

[3] G. Carrozza, M. Cinque, D. Cotroneo, and R. Natella, Operating System Suppor t to Detect Application Hangs, in *Proceedings of the Second international conference on Verification and Evaluation of Computer and Communication Systems*, 2008, pp. 117–127.

[4] (2012). [Online]. Available: http://googleenterprise.blogspot.it/2011/09/what-happened-wednesday.html. March 2013

[5] D. Siewiorek, R. Chillarege, and Z. Kalbarczyk, Reflections on industry trends and experimental research in dependability, *IEEE Trans. on Dependable and Secure Computing*, vol. 1, no. 2, pp. 109–127, april-june 2004.

[6] R. Chillarege, Understanding Bohr-Mandel Bugs through ODC Triggers and a Case Study with Empirical Estimations of Their Field Proportion, in *IEEE Third International Workshop on Software Aging and Rejuvenation (WoSAR)*, 2011, pp. 7–13.

[7] J. C. and M. M, Progress achieved in the research area of Critical Information Infrastructure Protection, IST-FP6 Projects CRUTIAL, IRRIIS and GRID, Tech. Rep., March 2007.

[8] A. Avizienis, J. Laprie, B. Randell, and C. Landwehr, Basic Concepts and Taxonomy of Dependable and Secure Computing, *IEEE Trans. on Dependable Secure Computing*, vol. 1, no. 1, pp. 11–33, 2004.

[9] N. Falliere, L. O. Murchu, and E. Chien, W32.Stuxnet Dossier, Symantic Security Response, Tech. Rep., Oct. 2010.

[10] [Online]. Available: http://www.oig.dot.gov/library-item/3911. March 2013

[11] P. Koopman and J. DeVale, The Exception Handling Effectiveness of POSIX Operating Systems, *IEEE Trans. on Software Engineering*, vol. 26, no. 9, pp. 837–848, 2000.

[12] K. Vaidyanathan and K. 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*, 1999, pp. 84–93.

[13] R. K. Sahoo, A. Sivasubramaniam, M. S. Squillante, and Y. Zhang, Failure Data Analysis of a Large-Scale Heterogeneous Server Environment, in *Proceedings of the 2004 International Conference on Dependable Systems and Networks*, 2004.

[14] R. Matias, P. Barbetta, K. Trivedi, and P. Filho, Accelerated Degradation Tests Applied to Software Aging Experiments, *IEEE Trans. Reliability*, vol. 59, no. 1, pp. 102 –114, march 2010.

[15] A. Bovenzi, D. Cotroneo, R. Pietrantuono, and S. Russo, Workload Characterization for Software Aging Analysis, in *IEEE 22nd International Symposium on Software Reliability Engineering (ISSRE)*, 2011, pp. 240 –249.

[16] A. Bovenzi, D. Cotroneo, R. Pietrantuono, , and S. Russo, On the Aging Effects due to Concurrency Bugs: a Case Study on MySQL, in *IEEE 23nd International Symposium on Software Reliability Engineering (ISSRE)*, 2012.

[17] M. Grottke and K. S. Trivedi, Fighting Bugs: Remove, Retry, Replicate, and Rejuvenate, *Computer*, vol. 40, pp. 107–109, 2007.

[18] S. Bhatia, A. Kumar, M. E. Fiuczynski, and L. Peterson, Lightweight, high-resolution monitoring for troubleshooting production systems, in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, 2008, pp. 103–116.

[19] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan, Magpie: online modelling and performance-aware systems, in *Workshop on Hot Topics in Operating Systems*, USENIX, Ed., 2003.

[20] I. Irrera, J. Durães, M. Vieira, and H. Madeira, Towards Identifying the Best Variables for Failure Prediction Using Injection of Realistic Software Faults, in *IEEE 16th Pacific Rim International Symposium on Dependable Computing (PRDC)*, dec. 2010, pp. 3 –10.

[21] J. A. Durães and H. S. Madeira, Emulation of Software faults: A field data study and a practical approach., *IEEE Trans. Software Engineering*, vol. 32(11), pp. 849–867, 2006.

[22] J. von Neumann, Probabilistic logics and the synthesis of reliable organisms from unreliable components, *Automata Studies*, vol. 34, pp. 43–99, 1956.

[23] A. Avižienis, Design of fault-tolerant computers, in *Proceedings of the November Fall Joint Computer Conference*, 1967, pp. 733–743.

[24] J. Laprie, Dependable computing and fault tolerance: concepts and terminology, in *15th IEEE Int. Symp. on Fault-Tolerant Computing*, 1985, pp. 2–11.

[25] A. Bondavalli, S. Chiaradonna, D. Cotroneo, and L. Romano, Effective Fault Treatment for Improving the Dependability of COTS and Legacy-Based Applications, *IEEE Trans. on Dependable Secure Computing*, vol. 1, pp. 223–237, October 2004.

[26] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton, Software Rejuvenation: Analysis, Module and Applications, in *International Symposium on Fault-Tolerant Computing*, 1995, pp. 381–390.

[27] J. K. A. Avizienis, Fault tolerance by design diversity: concepts and experiments, *Computer*, vol. 17, no. 8, pp. 67–80, Aug 1984.

[28] M. Grottke, A. Nikora, and K. Trivedi, An empirical investigation of fault types in space mission system Software, in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2010, pp. 447–456.

[29] T. Yoshimura, H. Yamada, and K. Kono, Can Linux be Rejuvenated without Reboots?, in *IEEE Third International Workshop on Software Aging and Rejuvenation (WoSAR)*, 2011, pp. 50–55.

[30] R. Gupta and T. C. Mowry, Eds., *Faults in Linux: ten years later.* ACM, 2011.

[31] R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, D. S. Moebus, B. K. Ray, and M.-Y. Wong, Orthogonal Defect Classification-A Concept for In-Process Measurements, *IEEE Trans. on Software Engineering*, vol. 18, pp. 943–956, November 1992.

[32] J. Christmansson and R. Chillarege, Generation of an error set that emulates Software faults based on field data, in *Proceedings of the The Twenty-Sixth Annual International Symposium on Fault-Tolerant Computing (FTCS)*, 1996, pp. 304–313.

[33] J. Gray, Why do Computer Stop and What Can be About it?, in *Symposium on Reliability in Distributed Software and Database Systems*, 1986, pp. 3–12.

[34] K. S. Trivedi, R. Mansharamani, D. S. Kim, M. Grottke, and M. Nambiar, Recovery from Failures Due to Mandelbugs in IT Systems, in *IEEE 17th Pacific Rim International Symposium on Dependable Computing (PRDC)*, 2011, pp. 224–233.

[35] N. W. Green, A. R. Hoffman, T. K. M. Schow, and H. B. Garrett, Anomaly trends for robotic missions to Mars: Implications for mission reliability, in *Proceedings of the 44th AIAA Aerospace Sciences Meeting and Exhibit*, 2006, pp. 1–9.

[36] A. R. Hoffman, N. W. Green, and H. B. Garrett, Assessment of in-flight anomalies of long life outer planet missions, in *Proceedings of the European Space Agency 5th International Symposium on Environmental Testing for Space Programmes*, 2004, pp. 43–50.

[37] D. P. S. Ting-ting Y. Lin, Error Log Analysis: Statistical Modeling and Heuristic Trend Analysis, *IEEE Trans. on Reliability*, vol. 39, pp. 419–432, 1990.

[38] R. I. M. Kalyanakrishnam, Z. Kalbarczyk, Failure Data Analysis of a LAN of Windows NT Based Computers, in *Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems*, 1999, pp. 178–187.

[39] I. Lee and R. Iyer, Diagnosing rediscovered Software problems using symptoms, *IEEE Trans. on Software Engineering*, vol. 26, no. 2, pp. 113–127, feb 2000.

[40] M.Cinque, D. Cotroneo, R. Natella, and A. Pecchia, Assessing and improving the effectiveness of logs for the analysis of Software faults, in *International Conference on Dependable Systems and Networks (DSN)*, 2010, pp. 457–466.

[41] [Online]. Available: http://aws.amazon.com/message/65648/. March 2013

[42] N. E. Fenton, *Software Metrics: A Rigorous and Practical Approach*, 2nd ed. Boston, MA, USA: International Thomson Computer Press, 1996.

[43] [Online]. Available: http://www-01.ibm.com/Software/tivoli/. March 2013

[44] M. Brodie, I. Rish, and S. Ma, Optimizing Probe Selection for Fault Localization, in *Operations & Management, 12th International Workshop on Distributed Systems*, 2001, pp. 88–98.

[45] I. Rish, M. Brodie, S. Ma, N. Odintsova, A. Beygelzimer, G. Grabarnik, and K. Hernandez, Adaptive diagnosis in distributed systems, *IEEE Trans. on Neural Networks*, vol. 16, no. 5, pp. 1088 –1109, sept. 2005.

[46] L. Falai and A. Bondavalli, Experimental Evaluation of the QoS of Failure Detectors on Wide Area Network, in *Proceedings of the 2005 International Conference on Dependable Systems and Networks*, 2005, pp. 624–633.

[47] J. Herder, H. Bos, B. Gras, P. Homburg, and A. Tanenbaum, Construction of a Highly Dependable Operating System, in *Sixth European Dependable Computing Conference*, oct. 2006, pp. 3 –12.

[48] C. Simache, M. Kaaniche, and A. Saidane, Event log based dependability analysis of Windows NT and 2K systems, in *Proceedings of Pacific Rim International Symposium on Dependable Computing (PRDC)*, dec. 2002, pp. 311 –315.

[49] F. Salfner and M. Malek, Using Hidden Semi-Markov Models for Effective Online Failure Prediction, in *26th IEEE International Symposium on Reliable Distributed Systems*, oct. 2007, pp. 161–174.

[50] M. K. C. Simache, Availability assessment of sunOS/solaris unix systems based on syslogd and wtmpx log files: A case study, in *Pacific Rim International Symposium on Dependable Computing (PRDC)*, 2005, pp. 49–56.

[51] A. Bovenzi and G. Carrozza, Monitoring Infrastructure for Diagnosing Complex Software, in *Innovative Technologies for Dependable OTS-Based Critical Systems*. Springer Milan, 2013, pp. 189–202.

[52] R. K. Sahoo, A. J.Oliner, I. Rish, M. Gupta, J. Moreira, S. Ma, R. Vilalta, and A. Sivasubramaniam, Critical event prediction for proactive management in large-scale computer clusters, in *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, 2003, pp. 426–435.

[53] A. Pecchia, A. Sharma, Z. Kalbarczyk, D. Cotroneo, and R. K. Iyer, Identifying Compromised Users in Shared Computing Infrastructures: A Data-Driven Bayesian Network Approach, in *IEEE Symposium on Reliable Distributed Systems (SRDS)*, 2011, pp. 127–136.

[54] M. Cinque, D. Cotroneo, and A. Pecchia, Event Logs for the Analysis of Software Failures: A Rule-Based Approach, *IEEE Trans. on Software Engineering*, vol. Pre Print, no. 99, 2012.

[55] A. Andrzejak and L. Silva, Using machine learning for non-intrusive modeling and prediction of Software aging, in *IEEE Network Operations and Management Symposium (NOMS)*, april 2008, pp. 25–32.

[56] M. Bertier, O. Marin, and P. Sens, Implementation and performance evaluation of an adaptable failure detector, in *Proceedings of the 32th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2002, pp. 354–363.

[57] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum, Failure Resilience for Device Drivers, in *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2007, pp. 41–50.

[58] OMG. [Online]. Available: http://www.omg.org/spec/FT/1.0. March 2013

[59] M. Brodie, I. Rish, and S. Ma, Intelligent probing: A cost-effective approach to fault diagnosis in computer networks, *IBM Systems Journal*, vol. 41, no. 3, pp. 372–385, 2002.

[60] M. Chen, E. Kiciman, A. Accardi, A. Fox, and E. Brewer, Using runtime paths for macroanalysis, in *Proceedings of the 9th Conference on Hot Topics in Operating Systems*, 2003, pp. 14–19.

[61] G. Khanna, P. Varadharajan, and S. Bagchi, Automated online monitoring of distributed applications through external monitors, *IEEE Trans. on Dependable and Secure Computing*, vol. 3, no. 2, pp. 115–129, april-june 2006.

[62] P. Huang, A. Feldmann, and W. Willinger, A non-instrusive, wavelet-based approach to detecting network performance problems, in *Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement*, 2001, pp. 213–227.

[63] Y. Zhang and V. Paxson, Detecting stepping stones, in *Proceedings of the 9th conference on USENIX Security Symposium*, vol. 9, 2000, pp. 13–23.

[64] S. Forrest, S. A. Hofmeyr, A. S. Ji, and T. A. Longstaff, A sense of self for unix processes, in *Proceedings of IEEE Symposium on Security and Privacy*, 1996, pp. 120–129.

[65] W. Lee and S. J. Stolfo, Data mining approaches for intrusion detection, in *Proceedings of the 7th conference on USENIX Security Symposium*, vol. 7, 1998, pp. 7–21.

[66] D. Wagner and D. Dean, Intrusion Detection via Static Analysis, in *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, 2001, pp. 156–168.

[67] F. Maggi, M. Matteucci, and S. Zanero, Detecting Intrusions through System Call Sequence and Argument Analysis, *IEEE Trans. on Dependable and Secure Computing*, vol. 7, no. 4, pp. 381–395, oct.-dec. 2010.

[68] L. Wang, Z. Kalbarczyk, W. Gu, and R. Iyer, Reliability MicroKernel: Providing Application-Aware Reliability in the OS, *IEEE Trans. on Reliability*, vol. 56, no. 4, pp. 597 –614, dec. 2007.

[69] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal, Dynamic instrumentation of production systems, in *Proceedings of the annual conference on USENIX Annual Technical Conference*, 2004, pp. 2–15.

[70] M. Agarwal, Eigen space based method for detecting faulty nodes in large scale enterprise systems, in *IEEE Network Operations and Management Symposium (NOMS)*, april 2008, pp. 224–231.

[71] A. Avritzer and E. J. Weyuker, Monitoring Smoothly Degrading Systems for Increased Dependability, *Empirical Software Engineering*, vol. 2, pp. 59–77, 1997.

[72] G. Khanna, M. Y. Cheng, P. Varadharajan, S. Bagchi, M. Correia, and P. Verissimo, Automated Rule-Based Diagnosis Through a Distributed Monitor System, *IEEE Trans. on Dependable and Secure Computing*, vol. 4, no. 4, pp. 266–279, oct.-dec. 2007.

[73] J. Case, M. Fedor, M. Schoffstall, and J. Davin. A Simple Network Management Protocol (SNMP). [Online]. Available: http://www.ietf.org/rfc/rfc1157.txt. March 2013

[74] [Online]. Available: http://docs.oracle.com/javase/tutorial/jmx/index.html. March 2013

[75] [Online]. Available: http://msdn.microsoft.com/en-us/library/windows/desktop/aa394582.aspx. March 2013

[76] [Online]. Available: https://www.oasis-open.org/committees/wsdm/. March 2013

[77] M. L. Massie, B. N. Chun, and D. E. Culler, The ganglia distributed monitoring system: design, implementation, and experience, *Parallel Computing*, vol. 30, no. 7, pp. 817–840, Jul. 2004.

[78] R. Van Renesse, K. P. Birman, and W. Vogels, Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining, *ACM Trans. Comput. Syst.*, vol. 21, no. 2, pp. 164–206, May 2003.

[79] K. Park and V. S. Pai, CoMon: a mostly-scalable monitoring system for PlanetLab, *SIGOPS Oper. Syst. Rev.*, vol. 40, no. 1, pp. 65–74, Jan. 2006.

[80] [Online]. Available: linux.die.net/man/8/ping. March 2013

[81] [Online]. Available: linux.die.net/man/8/traceroute. March 2013

[82] A. Frenkiel and H. Lee, EPP: A Framework for Measuring the End-to-End Performance of Distributed Applications, in *Proceedings of Performance Engineering 'Best Practices' Conference*, I. A. of Technology, Ed., 1999.

[83] [Online]. Available: http://sebastien.godard.pagesperso-orange.fr/. March 2013

[84] K. Yaghmour and M. R. Dagenais, Measuring and characterizing system behavior using kernel-level event logging, in *Proceedings of the annual conference on USENIX Annual Technical Conference*, 2000, pp. 2–15.

[85] V. Prasad, W. Cohen, F. Eigler, M. Hunt, J. Keniston, and B. Chen, Locating system problems using dynamic instrumentation, in *Proceedings of the Ottawa Linux Symposium*, 2005, pp. 49–64.

[86] S. Agarwala and K. Schwan, SysProf: Online Distributed Behavior Diagnosis through Fine-grain System Monitoring, in *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2006, pp. 8–15.

[87] [Online]. Available: http://technet.microsoft.com/en-us/library/cc771692%28v=ws.10%29.aspx. March 2013

[88] PerfMon monitoring tool. [Online]. Available: http://perfmon2.sourceforge.net/. March 2013

[89] M. D. Ernst, J. Perkins, P. Guo, S. McCamant, C. Pacheco, M. Tschantz, and C. Xiao, The Daikon system for dynamic detection of likely invariants, *Science of Computing Programming*, vol. 69, pp. 35–45, December 2007.

[90] K. Shen, M. Zhong, and C. Li, I/O system performance debugging using model-driven anomaly characterization, in *Proceedings of the 4th USENIX Conference on File and Storage Technologies*, 2005, pp. 23–36.

[91] Z. Liu, N. Niclausse, and C. Jalpa-Villanueva, Traffic model and performance evaluation of Web servers, *Perform. Eval.*, vol. 46, no. 2-3, pp. 77–100, Oct. 2001.

[92] L. Li, K. Vaidyanathan, and K. S. Trivedi, An Approach for Estimation of Software Aging in a Web Server, in *Proceedings of the International Symposium on Empirical Software Engineering*, 2002, pp. 91–100.

[93] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox, Capturing, indexing, clustering, and retrieving system history, in *Proceedings of the twentieth ACM symposium on Operating systems principles*, 2005, pp. 105–118.

[94] M. K. Agarwal, M. Gupta, V. Mann, N. Sachindran, N. Anerousis, and L. B. Mummert, Problem Determination in Enterprise Middleware Systems using Change Point Correlation of Time Series Data, in *IEEE Network Operations and Management Symposium (NOMS)*, 2006, pp. 471–482.

[95] M. Chen, E. Kiciman, E. Fratkin, and E. B. A. Fox, Pinpoint: Problem Determination in Large, Dynamic Internet Services, in *Proceedings of the 32th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2002, pp. 595–604.

[96] J. Dilley, R. Friedrich, T. Jin, and J. A. Rolia, Measurement Tools and Modeling Techniques for Evaluating Web Server Performance, in *Computer Performance Evaluation*, 1997, pp. 155–168.

[97] L. Cherkasova, K. Ozonat, N. Mi, J. Symons, and E. Smirni, Anomaly? Application change? or Workload change? Towards Automated Detection of Application Performance Anomaly and Change, in *Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, june 2008, pp. 452–461.

[98] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 2nd ed. Pearson Education, 2003.

[99] L. Li and A. Malony, Model-Based Performance Diagnosis of Master-Worker Parallel Computations, in *Proceedings of Euro-Par Parallel Processing LNCS 4128*, 2006, pp. 35–46.

[100] D. G. Benoit, Automatic Diagnosis of Performance Problems in Database Management Systems, Ph.D. dissertation, Queen's University, 2003.

[101] K. Hätönen, M. Klemettinen, H. Mannila, P. Ronkainen, and H. Toivonen, Knowledge Discovery from Telecommunication Network Alarm Databases, in *Proceedings of the Twelfth International Conference on Data Engineering*, 1996, pp. 115–122.

[102] S. Rovnyak, S. Kretsinger, J. Thorp, , and D. Brown, Decision trees for real-time transient stability prediction, *IEEE Trans. Power Syst*, vol. 9, no. 3, pp. 1417–1426, 1994.

[103] F. Salfner, M. Lenk, and M. Malek, A survey of online failure prediction methods, *ACM Computuping Survey*, vol. 42, no. 3, pp. 1–42, 2010.

[104] A. Daidone, F. D. Giandomenico, A. Bondavalli, and S. Chiaradonna, Hidden Markov Models as a Support for Diagnosis: Formalization of the Problem and Synthesis of the Solution, in *25th IEEE Symposium on Reliable Distributed Systems*, 2006, pp. 245–256.

[105] L. R. Rabiner, A tutorial on hidden Markov models and selected applications in speech recognition, in *Readings in speech recognition*, A. W. . K.-F. Lee, Ed., 1990, pp. 267–296.

[106] F. Salfner, M. Schieschke, and M. Michael, Predicting failures of computer systems: a case study for a telecommunication system, in *Proceedings of the 20th international conference on Parallel and distributed processing*, 2006, pp. 348–348.

[107] V. Jecheva, About Some Applications of Hidden Markov Model in Intrusion Detection Systems, in *Proceedings of International Conference on Computer Systems and Technologies*, 2006.

[108] K. R. Joshi, W. H. Sanders, M. A. Hiltunen, and R. D. Schlichting, Automatic Model-Driven Recovery in Distributed Systems, in *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems*, 2005, pp. 25–38.

[109] A. Nickelsen, J. Gronbaek, T. Renier, and H.-P. Schwefel, Probabilistic Network Fault-Diagnosis Using Cross-Layer Observations, in *Proceedings of the International Conference on Advanced Information Networking and Applications*, 2009, pp. 225–232.

[110] M. Jiang, M. Munawar, T. Reidemeister, and P. Ward, Efficient Fault Detection and Diagnosis in Complex Software Systems with Information-Theoretic Monitoring, *IEEE Trans. on Dependable and Secure Computing*, vol. 8, no. 4, pp. 510–522, july-aug. 2011.

[111] G. Jiang, H. Chen, and K. Yoshihira, Modeling and Tracking of Trans. Flow Dynamics for Fault Detection in Complex Systems, *IEEE Trans. on Dependable and Secure Computing*, vol. 3, pp. 312–326, 2006.

[112] R. Pietrantuono, S. Russo, and K. Trivedi, Online Monitoring of Software System Reliability, in *European Dependable Computing Conference (EDCC)*, april 2010, pp. 209–218.

[113] A. Bovenzi, D. Cotroneo, R. Pietrantuono, and G. Carrozza, Error detection framework for complex Software systems, in *Proceedings of the 13th European Workshop on Dependable Computing*, 2011, pp. 61–66.

[114] S. Ghanbari and C. Amza, Semantic-Driven Model Composition for Accurate Anomaly Diagnosis, in *Proceedings of the International Conference on Autonomic Computing*, 2008, pp. 35–44.

[115] A. Bondavalli, S. Chiaradonna, F. Di Giandomenico, and F. Grandoni, Threshold-based mechanisms to discriminate transient from intermittent faults, *IEEE Trans. on Computers*, vol. 49, no. 3, pp. 230 –245, mar 2000.

[116] N. N. Tendolkar and R. L. Swann, Automated Diagnostic Methodology for the IBM 3081 Processor Complex, *IBM J. Research and Development*, vol. 26, pp. 78–88, 1982.

[117] G. Mongardi, “Dependable computing for railway control systems, in *Proceedings of DCCA*, 1993, pp. 255–277.

[118] D. Powell, C. Rabéjac, and A. Bondavalli, Alpha-count mechanism and inter-channel diagnosis, ESPRIT Project 20716 GUARDS Report, Tech. Rep., 1998.

[119] L. Romano, A. Bondavalli, S. Chiaradonna, and D. Cotroneo, Implementation of Threshold-based Diagnostic Mechanisms for COTS-Based Applications, in *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems*, 2002.

[120] A. Casimiro, P. Lollini, M. Dixit, A. Bondavalli, and P. Verissimo, A framework for dependable QoS adaptation in probabilistic environments, in *Proceedings of the ACM symposium on Applied computing*, 2008, pp. 2192–2196.

[121] M. Serafini, A. Bondavalli, and N. Suri, On-Line Diagnosis and Recovery: On the Choice and Impact of Tuning Parameters, *IEEE Trans. on Dependable and Secure Computing*, vol. 4, no. 4, pp. 295–312, oct.-dec. 2007.

[122] A. Bulut and A. Singh, A unified framework for monitoring data streams in real time, in *Proceedings. 21st International Conference on Data Engineering*, april 2005, pp. 44–55.

[123] D. Cotroneo, D. Di Leo, and R. Natella, Adaptive monitoring in microkernel OSs, in *International Conference on Dependable Systems and Networks Workshops*, 2010, pp. 66 –72.

[124] [Online]. Available: www.bmc.com/. March 2013

[125] [Online]. Available: http://www.netuitive.com/. March 2013

[126] T. Idé and H. Kashima, Eigenspace-based anomaly detection in computer systems, in *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, 2004, pp. 440–449.

[127] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat, Pip: detecting the unexpected in distributed systems, in *Proceedings of the 3rd conference on Networked Systems Design & Implementation*, vol. 3, 2006, pp. 9–23.

[128] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste, Rainbow: architecture-based self-adaptation with reusable infrastructure, *Computer*, vol. 37, no. 10, pp. 46–54, oct. 2004.

[129] A. Daidone, Critical Infrastructures: a Conceptual Framework for Diagnosis, Some Applications and Their Quantitative Analysis, Ph.D. dissertation, Università degli Studi di Firenze, April 21th 2010.

[130] W. Chen, S. Toueg, and M. K. Aguilera, On the Quality of Service of Failure Detectors, *IEEE Trans. on Computers*, vol. 51, no. 1, pp. 561–580, 2002.

[131] M. Basseville and I. Nikiforov, *Detection of abrupt changes: theory and application.* Prentice-Hall, Inc., 1993.

[132] M. Jiang, M. Munawar, T. Reidemeister, and P. Ward, Detection and Diagnosis of Recurrent Faults in Software Systems by Invariant Analysis, in *Proceedings of the 11th IEEE High Assurance Systems Engineering Symposium*, 2008, pp. 323–332.

[133] I. S. Board, IEEE Standard Classification for Software Anomalies, in *IEEE Std 1044*, 1993.

[134] A. Bondavalli, F. Brancati, and A. Ceccarelli, Safe Estimation of Time Uncertainty of Local Clocks, in *Proceedings of International IEEE Symposium on Precision Clock Synchronization for Measurement, Control and Communication (ISPCS)*, 2009, pp. 47–52.

[135] S. Ross, *Introduction to probability and statistics for engineers and scientists.* Elsevier Academic Press, 2003.

[136] H. Risken, *The Fokker-Planck equation: methods of solutions and applications*, 2nd ed. Springer, Berlin, 1989.

[137] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering: an introduction.* Norwell, MA, USA: Kluwer Academic Publishers, 2000.

[138] K. M. Chandy and L. Lamport, Distributed snapshots: determining global states of distributed systems, *ACM Trans. Comput. Syst.*, vol. 3, no. 1, pp. 63–75, Feb. 1985.

[139] D. Montgomery, *Design and Analysis of Experiments.* Wiley, 2008.

[140] M. D. Cin, K. Kanoun, K. Buchacker, L. L. Zuinga, R. Lindstrom, A. Johanson, H. Madeira, V. Sieh, , and N. Suri, DBench - workload and faultload selection, DBench Project, Tech. Rep., June 2002.

[141] A. K. Mishra, J. L. Hellerstein, W. Cirne, and C. R. Das, Towards characterizing cloud backend workloads: insights from Google compute clusters, *SIGMETRICS Perform. Eval. Rev.*, vol. 37, no. 4, pp. 34–41, Mar. 2010.

[142] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell, Fault Injection for Dependability Validation: A Methodology and Some Applications, *IEEE Trans. on Software Engineering*, vol. 16, no. 2, pp. 166–182, 1990.

[143] R. Natella, D. Cotroneo, J. Duraes, and H. Madeira, On Fault Representativeness of Software Fault Injection, *IEEE Trans. on Software Engineering*, vol. 39, no. 1, pp. 80–96, 2012.

[144] I. C. Society, Ed., *Second Workshop on Sharing Field Data and Experiment Measurements on Resilience of Distributed Computing Systems*, 2010.

[145] M. Vieira, J. Costa, and H. Madeira, The OLAP and Data Warehousing Approaches for Analysis and Sharing of Results from Dependability Evaluation Experiments, in *Proceedings of the 33th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2003, pp. 86–91.

[146] N. Laranjeiro, M. Vieira, and H. Madeira, Robustness Validation in Service-Oriented Architectures, in *Architecting Dependable Systems VI*, R. de Lemos, J.-C. Fabre, C. Gacek, F. Gadducci, and M. ter Beek, Eds. Springer Berlin / Heidelberg, 2009, vol. 5835, pp. 98–123.

[147] D. A. Menascé, Workload Characterization, *IEEE Internet Computing*, vol. 7, no. 5, pp. 89–92, 2003.

[148] A. Johansson and N. Suri, On the impact of injection triggers for OS robustness evaluation, in *International Symposium on Software Reliability Engineering*, 2007, pp. 127–136.

[149] R. Jain, *The Art Of Computer Systems Performance Analysis:.* Wiley, 1991.

[150] N. Juristo and A. M. Moreno, *Basics of Software Engineering Experimentation*, 1st ed. Springer Publishing Company, Incorporated, 2010.

[151] R. Matias and P. Filho, An Experimental Study on Software Aging and Rejuvenation in Web Servers, in *30th Annual International Computer Software and Applications Conference (COMPSAC)*, vol. 1, sept. 2006, pp. 189 –196.

[152] A. Bovenzi, F. Brancati, S. Russo, and A. Bondavalli, A Statistical Anomaly-Based Algorithm for On-line Fault Detection in Complex Software Critical Systems, in *Lecture Notes in Computer Science (LNCS)*, vol. 6894, 2011, pp. 128–142.