Pontifícia Universidade Católica
do Rio de Janeiro

Thiago Pinheiro de Araújo

Using runtime information and maintenance knowledge to assist failure diagnosis, detection and recovery

TESE DE DOUTORADO

Thesis presented to the Programa de Pós-Graduação em Informática of the Departamento de Informática, PUC-Rio as partial fulfillment of the requirements for the degree of Doutor em Informática.

Advisor: Prof. Arndt von Staa

Rio de Janeiro
October 2014

Pontifícia Universidade Católica
do Rio de Janeiro

Thiago Pinheiro de Araújo

Using runtime information and maintenance knowledge to assist failure diagnosis, detection and recovery

Thesis presented to the Programa de Pós-Graduação em Informática of the Departamento de Informática do Centro Técnico Científico da PUC-Rio, as partial fulfillment of the requirements for the degree of Doutor.

Prof. Arndt von Staa
Advisor
Departamento de Informática – PUC-Rio

Prof. Roberto da Silva Bigonha
UFMG

Prof. Guilherme Horta Travassos
UFRJ

Prof. Renato Fontoura de Gusmão Cerqueira
IBM Research – Brazil

Prof. Hélio Côrtes Vieira Lopes
Departamento de Informática – PUC-Rio

Prof. José Eugenio Leal
Coordinator of the Centro Técnico Científico da
PUC-Rio

Rio de Janeiro, October 7th, 2014

Thiago Pinheiro de Araújo

Graduated in Computer Engineering from Pontifícia Universidade Católica do Rio de Janeiro (2007, Brazil, Rio de Janeiro), and obtained the degree of Mestre em Informática also from Pontifícia Universidade Católica do Rio de Janeiro (2010, Brazil, Rio de Janeiro).

Bibliographic data

Araújo, Thiago

Using runtime information and maintenance knowledge to assist failure diagnosis, detection and recovery / Thiago Pinheiro de Araújo; advisor: Arndt von Staa. — Rio de Janeiro : PUC–Rio, Departamento de Informática, 2014.

v., 192 f.: il. ; 29,7 cm

Tese (Doutorado em Informática) – Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 2014.

Inclui bibliografia.

1. Informática - Teses. 2. Engenharia de Software. 3. Qualidade de Software. 4. Diagnóstico de falhas. 5. Detecção de Falhas. 6. Recuperação de falhas. I. Staa, Arndt von. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

To my parents, Ítalo and Antônia,
my aunt Maria do Carmo,
and my wife Ana Carolina.

Acknowledgements

To my father, Ítalo José de Araújo, the main responsible for my upbringing. Without your presence, your life experience, your participation, your support, your joy, and your interest, I would not have gotten this far. Thank you for your relentless dedication while you were among us. Thank you very much for everything and watch over us.

To my mother, Antônia dos Santos Cunha Pinheiro, and my aunt, Maria do Carmo Cunha Pinheiro, for the support and love in all through my moments of hardship. Thank you for your steadfast dedication in providing the necessary means so I could always keep on with my education.

To my wife, Ana Carolina Froes da Fonseca Martins de Andrade, for your love, support, and great patience, especially in the final months I spent working on this thesis. Thank you also for your advices, always very useful in all the moments of our lives.

To my advisor, prof. Arndt von Staa, for all your dedication and interest in my research. Thank you for your advice, your guidance, and for all our meaningful conversations, so constructive to my academic, professional, and personal life. Without your support and teachings, this work would not have been possible. It has been an honor working with you for the last 9 years, which I hope will be only the beginning of a long interaction in joint academic and professional projects.

To prof. Renato Cerqueira, for all the conversations, ideas, and teachings that in several ways influenced and contributed so much to the production of this thesis. Thank you, also, for your support and for the high standards set in all the work I produced under your guidance, ever since I got into the university. Thank you, still, for offering me the chance to teach, awakening in me a growing desire to become an educator.

To my best friend, Roberto Maia, for all the support and patience in these last few years. Thank you for helping me with my personal issues whenever I needed it, for the steadfast

believe in my work, and for the words of encouragement ever said in difficult moments.

To my great friend and co-worker, Pedro de Goes, for your interest in my research, for making yourself available for our discussions, and for your help in spreading this new technique among other co-workers.

To all my colleagues at Aevo, for the exchange of ideas and observations during the experiments and case studies.

To all my family and friends, who somehow contributed to this work. In particular, Patrícia Correa and Thuener Silva.

To CNPq, for the financial support without which this work would not have been possible.

To all the members of the examining committee.

To all the teachers of the department, for sharing their great knowledge.

Abstract

Araújo, Thiago Pinheiro; Staa, Arndt von (Advisor). **Using runtime information and maintenance knowledge to assist failure diagnosis, detection and recovery.** Rio de Janeiro, 2014. 192p. Thesis - Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Even software systems developed with strict quality control may expect failures during their lifetime. When a failure is observed in a production environment the maintainer is responsible for diagnosing the cause and eventually removing it. However, considering a critical service this might demand too long a time to complete, hence, if possible, the failure signature should be identified in order to generate a recovery mechanism to automatically detect and handle future occurrences until a proper correction can be made. In this thesis, recovery consists of restoring a correct context allowing dependable execution, even if the causing fault is still unknown. To be effective, the tasks of diagnosing and recovery implementation require detailed information about the failed execution. Failures that occur during the test phase run in a controlled environment, allow adding specific code instrumentation and usually can be replicated, making it easier to study the unexpected behavior. However, failures that occur in the production environment are limited to the information present in the first occurrence of the failure. But run time failures are obviously unexpected, hence run time data must be gathered systematically to allow detecting, diagnosing with the purpose of recovering, and eventually diagnosing with the purpose of removing the causing fault. Thus there is a balance between the detail of information inserted as instrumentation and the system performance: standard logging techniques usually present low impact on performance, but carry insufficient information about the execution; while tracing techniques can record precise and detailed information, however are impracticable for a production environment. This thesis proposes a novel hybrid approach for recording and extracting system's runtime information. The solution is based on event logs, where events are enriched with contextual properties about the current state of the execution at the moment the event is recorded. Using these enriched log events a diagnosis technique and a tool have been developed to allow event filtering based on the maintainer's perspective of interest. Furthermore, an approach using these enriched events has been developed that allows detecting and diagnosing failures aiming at recovery. The proposed solutions were evaluated through measurements and studies conducted using deployed systems, based on failures that actually occurred while using the software in a production context.

Keywords

Runtime information extraction; Failure diagnosis; Failure detection; Failure recovery.

Resumo

Araújo, Thiago Pinheiro; Staa, Arndt von. **Utilizando informações da execução do sistema e conhecimentos de manutenção para auxiliar o diagnóstico, detecção e recuperação de falhas.** Rio de Janeiro, 2014. 192p. Tese de Doutorado - Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Mesmo sistemas de software desenvolvidos com um controle de qualidade rigoroso podem apresentar falhas durante seu ciclo de vida. Quando uma falha é observada no ambiente de produção, mantenedores são responsáveis por produzir o diagnóstico e remover o seu defeito correspondente. No entanto, em um serviço crítico este tempo pode ser muito longo, logo, se for possível, a assinatura da falha deve ser utilizada para gerar um mecanismo de recuperação automático capaz de detectar e tratar futuras ocorrências similares, até que o defeito possa ser removido. Nesta tese, a atividade de recuperação consiste em restaurar o sistema para um estado correto, que permita continuar a execução com segurança, ainda que com limitações em suas funcionalidades. Para serem eficazes, as tarefas de diagnóstico e recuperação requerem informações detalhadas sobre a execução que falhou. Falhas que ocorrem durante a fase de testes em um ambiente controlado podem ser depuradas através da inserção de nova instrumentação e re-execução da rotina que contem o defeito, tornando mais fácil o estudo de comportamentos inesperados. No entanto, falhas que ocorrem no ambiente de produção apresentam informações limitadas à situação específica em que ocorrem, além de serem imprevisíveis. Para mitigar essa adversidade, informações devem ser coletadas sistematicamente com o intuito de detectar, diagnosticar para recuperar e, eventualmente, diagnosticar para remover a circunstância geradora da falha. Além disso, há um balanceamento entre a informação inserida como instrumentação e a performance do sistema: técnicas de *logging* geralmente apresentam baixo impacto no desempenho, porém não provêm informação suficiente sobre a execução; por outro lado, as técnicas de *tracing* podem registrar informações precisas e detalhadas, todavia são impraticáveis para um ambiente de produção. Esta tese propõe uma abordagem hibrida para gravação e extração de informações durante a execução do sistema. A solução proposta se baseia no registro de eventos, onde estes são enriquecidos com propriedades contextuais sobre o estado atual da execução no momento em que o evento é gravado. Através deste registro de eventos com informações de contexto, uma técnica de diagnóstico e uma ferramenta foram desenvolvidas para permitir que eventos pudessem ser filtrados com base na perspectiva de interesse do mantenedor. Além disso, também foi desenvolvida uma abordagem que utiliza estes eventos enriquecidos para detectar falhas automaticamente visando recuperação. As soluções propostas foram avaliadas através de medições e estudos conduzidos em sistemas implantados, baseando-se nas falhas que de fato ocorreram enquanto se utilizava o software em um contexto de produção.

Palavras-chave

Extração de informações da execução do sistema; Diagnóstico de falhas; Detecção de Falhas; Recuperação de falhas.

Summary

Figures

Tables

1
Introduction

Despite the effort spent in fault prevention techniques, every software system may expect failures during its lifetime (Brown & Patterson, 2001). Since software development is produced or guided by humans, the result is vulnerable to human mistakes too. Although tools and techniques are continuously improved to aid developers to avoid these mistakes, it is impossible to guarantee that the specification of a system or even the deployed instance of a system is fault-free. Faults can be due to different reasons: specification errors, inadequate architecture, wrong implementation, incorrect or missing configuration, hardware defects, defects in third party systems, and misuse due to inexistent, misleading or incomplete documentation (Brown & Patterson, 2001; NIST, 2002). Even if successfully avoiding the insertion of faults during development, the system may be vulnerable to faults due to external causes, such as third-party libraries (Thomas, 2002), frameworks, and remote services. Unfortunately, in modern system development, application developers do not own all the source-code, neither can third-party developers predict all configurations and environments where their software will be used. Thus, although software development must prevent fault insertion, software engineers cannot assume that this approach alone is sufficient to avoid runtime failures. It is, therefore, extremely important to incorporate the notion of run time failure handling in all phases of system development. The terminology used in this work is defined in Section 1.1.

The observation of a failure should be followed by a diagnosis session, in order to determine its cause and proceed with the correction. However, during the time elapsed between the first observation of a failure and the deployment of a proper correction, the system is vulnerable to other occurrences of failures caused by the same fault. One of the characteristics of a fault is its criticality, which is based on the severity of the consequences of the failures it provokes, which depend on the application domain. The consequence of simple application failures is the loss of work, which causes rework and dissatisfaction with the system;

however, the consequences in mission-critical systems may cause damage to high-value equipment, loss of business credibility, environmental disasters, or even the loss of human life. Thus, the occurrence of a critical failure frequently requires instant reactions to avoid disasters even before the causing fault can be duly removed. Among the actions needed to recover, another form of diagnosis occurs. This form must identify which data have been corrupted and determine how to bring them to a valid state. The state must be structurally correct (e.g. satisfies the data model) as well as accurate (the state corresponds to the real world state).

We call *recovery* the sequence of actions that must be performed to prevent serious consequences of a failure and to reestablish the functional integrity of the system. After recovery, the system may resume working, even if slightly degraded. Recovery does not necessarily mean that the causing fault has been properly removed. The time elapsed between the moment a fault is exercised generating an error, and the moment the corresponding failure is observed, is the *mean-time-to-observe* (MTTO). The longer this elapsed time is, the more difficult it tends to be to diagnose the fault. The mean elapsed time between the observation of the occurrence of a failure and the end of its recovery is called *mean-time-to-recover* (MTTRc). The mean elapsed time between observation of a failure's occurrence and the moment the properly corrected system is available to be deployed is called *mean-time-to-repair* (MTTRp), which does not include the time to redeploy the corrected components. When actions are executed to handle a failure, the MTTRc is expected to require much less time than the MTTRp. There is also the *mean-time-between-failures* (MTBF), which represents the time between successive failures. A larger MTTF does not guarantee system stability; a short MTTRc, however, contributes to reduce the risk of disasters due to failures (Patterson & Brown, 2002). The conclusion is that faults and failures are facts to be coped with, and should not be treated just as development errors that can eventually be removed. The team must expect run time failures, and take advantage of all failure occurrences to learn more about the weaknesses of the system, thus investing more effort in its evolution in order to minimize the chances of similar failures in the future.

Since failures may occur, system maintainers must be prepared to diagnose them in a short time and implement, as quickly as possible, a mechanism to detect and handle future occurrences, in order to prevent or minimize disasters. There are two diagnosing approaches: the most effective, in order to determine the root cause and remove the fault; and the most efficient, looking for the failure signature in order to produce the failure handler. Despite the need of a manual diagnosis to determine the fault when a failure is first observed, future failures occurrences of the corresponding fault should ideally be automatically detected and handled without human intervention. Ideally, the first failure occurrence of a fault should be automatically handled too. However this would require automatic diagnosis, for which the available solutions are usually limited to superficial faults, such as network, disk access, and performance issues (Chen et al., 2004; Yuan et al., 2010; Meng et al., 2012), and are thus ineffective when diagnosing logic-specific failures, since for these type of failures the diagnosis needs specific information about the target system. Moreover, solutions that use a system's specification written by humans (in a given format) to evaluate the runtime execution require a huge effort to implement and are also susceptible to faults, since they, too, may contain errors. This topic is further discussed in chapters 2 and 8.

Fault diagnosis, failure detection, and recovery capabilities are requirements of modern systems that must be available 24 hours per day and for which acceptable downtime is very short, measured in minutes or even seconds, depending on the application domain. It is undeniable that faults must be removed from the system as soon as possible. However, some of them impose a larger MTTRp due to different factors: the code modification or the redeployment may increase the risk of other faults; the effort to remove and redeploy may not compensate; the system platform may impose time limitations (for example, the review time in AppStore submissions can take up to three weeks); or, even, all of the previous factors together. Sometimes, it may be safer to avoid fault removal until the next release or even until a given error threshold, proportional to the acceptable consequences, such as number of errors or amount of resources lost. However, in these cases a recovery solution must be available to handle failure consequences.

It has been observed that due to the complexity of modern systems, faults will not be completely eliminated from deployed applications (Business Internet Group, 2004), even after several maintenance patches. It is safe to assume that even if faults are correctly removed, some will remain due to incorrect diagnostics or will reappear due to specification ambiguity. As an example we may consider a very complex system, with an asset shared between two components (A and B). The asset specification contains an ambiguity that favors one of these components, and causes a failure when executing the other. When the fault is diagnosed, the maintainer does not notice that it is in fact a specification error, and hence just corrects the asset without analyzing the consequences in other components, which had until then presented a correct behavior but will eventually fail in future executions of the yet undetected fault. This fault will be unsolved until the maintainer finds out that it is a specification error and asks to meet the developers of these components to talk about the issue. The conclusion will be that both implementations are correct from each point of view, and that an evolution of the specification is necessary to define how the asset will comply with both needs. Observe that the time to repair will be inevitably long, and a solution must be applied to handle new failures during this period.

Some faults cannot be removed due to technical or even business limitations. For example, consider a project with recently purchased hardware, whose library implements a proprietary state-of-art algorithm to process and provide a excellent results. Unfortunately, the project is based on an operating system that is not fully supported by the library needed by this hardware, thus an unstable beta version is the only option available. The system will have to handle the consequences of these library failures until the hardware provider publishes a new version of the library, something that may never happen. Observe that, due to business decisions, it is impossible to change the hardware to a supported one and, therefore, a recovery mechanism is the only solution to maintain system stability during occurrences of failures of this type.

There are also mission-critical systems that require a high-level of fault tolerance due to the possibility of failures leading to irreparable disasters. Hence, these failures must be detected and handled promptly, and cannot wait for a correction diagnosis, fault removal and redeployment. Therefore, faults must be foreseen during development to enable the implementation of mechanisms that detect and handle failures, where this must be done before the system is deployed in a production environment. Such systems usually rely on the use of replicas and a voting mechanism (Avizienis & Chen, 1977; Avizienis et al., 2004). Unfortunately, even these approaches do not work if failure independence cannot be assured (Knight & Levenson, 1986), not even when considering hardware, since these may suffer from design faults (BEA, 2009).

Failure handling and fault removal tasks are classified as corrective maintenance, which represents about 17% of the effort spent in general maintenance (Nosek & Palvia, 1990). There are other forms of maintenance, such as evolution and adaptive maintenance, which represent respectively about 65% and 18% (Nosek & Palvia, 1990). System monitoring and failure detection are also corrective maintenance tasks, since they aid in leading the system to a correct and accurate state. Compared to the total effort spent in software development, the cost of maintenance is usually high, representing about 80% of the total lifecycle cost of a software system, of which 40% is invested in understanding the software (Diehl, 2007; Lanza & Marinescu, 2008; Telea et al., 2010). This topic will be further discussed in Section 2.1. It is worth mentioning that in the last decade the NIST (2002) reported that corrective maintenance cost the US economy around 60 billion dollars per year.

Usually, small businesses and startups[1] cannot handle the cost of high dependability, thus they relax on quality while striving to assure an acceptably working system. Hence, low cost methods and tools are needed to aid those businesses to reduce the effort of corrective maintenance. For the moment, however, software systems must be able to coexist with faults. Some will remain active for short periods (a release and a subsequent patch), others for the system lifetime (when the removal is not worth the effort, or is impossible). Hence, for each known failure, a recovery mechanism should be available to handle its occurrences. These mechanisms must be easily removed when no longer needed. Newer failures need to be diagnosed to understand what went wrong during execution, and then proceed with the creation of this recovery mechanism. Last but not least, all of this should be provided in a way that common developers can apply into their own software to increase reliability, with traditional paradigms and minor changes in the way the code is written.

[1] These kinds of businesses are relevant to the Brazilian market, where they represent 70% of the software production (MCT, 2001).

Tools and techniques designed for fault diagnosis, failure detection, and system recovery require, regardless of the approach adopted, a technique to extract system's runtime information, which can be evaluated inside or outside each process, synchronously or not, but it must exist to enable behavior understanding. Sometimes, this information is combined with other data, such as maintenance knowledge and software source-code, to produce a richer content for these support tools. Available solutions for extracting this information expose a drastic trade-off between the effort to apply the methodology and the quality of the information content. Current solutions will be discussed in Chapter 8. There are also the requirements to apply the methodology, which even when presenting low-cost, in some cases can influence the system or the software architecture in an unwanted way by their designers. This topic will be further discussed in Chapter 2.

A survey conducted during this work in the field of information extraction identified that the effectiveness of the approach must be evaluated based on four aspects: context completeness, abstraction preservation, effort to instrument, and technology limitations. In order to provide a solution for traditional projects and development teams, its implementation must demand little effort and it must be free of technology, such as language and frameworks that will be used. However, current solutions that fit this profile usually present low information quality.

The problem addressed in this thesis is how to design a solution that extracts runtime data with adequate levels of contextual properties, and how to embed them with software abstractions created during development, in order to provide richer information for diagnosing tools and failure recovery mechanisms, with the objective of reducing the effort of failure diagnosis tasks and failure handling implementation.

The solution must impose as little technological limitations as possible, and must have low implementation costs. This extraction technique must support failure diagnosis tools and failure handling mechanisms. It is not designed for fault tolerance, however, as we shall present in the evaluation chapter, some fault tolerant solutions may use the proposed information extraction solution as an information source. Therefore, instead of relying on replicas and a voting mechanism, it relies on observing the runtime history and taking appropriate action to avoid, or recover from, failures. This solution has been chosen since it is relatively low cost and because software replicas are not necessarily reliable (Knight and Levenson, 1986). Furthermore, hardware replicas are also not necessarily reliable, since the hardware may not satisfy the independent fault hypothesis, or may trigger incorrect human response that leads to a disastrous behavior[2] (Reason, 2003).

The solution presented in this work is based on a novel hybrid logging technique capable of extracting properties from the full execution state, without breaking the encapsulation neither imposing great effort from the developer. The solution is slightly more expensive than the traditional logging technique, is technology-free, and drastically improves the context completeness and abstraction preservation of the extracted information. Developers insert instrumentation in a semi-automatic way following an established instrumentation policy, which is written for each project using some guidelines that are explained in Chapter 3. These guidelines suggest how to represent abstractions as properties and insert them in the log content in a way that can be used further to correlate events in a diagnosing tool or in failure handling mechanisms. Following our approach, tools and mechanisms that depend on log to detect, diagnose, and recover from failures will receive richer data to analyze. Thus, the outcome will be a reduced mean time to recover (MTTRc), since these tools and mechanisms will have (1) a better description of the system execution history to determine the root cause, (2) known-failure detectors will be more powerful, and (3) recovery mechanisms will be fed with precise information to execute the handling routines.

The technical solution is composed of (1) a code instrumentation technique, which supports the information extraction approach (described in Chapter 4), and (2) tools to assist failure diagnosis by providing an interface capable of selecting only events related to a maintainers perspective of interest (described in Chapter 5); and (3) a self-healing mechanism that uses both the events produced by the instrumentation and the knowledge about the system faults to detect and handle failures (described in Chapter 6).

[2] The Air France 447 crashed due to human error after the autopilot disengaged after non-recoverable incorrect readings of the three pitots (speed meters). The pitots suffered from a known design flaw that could lead them to freeze if flying in certain kinds of turbulence (BEA, 2009). Hence, they failed the independent fault hypothesis.

The solution has been evaluated on four real systems developed by a small software company (described in Chapter 7), which were used to identify problems and execute experiments. Some studies were executed in a controlled environment capable of measuring the solution's effectiveness during the assessments. These systems are from different domains and developed by different teams. The solution was applied during the development phase, without influencing the technology definition, neither impacting the way software is modeled or coded. The instrumentation inserted in these four systems was measured in order to evaluate the effort in applying the technique. The result showed us that the average implementation overhead is 4% of the full development effort. Also, two studies were conducted: (1) faults once discovered in the deployed version of the system were injected into the controlled environment and maintainers were submitted to the task of diagnosing them with the inspection tool; and (2) failure handlers were developed for failures caused by hard-to-remove faults, in order to verify the efficacy of our solution for failure detection and recovery. These failures were selected from all those identified on systems that participated in the evaluation. The results showed that both the inspection tool and the failure handler mechanism are capable of reducing the effort of the maintainer in the tasks of diagnosing and implementing failure handlers, respectively.

1.1 Terminology definition

We will adopt the following terminology — adapted from IEEE (2010) —: *fault* is an incorrect code fragment or configuration in the software which, when executed or accessed, may cause the system to perform in an unintended or unanticipated manner. Faults may be due to incorrect implementation, incorrect maintenance, incorrect specifications, incorrect third party software, or platform faults. Executing a fault may generate an *error*, which is a discrepancy between the instantaneous computed state and what it is expected to be. Faults may correspond to vulnerabilities that may lead users to, accidentally or willfully, provoke an error. The sequence of instantaneous, possibly parallel, states establishes the *behavior* of the system. Observing that an error occurred corresponds to *detecting* a *failure*. A failure is, thus, the observed inability of a system to perform its required functions within expected performance requirements. This means that a failure corresponds to an error that has been observed. There is a *latency* between the moment the error is generated and the moment it is detected and reported as a failure. Unfortunately, it might happen that an error is never detected, or is detected only a long time after having been generated. When a failure is observed at use time, the corresponding fault must be *diagnosed*. The longer the latency, the harder it is to *diagnose* the failure in order to precisely determine the corresponding fault. A *diagnostic* should describe the exact *root cause*, i.e. the very fault that lead to the failure. *Log* is a set of events generated by the system execution, which are sorted and presented in chronological order. The *diagnosis process* investigates the *log set* looking back searching a *failure footprint*. It starts at the state were the failure was detected and ends at the state that exercised the fault. The footprint should convey the necessary information to create the diagnostic. The process of evaluating a hypothesis over the log set is an *inspection*, which may restrict events based on some properties considered more relevant for the analysis. This set of restrictions is the *perspective of interest*. As mentioned before, the root cause may be other than just a faulty code. To eliminate the fault, either code fragments must be *removed*, *added*, *replaced* or *encapsulated* in a control wrapper, or configurations must be corrected. The encapsulation solution is often required when using third party code. *Failure recovery* is the system's capability of detecting and handling failure occurrences. The *recovery handler* is an entity that targets a specific type of failure and that is developed using the failure footprint to generate the *failure signature*, which is an event correlation capable of detecting when a failure will occur, or has occurred in the past few moments. *Debugging* corresponds to performing the three operations: detecting the occurrence of an error; diagnosing it to find the root cause; and correctly and completely removing the causing fault.

1.2 Document Structure

The rest of this work is organized in the following sections:

2) *Problem Formulation* - Discusses the corrective maintenance problem and recovery solutions in more detail based on the existing literature, defines the problems addressed by this work and enumerates the requirements the solution must fulfill to be acceptable.

3) *Solution Overview* – Presents a solution for the outlined problems and discusses how it complies with the requirements.

4) *Logs Annotated With Meta-Information* – Describes a solution to extract the system execution information in a format that supports diagnosis and failure handling.
5) *Lynx: a Diagnosing Tool based on Contextual Information* – Describes the tool provided for diagnosing, capable of studying the system execution restricted to a perspective of interest.
6) *Hydra: a Low-Cost, Self-healing Mechanism* – Describes how the proposed self-healing mechanism works, how it must be implemented, and how the maintainer should evolve the system's specific knowledge during its lifetime.
7) *Evaluation* – Describes how this work was evaluated on real systems, all executed in controlled environments capable of measuring the solution effectiveness.
8) *Comparing with Related Work* – Discusses how this work is compared with others in the field of fault diagnosis and failure handling.
9) *Conclusions and Future Work* – Discusses the contributions and proposes some future work.

2 Problem Formulation

Since faults are a concern to be coped with, software systems must provide mechanisms to detect failures at run-time and attempt to avoid the correspondent consequences by employing recovery strategies. Obviously, these mechanisms should be designed and inserted at development time. However, faults remain undetected until they cause the first observable failure, otherwise they could have been removed or encapsulated during development. Unfortunately, there are faults that, despite being known, cannot be removed – faults due to external conditions such as hardware defects, for example, or faults localized in a third-party asset, which will go along with the system during its lifetime, or until a new version of the asset is released. Such faults are often identified while the system is being developed or evolved.

When a failure is observed at run-time, it must be diagnosed with the goal of determining the root cause, which corresponds to a failure signature (the event correlation that represents the misbehavior), thus enabling the development of a recovery handler for this kind of failure. After that, the effort of removing the fault must be estimated and compared to the effort of implementing the recovery mechanism. This effort, combined with other factors, such as the criticality of the consequences and the availability of the development team, should be used to decide if the recovery mechanism must be implemented, in order to handle future occurrences until the fault is removed. Observe that some systems require more effort to remove faults due to several factors, such as redeployment time, the absence of the development team, or simply the fact that the code modification is complex and a proper solution requires substantial effort to be designed and developed. In short, fault removal can take up days, weeks or even months. While waiting for a new version, the system in use is vulnerable to repeated occurrences of the same kind of failure. Therefore, if a temporary recovery handler could be implemented in a reasonably short time, it would be worth the effort to handle future occurrences until the fault is removed and a new corrected version of the system is deployed.

After deploying the recovery handler, maintainers can plan when and how the causing fault will be removed, based on its severity and the human resources needed to execute the necessary tasks. Hence, there are two diagnosing goals:

1. Determining the root cause in order to remove the fault.
2. Identifying the failure signature in order to develop the recovery mechanism.

If the first goal cannot be made in a period of short time, the second should be sufficient for properly handling the failure. Therefore, a promising solution to cope with known failures is learning from the diagnosis result and developing a specific recovery handler for these failures. The shorter the execution time for these tasks (diagnosis and recovery handler deployment), the sooner the system will be protected from the recently discovered fault.

The rest of this chapter will (1) explain why the information extraction problem plays the main role in the solution and, thus, why it is the focus of this thesis; (2) discuss which are the limitations of the available approaches and what can still evolve; (3) present a list of requirements the solution must fulfill to become acceptable; and, finally, (4) present an overview of the proposed solution, which will be described in Chapter 3.

2.1 General runtime information extraction problems

Adequate information about system execution is needed to aid maintainers in the process of error detection and diagnosis. Traditional techniques are based on logs, which are sequences of messages in a human-readable format (Hansen & Siewiorek, 1992), where the corresponding instrumentation has been written by developers. A log message is called an event. It may contain values that describe the state of the execution at the moment it is notified (Gülcü, 2002; Liu, 2007), such as context variables and, if possible, the stack trace of the execution, usually inserted in events that represent errors. Libraries such as Unix syslog (Lonvick, 2001), log4J (Gülcü, 2002) and Microsoft Event Logging (Murray, 1998) support this approach. When an event is generated, these libraries also append the current local timestamp to enable temporal analysis, before storing the event in a local file or in a remote database.

Although this technique may produce some result, the effort required to analyze the log is usually huge (Mendes & Reed, 2002). Distributed system failures are often hard or even impossible to replicate, hence the data available at the moment of failure detection should ideally provide sufficient information to support diagnosis and removal of the causing fault without the need for replicating the error (Skwire, 2009). Furthermore, in distributed or multi-programmed systems, an incorrect state may be itself distributed, which may involve states of several processes. Hence, the data available at the point of detection (i.e. within a specific process) is not necessarily sufficient to provide all the data needed for a proper diagnosis.

Undoubtedly, the log technique helps understanding system execution. However, several authors have identified limitations in this approach:

- The log set is often very extensive and presents information from different contexts mixed in the same dimension (Mariani & Pastore, 2008), reducing the visibility of information that is needed to detect and diagnose the failure.
- The log files are usually distributed over various machines (Liu, 2007), imposing an additional effort to access and adequately organize them in a chronological order, needed for determining inter-state faults.
- The available information is very often insufficient whenever the application context is not represented in the events (Oliner & Stearley, 2007).

Considering all the different contexts, it is hard to correlate events creating logical links that could explain the undesired behavior. While diagnosing, simply searching for the keyword “error” in logs may find evidence that a failure has been detected, but this is usually insufficient to determine the failure footprint (sequence of events that matches the failure signature) and, hence, to create a precise diagnostic (Oliner & Stearley, 2007; Jiang et al., 2009), as we need much more information about the system’s execution to understand the scenario that led to the failure. The most challenging failures are not the ones that will crash the system immediately, but the ones that corrupt some data and drive the system to unexpected behavior after long runs (Liu, 2007). To diagnose these failures, we need to study execution histories and have access to properties that could explain the unexpected behavior.

Notice that we are addressing the problem of diagnosing and handling failures in a deployed system while running in a production environment. During the development and test phases, other approaches are available for the problem, which may not suffer from the limitations discussed in the following sections.

2.2 How to extract relevant information?

Several works present solutions to aid behavior understanding (discussed in Chapter 8). However, the information extraction technique always relies on three main approaches:

- Logs written manually by developers (Ruffin, 1995).
- Static traces inserted by a language or binary preprocessor (Lindlan et al., 2000).
- Dynamic traces inserted at runtime (Maebe et al., 2002).

Log and trace approaches are very similar. The main difference is that logs are usually written directly into the source-code with some abstractions, but without a fixed structure, and are intended for system administrators; traces, on the other hand, are generated automatically, based on some insertion criteria defined by the developer and are usually designed to track execution behavior or to measure non-functional requirements, such as performance and resource allocation. Each logged event may be categorized in a few, fixed set of types, later used to filter events using a perspective of interest. This solution is usually ineffective, due to incomplete information stored in events and the inability to foresee which perspectives of interest will be needed, since each of these perspectives must be created from the specific characteristics of each failure. Traces present a higher frequency than logs, and usually capture the current state of system properties, which are, in addition, stored in a more accessible way than logs, since they are indexed by their name, not blended in text messages. Due to this well-defined format, traces are more appropriate for automated analysis, since they present the event properties in an indexable form, avoiding the use of a mechanism to extract them heuristically, thus introducing errors or imprecision as discussed before. However, this approach also imposes a higher overhead than logs, due to event notification frequency. The solution presented in Chapter 3 is a hybrid approach, based on logs generated using automated features that allow them to be classified as specialized traces. The rest of this section will discuss in more depth the problems found in extracting relevant information that are intrinsic to log and trace approaches.

One of the hardest problems in the field of runtime information extraction is the definition of the data that will be necessary in further investigations. It is particularly hard due to two issues: (1) acquisition granularity and (2) the set of properties that must be present in each event. The granularity is associated with the number of event notifications along the system behavior, which usually follows an instrumentation policy, even when not formally defined (developers tend to insert event notifications in every code block controlled by conditionals and repetitions). The second problem, the definition of the property set, is controversial, since it is impossible to know beforehand which information will be needed during a diagnosis session. The main reason is that every diagnosis session investigates newer failures, with little and sometimes no guess to formulate hypotheses. As previously mentioned, an unstructured log usually contains insufficient information due to the difficulty in expressing the full context in each log event. In this approach, event notifications are written manually and if one wants to notify all properties of all scopes in each event notification, the task would consume an effort that invalidates the approach.

When the information is insufficient to diagnose a failure, developers try to replicate the scenario that lead to the failure in a controlled environment. This allows them to inspect the execution environment with a debugging tool such as GDB (Stallman et al., 2002), which enables the operator to pause the execution at each point of interest to evaluate the state of the process. In large systems, with many concurrent users, it is very difficult to find out how to build the initial scenario, since the internal state related to the failure is unknown, and the user may not be available or might be unable to explain what he did or was trying to do. Furthermore, these approaches are subject to *heisenbugs* (Gray, 1985), which are failures that disappear (or appear) when inserting or removing instrumentation, and are hence almost impossible diagnose.

There are solutions based on capture & replay techniques (Wittie, 1988) (Steven, 2000), also for distributed systems (Geels et al., 2006), which store the full state of each process following a given execution granularity and, after a failure is detected, the initial scenario can be precisely recreated. However, this type of solution has a considerable impact on the system's performance, since it requires much storage or disk space and consumes a fair amount of network bandwidth in spite of trying to use lightweight approaches. These problems take us back to the original information extraction problem: how to extract a system's runtime information in such a way that it exposes relevant properties just when the failure occurs for the first time without facing a tradeoff between performance overhead and information completeness?

There are several works based on tracing that automate the task of event notification (Maebe et al., 2002; Mirgorodskiy et al., 2005; Toupin, 2011). These approaches capture the entire state at the moment of the event creation, which contains all the properties available in the runtime stack, and obviously provide sufficient information to aid in the diagnosis process. However, they tend to overwhelm the user with much more information than is needed, since it captures auxiliary variables and complex objects that will not present a readable description. Also, they cannot be applied to large systems, since capturing all properties would impact the system's performance and the cost of storing all the generated data would be too high.

Lightweight approaches that focus on distributed systems (Hendrickson et al., 2003; Reynolds et al., 2006) solve this problem by capturing only events related to node interaction. These works claim that distributed systems are hard to diagnose and their solutions focus on understanding communication behavior between nodes, keeping each component as a black-box, and discarding the possibility of diagnosing internal logic failures. However, the internal logic of these components is also difficult to debug due to all the problems listed in the previous section.

Another concern related to log content is abstraction preservation, which is divided into multiple levels, from hardware to software conceptual models (Maebe et al., 2002). An *abstraction* is the result of a cognition process where software engineers and developers remove details from complex definitions and generalize them as a virtual entity, design, or structure, which receives a name that will thereon be used to reference them, reducing the effort in describing its full complexity (Timpf, 1999; Kramer & Hazzan, 2006). When a software model is transformed into a source-code, some high-level abstractions are lost since programming languages usually do not provide means to represent them. Then, when source-code is converted to a machine language (a set of instructions and symbols), application-specific abstractions are lost because the machine does not need them to execute the software. Reflection and introspection techniques available in some modern languages (Cazzola, 2004) provide means to retrieve some language abstractions such as class definitions, methods, parameters, etc. These techniques enable approaches, such as automatic traces, to extract the software call hierarchy and notify events when each method is executed. However, high-level abstractions are not represented since they were lost in the coding phase. The presence of these abstractions in the runtime information would reduce the effort of the cognitive process of understanding an expected behavior, since they would remove complexity by representing the execution using the same entities, designs and structures created by developers and learned by other maintainers. A solution is, therefore, needed to bind the runtime information with these abstractions, enabling the process of diagnosis to act not only at source-code level, but also at design level.

Despite of the effort demanded by manual instrumentation, it achieves better results in terms of abstraction preservation than the automatic approach, since developers describe each event using application-specific abstractions. However, in addition to properties in the instrumentation data possibly being insufficient, they are blended into the notified messages without maintaining homogeneity among different events. Assuring that the entire software will follow a fixed set of terms to represent abstractions is very difficult, due to the fact that large systems are written by different teams, which may adopt different names for the same abstractions. We must stress that property name uniformity is extremely important to reach maximum precision in further event correlation (Hendrickson et al., 2003), thus improving the results of log analysis. Another difficulty is property accessibility due to modularization. In some situations, it is impossible to identify the context that leads to a local scenario, because the necessary information to identify it is contained in outer scopes or even in a different process. Dependence mining techniques try to extract this information from raw logs to create the relations (Lou et al., 2010). However, the effectiveness of the result relies on the log uniformity discussed before.

There are also technology limitations imposed by each extraction approach: (1) log events require a log library implemented for the target programming language; (2) static traces require a tool capable of interpreting the source code and injecting calls for the tracer; and (3) dynamic traces require a specific library for the target language that provides introspection and reflection capabilities. Solutions that are more easily applicable to different languages and become portable between operational systems are more appropriate for modern systems, since their subsystems are usually implemented in different languages and executed in many environments (desktop, mobile, cloud, etc).

The conclusion is that automatic instrumentation usually induces a small extra development cost and implies some technological limitations, but generates a large volume of data, many times containing information of little use. Manual instrumentation, on the other hand, is inserted by the developer and requires a noticeably greater effort to implement, exposes few context properties, and poses a considerable risk of inadequacy, but it tends to keep a higher level of application-specific abstractions and has almost no limitations. Therefore, a technique is needed that better balances these four aspects: context completeness, abstraction preservation, instrumentation effort, and technology limitations. Since abstractions are an intrinsic knowledge to humans, and manual instrumentation takes advantage of the developer's expertise and unveils promising paths, it was defined as the starting point for this research. However, solutions are needed to increase context completeness with precise data neither increasing implementation effort, nor introducing considerable technology limitations.

2.3 Richer runtime information are needed for failure diagnosis

Every diagnosis session starts with a failure report created by the person who identified the failure occurrence, which can be a maintainer, an end user or even a software assertion. In the first case, the report is expected to contain technical details, steps to reproduce and few hypotheses for the fault. However, when created by the end user, in a best case scenario it will provide some minimum information on which to formulate the first hypothesis, because ordinary users do not know the application architecture, their specific abstractions or even general software abstractions, needed to precisely describe the unexpected behavior. Furthermore, failures that occur in production are usually harder to diagnose, since the software components have already passed through unit and integration tests during development, where simple failures were identified and removed. Some of the failures identified in production are related to exogenous causes, like hardware performance or network instability, and can be diagnosed by seeing components as a black-box. The occurrence of internal failures, however, needs detailed information about each component behavior, which are difficult to obtain, as we shall see in the final sections.

During a diagnosis session, maintainers and developers use all available information they can access to understand the unexpected behavior. This information is usually in the software source-code, in logs or traces that reflect the runtime information (containing the failed execution), in the version control history, and in additional configuration files, when applicable. A common approach starts by localizing the event that represents the failure occurrence in the log or trace, then back stepping through the execution log, discovering the conditional decisions made, discarding irrelevant events, and gathering properties from some events, until the selected set corresponds to the failure footprint that enables behavior comprehension and the establishment of a diagnostic for the failure. This approach produces good results when events near the start event contain the necessary information, even when it is unstructured. Some failures, however, are more complex and this approach can be hindered by extensive logs, sometimes spread over different threads and routines, requiring the maintainer to intertwine events from different threads, and identify the start and end events that connect two logically-connected routines.

It is important to keep in mind that for an extensive log, a considerable effort is required to answer the most basic questions, like: what was the action being performed, triggered by which feature, from which user, using which environment, what is the client’s version, etc. Note that the failure can be observed inside a function of a common library, while the context of interest is far away, at the beginning of the footprint, and triggered by some user input. The second problem is that the footprint may be segmented along several routines, processes, machines, and environments. The logical relation between these routines can be due to local or remote *call* or *shared data*.

The *call relation* can be explained through the following example: consider a failure identified in a cloud service that interacts with thousands of mobile devices per minute. The maintainer starts the inspection in the server application, but the footprint backtracking reaches the web service interface and the diagnostic is still incomplete. To continue diagnosing, the maintainer needs to find the last event before the remote call, from the exact device that triggered the fault. However, even when these events are available, they are usually intertwined with many of other events, from all devices, requiring a huge effort to find the exact set of events that represents the footprint associated with the failed call in the remote server.

The *shared data relation* can be explained through another example: an application enables editing a given type of record by many use cases, which are implemented in separated source-codes, where some are automated and others are executed through human interaction; the result of a record editing is stored in the application database; many features read these records from the database, and most of them use an auxiliary function to convert the record data to another format. During retrieval, a failure is identified in this auxiliary function. After some inspection steps, the maintainer concludes that the data stored is in a wrong format and that the corresponding fault must, thus, be in one of the editing features. However, it is very difficult to track it back, since the failed routine does not present a clue of which feature wrote the last data into the record. When the diagnosis footprint is not clear, as in this situation, maintainers must hypothesize and evaluate each possibility searching for the traces that modified that exact record. In most cases, this approach is unfeasible, since an unstructured log does not provide the necessary organization to filter routines based on a given perspective of interest.

In addition, there is also a transversal problem that affects every footprint: log events are intertwined, mixing events from different contexts and reducing the visibility of the ones that are interesting for the diagnosis session. The set of relevant events that correspond to a footprint is usually extremely small when compared to the full log. However, to select them the log set must be indexed according to these contexts, which are unknown since their selection depends on the failure under analysis. This leads us to the problem of extracting events with information that makes them indexable. This problem is indirectly addressed by some state-of-art solutions, discussed in Chapter 8. The most relevant is dependency mining (Hellerstein et al., 2002; Zheng et al., 2002; Zhang et al., 2009; Lou et al., 2010), referenced before, which is an approach that aims at extracting properties from traditional logs and traces. Unfortunately, this solution has a considerable lack of precision due to the problems related to runtime information extraction techniques, discussed along this section. The proper contextual properties are usually not represented in the log information, since they are defined at a higher level of abstraction. Therefore, a technique that extracts richer information from a higher level of abstraction is needed to aid in event indexing during a diagnosis session.

Finally, there are some works describing automatic diagnosis techniques (Chen et al., 2004; Mirgorodskiy et al., 2006; Yuan et al., 2010), which aim at dismissing human intervention when handling failures. We believe, however, that human knowledge is still a fundamental part of the diagnosis process, as discussed before, and even if manual work could be partially automated, it cannot be ignored. It is worth mentioning that others, as for instance Bodik et al. (2005) and Xu et al. (2008), follow the same assumption. Current automated diagnosis techniques are appropriate only for superficial failures, not for those that need internal-logic hypotheses investigation, thus requiring human knowledge to complement the necessary information and determine the root cause.

2.4 Automated failure detection and recovery

Once the failure’s cause has been identified, the knowledge acquired can be used to develop mechanisms capable of handling future occurrences of the failure while the causing fault is awaiting to be removed and to redeploy the corresponding artifact. In order to identify the scenario that can lead to a failure, a signature of the failure is needed, and also mechanisms capable of detecting it during the system execution. The resulting detection routine is a translation of the diagnostic applied to a running system, since the diagnostic was defined based on the knowledge acquired during the diagnosis session, which was learned from an execution footprint, composed by a sequence of events exhibiting the state of the system’s properties. This detection rule must attempt to identify the failure signature using the properties available in the execution context. However, the chosen set of properties, and the relations between them, must not be corrupted or masked by those properties that do not contribute to precisely detect the generalization of the failure (i.e. the unique characteristics that classify all occurrences of the same failure).

The most common form of implementing detection and recovery mechanisms is the *ad-hoc* way, based on the developer's knowledge about the fault and his experience with the software domain. The result is not a well-thought out solution, carefully designed for future maintenance; it demands minimum effort to implement and should be removed when the fault has been corrected. This approach is usually implemented directly in source code, sometimes spread over different modules, making it difficult to track back when removing the fault, since the workaround code blends with the normal code (experienced programmers annotate their code with comments to quickly find them in future maintenance). Thus, since the inserted code is just a quick and fast way to handle the fault occurrences, and not a true correction of the normal behavior, it pollutes the logic of the software, increasing code complexity and reduces maintainability.

There are better approaches than *ad-hoc* implementations, which formalize some aspects in order to reduce design degradation. For example, in the context of failure detection, systems developed with design by contract (Meyer, 2002) can insert into the source code executable assertions for each contract defined. These contracts are evaluated at runtime to validate the context, and when one fails, it is possible to act on it. Defensive programming can be applied stopping execution to prevent further consequences. Nevertheless, when a recovery action is known and feasible, a proper handler can be implemented and associated with the executable assertion. Another approach is based on the language exception mechanism, which aids in failure handling by passing the error descriptor through an exception flow in the opposite direction of the call hierarchy, until a proper handler is reached. This approach is less polluting to the software logic and its architecture, since it formalizes the communication between the module that identifies the failure and the one that will handle it. However, this approach still does not reach the expected result, since 70% of the failures are manifested by exceptions, mainly because it is difficult to write an exception handler that correctly recovers the failure signaled by the exception (Li et al., 2007).

Ad-hoc assertions and language exception mechanism approaches are only applicable in situations where part or the entire system can easily be modified and redeployed, and, even in those cases, there are situations when a fault cannot be removed, such as when it occurs in third-party services. When the system can or should not be modified and redeployed, sophisticated solutions are needed to handle failure occurrences. Another limitation of *ad-hoc* assertions is that concurrent system contracts cannot be evaluated using these approaches, since they may require access to states of more than one component, which may be executed in separated processes. Therefore, there is a trade-off between performance impact and consistency while acquiring the global state. Moreover, for some types of failure, it would not be reliable to implement their corresponding failure handlers inside the process, since the error may compromise the recovery routine. The *design by contract* concept is applied in the Hydra solution (Chapter 6), guiding how to write failure detectors over the event flow.

The most primitive solution for fault tolerance without modifying and redeploying the system is the action-base approach (Hansen & Atkins, 1993), which executes a recovery handler based on log patterns, relying on the log content, as discussed before. There are more general solutions that do not depend on execution runtime information, such as checkpointing & restart (Johnson & Zwaenepoel, 1990; Sankaran et al., 2005; Hursey et al., 2007), N-version software (Avizienis, 1995), data diversity (Ammann & Knight, 1988), and different combinations of all these methods (Kazinov & Mostafa, 2009). These works will be discussed in Chapter 9. The fact is that checkpointing has a considerable impact on system performance, and its rollback solution does not guarantee that failure consequences will be properly handled, since the output may have already been propagated for other systems. The other two techniques require a huge effort to implement because they depend on logic and data redundancy, and — even with this robustness enhancement — produce systems with questionable reliability, since failures may occur due to faults being inserted in all versions (Holloway, 2007). In addition, these approaches handle failures with generic solutions, while some failures require specific knowledge about its fault to efficiently handle the consequences.

Most of the work done in the field of failure handling in the last decade is based on self-healing approaches, which are a facet of Autonomic Computing (Murch, 2004). In short, this concept states that every component must be aware of itself; reason about its own behavior and its relation with the system environment; and take actions to better achieve high-level system goals. They are also called self-adaptive systems and present two main control loops: local, for the component, and global, for the environment. This approach enables self-healing mechanisms to target failures with specific knowledge about the fault, learned previously or during reasoning about the cause of the failure. In addition, this technique offers the flexibility to handle failures inside the component execution (local loop) or in the system environment (global loop). However, this approach impacts directly on the component design, thus requiring specialized developers that master the technique and must also be familiar with high-level software concepts. This approach is very powerful and has had great influence in modern software system development, but is unfeasible for ordinary software, since a team with the necessary skill is usually not worth the cost. Solutions are needed, thus, to address this problem without requiring high-level expertise from all developers involved.

Traditional methods and tools, spread among developers and properly adapted to most application domains, must be used as a basis on which to build a solution to aid in failure diagnosis and handling, since they are common knowledge for most developers and would not require specific training. As long as specific information about the failure is required to properly heal the system, the efficacy will continue to rely on the completeness and precision of the recovery footprint content extracted from the system runtime. In other words, to become able to detect and handle new failure occurrences, events generated by the failed execution of the software must expose relevant information. Therefore, the richer the runtime information, the greater the chances of directly using diagnosis results to create the detection rule, thus reducing the effort in detection and recovery implementation tasks.

Finally, there is the problem of knowledge distribution among all team members. Since fault descriptions may vary considering the level of abstraction, some team members may have limited knowledge about the best way to detect and handle the corresponding failure occurrences. For example, in a team composed of developers with different expertise and experience levels, a novice programmer may have developed a software component that presents a risk of failure when integrated into the system. This developer does not have sufficient knowledge to understand the failure scenario in order to develop the solution to handle the failure consequences. In this case, the solution may be developed by an experienced developer and installed at the system level. The opposite case may also occur if the software engineer observes an internal failure in a component and is capable of developing the detection strategy for future failure occurrences, but the recovery routine requires the expertise of the component's developer. In this case, the solution may be hybrid, developed by both team members. This does not mean that team members cannot handle failures outside their expertise, but effort necessary for doing so may vary greatly. Thus, the appropriate solution must provide mechanisms to enable writing failure handlers to be installed: (1) inside a single component, (2) at the system level, and (3) in a hybrid form, as in the Autonomic Computing solution.

2.5 Solution Requirements

This section summarizes the discussion of this chapter, and presents a list of requirements that the solution must fulfill to be acceptable for the problem. Foremost, the main problem addressed by this thesis is the runtime information extraction. However, solutions for this problem are evaluated through the information usage, which is mainly represented by failure diagnosis, detection and recovery capabilities. Therefore, the solution must address these four aspects. The list of solution requirements elaborated after discussion is:

- Must be based on low-cost methods and tools.
- Must rely only on traditional programming paradigms.
- Must be applicable to failures in local, as well as in distributed software.
- Must be portable between different programming languages and environments without implying in considerable development effort.
- The use of libraries and frameworks must not restrict project decisions.
- Must address both scenarios where code can and cannot be changed to detect and recover from a recently diagnosed failure.

There are some aspects that need more attention, such as providing methods and tools that can be applied to ordinary projects, guided by usual developers, and without requiring new paradigms, theoretical concepts, language extensions, experimental tools to assist coding, etc. In short, the objective is to provide means for existing systems to increase their reliability with a solution that does not disincentive its usage due to an overwhelming effort in changing the development course, or even its impossibility due to technical and business issues. Therefore, the solution must be easy to integrate and only minimally impact the way developers write their code.

2.6 Overview of the Solution

With the purpose of achieving a good result in recovery diagnosis and removal diagnosis, tools must help expose the system's runtime information in a way that maintainers can easily understand the execution, elaborate hypotheses about possible causes of the observed failures, and efficiently verify them against the runtime information, until the diagnostic is complete.

After the failure signature is known, a mechanism may be developed to detect future occurrences, through the corresponding signature, and proceed with a recovery routine. Although such a routine is implemented for an exact kind of failure, it must be complemented with runtime data about the failure occurrence to enable proper recovery. Observe that different occurrences of a same type of failure may depend on occurrence specific data, and, hence, the recovery routine must be able to use the footprint data as parameters, in order to properly recover from the current failure instance.

When writing the failure handler, the necessary information must be gathered from the footprint generated by the failure occurrence, from where one must retrieve the set of properties that is needed to write (1) the failure signature for the detection strategy, and, if needed, (2) the set of properties which values will be extracted from future failure footprints and passed on to the recovery routine.

Moreover, the usual *ad-hoc* approach for failure handling tends to modify the behavior that was initially designed to address the system requirements, thus polluting the system's logic as new recovery handlers are inserted without proper reasoning about engineering concerns.

Therefore, the main research questions are: (1) how to extract rich information about the system execution considering the restrictions presented in the previous section; (2) how to enable developers and maintainers to diagnose failures using this rich information source; (3) how to use this information source to support the development of mechanisms to recover from identified failures; and (4) how to implement these mechanisms avoiding spreading temporary modifications in the system's source code (i.e. the *ad-hoc* failure handling approach).

3
Logs Annotated with Contextual Meta-information

The main problem addressed by this thesis is the search for a solution capable of exposing relevant contextual information about the execution behavior, which must be useful for further analysis and must not impose a considerable overhead on performance. As discussed in the previous chapter, the tracing approach imposes an overhead that prevents it from being used in a production environment. On the other hand, traditional logs are not suitable for runtime analysis, due to their incompleteness and absence of structure to handle information. The rationale used as basis in our research is that there must be a way to balance the benefits of logging and tracing through a hybrid technique that imposes a small overhead when applied, while offering a better cost-benefit regarding the quality of the extracted information.

From the problem diagnosis point of view, the three main difficulties encountered by a developer who is using traditional methods to study system logs are:

1. Finding out which machines are involved in the failure under analysis and gather its log files.
2. Correlating events between different machines in order to retrieve the complete footprint between the triggering-event and the point where the failure was observed — an executable assertion, for instance, may identify that an object has an inconsistent state, which is the observation of the failure, but the triggering-event that produced the error may be in the past, where this object was modified, possibly in a different context.
3. Viewing only the relevant events and properties, discarding the enormous amount of irrelevant information from contexts unrelated with the failure.

These difficulties are the reason why manual inspection requires a great effort. The first one is addressed by logging tools that aggregate events from different machines in a central repository. For the second one, there are some proposed solutions in the state-of-art, discussed in Chapter 8, that can infer relationships between executions in different machines, processes and threads. However, these solutions impose either a high impact on performance, or produce inaccurate results leading to many false positives. The third problem presents a greater difficulty: how to select, from the set of events, only those events that are interesting for the diagnosis session.

The simpler and direct solution of separating events into different profiles does not solve the problem, since most events can be classified in more than one profile. For example, an event that represents an authentication fault during login can be interesting both for the developer who is verifying why his component does not interact correctly with the system, and for the infrastructure administrator who is verifying why an user cannot execute the login action. Note that not every security event will be interesting for the developer, neither will every login-related event be interesting for the system administrator. Moreover, if we try to generate profiles based on groups of contexts, a recently discovered fault may require a profile based on an unexpected combination of contexts, for which the possibilities are countless. Thus, the profile — or, better, the perspective of interest — must be precisely created for the specific failure under analysis, embracing all contexts that may have some connection with it. However, it is impossible to foresee all possible profiles that will be needed during a diagnosis session, since one does not know which faults a system will present after deployed. Therefore, it is necessary that the solution enables the generation of perspectives of interest on-the-fly, based on the contexts the developer desires to inspect, which are learned from the failed scenario and combined with acquired knowledge about the system.

Information extraction also presents some problems for detection and recovery mechanisms. Since the tracing approach is impracticable for a production environment, logging becomes the more appropriate applicable solution. However, due to its unstructured format, the system properties embedded in the extracted information are hard to access, and even when some of them are available, they are usually insufficient to precisely diagnose de cause of the failure. Mining solutions (discussed in Chapter 8) produce an imprecise result, and are, furthermore, limited to the information available in the string that represents the event, which is restricted by the way developers report events during the coding phase, and hampered due to scope visibility, as we shall discuss later.

While developing failure handling mechanisms, actuators are implemented by transferring developer and maintainer knowledge into code fragments, scripts, configuration files, and whatever is needed for these mechanisms to identify and handle failure occurrences. During this task, developers study the system's behavior through the information in the execution flow, in order to determine the scenario characteristics that represent the failure and, thus, must be used to detect future occurrences. As a simple example, it is possible to determine that an operation failed looking for an event that represents the beginning of an action without a corresponding event that represents the end of this action. This scenario is the signature of the failure, which ends up consisting of relations between events and their properties, which are the atomic fragments in the execution flow and, therefore, the main source of information for detection and recovery mechanisms. In other words, these mechanisms must use the sequence of instantaneous, possibly parallel, states to gather the necessary information to:

(1) Identify scenarios that can lead to or represent a failure.

(2) Feed the recovery mechanism, specifying how it must behave to handle that specific failure occurrence.

However, here we face the problems discussed in the beginning of this section: logging techniques producing lean results, with few and hard-to-index properties, and a resulting scenario requiring events and properties that do not exist on the extracted information, thus imposing software modification and redeployment in order to attempt identifying the missing information in future failure occurrences.

Even when considering systems that can be partially redeployed and allow evolutions of the instrumentation, some of the properties must be fetched from scopes that are not visible at the place where the event is reported, such as outer scopes, other modules, or even other components. To address this problem, the instrumentation solution must consider the execution context, and not only the data available at the point where the event is notified. Additionally, the solution must also provide a mechanism to index event properties in a straightforward way, necessary for detection and recovery mechanisms.

Therefore, the extracted runtime information design plays a fundamental part in failure diagnosis and failure handling, by influencing the event selection capabilities and easing the knowledge transcription process, respectively, thus influencing these tools and mechanisms effectiveness. Ergo, the information extraction mechanism is the main problem addressed by this thesis, which proposes, as a solution, a novel approach based on hybrid instrumentation that enriches events based on the software context, increasing the chance of the required information being available during failure diagnosis, detection and handling and, thus, ensuring the mechanisms effectiveness. The main requirements for this solution were listed in Section 2.5.

3.1 A Hybrid Instrumentation Approach

The way to represent and select the events among all those available in the execution flow is the key to develop solutions for diagnosis and failure handling. Ideally, while performing a diagnosis, only the events that match a context of interest should be selected and displayed in a unique temporal list combining the execution flow originating from different components. It is interesting to find out location information — e.g., from what machine the event was sent, which process was running, in which component, in which procedure, triggered by what user, some local and global variable values, the execution stack point, etc. It is desirable to identify non-explicit relationships between the events, which can often provide clues to determine the cause of the failure. Finally, it is important that only the necessary information from each event is presented to the user, according to his/her perspective of interest. The needs for detection and recovery mechanisms are similar, since they require events with properties that can be easily indexed, in order to evaluate if each known failure scenario has been activated. Therefore, those system properties are the key-solution for developing tools to support diagnosis, and mechanisms to support failure handling implementation. Chapters 5 and 6 explain how to combine these tools and mechanisms with this instrumentation approach.

Current logging techniques insert these properties in an unstructured form, within a human-readable message, which is not appropriate for indexing. Hence, these properties must be appended as meta-information instead of raw text in the message. In our solution, every event is a composition of *tags*, which are key-value pairs representing the state or value of a property at the exact moment when the event is generated. The key is the name of the property being notified. Values are optional, since in some cases the goal is merely to inform the presence of a property — such as *Error*, for example —, sometimes not conveying any other information. In other cases values inform some state, for example, the id of the component responsible for a computation. The set of properties that may be recorded in an event is not limited and does not have to follow a given schema: the developer may inform any property he/she wants, in any point of the execution. However, there is a basic set of tags that must be present in every event, which is guaranteed by the log library. These tags are:

- A *timestamp* used to sort the events from each thread into a timeline.
- A *message* representing a human-readable description of the event (the traditional log).
- An *action* describing the high-level intention of the current procedure under execution.
- The *location* representing the runtime origin of the event, defined by the application domain (ex: component X, thread Y).
- The pair *file*/*line* representing the place in the source code where the event is notified.

The rest of the tag set must be defined based on the project's artifacts, such as design, architecture, domain, etc. This topic will be discussed in the next session.

The event notification calls are inserted into the software using a novel instrumentation technique that enables developers to insert properties about the current context as meta-information into the event. During the execution, each routine exposes its runtime information through these notifications to a central repository. This repository stores all the events received, keeping the structured format, which will be exploited by tools and mechanisms later on. Nevertheless, applying this approach, alone, imposes more effort than the traditional log approach, since it requires developers to expose more information and also reason about what information might be important in the future.

The recorded event must be composed of all the relevant properties about the context, leading us to two potential problems: (1) breaking encapsulation, due to the need to access variables defined in outer scopes, or even in other modules; and (2) abhorrence, since all context variables must be inserted in every event notification. To solve these problems, our solution introduces the concepts of the *scoped-tag* and the *tag-stack*. The *tag-stack* contains tags that represent high-level abstractions and must grow and shrink synchronously with the execution stack. There is only one stack per thread, and sub-threads are treated as new threads, identified by name. During the execution, each new scope may insert information into the *tag-stack* using *scoped-tags*; and, when the scope ends, all corresponding *scoped-tags* must be removed. In order to avoid the risk of forgetting a *scoped-tag* in the end of the scope mechanisms based on the current programming language are employed, which are explained in Chapter 4. Following this way, all event notifications executed inside this scope are enriched by tags present in the *tag-stack* of the current thread, thus solving the abhorrence problem, since the effort of inserting the specific tag is made only once per scope (considering, mainly, inner scopes). For illustration, consider the following code:

```
int myFunc(char strategy) {
    log.push('choice', strategy);
    int someValue = f();
    if (someValue > 0) {
        log.push('curr_value', someValue);
        if (strategy == 'x') {
            other.executeX();
        } else if (strategy == 'y') {
            other.executeY();
        } else {
            log.notify('Invalid strategy', 'error');
        }
    }
}

...

void executeX() {
    log.notify('Executing X strategy');
    ...
}

...

void init() {
    log.push('component', 'myNode');
    log.notify('Starting');
    myFunc('x');
}
```

Then, if we call *init()* we may have an output similar to:

```
[component:myNode] Starting
[component:myNode] [choice:x] [curr_value:12] Executing X strategy
```

Observe that all tags defined in outer scopes enrich the inner notification, despite being inserted due to a function or language control scope. There is also an example hereafter to demonstrate that properties intrinsic of the notification may be inserted as additional arguments in the notification call. Therefore, following the *scoped-tag* approach the required effort is dramatically reduced, since developers will only need to annotate scopes with properties that must be present in all related events. In addition, encapsulation will remain unchanged, as inner notifications will take advantage of already recorded outer properties without needing to explicitly access them.

Finally, this instrumentation approach solves the problem of extracting properties from higher levels of abstraction in the execution flow, since these properties are inserted by developers during the coding phase, when they can transcribe more knowledge about the system's design and architecture than while just executing maintenance activities. This approach is classified as hybrid since it enables developers to manually transpose high-level abstractions, as in the logging technique; however, being automatically extracted in an indexable format as in the tracing technique; thus, providing richer information that better supports tools for diagnosis and mechanisms for failure handling.

3.2 Defining the Instrumentation Policy

An existing issue in logging techniques is defining the instrumentation policy in such a way that it assures a coverage rate that sufficiently describes the execution behavior. Following the presented hybrid approach, sufficient coverage can be achieved since the activity of inserting event notifications also inserts contextual properties, which may occur during the event creation or in source-code scopes. In addition to the definition of granularity and strategy of event notifications, the policy must also inform rules to notify scoped-tags.

There is no generic formula to address this issue, since information extraction requirements change from system to system. Hence, all team members involved in system creation must participate in the policy definition, contributing with:

- Knowledge about the domain;
- Abstractions developed during the software modeling; and
- Their own creativity, as commonly used in the development of functional requirements for novel solutions.

Thus, the policy manifest must emerge from discussions that use these three core elements (listed above) as a starting point, and the result must be presented as a document with two sections: the first one with a list of rules for inserting event notifications, each one with an indicator defining if the pattern must always be followed or if it may be skipped depending on component's risk for the system; and the second with a list of tags with the property name, an associated description, and when applicable an indicator if it must be used as a *scoped-tag* — meaning that this tag always represents a contextual property and must be present in a group of events in order to make them relate.

When instrumenting, it is crucial to know the risk represented by each component, in order to guide the developers while following the indicator on the first section of the policy's document. Thus, it becomes necessary to define the degree of instrumentation that must be applied to each of them. Also, after the conclusion of the policy document, the developers must discuss and decide how to rate all components defined in the architecture. This rating will define the instrumentation granularity for each component, which must be directly related to the expectations of using its runtime information for any of the further applications: fault diagnosis, failure detection, and recovery. Observe that these definitions are intrinsic to each specific system and must, thus, be defined based on the risk each component implies to the system's goals. While the system is being developed, the complexity of each component, directly related to the risk of inserting a defect, might be evaluated based on metrics (McCabe & Butler, 1989; Chidamber & Kemerer, 1994; Macia, 2013), such as McCabe's cyclomatic complexity, total number of lines of code, fan-in, fan-out, number of services that share the component or even the running instance, or, yet, on the intuition the developers have, based on the component's responsibilities into the solution. However, when the system is in productive use, this risk might be defined based on the component's history of failures, flagging the ones that have participated in the footprint of previous failures. In addition, this definition can be automatically evaluated based on the extracted information itself, through applications implemented over our solution basis — these, however, will remain a theme for future work.

Moreover, this technique is meant to be used as a lightweight approach, allowing the maintenance team to learn from the flaws during the software's lifetime and to add new instrumentation as needed. The rest of this session will describe some guidelines to aid software developers in the task of defining the instrumentation policy. The resulting rules of the policy may not be complete in the early phases of the system's lifetime. Some failure occurrences would require more information than which is exposed, thus imposing additional instrumentation. As risk and requirement specifications, this policy must be refined along the software lifetime while learning from its weaknesses.

3.2.1 Programming Language Abstractions

The most generic approach for applying the proposed instrumentation technique is using programming language abstractions, since they are very similar among different languages. For example, it is undeniable that the set of abstractions provided by all C-inspired languages (C++, Java, C#, etc) have a large intersection between them, such as classes, methods, parameters, operators, and flow controls. However, these are also the abstractions that can be retrieved from automated solutions, such as traces, since the information needed is already available on the software itself. Nevertheless, they are also valuable when combined with high-level abstractions, and should, thus, be present on the instrumentation policy.

The most trivial approach is to insert at least one event notification per edge in the syntax tree, leaving breadcrumbs to track runtime decisions when needed, as the example in the following code:

```
log.push('action', 'generate_report');
log.notify('Initializing the report generation', ('group',
   request('param_group']));
if (report == 'simple') {
    log.notify('Simple report chosen');
    ...
} else if (report == 'full') {
    log.notify('Full report chosen');
    ...
    for (obj in report_list) {
        log.notify('Processing item', ('item', obj));
        ...
    }
} else {
    log.notify('Invalid option', 'error');
}
```

Observe that whichever path the program executes, the footprint will be able to retrieve all decisions made, thus supporting further behavioral analysis.

Moreover, there is the problem of information content in event messages, which might be assisted by applying coding techniques such as writing comments into a logic sketch (a source skeleton with defined flow controls), disclosing an explanation of what the following code must do, before its writing. The same content on the comments may also be used as content in event notifications. In addition, this coding technique is renowned for forcing the developer to think about the logic before writing it.

Another guideline is to insert notifications into the usage of language exception mechanisms. For example:

```
try {
    log.notify('Synchronizing');
    ...
    while (something != otherthing) {
        log.notify('Current state', ('state', something));
        ...
        if (!result.valid()) {
            log.notify('Invalid result', ('data',
               request.data));
            throw exception('Invalid result');
        }
    }
} catch (exception e) {
    log.notify('Unexpected operation', 'error',
       ('stacktrace', e.stacktrace()));
}
```

The notification should explicitly inform an *error* tag when the state is surely wrong, and a *warning* if the unexpected local result might be considered valid for the external environment. This decision relies on the knowledge the developer has about the system. The *warning* approach is controversial and is considered a *bad smell*, however one may disagree. At least, this warning tag may be a tip for the diagnosis session, if a failure actually occurs. Moreover, the throw scope must also be instrumented in order to expose properties that may be involved in the unexpected behavior — for example, the parameters of a failed request.

Another suggestion is to associate the event notification with other development methods, such as *contract-driven development*, and use an executable assertion for each contract item to notify the failed ones. This approach can be further enhanced by also reporting valid assertions, which confirm that the system is presenting a correct, expected, behavior. Moreover, these assertions can be used as checkpoints for verification and actuation of recovery mechanisms.

Observe that all previous examples insert an event notification in every decision edge of the program. While instrumenting the developer's task is to expose as much as he can, considering all decisions made by the software during execution, including, mainly, those that are unexpected, which may aid as tips for the root-cause diagnosis. In addition, there is the problem of execution edge completeness (granularity in path coverage) versus system performance. Thus, in order to address instrumentation for deployed systems, the developer must avoid decision edges that would have higher impact on performance and would contribute little for a diagnosis session. For example, the *size* method of a generic array may not be instrumented since it is called several times along the execution. On the other hand, it should be notified when it is used by a more complex code (for example, when used in a parser that processes a command received by a data stream). Furthermore, the key to enhance the instrumentation's efficacy is the adaptation for using the technique presented here paired with existing methods that suggest better coding practices, such as *contract-driven development* (Meyer, 2002), *correctness by construction* (Hall & Chapman, 2002), or even the *pragmatic programmer* guide (Thomas & Hunt, 1999).

3.2.2 Architecture and Design Abstractions

The second main guideline is to define the policy based on software models, which must expose the decisions made regarding the system's architecture and design. Developers must take advantage of high-level abstractions created during the modeling phase to define them as properties that will be notified in events, inserted as scoped-tags or not. These are the same abstractions the person who is diagnosing will attempt to find in the footprint, even when possibly not being aware of it.

From software developed by model-driven methods to software developed using simple, ad-hoc sketches, the process is the same: model artifacts must be studied and each abstraction identified must be rated based on its relevance to the system's main goals, then selected or not to be represented as a tag when referenced in the source code. The set of properties that must be inserted into the instrumentation depends on the architecture, and some examples will be presented and discussed hereafter. However, from class and sequence diagrams (if available), we might apply a generic approach: abstractions derived from sequence diagrams must answer questions about the program behavior, while abstractions derived from class diagrams must answer questions of which entities were participating, and which data was being handled. In order to accomplish these goals, we introduce the tags *action* and *record*, which are native tags in the solution. A tag *action* must be created as a scoped-tag for each high-level operation being executed, with the possibility of being nested. The value of the action tag is the operation name, for example:

```
log.push('action', 'sync')
...
if (invalid_report) {
    log.push('action', 'build_report');
    ...
}
```

Similarly, a tag *record* must be generated when an object representing a data record is used (created, removed, or modified), having as tag-value, in the worst case, a tuple composed by the record type and the identifier, and, in the best case scenario, the full state of the record. The problems with always storing the full state regard the space required and language limitation, since those without reflection mechanisms would require an additional effort to serialize each object type. Here is an example, considering a language with reflection capabilities:

```
database.save(new_user);
log.notify('Finised user registration', ('record',
   new_user));
```

Additionally, since this is a novel technique, future research is encouraged to address other model types and formalize how they can be studied in order to guide instrumentation policy definition. Furthermore, each architecture type exposes its intrinsic abstractions, which must be transformed into tags in order to enable developers to benefit from them while diagnosing or implementing failure-handling mechanisms. For example, every distributed system architecture may have a tag indicating the entity identifier, and a tag indicating the remote procedure call, since these are the most basic properties in software divided into several independent parts. However, the semantics of these properties may vary between different architectures: a client-server application will use a *client_id* tag to identify each piece of software interacting with a central server (assuming there is only one) and a tag request to reference each RPC; a parallel architecture using map-reduce will prefer *mapper_id* and *reducer_id* as tags for identifying each node in the cluster; and, finally, component-based systems, with much more complex abstractions, will use *instance_id* for each running component and *instance_type* to annotate the type of component.

Moreover, depending on the requirements of the instrumentation design, more abstractions may be necessary to achieve the expected efficacy when using the runtime information. The client-server architecture may also expose the client environment and the request feature. Observe that it is possible to aggregate different actions through a common tag, for example, using the tag [feature:info_update] to group three request types that are used together for the same feature, albeit in separated requests: (1) load the webpage template; (2) load available controllers based on the user profile; and (3) retrieve the most recent data related to the information service. Following this policy, a developer inspecting this system's behavior will be able to select all events from a feature by using this abstraction, which aggregates all actions associated to it. Another good example is the component-based architecture, with running systems presenting heterogeneous entities with different types of components and more than one instance per component. This type of architecture requires, for instance, one tag to represent the component type (*instance_type*) and another one for representing the running instance (*instance_id*), thus requiring more tags to represent the service usage, depending on the implementation. A message-based system, for example, will require tags for identifying the message instance along the execution flow (*message_id*), its type (*message_type*), and, if possible, the content (*message_content*). Finally, architectures such as Model-View-Controller (MVC) may also take advantage of this technique by extending the *record* tag to map the model elements, then create event notifications for controller manipulations and update notifications for these elements.

One of the key-characteristics of this technique is its flexibility to adapt to the system's specific architecture, therefore it is not expected that a single, thus generic, instrumentation process will achieve success in exposing the system's abstractions, since these abstractions are derived from very specific decisions made based on the problem each system is solving. Again, we encourage future work aiming at refining and formalizing these guidelines based on other architecture types.

3.2.3 Domain Abstractions

The third main guideline is related to abstractions identified in the software domain. They must be gathered from requirement specifications and knowledge acquired from the environment where the system will be used. For example, a system designed for chatting applications may involve abstractions such as *room*, *conversation*, *message*, *sender*, *receiver*, *profile*, among others, which will be transformed into tags and appended to events such as "Sending message", "Receiving message", and "Network down, failed to send". Another example, enumerating a completely different set of abstractions, would be a mission-critical embedded software for monitoring oil pipelines, which might require tags to represent *joint positions*, *level of detected leak*, *level of battery*, *radio status*, among others; and notifications such as "Sampling oil sensor", "Evaluating risk", "Notifying alarm", "No satellite network available", etc.

Furthermore, the resulting set of tags from the domain should follow a terminology that is familiar to both the users and the maintainers that will provide support for the system. This approach increases the chances of a maintainer who did not participate in the development, or even a non-technical user, investigating hypothesis about unexpected behaviors; we do not expect, however, with the present technology, to enable these actors to produce detection and recovery mechanisms, since this might require internal knowledge about the system.

3.3 Threats and Solutions

There are two main threats in this approach for extracting runtime information. The first one is that it requires developers to insert event notifications and manage tags between scopes, which may be exhaustive even with the *tag-stack* solution that avoids re-inserting contextual properties in every event notification. Observe that some guidelines may require an enormous effort in their implementation, such as in saving properties of each remote request: developers following this approach will have to write a code block at the beginning of each request handler in order to stack up a bunch of properties related to the specific request. Other guidelines may be error-prone, requiring a log for each database manipulation, for example, something one may easily forget to insert. These difficulties could, in theory, reduce adherence to the technique, but the problem is solved combining basic software reuse principles, which we shall discuss in Chapter 4 and assess in Chapter 7.

The second threat is the necessary disk space and the communication bandwidth for transferring and storing all generated data. Since each event contains a fair amount of contextual information, which increases the volume of the required space, a log database using this approach may grow quite fast. If not addressed, this problem would turn the solution proposed here non-cost effective. The required space issue may be mitigated, however, by creating an event discard policy to maintain the stored data volume near to a given size limit. The rest of this section will describe some conjectures about this discard policy, which was implemented on some of the systems used in the evaluation of this thesis (Chapter 7), however the discarding technique needs further experiments to define how it must be designed and applied for each system, and will, thus, be addressed in future research.

Observe that the proposed approach generates less data than tracing, since it does not save the full state of the system, however having a log size limit, eventually some data will be needed to be discarded. It may compromise the efficacy in failure diagnosis, which implies a tradeoff between the log size limit and the quality of information to support diagnosis. The proposed solution leverages the chances of having the required information when needed.

The discarding solution works as follows: when the database size reaches its size limit, some of the stored information is selected for elimination. In this case, which information should be removed? Traditional log approaches uses a *first-in first-out* (FIFO) policy, which removes the oldest events first, and are not necessarily the less relevant information in the log. When dealing with a log where each event contains lots of contextual properties, sophisticated methods can be applied in order to avoid removing relevant information, which could aid in diagnosing failures. Observe that the information can be only part of the event, i.e. just a few tags, not the entire event.

Our proposed solution is based on the assignment of a *time-to-live* property (TTL), representing days, for each generated event. The value for this property must be defined based on the information the event is carrying, thus enabling that relevant information remains more time in the log. Our methodology suggests that the instrumentation tool must propose an initial value, and then sequentially apply a group of rules capable of measuring the event's relevance, in order to reach a final value that exhibits the estimated relevance for that specific event. This value will be appended to the event as a hidden tag. Also, the rule may change the relevance of previously evaluated events, in order to adequate them to the knowledge acquired later on. Each rule must receive as input the event data and the current TTL value; and return the new computed TTL value. Moreover, during the system's lifetime, a daily routine must be executed to decrement the TTL value of each event and remove those that reach zero.

There is also an extension of this approach that defines patterns for tag removing, thus reducing the space required to store the event instead of removing it completely. This pattern can be (1) a given size limit for a tag type or (2) a set of tags together; both with a threshold based on TTL or event age (lifetime) to remove them. This approach is extremely useful when applied to events that contain large data tags, such as the tag *record*, since the content after the removal represents a summary of what the event was before, indicating, for example, that an object was saved into the database, but in this case without having data to inform what happened exactly. This approach is valid, since having to choose between recent and older serialized data, one will prefer the recent but will keep the option of not removing all of the information about the older event, thus leaving a breadcrumb to its footprint.

The development team must create these rules based on the specific domain requirements, following the relevance of each defined abstraction, and considering the use of heuristics to define how each rule must be applied to a specific system. There are also some generic guidelines that may be used, or at least adapted, to every system, which are exemplified using some values to ease the comprehension of the relation between them:

- Every event starts with TTL=30 (1 month)
- If the event has an *error* tag, its TTL is increased by 11*30.
    - Events with the same *request_id* have their TTL raised by 5*30.
    - Events with the same a*ction* tag have their TTL raised by 2*30.
    - Events with the same *device_id* tag have their TTL raised by 30.
    - Events that present a timestamp with a difference of 1 second have their TTL increased by 5.
- If the event has a [*feature:sync*] tag, its TTL is increased by 30 (forcing events of a specific feature to remain longer in the database).

Thus, there are also tag-based rules, which must be defined based on the TTL or living days (AGE):

- Remove tag *cpu* when AGE > 7.
- Remove tag *memory* when AGE > 7.
- Remove tag *disk_space* when AGE > 7.
- Remove tag *record* when AGE > TTL * 0.7.
- Remove tag *request_post* when AGE > TTL*0.5.

The definition of these TTL increment is subject to future research and were defined empirically during this thesis. The guidelines above just exemplify how to implement the technique. Finally, we have also observed that an analysis involving the computed TTL and tags such as *location*, *file*, and *action* can be used to determine modules, components, or even services that present higher risks, aiding developers to determine flow inspection priority when diagnosing a failure. However, this type of analysis will be addressed in future work.

4
Architecture of the Solution

This chapter presents how the requirements described in the previous chapter can be implemented in a software system. An overview of the solution is shown in Figure 1, which is based on an instrumentation library and a central repository. The instrumentation library must provide primitives for notifying events and handling the *tag-stack* state, while the central repository is responsible for storing the events received, handling the information lifetime, and providing mechanisms to access the stored events based on a perspective of interest. Therefore, the solution is based on a client-server architecture, where each software entity that generates events must notify them through the instrumentation library.

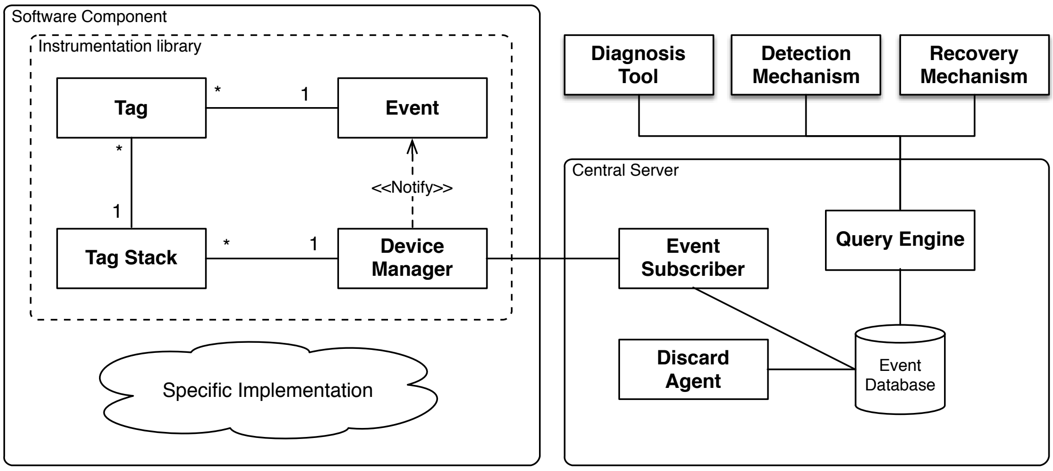


Figure 1 - Solution Architecture

The instrumentation library implements an entity for each abstraction discussed in the theory: an *Event* descriptor consisting of a *timestamp* and a set of contextual properties, all represented as a list of *Tags*; each *Tag* being represented as a key-value pair, where the *key* is the name of the property, and the *value* part represents the property's state at the moment of the associated timestamp. There is also a *tag-stack* abstraction, which handles a stack with contextual properties, easing the information gathering process, and a d*evice manager*, which holds the logic for the notification process.

This Central Server Repository (CSR) contains an *event subscriber*, which provides a remote interface for receiving event notifications. This entity is responsible for adjusting the event data and storing them into a database. It also contains a *discard agent*, which implements the logic for information removal, with a hot-spot to register specific policy implementations. Last but not least, the *query engine* entity provides mechanisms to query data based on a perspective that limits the result according to restrictions selected and passed on as parameters by the maintainer or the automated failure handling mechanisms.

The following sections will specify the instrumentation library interface and explain in more detail the behavior of the CSR.

4.1 The Instrumentation Libraries

An instrumentation library must be implemented for each domain and language used. For example, the C++ instrumentation library may be used in desktop applications and in components of a distributed system, but the requirements are not the same, since the distributed system will need a more sophisticated design in order to handle transmission failures. There are also minor differences between each language implementation, since the chosen language may present some limitations or an intrinsic characteristic that enables a refined approach. For example, in Python language we can define as a tag-value any object type, which is transparently serialized, while in C++ this approach cannot be implemented without a third-party solution for serialization. The C++ language provides, nevertheless, a solution for scope-tag implementation that only works in languages in which destructors are called immediately when the object is deleted, and is impossible to implement in most languages with garbage collection (for it requires a handler synchronized with the moment the object is dereferenced by the allocation scope).

The library interface must exhibit only the operations related to tag handling and event notification, while the operations that deal with multi-threading, event recording, and data transmission must be encapsulated into the library's implementation. Hence, it must provide the following interface — written in IDL (Lamb, 1987):

```
Tag {
    attribute string key;
    attribute string value;
}

typedef sequence<Tag> TagList;
TagDictionary {
    attribute TagList tags;
}

module EventMonitor {
    void notifyEvent(in string message);
    void notifyEvent(in string message,
                     in TagDictionary dict);
    void pushTag(in string name);
    void pushTag(in string name, in string value);
    void popTag();
}
```

Observe that except for tag handling, this instrumentation does not differ much from traditional logging. For example, in Python, a notification could be written as:

```
logger.notify(’Invalid client settings’, {
    ’platform’: ’web server’,
    ’request_id’: ’1234’,
    ’step’: ’verifying client settings’
})
```

This example exposes a problem discussed in the previous chapter, which is the exhaustive and error-prone effort of writing the same property in all event notifications. To illustrate, let us consider that the tag *platform* should be included in every event, and that the tag *request_id* will possibly be included in several events of a feature that handles requests. In such case, to ease the instrumentation task, we use a *tag-stack* mechanism provided by the *pushTag* and *popTag* primitives, which control the abstraction information of the current routine that will be appended to notified events. Refactoring, the code looks like this:

```
# In the 'main function'
logger.push_tag(’platform’: ’web server’)
...

# In the request handler function
logger.push_tag(’request_id’: request.id)
...

# At the notification raising point
logger.notify(’Invalid client settings’, {
    ’step’: ’verifying client settings’
})
```

In addition to eliminating the need of rewriting tags, this approach also eliminates encapsulation violations. The tag *platform* should be present in all events and its value is constant, so it may be pushed directly in the main function of the application. The tag *request_id* is also present in all events that handle a specific request, however its value changes as requests are made and, therefore, it must be pushed in the scope of a specific request.

The function calls for *push_tag* and *pop_tag* must always form a pair; hence, each call to *push_tag* must be associated with exactly one call to *pop_tag*, limiting the scope of the tag. Following the previous example, the *pop_tag* calls would be inserted as follows:

```
# At the end of the request handler function
logger.pop_tag()
...

# At the end of the 'main' application
logger.pop_tag()
```

This approach is obviously risky, since the developer may forget to pop some tags, making the stack inconsistent until the end of the execution. To overcome this problem we suggest adapting the instrumentation library according to the implementation language. The main idea is to consider the tag as a resource and ensure that the allocator entity, i.e. method, is also responsible for its deallocation. For example, the *scoped tag* is implemented in C++ language as a class that allocates a variable on the stack whose constructor pushes the tag, and the destructor automatically pops it at the end of the current scope. In both normal and exception paths the variable will be deallocated. A simplified example of this class is presented below:

```
class ScopedTag {
    ScopedTag(string name, string value) {
        TagStack::push_tag(name, value)
    }

    ~Scopedtag() {
        TagStack::pop_tag()
    }
};
```

The usage of this class can be exemplified as:

```
Response authenticateUser(Resquest req) {
    ScopedTag request('request_id', req.id);
    ScopedTag user('user_id', req.user.id);
    ScopedTag action('action', 'authentication');
    ...

    if (is_superuser) {
        ScopedTag user_type('user_type', 'superuser');
        ...

        if (user_does_not_exist) {
            logger.notify('Invalid user ID', 'error');
        }
    }
    ...
}
```

The same approach cannot be implemented in languages that automatically manage memory, hence do not have a destructor called deterministically when the object reaches the end of scope (dereferenced), only when it is destroyed by the garbage collector. Java and Python programming languages, for example, have this limitation, imposing the use of other approaches for implementing this *syntax-sugar*. In Java it can be implemented through the combination of Aspects (Gradecki & Lesiecki, 2003) with the language *annotation* capabilities, which is found in Python as *decorators*. Both can create function wrappers in order to surround the function call with some extra-operations. These language capabilities enable the implementation of mechanisms in the instrumentation library to create aspects (or decorators) that receive tags as parameters, push them before the call and pop them after, in normal or exception flow. This wrapper may also transform some of the function parameters into tags, register the function name as a tag action, and notify events after or before the call. Here is an example of a Python decorator extracted from one of the systems used to evaluate the theory:

```
def action_tag(f):
    """
    Decorator to insert a tag with the function name as an
    action in the stack, and automatically remove it in
    the exit (normal or exception). Also the time required
    to execute the function will be notified as an event.
    """
    def wrapped(*args):
        start_ts = datetime.now()
        push_tag('action', f.__name__)
        result = f(*args)
        log('Time to execute',
            [('elapsed',
               human_date_format(start_ts, precise=True))])
        pop_tag()
        return result
```

However, this approach does not handle tags that must be pushed onto the stack in inner scopes of a function such as inside an “if” or “for” flow control. There is a solution in Python, using the “*with”* statement, which is not straightforward but enables developers to safely insert tags into inner scopes. It requires a class to represent the scope handler, similar to the *ScopedTag* class presented before for C++:

```
class ScopeTag:
    """
    Container to be used with the 'with' statement, to
    push tags in the middle of a function.
    """
    def __init__(self, key, value=None):
        self.key = key
        self.value = value

    def __enter__(self):
        push_tag(self.key, self.value)

    def __exit__(self, type, value, traceback):
        pop_tag()
```

And it can be used as follows:

```
def myFunc():
    f1()
    if (something):
        myVar = f2()
        with ScopedTag(‘some_var’, my_var):
            f3()
            ...
```

Notice that the property *some_var* is pushed into the stack in the middle of an “if” statement, and will be valid until the scope of the “*with”* statement exists. The same approach cannot be applied to Java, but each language has its mechanisms or extensions to implement a similar solution — for example, in Java language the *Aspect* concept (Gradecki & Lesiecki, 2003) may be used to generate a variety of instrumentation helpers.

Therefore, its imperative that the *pop* primitive is not called directly by the developer, being always used through the *ScopedTag* in order to avoid mistakes that render the extracted data useless.

4.2 Tips to Reduce Instrumenting Effort

Despite of the *ScopedTag* concept, the effort to apply every guideline presented in the last chapter may be annoying for some development teams. While it is impossible to reach an effective result without employing some effort in tag insertion along the code, the software architect is encouraged to search for technical solutions to reuse the code inserted for most general tags.

A practical example is the guideline to address abstractions in the interaction of a remote client with a webserver: in all requests, a set of properties will be gathered, such as user id, session id, request id, POST or GET content, among others. Regardless of the implementation, there is always a point in the code to intercept incoming requests. When implementing from scratch, the developer may use a *command pattern* to execute a given action after and before the request execution; and when using web frameworks, there is usually a hot-spot that achieves the same objective. The suggestion is to use this mechanism to gather as much properties as possible from the request, in order to minimize the effort of other developers during the functional behavior implementation. Observe that the way this approach is designed can be adapted to other guidelines, such as the guideline related to generating an event for every database object manipulation. A solution can be implemented using the *dynamic proxy pattern*, by creating a proxy for the database descriptor, then using the proxy object to notify every action executed, without even having access to the object class. The simpler *proxy pattern* can also be used if the language does not provide reflection mechanisms, but it will require more effort than the previous one.

Another example concerns components. According to the guidelines, each component instance is expected to push the following tags: *device_id*, *component_type*, *component_id*, and *execution_params*. The base implementation of a component is usually a loop to gather inputs, process something, and export outputs. Therefore, the best code point to insert all these properties into the stack is before the loop, thus reducing the implementation effort by inserting a framework as a layer between the middleware used for componentization and the application code. This framework must handle the component initialization by gathering those properties and inserting them in each thread created. Another way to achieve this result is by using the Aspect concept (Gradecki & Lesiecki, 2003) instead of developing a framework.

In order to avoid misunderstanding of the proposed approach, we must discuss why the use of Aspect-Oriented programming (Gradecki & Lesiecki, 2003) cannot be proposed as the only method of instrumenting the code (with the objective of reducing the effort in the task). Since Aspects are essentially abstractions to represent crosscutting concerns, they cannot be used to represent specific abstractions related to each piece of the code. The developer's knowledge is required to identify and expose these abstractions, thus the motivation to adopt manual instrumentation.

As a rule of thumb, the most important advice is to reason about how the guidelines can be easily implemented through the introspection and reflection features available in the technology of choice.

4.3 Event Transmission Issues

When an event is notified, its record is converted into a serialized form and is eventually sent to the central server repository. However, considering that target systems are distributed, and are assembled using a variety of devices and components, the sending process becomes vulnerable to several problems, such as: message loss, network availability, and bandwidth priority. The sending device must guarantee that the event was sent and stored before deleting it. This process must not compete with other requests made from that device, hence avoiding noticeable losses of quality due to interference in the system's normal behavior. Furthermore, low quality networks may require multiple retransmissions of the same event until it is correctly received. Also, mobile applications do not have constant network availability and tend to be more susceptible to power failures, since many devices may pass through regions without network signal and usually rely on battery power. These difficulties impose the need to keep unsent events in local persistent memory until their successful transmission is confirmed.

In order to solve these problems, the libraries used in mobile applications must implement a producer-consumer pattern to transmit events. Events must be saved into files, which then wait in a queue until successfully transmitted. When an event is notified, it is immediately appended to the current file, i.e. the last file in the queue. When this file reaches a given size limit, it is closed, tagged for shipment, and a new file is created and appended to the file queue. In case of restarting after, for instance, a crash, a power failure, or loss of connection, the application continues to write new events into the last file and attempts to transmit all the files that are already closed but remain in the queue. In case of a disaster — for example, when attempting to access invalid memory —, this approach assures that the centralized log set will contain information near the point where the failure occurred, helping to locate the faulty code.

Another problem concerns timestamp normalization among all devices in the system. Considering that each device has its own clock, which may differ from the central server's clock, it is necessary to normalize the timestamps of all events received by the central server so they are all congruent with the server's clock. Therefore, when starting the transmission of a package, the device's current timestamp is appended to this package. By using this timestamp, the server calculates the temporal delta between its own current clock value and the package's timestamp. The computed difference is then applied to all timestamps contained in the events of the received package, normalizing them to the server clock. This approach does not consider the transmission delay, which might produce inconsistencies. It is also vulnerable to clock updates occurring after recording events and before transmitting the corresponding package. For the time being, we accept that risk, which will be addressed in future work. Meanwhile, a rather easy way of overcoming this risk is by using a network time protocol (NTP) server for synchronizing all clocks in the system.

4.4 The Query Engine

In addition to the event handling described in this chapter, the implementation of tools based on runtime information flow needs the support of a mechanism to access the stored runtime information. This query mechanism must be capable of selecting only events that are relevant for the scenario under analysis, independent of whether the requisition came from a human or from an automated mechanism. This scenario must be described using properties present on the runtime information to create a perspective of interest, which restrictions can be transformed into filters to be applied over the execution flow in order to discard all undesirable events, leaving only those that are relevant for the current objective.

Therefore, the *Central Server Repository* must provide a primitive to query for a sequence of events. This primitive receives, as parameters, a list of *restrictions*, which are transformed into filters. The result of this primitive is a sequence of events, ordered by date, which matches all the restrictions. There are three types of restrictions: temporal limit, interesting property pattern, and undesirable property pattern. The temporal limits are expressed as a start and an end date. Every event that has its timestamp between those dates is considered a match. The interesting and undesirable property patterns are two lists of *tag restrictions* — a tag restriction is composed of a key, representing the property name, and a pattern, representing a set of values. The set of restrictions may be empty, in this case, instructing the engine to match every possible value. With this approach, it is possible to query for events that have only a specific tag, with any associated value, for events with values that match a given pattern (implemented using regular expressions), or for events that have a tag with an exact value. The final result of the interesting properties restriction is a composition of filters that matches an event if, and only if, it has a tag matching each property, and the values of these tags are matched to the corresponding pattern. The result of the undesirable properties restriction, on the other hand, is the opposite: a composition of filters that holds if the event does not match any of the *tag restrictions*. Therefore, when the query is evaluated, the search algorithm selects, by means of the restriction lists, all the events between the start and end date that have all the interesting tags and do not have any of the undesirable tags.

As a simple example of this query feature, consider the following log sequence stored into the database, comprising events from a routine being executed by two different execution threads (with less tags than usual and abbreviated names, avoiding visual pollution):

```
[env:mobile][dev_id:123][action:get_list][req_id:456] Requesting for items
[env:mobile][dev_id:123][action:get_list][req_id:456] Creating filters
[env:mobile][dev_id:789][action:get_list][req_id:321] Requesting for items
[env:mobile][dev_id:789][action:get_list][req_id:321] Creating filters
[env:mobile][dev_id:789][action:get_list][req_id:321] Generating the request obj
[env:mobile][dev_id:123][action:get_list][req_id:456] Generating the request obj
[env:mobile][dev_id:123][action:get_list][req_id:456] Executing the async request
[env:server][action:get_list][req_id:456][security] Authenticating
[env:mobile][dev_id:789][action:get_list][req_id:321] Executing the async request
[env:server][action:get_list][req_id:321][security] Authenticating
[env:server][action:get_list][req_id:456][security] Verifying permissions
```

```
[env:server][action:get_list][req_id:456][security] Checking params
[env:server][action:get_list][req_id:321][security] Verifying permissions
[env:server][action:get_list][req_id:321][security] Checking params
[env:server][action:get_list][req_id:321][user:ABC] Reading GET data
[env:server][action:get_list][req_id:456][user:ABC] Reading GET data
[env:server][action:get_list][req_id:456][user:ABC] Loading filters
[env:server][action:get_list][req_id:456][user:ABC] Reading related items
[env:server][action:get_list][req_id:321][user:ABC] Loading filters
[env:server][action:get_list][req_id:321][user:ABC] Reading related items
[env:server][action:get_list][req_id:321][user:ABC] Processing data for response
[env:server][action:get_list][req_id:456][user:ABC] Processing data for response
[env:server][action:get_list][req_id:456][analytics:56ms] Reading GET data
[env:mobile][dev_id:123][action:get_list][req_id:456] Unpacking response
[env:mobile][dev_id:123][action:get_list][req_id:456] Notifying the view
[env:server][action:get_list][req_id:321][analytics:56ms] Reading GET data
[env:mobile][dev_id:789][action:get_list][req_id:321] Unpacking response
[env:mobile][dev_id:789][action:get_list][req_id:321] Notifying the view
```

Observe that events from different routines — i.e. events from different devices executing a server request and events from the server processing these requests — are mixed together. Now, consider that, while diagnosing, a maintainer needs to inspect the request 456, avoiding non-functional information. He would generate a perspective of interest using as interesting property the [req_id:456], and as undesirable properties [security] [analytics]. When submited to the query engine, the result would be a single execution combining mobile and server events from the same request, discarding security and measurement events, as presented below:

```
[env:mobile][dev_id:123][action:get_list][req_id:456] Requesting for items
[env:mobile][dev_id:123][action:get_list][req_id:456] Creating filters
[env:mobile][dev_id:123][action:get_list][req_id:456] Generating the request obj
[env:mobile][dev_id:123][action:get_list][req_id:456] Executing the async request
[env:server][action:get_list][req_id:456][user:ABC] Reading GET data
[env:server][action:get_list][req_id:456][user:ABC] Loading filters
[env:server][action:get_list][req_id:456][user:ABC] Reading related items
[env:server][action:get_list][req_id:456][user:ABC] Processing data for response
[env:mobile][dev_id:123][action:get_list][req_id:456] Unpacking response
[env:mobile][dev_id:123][action:get_list][req_id:456] Notifying the view
```

Furthermore, the implementation of the query algorithm is trivial, since most of the complexity is solved by modern databases and provided as features for the application layer. During our evaluation, the query engine was implemented using a NoSQL database, which enables unstructured data manipulation. The specific implementation was MongoDB (2013), which provides some features that facilitate the implementation of these restriction filters. However, nothing prevents the solution from being implemented in an SQL database.

4.5 Conclusions and Technical Requirements

The solution presented here allows the instrumentation and the event storage to be designed without needing a fixed set of maintainer profiles, a problem discussed in the previous chapter. Furthermore, it is not necessary to specify the set of all possible properties of interest in the development onset, as new tags may be defined whenever they are needed. However, as already mentioned, tags are identified by their names and, thus, a document must be available allowing developers and maintainers to know all available tags and the instrumentation policy for a given system. If developers correctly use tag names, profiles can be defined at runtime according to the needs of each query perspective of interest.

Moreover, the solution can be used by most development teams, since there is no technical requirement outside the set of traditional methods and tools. The instrumentation libraries must be implemented accordingly to the target programming languages, relying on the traditional programming paradigms. It also does not impose architecture or design decisions and is used as usual log libraries. The application on the central repository may be implemented using any programming language that allows accessing a database. Furthermore, the effort to instrument the software code is small, as we will show in Chapter 7.

5
Lynx: Diagnosing with Contextual Information

This chapter presents a diagnosis technique developed using the query engine presented in Chapter 4. This is the first example of the benefits provided by a log containing contextual information. Chapter 6 presents a failure handling mechanism that takes advantage of this characteristic. This diagnosis technique is based on: (1) an inspection approach, presented in Section 5.1, which describes how maintainers may take advantage of the contextual information when investigating hypothesis; (2) a diagnosis process, presented in Section 5.2, which describes how the maintainer may use this approach for failure diagnosis; and (3) a tool named *Lynx*, presented in Section 5.3, which provides features to aid maintainers in the process of failure diagnosis.

The problem addressed by this technique has already been discussed in chapters 2 and 3, while formulating the thesis problem and describing the motivations for a log annotated with meta-information. In brief, while diagnosing a failure maintainers elaborate some hypothesis and attempt to investigate them in the execution log. However, common log techniques present limitations, such as mixed contexts, execution flow spread along files in different machines, and insufficient information on events. Thus, the large amount of irrelevant events displayed hampers efficiency, while the insufficient information associated with relevant events identified reduces efficacy. These problems were deeply discussed in Chapter 3, where we concluded that the appropriate solution must provide a mechanism to select events based on a perspective of interest, created and evolved on-the-fly by the maintainer who is diagnosing the failure. Hence, the assumption behind this novel technique is that if we are able to extract and display only the sequence of relevant events, both the efficacy and the efficiency will be improved.

5.1 The Inspection Approach

The proposed approach is based on the query engine described in the previous chapter, which provides a mechanism to select events based on a perspective of interest. This approach consists of investigating each formulated hypothesis using the query mechanism, by filtering events based on properties identified in the failure occurrence description. With this mechanism, each hypothesis can be transformed into a perspective of interest, which is transcribed as a set of restrictions used to retrieve a sequence of rich events that may be capable of aiding the maintainer to confirm the hypothesis. For example, a hypothesis may be "the port permission was not set", which will stimulate the maintainer to inspect the footprint looking for events related to port permission, transcribed as a restriction that requires the event to have the tag [*action : set_port_permission*]. In addition, the perspective of interest should be refined during the inspection, by the evolution of the restriction set based on what is learned from the footprint under analysis. For example, an inspection starts looking for errors occurred while a specific user was working with the system. This perspective of interest may be represented with restrictions that require the tags [*error*] [*user_id : john doe*]. Then, while studying the result, the maintainer identifies that all errors occur in the same action, hence the perspective of interest must be refined to consider this property, in this case also removing the tag error, in order to reach all footprints of this specific action when triggered by the given initial user.

Moreover, some hypothesis may be broken into a tree, with some possibilities that must be verified. The maintainer may use a depth-first search algorithm to walk through this tree, inspecting each path and discarding those with leafs that do not verify the hypothesis. It is also necessary to consider that the tree may grow during the verification, since retrieved information may lead to other paths. A simple example of a tree scenario is: a maintainer following a hypothesis discovers that the erroneous data were retrieved from the database. Its is known that it was written that way into the database, however when there are more than one piece of code executing insert-operations on this type of record, new branches are created in the hypothesis tree, one for each writing call, and all must be verified.

On yet a last different hypothesis, consider a system composed of mobile devices communicating with a server in the cloud. Each event created by this system has a tag name that represents the device's origin, and a tag action, which represents the current operation the device was performing. Suppose that, among other operations, each mobile device triggers data sync actions on the server. We know that when a specific device triggers this operation, the server fails to process. We do not know, however, which device is causing the failure. Inspecting in the traditional way, by collecting manually the logs of each device, would take a great effort to correlate them in a single timeline and filter the events, leaving only those related to the failed sync action. Using the approach suggested here, a system maintainer would only need to specify his perspective of interest — in this case, informing the tags *[action:sync][error]* — and, after finding an event that represents the failure, use the name of the source device to refine the search, i.e. a new query using the tags *[action:sync][name:tablet_1]*, for example. Continuing this way, the maintainer may come to a clear view of the footprint that displays only events related to the failed execution.

Observe that the examples above are hard to diagnose using the traditional log. Therefore, the approach presented in this work may increase inspection efficiency by enabling maintainers to restrict the number of events they need to study by using a mechanism to select only these related to the hypothesis under analysis. This approach, however, is limited by the events and properties on these events, gathered by the instrumentation mechanism. If a property that is relevant for the hypothesis investigation is not present on the event set, the diagnosis may be compromised or require more effort than strictly necessary. Thus, the success in this diagnosis approach relies on the system's instrumentation policy.

5.2 Failure Detection and The Diagnosis Process

The approach presented in the last section has been devised for use in the following scenario: a failure is observed, and a maintainer needs to produce its diagnostic in order to remove the fault and, if necessary, implement a detection and recovery mechanism to handle future occurrences while the fault is being removed. The failure observation may occur by human detection while using the system or by an automated solution that identifies unexpected behaviors (assertions, exception notifications, etc.). Whichever the observation mode, a failure report must always be filled up describing all possible characteristics of the failed scenario, what would vary depending on the project domain. In a system composed of mobile devices and a web service, for example, the report is expected to register the username that triggered the failure, the type of his device, the feature that failed, the input if applicable, the observed error, etc; in an embedded supervision system, on the other hand, the characteristics would be the state of all logical and analog ports in the microcontroller; thus the full memory content, which in this case is predictably small.

When the occurrence of a failure is automatically detected, the system may also automate the report generation. However, when observed by a human, someone in the production team must be responsible for creating this report. The automated option is preferable, since it does not depend on the final user's goodwill to inform the occurrence. The automatic detection may be enhanced by creating a software agent to monitor the event flow in order to find events with error tags. When an event with this tag is identified, an alarm must be activated, generating human notifications with an appropriate implementation based on the application domain. In a web service, for example, it can be implemented as a simple e-mail, while in an embedded system, isolated from the network, the notification may be an SMS message. This approach must be supported by the instrumentation policy, which must determine rules for notifying every error detected during execution. Furthermore, the moment the software detects a failure occurrence, it is of utmost importance to gather as much information as possible about the current execution state, in order to aid future diagnosis, even if this requires inserting larger values into tags, such as the *stack trace* of the execution or database records.

The proposed diagnosis process starts analyzing the failure report, which provides the observed error and the context environment where the failure occurred. After that, the maintainer must execute the following steps:

1. Generate a set of hypothesis that may explain the observed error.
2. Generate a base perspective of interest, using properties identified in the contextual environment described in the report.
3. Order the set of hypothesis in a list, using as criterion the expected holding possibility of each one.
4. For each hypothesis,
    a. Reason about which contextual properties related to the hypothesis may be transformed into restrictions.
    b. Append the restrictions created in the last step to the perspective of interest.

c. Analyze the resultant footprint.
    i. If the root cause of the failure was found, stop and conclude the process.
    ii. If the root cause of the failure was not found, then evaluate:
        1. If the hypothesis was proven wrong, walk in the tree to the next hypothesis to be tested, then go to step 4.
        2. If the hypothesis remains open, or elaborate nested hypothesis, walk to the first one and go to step 4.

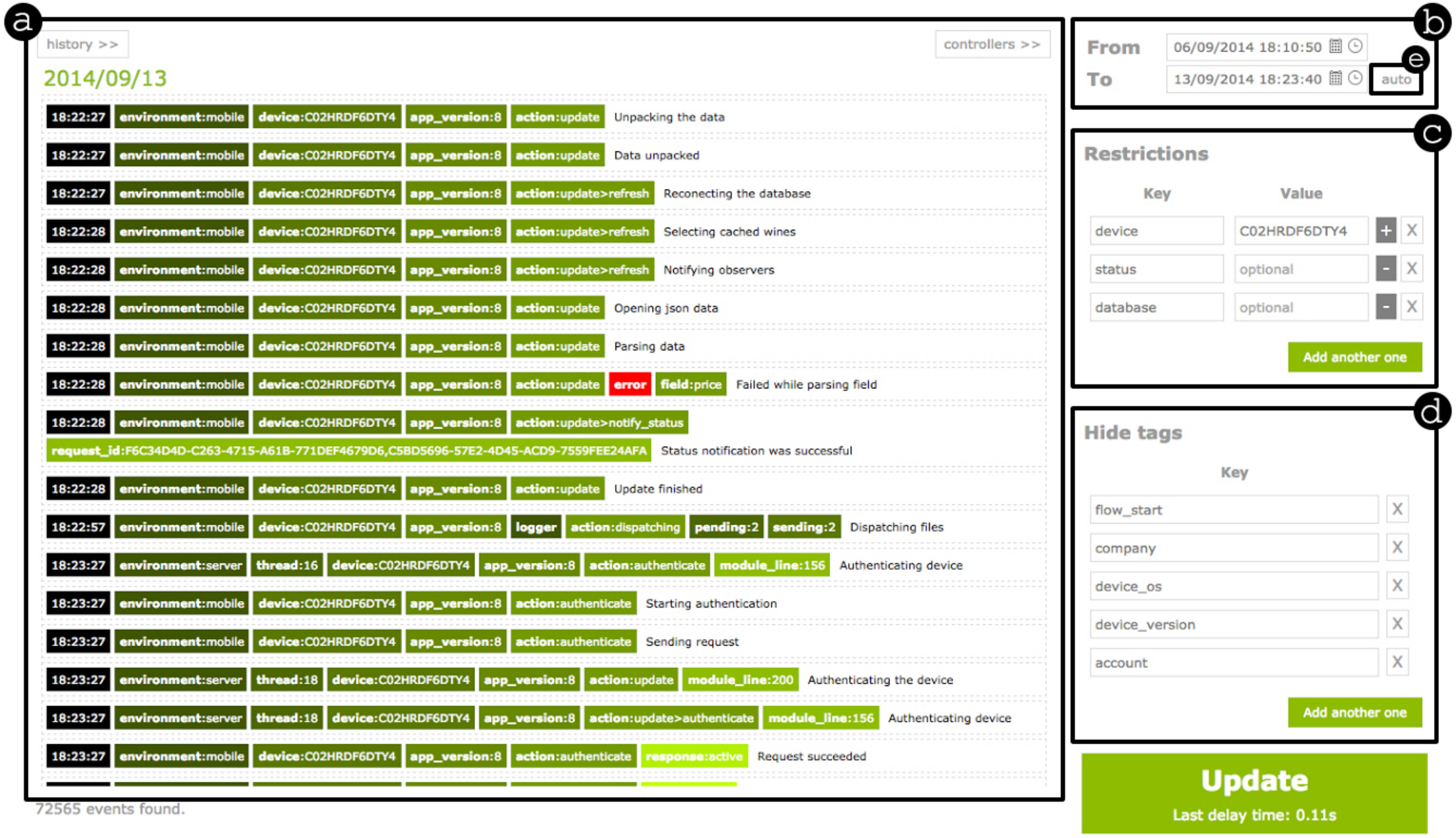
Observe that, for each perspective of interest, the maintainer will study the resultant footprint and refine the perspective of interest, in order to query a perspective that presents the exact footprint that contains the explanation of the unexpected behavior, i.e. the root-cause of the failure. In addition, the footprint that represents the diagnostic may provide events and properties' relations that may be used to develop a failure handler for detecting and recovering future occurrences of the recently diagnosed failure. This approach will be explained in Chapter 6.

5.3 The Inspection Tool

An inspection tool is needed to support the process described in the last section. This tool must provide mechanisms for the maintainer to describe the perspective of interest as a set of restrictions, which will be evaluated by the query engine in order to select the set of events that are relevant for the hypothesis under analysis. The result must be displayed in an appropriate format, exhibiting the sequence of events in chronologic order, and the contextual properties of each one. Also, the tool must previse that the perspective of interest will change during the inspection, and update the sequence of events accordingly.

With the purpose of assessing the inspection technique, this tool was implemented as a web application. An example of its interface can be seen in Figure 2, which shows fields that define the perspective of interest and the extracted event list corresponding to this perspective (Figure 2a). These input fields are: (1) fields for temporal restrictions: start and end dates (Figure 2b); and

(2) fields for tags restrictions (Figures 2c and 2d). The maintainer can represent tag restrictions using two lists: the first containing tags that must be present, and the second containing tags that must not be present in the events. Restrictions can be specified either using only the tag name or a regular expression.



2014/09/13

```
18:22:27 environment:mobile device:C02HRDF6DTY4 app_version:8 action:update Unpacking the data
18:22:27 environment:mobile device:C02HRDF6DTY4 app_version:8 action:update Data unpacked
18:22:27 environment:mobile device:C02HRDF6DTY4 app_version:8 action:update>refresh Reconecting the database
18:22:28 environment:mobile device:C02HRDF6DTY4 app_version:8 action:update>refresh Selecting cached wines
18:22:28 environment:mobile device:C02HRDF6DTY4 app_version:8 action:update>refresh Notifying observers
18:22:28 environment:mobile device:C02HRDF6DTY4 app_version:8 action:update Opening json data
18:22:28 environment:mobile device:C02HRDF6DTY4 app_version:8 action:update Parsing data
18:22:28 environment:mobile device:C02HRDF6DTY4 app_version:8 action:update error field:price Failed while parsing field
18:22:28 environment:mobile device:C02HRDF6DTY4 app_version:8 action:update>notify_status
request_id:F6C34D4D-C263-4715-A61B-771DEF4679D6,C5BD5696-57E2-4D45-ACD9-7559FEE24AFA Status notification was successful
18:22:28 environment:mobile device:C02HRDF6DTY4 app_version:8 action:update Update finished
18:22:57 environment:mobile device:C02HRDF6DTY4 app_version:8 logger action:dispatching pending:2 sending:2 Dispatching files
18:23:27 environment:server thread:16 device:C02HRDF6DTY4 app_version:8 action:authenticate module_line:156 Authenticating device
18:23:27 environment:mobile device:C02HRDF6DTY4 app_version:8 action:authenticate Starting authentication
18:23:27 environment:mobile device:C02HRDF6DTY4 app_version:8 action:authenticate Sending request
18:23:27 environment:server thread:18 device:C02HRDF6DTY4 app_version:8 action:update module_line:200 Authenticating the device
18:23:27 environment:server thread:18 device:C02HRDF6DTY4 app_version:8 action:update>authenticate module_line:156 Authenticating device
18:23:27 environment:mobile device:C02HRDF6DTY4 app_version:8 action:authenticate response:active Request succeeded
```

72565 events found.

From 06/09/2014 18:10:50
To 13/09/2014 18:23:40 auto

Restrictions

Key	Value
device	C02HRDF6DTY4
status	optional
database	optional

Hide tags

Key
flow_start
company
device_os
device_version
account

Last delay time: 0.11s

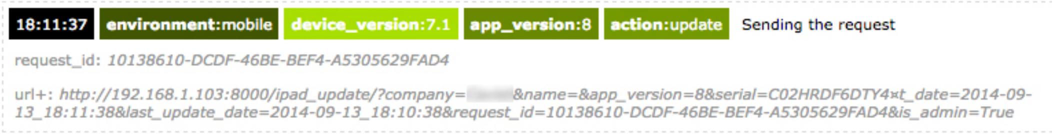
Figure 2 – Inspection tool interface.

When an event is displayed and the set of tags is extensive, the entire representation may be confusing. In order to reduce this visual pollution, two solutions are provided. The first one is based on controllers to hide undesired tags: a meaningful event must contain all tags, but only a few of them should be displayed, depending on the ongoing investigation. For this reason, this tool implements a feature that allows selecting only those tags that should be shown (Figure 2e). Consider the following example, which shows an event that is interesting both to evaluate the application performance (tags *cpu* and *memory*) and to inspect screen flow:

```
[environment:mobile] [application:hello world] [cpu:80]
[memory:2524] [version:3] [flow:main] [message:window loaded]
```

An inspection that does not need to evaluate device resources may hide cpu and memory tags, in order keep only relevant tags visible.

The second solution is a mechanism to automatically generate a collapsed area for each event, where tags with large values are displayed, as presented in Figure 3. This solution is appropriate for tags such as *stack traces*, *urls*, *records*, etc.



```
18:11:37 environment:mobile device_version:7.1 app_version:8 action:update Sending the request
request_id: 10138610-DCDF-46BE-BEF4-A5305629FAD4
url+: http://192.168.1.103:8000/ipad_update/?company=[redacted]&name=&app_version=8&serial=C02HRDF6DTY4¤t_date=2014-09-
13_18:11:38&last_update_date=2014-09-13_18:10:38&request_id=10138610-DCDF-46BE-BEF4-A5305629FAD4&is_admin=True
```

Figure 3 – Example of an event with collapsed area.

Finally, the inspection tool also provides a history of restrictions' sets used in previous queries, with the goal of supporting a fast walk into the hypothesis tree: when the maintainer needs to change the course of his/her investigation, he or she may use the history to quickly change the perspective of interest.

5.4 Closure

With this inspection technique, we allow an operator to study the behavior of the system by filtering the events to be displayed according to the perspective of interest. This, in turn, is formed by a set of restrictions created from tags related to the hypothesis under investigation. The solution can be applied to diagnose failures in local as well as in distributed software, addressing both scenarios where code can and cannot be changed to add more information about the execution. This technique is based on manual analysis in order to take advantage of maintainer's knowledge about the system's history of failures; and software design and architecture. We believe that maintainers with adequate tools are often more efficient when diagnosing than fully automated techniques for two main reasons: (1) humans can reason in a level of complexity that software still cannot; (2) and humans also hold tacit knowledge that has not been transferred to the source code during development, hence limiting automated reasoning about the software. Developers and maintainers hold knowledge such as system's high-level abstractions, architecture anomalies, issues that arose during development, and history of errors — among others —, which are useful to elaborate effective hypotheses that lead to the determination of the root cause of the failure.

Moreover, semi-automated techniques may reduce the effort in diagnosing without removing the benefits of human reasoning, and can also be implemented over this type of log. There are two future works prevised in this regard:

- Extracting the system's state machine from a structured log, in order to attempt to detect anomalous behavior (related work discussed in Chapter 8). By using annotated logs, the efficacy should be higher than solutions based on traditional log.
- Investigating how the maintainer's experience with the tool can be used to learn how types of failures are individually diagnosed, then using this knowledge as tips for future diagnosis.

Furthermore, the objective of the technique introduced here is to reduce the diagnosing effort, by providing mechanisms to investigate hypotheses in a more efficient and effective way. The efficacy problem is addressed by the instrumentation technique, through the contextual information annotated on events, which improves the maintainer's view during an investigation. This approach alone makes the resulting log sequence capable of unifying events from different devices in a single view and informs for each one a range of properties that goes from language to application abstractions — such as the source-line number, the modules involved, the high-level action, the user that triggered it, request parameters, etc. Moreover, this meta-information also addresses the efficiency problem, since they turn the event descriptor into a comparable structure. This enables the mechanism to filter events based on a set of restrictions, which in turn uses properties' relations to determine if an event must be considered relevant or not for the perspective under analysis. Therefore, the technique provides means to analyze the hypotheses under investigation selecting only those events that are directly related to the failure occurrence, which is usually formed by a small set of events compared to the full log. As previously said, however, both the efficacy and the efficiency rely on the log content, which should provide the necessary information for the diagnosis. Thus, the instrumentation policy is of paramount importance, since it will guide developers while coding, making them leave the necessary information for future, unexpected, diagnosis sessions.

6
Hydra: A Tag-Based Self-Healing Mechanism

This chapter presents a mechanism for supporting failure detection and recovery implementation for systems in production. This is the second example of the applicability of the logging technique presented in Chapter 4. The problem addressed by this mechanism is the development of failure handlers, which are modules that will aim at detecting failure occurrences and, when possible, recovering the system to a valid state. Observe that some handlers will be capable of prevent the failure consequences, whilst others will at least minimize them.

During our research, we have encountered two main scenarios where this type of mechanism is needed. The first one aims at future occurrences of a recently discovered failure that requires a fair amount of time to remove the corresponding fault; or is located in a component that exhibits some deployment issues, thus requiring a solution without redeploying the entire system. The solution addresses this problem by deploying the specific detection and recovery routines for the corresponding failure, without modifying components and redeploying them. An example of this scenario would be a recently discovered fault in a web-service protocol that would take a week to be removed due to team unavailability, thus during this period a recovery handler would avoid failure occurrences by identifying the corresponding failure signature and proceeding with the recovery.

The second scenario is the development of failure handlers for faults that cannot be removed from the system, thus will remain during its lifetime. Since handlers for these failures will always be present in the software, the detection and recovery concerns should be decoupled from component implementation, thus avoiding the *ad-hoc* design that so often degrades the code by mixing the failure handling instructions with the functional instructions. Examples of this scenario are the usage of a faulty third-party library or even the possibility of hardware malfunction, which cannot be avoided.

Observe that the requirements of this second scenario also benefit the first scenario, since the developer can define and implement the failure handler without adding complexity to the functional code, which also reduces the effort when later removing the failure handler from the system. Observe that after the fault is removed, the handler may no longer be needed and can be uninstalled. By following this approach there is no need to revert *ad-hoc* instructions along the functional code, since they were never inserted. Therefore, the main requirements for this solution are:

- Detect failures signatures without explicitly writing code in the functional implementation.
- Avoid, whenever possible, modifications in the functional implementation, what depends, however, on the failure being handled.
- Deploy the failure handler without redeploying the system, and keeping it loosely coupled for easy removal.

The rest of this chapter presents the solution overview, discusses how the event flow can be used to detect failure occurrences and extracts the necessary information to recover the system; and describes how this mechanism can be implemented in a software system.

6.1 Solution Overview

The logging technique presented in Chapter 3 can help extracting a better representation of the failure signature due to the extra information about the event flow, which improves the developers' (or maintainers') efficacy when designing the failure detection handler that will attempt to identify future occurrences of the correspondent failure. Observe that some known failure signatures are difficult to transcribe into a verification routine due to accessibility issues, such as modularization limitations or even the desired set of properties belonging to different processes. The first issue is usually solved by *ad-hoc* approaches when violating the encapsulation, while the second requires more effort to generate remote calls. When using events with contextualization properties, however, there is no need to break the encapsulation, since the information required is in the tag stack or in the event flow. It is an information source.

The failure handling mechanism presented here was inspired by the *Autonomic Computing* concept (Murch, 2004). Figure 4 presents the solution overview of our approach, describing the process of detecting and handling failures. The event flow and the tag-stack content are the input of the process, which is handled by the *Event Monitoring* process in order to unify the event data representation and provide mechanisms to inspect past information. The *Failure Detection* process attempts to detect known-failures in the event flow by using recent events and individual tags, explained in depth in the next sections. The detection step may also need to gather extra information about the system's state through sensors (when available). When an occurrence is found, the *Failure Handling* process executes the failure's corresponding recovery routine, in order to handle its consequences. This task is done by modifying the system's state through actuators and by changing environmental settings.

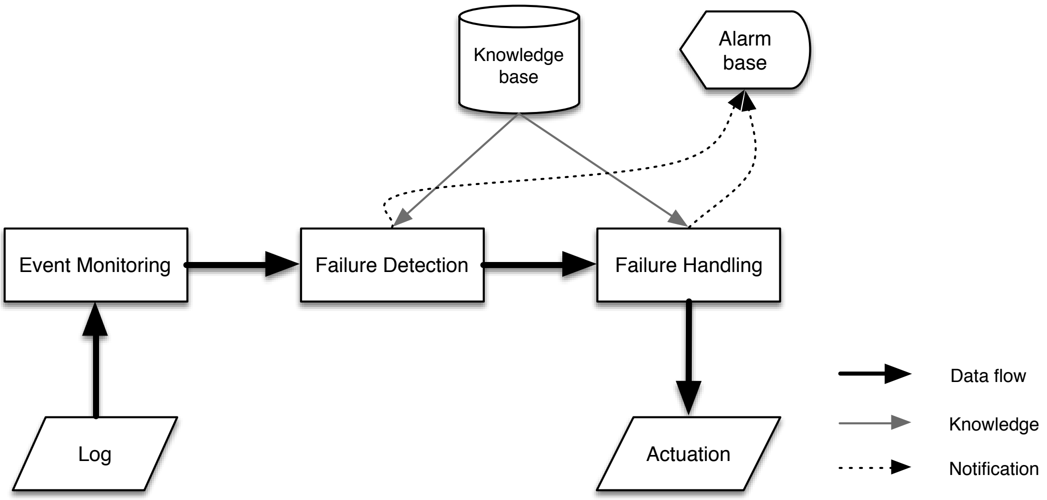


Figure 4 – Solution overview.

It is important to observe that the specific detection and recovery knowledge is provided by humans, based on their previous experience with the system. This knowledge is transcribed into *Failure Handlers*, which are software modules stored in the *Knowledge Base*, which evolves constantly during the system's lifetime by the addition or removal of failure handlers in response to failure diagnosis and fault removals. For example, the first system deployed by an organization would, therefore, start with an empty knowledge base, so when a first failure occurs, it will be diagnosed by looking for a signature that may be used to detect future occurrences. Based on the diagnostic and the knowledge about the system's logic, the developer (or maintainer) will develop a software module that complies with one of the *Failure Handler's* interfaces, presented hereafter. This *Failure Handler* will provide a mechanism for detecting the failure signature and another for handling the occurrence. This handler will be installed in the production environment as the first *knowledge-item*. For each new failure, the process will be the same, and then the base will be populated. When a corresponding fault is removed, the failure descriptor may be removed with it. This decision is made considering failure characteristics: some handlers may be helpful to keep, while others may lead to unnecessary perturbations. Moreover, when the organization develops the next system, some handlers may be availed, if generic enough, to protect the deployed instance from common known-failures. This is also the case for those failures originated from faults that cannot be removed, and which receive a failure handler already during development.

When the mechanism detects and handles a failure while executing, events are generated to feed an alarm base, which informs properties about the failure being handled. This alarm base is a component that keeps updated information about the failure occurrences and can be used to develop tools for human operators, keeping them aware of the unexpected executions in the system and, thus, avoiding bad decisions. For example, in a robot control system, an alarm informing that the position information is compromised may lead the operator to adopt a more cautious attitude.

6.2 Detecting Failure Signatures Through the Execution Flow

The technique of enriching logs with contextual properties enables sophisticated and precise analyses about the system's execution. The proposed failure handling mechanism takes advantage of these analyses' benefits, by attempting to identify failure occurrences in the execution flow. This can be done (1) through the meta-information contained on events, or (2) through the tag-stack state in some checkpoints along the execution. As mentioned in the last chapter, the diagnosis of a known failure aims at identifying the signature of this failure, and this signature must contain sufficient data to allow a specific handler to identify the occurrence and properly recover from the failure.

Therefore, there are two variants in the failure handling mechanism of our solution, which differ in the detection approach — both approaches will be explained in this section, while more details about the mechanism will be given in the next one. The first approach is asynchronous, since the mechanism analyses the event flow looking for the system's history, thus with a small delay between the failure occurrence and the moment it is detected. This method is implemented though an external agent that monitors the system execution, enabling it to write detectors that relate events between different component instances and do time relations. However, this approach presents some limitations for the recovery implementation, since it is executed outside the failed process's memory space, hence imposing the need for actuators, which are procedures implemented in the target component.

The second approach is synchronous, executed at checkpoints along the execution. It aims to identify a failure signature through the current state of the tag stack. Observe that some configuration of the values in the tag stack may represent a failure signature (however, sometimes it is necessary to look in the event history in order to complement the verification input). Since this approach is synchronous, the execution only continues after the verification has been completed, guaranteeing that any recovery applied will be effective from that point on. For example, consider a system composed of different types of devices interacting with a web service; and when a specific type of device (*d314,* for instance) makes a request named *process_some_data* for this web service, a failure occurs, resulting in a *bad request* response. When the failure is diagnosed, the maintainer identifies that this type of device sends one of the request parameters using an unexpected data format, which produces an error in the internal function *data_to_object*. Therefore, a mechanism can be created to detect the occurrence of this exact context by monitoring the tag-stack for the presence of tags [device: *d314*] [action: *process_some_data*, *data_to_object*]. Hence, if a recovery routine is available (for example, one which converts the data to the correct format) it can be called at this point of execution to avoid the cause of the failure. However, depending on the state of the system, it will only be possible to avoid or minimize the failure consequences. Observe that this failure handler could not be implemented using the previous approach, since it must be executed during the system's execution flow. However, there are two drawbacks in the synchronous approach:

1. Since the verification is done over the tag-stack state, it is limited to the information contained in the stack. An extra-verification step can be made using the query engine to inspect the event flow, but the impact on performance will be increased from retrieving extra information from the database.
2. If the failure cause has already been triggered and it leads the process to be closed (*segmentation fault*), the handler may not be executed.

The implementation of both approaches will be explained in Section 6.4.

6.3 Solution Architecture

Similar to the *Autonomic Computing* concept, this solution presents an autonomic global cycle (Figure 5), executed by an independent component in the system's environment.

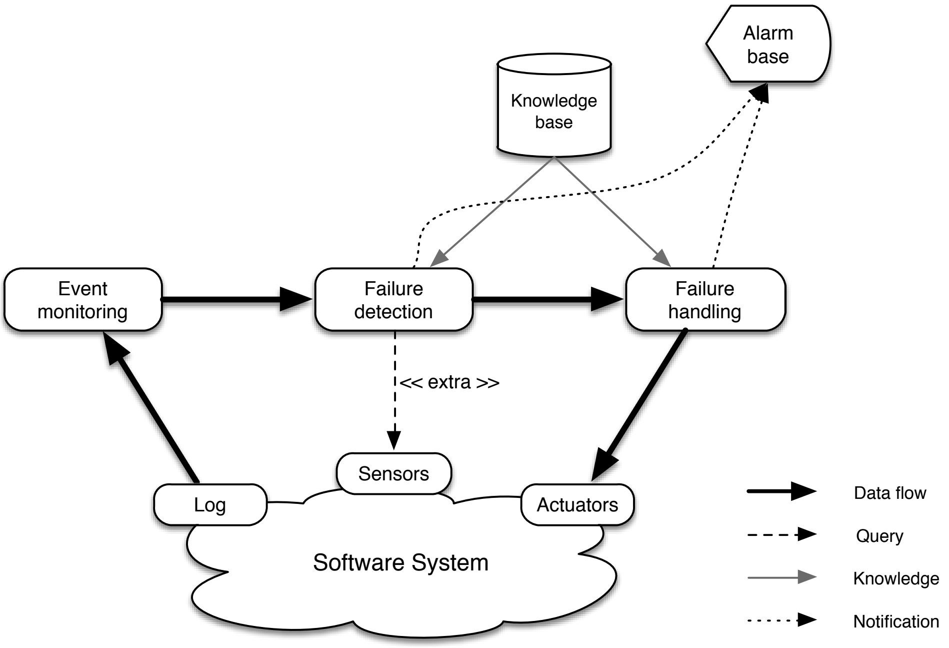


Figure 5 – Autonomic global cycle

This cycle monitors the system's behavior, attempting to detect failure signatures through the event flow, as explained in the first approach of the previous section. This approach is asynchronous and enables the detection of failure signatures based on temporal relations. For example, consider a failure with a signature described by the absence of the last action event in a given request type. In this case, the detection handler may attempt to identify this situation by looking for a sequence of events that represents the request execution — events A, B, and C, with the last event, D, missing from the log — and by using an error threshold in seconds to consider this scenario a failure occurrence. When a failure occurrence is detected, the corresponding recovery routine must be executed, optionally with contextual properties as parameters, which may specialize the procedure. However, a failure recovery that needs immediate action in the execution context (before continuing the execution) will not be supported by this approach, as discussed before. Beyond this asynchronous limitation, the recovery routine is also executed outside the *component-under-failure* space. In other words, it is executed in another process, possibly in another machine, thus requiring actuators implemented in the target-component to modify its current state when needed, in order to handle the failure consequences.

There is also an autonomic local cycle (Figure 6), which is slightly different from the original autonomic concept: the mechanism’s cycle in our solution is triggered by instrumentation code, instead of being executed in a parallel routine.

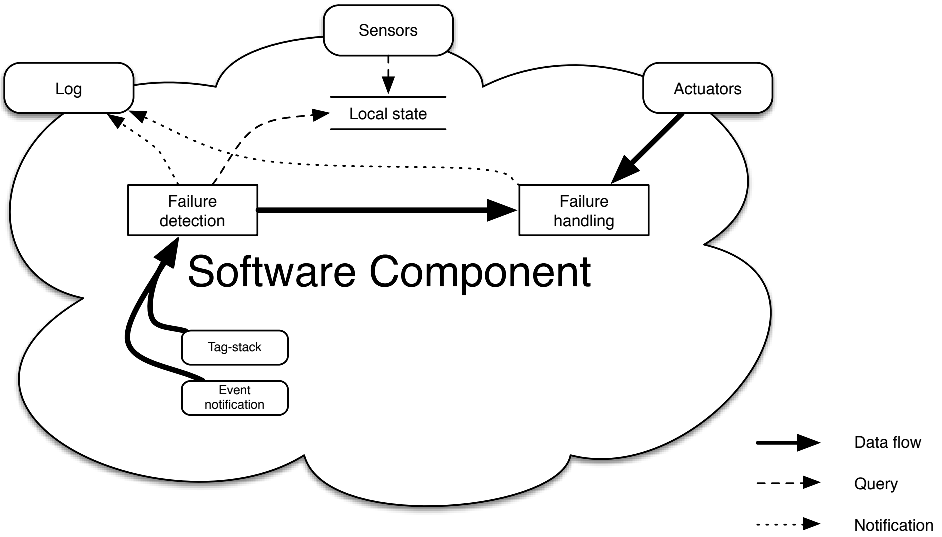


Figure 6 – Autonomic local cycle

The goal of this approach is to fill the gaps left by the shortcomings of the global cycle, which are: the possibility to handle failures synchronously and to have direct access to the component’s state. From the failures we could observe in the set of systems used in our evaluation (Chapter 7), we managed to identify two distinct recovery groups: those needing to modify the process or the environment state, and those needing to modify the parameters passing through method calls. Therefore, we have developed two mechanisms to detect and handle failures synchronously for each of these groups — explained in the next section. The underlying concept shared by these mechanisms is having the tag-stack as a source of information that can be used to identify a failure signature — or at least part of it —, consequently putting the execution in a "state of alarm" while running a vulnerable scope. While in this state, the corresponding mechanism is allowed to make high-cost verifications, which may have an impact on system performance. Hence, the smaller the scope, the lower will be the overhead for the system.

Therefore, this approach is appropriate for recovery routines that need to modify the component's state or the data flow in function calls immediately after the failure detection and before the execution continues. The trigger of the local cycle is embedded in the instrumentation contained in the code, being executed from the following operations: push tag, pop tag, notify event, start action, and end action (these last two are coupled in the *action_tag* interceptor, described in Chapter 4). The implementation design of both cycles will be described in the next section.

6.4 Implementing Failure Handlers

Each failure handler must be developed for a specific failure, designed in a way that better detects and handles its occurrences. The handler must also provide two main primitives: one for evaluating the software state, seeking for its failure signature, and another for handling the failure occurrence, when an instance is detected. The following subsections will discuss how to implement the handler as a failure descriptor, which must be supported by a framework in order to reduce the effort employed in the handling implementation task. Each subsection will also discuss for which types of failure a given handler is the most appropriate, since the first decision when developing the handler is whether the descriptor must be implemented for the local or the global cycle.

6.4.1 A Framework for the Global Autonomic Cycle

The global autonomic cycle must be implemented in an independent component, in order to avoid compromising its execution when a failure corrupts the system's environment. The most basic example is when a *segmentation fault* occurs in a software component and consequently ends its process – in this case, the global cycle is not affected and may proceed with a re-execution type of recovery in the failed component.

In order to support the failure descriptor implementation, a framework is proposed and presented in Figure 7. *Hot spots* are the classes shown in red. The *Autonomic Manager* is a singleton class, responsible for the entire cycle, created when the system starts. During the initialization, it looks for available *Failure Descriptors*, which are the implementation of the failure handler, and loads all those it finds through a reflection mechanism (Smith, 1982). Observe that this solution enables the installation of handlers without having to change or redeploy the system.

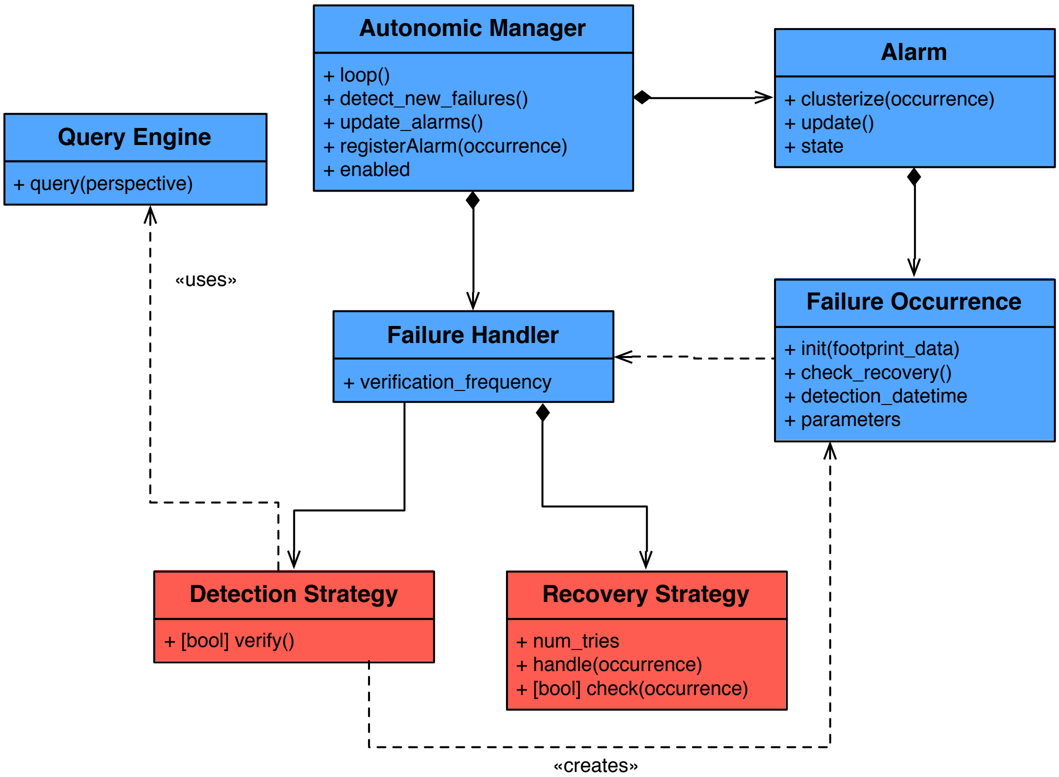


Figure 7 –Framework for global cycle implementation.

After the system is started, the *Autonomic Manager* initiates its verification cycle. During each loop of this cycle, the *Detection Strategy* of each *Failure Handler* is used to detect failure occurrences, which are represented as *Failure Occurrence* instances. A *Failure Occurrence* holds the information about that specific occurrence until it is recovered. Each *Failure Handler* must also provide the verification frequency, which must be configured in a way that avoids impacting the performance of the system. During our evaluation, this frequency was empirically defined based on the characteristics of each failure, resulting in a low overhead as presented in Chapter 7.

Once a *Failure Occurrence* is created the *Autonomic Manager* attempts to cluster it in a previous *Alarm*. This clusterization routine consists of verifying if the failure occurrence corresponds to a previously detected occurrence, for which a recovery routine is in progress. The objective of this approach is to avoid multiple alarms due to the same failure occurrence. The *clusterization* method considers a re-occurrence when the *footprint data* from both instances are equal. The data scheme is free, and can be composed by events or tags extracted from events in the footprint (that contains the failure signature). This data is gathered in the detection phase, and passed on in order to contribute to the recovery mechanism by specifying the occurrence context. When there is no preexisting identical alarm, a new instance is created.

When a failure is being handled, the *Autonomic Manager* applies each *Recovery Strategy* associated to the *Failure Handler* (in the same order they were registered on the handler's list). The handling operation of each strategy is composed of (1) the invocation of the *handle* method to execute the recovery routine, (2) and the continuous verification for the result through the *check* method, indicating if the recovery was successful, if it wasn't, or if it is still being applied. A successful result proceeds with the next step of the failure handling, by executing the next *Recovery Strategy* operation; an unsuccessful result re-executes the handling operation based on the number of tries configured for the strategy, or terminates the operation if the maximum number of attempts is reached; and the third option, still being applied, just wait until the next *check* verification. The base implementation of the *check* method is to call the *verify* method of the corresponding *Detection Strategy*, but it may be evolved when implementing a specific handler. The *Alarm* is initialized in the *open state*, and when the *check* execution of the last *Recovery Strategy* confirms success, the alarm transitions to *resolved state*. Similarly, when a check fails the *Alarm* is set to *unresolved state*.

Finally, for each failure descriptor the developer (or maintainer) must only implement a concrete class for the interfaces' *Detection Strategy* and *Recovery*

Strategy. The first one may use the query engine presented in Chapter 4 to search for the failure signature in the event flow. A failure may also be detected in the local cycle and handled by a *Failure Handler* in the global cycle. This approach can be implemented by a *Detection Strategy* that looks for an event (emitted by the component) with the tag *failure,* as well as other tags representing the *signature data*.

As a simple example, consider a failure handler that works as a watchdog for a given component named *MotorManager*. The detection consists on verifying the component's availability, and the handling on killing the zombie process and restarting the component's instance. The detection and handling strategies may be implemented as the following pseudo-code:

```
class WatchDogDetectionStrategy
    : extends DetectionStrategy
{
    bool verify () {
        // Every component must notify a keep-alive
        // tag at every 30 seconds.
          result = query_engine.lookup(
              '[component:MotorManager][keep-alive]',
              date.now() - 30,
              date.now()
        );
        if (result.count() == 0) {
              result = query_engine.lookup(
                  '[component:MotorManager][PID]'
              );

              // Get the last PID registered and pass it
              // as parameter
              parameters = List(result.last_event['PID']);

              // Create the failure occurrence
              AutonomicManager.registerOccurrence(
                    new FailureOccurrence(parameters);
              );

              return true;
        }
        return false;
    }
}

class WatchDogHandlingStrategy
    : extends HandlingStrategy
{
    void handle(FailureOccurrence occurrence) {
        // Kill the last process
        pid = occurrence.parameters['PID'];
        System.execute('kill -9 %s', pid);

        // Re-launch the component instance
        SomeMiddleware.launch(
            'com.projectX.MotorManager',  // Instance type
            'Motors' // Instance name
        );
    }

    bool check(FailureOccurrence occurrence) {
        result = query_engine.lookup(
             '[component:MotorManager][keep-alive]',
             date.now() - 30,
             date.now()
        );
        return result.count() > 0;
    }
}
```

This is just a simplification of the watchdog implementation. In a real system, the implementation would be generic enough to be reused for different component types and instances, which is one of the contributions of the proposed mechanism. The complete example will be presented in the evaluation (Chapter 7).

6.4.2 The Local Autonomic Cycle as an Instrumentation Library Extension

The local autonomic cycle must be implemented as an addition to the Lynx library in order to take advantage of triggers already available in the component's process space. With this approach, it is possible to develop handlers for failures that require a recovery action before continuing the execution. An example can be a recovery routine that avoids a failure by modifying the parameter values of a method call to a correct format. Observe that this failure handler must be synchronous, in order to change the passing data in the execution flow.

The instrumentation library extension is composed of an evolution in the Logger module and of a framework for the local cycle. The evolution in the Logger module consists of providing listeners for the following operations: *push tag*, *pop tag*, *notify*, *start action*, and *end action*. The first three are called on its respective primitives, and the last two in the *action_tag* interceptor (described in Chapter 4), respectively before and after the function call. Each listener must provide a mechanism to register observer modules, which in this case is the local cycle framework presented in the Figure 8, with *hot-spot* classes shown in red.

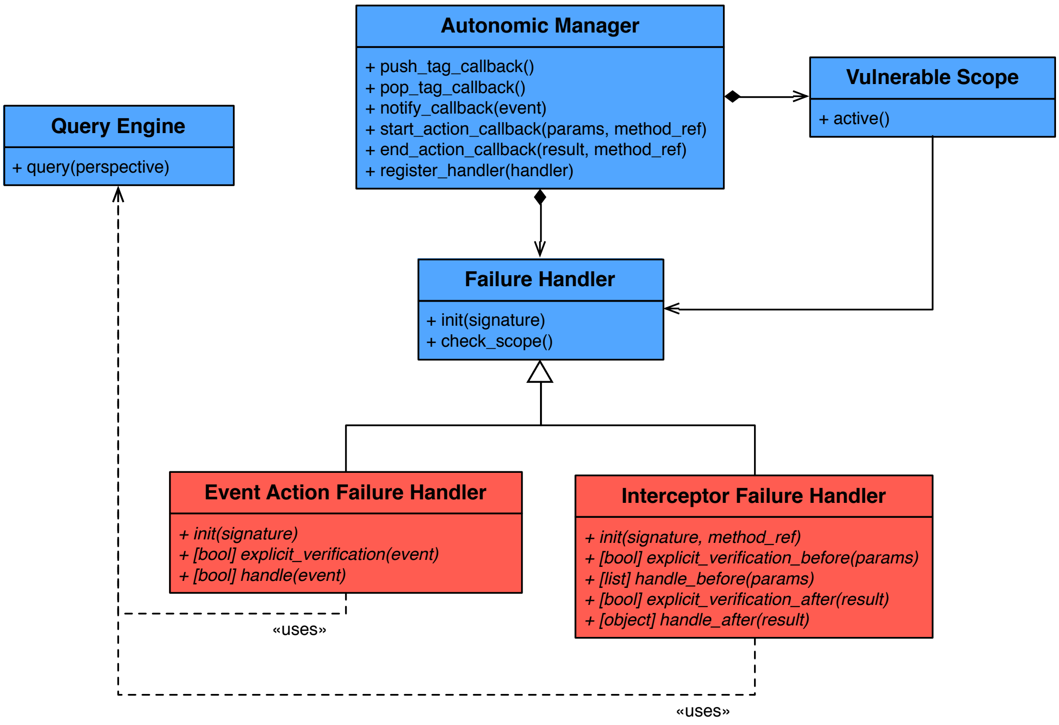

Autonomic Manager
+ push_tag_callback()
+ pop_tag_callback()
+ notify_callback(event)
+ start_action_callback(params, method_ref)
+ end_action_callback(result, method_ref)
+ register_handler(handler)
Vulnerable Scope
+ active()
Query Engine
+ query(perspective)
Failure Handler
+ init(signature)
+ check_scope()
Event Action Failure Handler
+ init(signature)
+ [bool] explicit_verification(event)
+ [bool] handle(event)
Interceptor Failure Handler
+ init(signature, method_ref)
+ [bool] explicit_verification_before(params)
+ [list] handle_before(params)
+ [bool] explicit_verification_after(result)
+ [object] handle_after(result)
«uses»
«uses»

Figure 8 - Framework for local cycle implementation.

The *Autonomic Manager* class is the observer of the *Logger* module, thus it implements a callback for each listener listed before. These are the triggers for the autonomic loop cycle, explained hereafter. This manager class also holds a list of *Failure Handlers* descriptors, which can be loaded by a registration method (*register_handler*). There are two types of failure handlers: *EventActionFailureHandler* and *InterceptorFailureHandler*. The first one is for handlers that must modify the state at an exact point of the execution (Figure 9), and the second one for handlers that must modify passing parameters (Figure 10).

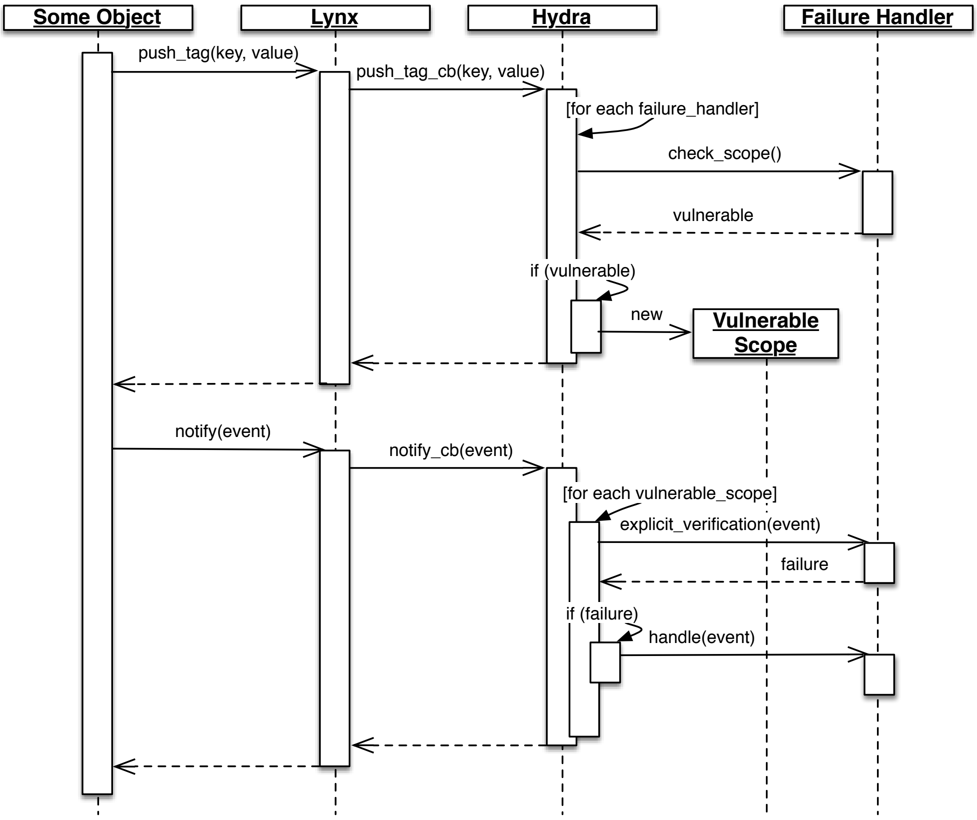


Figure 9 – Event Action Failure Handler.

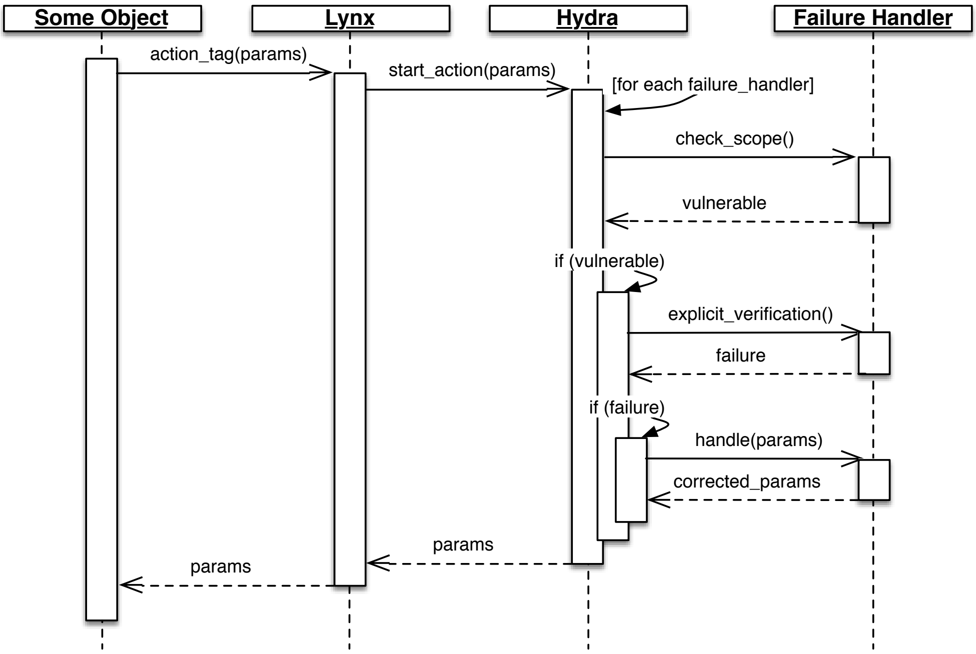


Figure 10 – Interceptor Failure Handler.

Both handler classes are initialized with a list of tags, representing the failure signature or part of it, which are used by the *check_scope* method to evaluate if the current state of the tag stack corresponds to a failure or a vulnerable scope. The explanation for this vulnerable scope representation is that some failure signatures cannot be completely represented on the tag stack, thus requiring an additional verification based on explicit and specific calls, which may impact on performance if called in every detection cycle. This is why the explicit verification is called only after confirming that the tag-stack state matches a possible failure signature. Therefore, subclasses of these handler classes must implement the *explicit_verification* method, which can immediately return true if the signature is completely represented on the stack, or execute further investigation through the current event or the log history (using the query engine).

Let us consider, for example, a failure signature described by the presence of the tags [*action*, *create_entry*] and [*user_type*, *manager*] on the stack, which also requires an explicit verification if the associated company has more than one manager (which would require a database query). In most cases, the current user will not be a manager, so before executing the expensive call that will make a database request in order to verify the explicit verification, the mechanism must confirm that all related tags are matched, since a false response avoids an impact on performance. Observe that the tag-stack state verification is done in the same process space of the execution, avoiding remote calls to evaluate the signature. Moreover, the normal execution continues only when the verification and recovery routines return, since this handling mechanism is designed for synchronous purposes.

The autonomic cycle is triggered by the lynx instrumentation, as already explained. When the cycle is triggered, the normal execution stops and all registered failure signatures are evaluated, thus the recovery routine is called if a signature is found. The cycle is triggered from two approaches: (1) based on the tag-stack manipulation and event notifications, which is implemented through the *EventActionFailureHandler*; and (2) based on actions notified by interceptors in method calls, which is implemented through the *InterceptorFailureHandler*.

The first approach is related to the *push tag*, *pop tag*, and *notify* operations, and hence uses the corresponding listeners. The *push tag* callback is used to execute the *check_scope* verification for each descriptor, in order to identify if the current tag stack represents a vulnerable scope, and if one is found, an instance of *VulnerableScope* is created and registered in the manager. For each notify operation, the *explicit_verification* of handlers with a vulnerable scope are called, passing the event as parameter; if it detects a failure, the corresponding *handler* method is called, also with the event used as a source to identify the occurrence. Finally, the *pop tag* callback is used to identify when the vulnerable scope ends, in order to remove its corresponding instance from the manager, thus avoiding the explicit verification in future execution.

The second approach addresses failure handlers that must access passing variables to and from a specific function, and change them according to the recovery needs. This approach uses the *start action* and *end action* callbacks to verify if the specific function called is addressed by any of the registered failure handlers. This verification is made by comparing the *method reference* received from the callback with the one received during the handler initialization. The *method reference* representation may vary depending on the programming language, since it can be as simple as the function name, which may be ambiguous, or a descriptor generated by an introspection mechanism, which is expected to be precise. Hence, if a handler for the specific method is found, the *check_scope* verification is called in order to verify if the current scope is vulnerable; if it returns true, the corresponding explicit verification is called (*explicit_verification_before* or *explicit_verification_after*) in order to attempt to detect the failure.

When a failure is detected, the *handle_before* method is called with the function parameters (received from the callback). This handler method must return a modified version of these parameters with corrected values, which are passed on to the specific function (through the callback return) in order to continue the execution. The same process is made with the specific function result, which is passed on to the *handle_after* method in order to be modified to a corrected value (if needed), and then returned to the caller function. Observe that in our solution the listeners were installed on the *action_tag* interceptor, which is implemented as a decorator in Python, for example. These listeners can be implemented, however, through other mechanisms, such as Aspects (Gradecki & Lesiecki, 2003) in Java, or as a simple object proxy.

The following example (pseudo-code hereafter) uses the second approach to show how a data inconsistency failure may be handled by a synchronous routine that intercepts the method call. The scenario consists of a function (*generate_report*) of a web-service that is used by different applications (a website, a mobile app, and a desktop application), which sometimes receives an inconsistent record (*report_data*) to process. Since the fault cannot be removed immediately, for example, because it requires a huge coding effort, a handler must be implemented in order to cope with the failure. Moreover, from the diagnostic of this failure we have identified that it is only triggered when the function is called by a request originated from a recent version (2.26) of the mobile app. This generates, in some cases, a record *report_data* with a reference to a s*hip* record (located in the server database) that does not have an owner (which should be an instance of the *company* class). The failure cause is a lack in the specification, since it does not determine that mobile applications may generate incomplete ship records. The failure observation is an exception while attempting to access the ship's owner. Since the association with a generic owner is irrelevant for all operations in the web-service, the handling routine associates the incomplete ship record with a generic owner named "No owner", which was manually created in the database.

```
class DataInconsistencyFailureHandler
    : extends InterceptorFailureHandler
{
    void init() {
        super.init(
            '[from:mobile][version:2.26]',
            ReportEngine.generate_report);
    }

    bool explicit_verification_before(Object[] params) {
        data = params['report_data'];
        ship = database.load(data.ship_id, Ship.class);
        return (ship.owner_id == -1);
    }

    Object[] handle_before(Object[] params) {
        data = params['report_data'];
        ship = database.load(data.ship_id, Ship.class);
        ship.owner_id = NO_OWNER_ID;
        ship.save();
        return params;
    }

    bool explicit_verification_after(Object result) {
        return false;
    }
```

```
        Object handle_after(Object result) { return result; }
    }
```

Observe that this approach is only feasible due to the information in the tag-stack, which enables the evaluation on whether the recovery must be applied. This approach reduces the impact on performance, since the high-cost verification (*explicit verification*) is only executed when a vulnerable scope is found. Moreover, the installation of both handlers is transparent for the functional code (thus avoiding polluting it), since it is encapsulated in instrumentation already written on it. In other words, the software methods remain unchanged, leaving the traditional way of writing code.

6.5 Threats and Validity

There are three main threats that may compromise the mechanism discussed above. The first one is the absence of tags required for matching the failure signature and extract additional properties to feed the recovery routine. It is impossible to predict which information will be needed to this end; however, if the most common properties are available, the possibility of achieving an effective result is higher. This threat is mitigated by the instrumentation policy described in Section 3.2, which is the most precise definition we can contribute with at this moment. These instrumentation guidelines were developed based on our experience with the four systems described in Chapter 7. However future research will be made along these systems' lifetime, in order to verify if it is possible to produce a more accurate set of guidelines. We have also observed that the tag action is required in most failure signatures, since it is a filter that drastically narrows down the signature for a single operation in the system. However, these signatures also require information about the execution state, for which there is still no defined pattern.

The second threat is the impact of the verification tasks over the performance, since these tasks must not compromise the main functionalities of the system under protection. This issue is addressed in the global cycle by adjusting each failure verification frequency to the maximum acceptable interval, which depends on both failure and the system's characteristics. The local handlers were designed to have a low impact, since the high-cost verification is only executed when a vulnerable scope is detected. The impact of a local handler over the performance is defined by a function of the number of handlers installed, the number of tags in each handler's failure signature, the number of tags in the tag stack, and the granularity of the instrumentation that triggers the cycle. For now, the trigger is associated with the *push tag*, *pop tag*, *notify*, *start action*, and *end action* operations, which are a coarsely grained instrumentation; hence, with a few failures being handled, its impact on performance is smaller than that of a continuous verification would have.

The third threat is the inability of installing a failure handler after deployment. The solution depends on the system's architecture and the technology available. As suggested in this chapter, when the mechanism is being implemented with a language that provides a reflection mechanism, it must be used to load all classes that represent failure descriptors (ex: from a given folder). Therefore, by using the reflection mechanism, the problem is reduced to the task of installing the failure handler module into the production environment, which in most cases will be a file copy. However, software written in languages that do not provide a mechanism to load and execute external code may also take advantage of the proposed solution. They will, however, be limited to a simpler approach, which would be using an external configuration (xml file, table in the database, etc.) to associate failure signatures to static pre-defined commands, which have to be implemented in the deployed component.

Finally, analyzing the visibility of the local handlers, there is the impossibility of modifying the state of variables that are not accessible from a global reference, such as those located on functions scopes (local variables). In some situations, this capability will be mandatory for the recovery approach, but the current solution does not provide mechanisms to inspect higher scopes in the call stack, neither to modify its content. This is a desired improvement and will be a topic for future research.

7
Evaluation

This research was performed using systems developed and maintained by a small software company. Most of the problems that were studied showed up as a need to solve some kind of difficulty in developing and maintaining these systems, which acted as a workbench, challenging with real-world problems. This was a major contribution to the solutions, as they were conceived in scenarios with the same complex variables expected to be found in the problems targeted by this thesis — variables such as technology, architecture and effort limitations; human fallibility; team competence; and most importantly, failure occurrences. The proposed solutions were, therefore, evolved along the systems' development. For example, an initial version of the information extraction solution was provided to the development teams, which provided feedback on every difficulty observed while instrumenting or inspecting the execution, allowing us to further improve the techniques. The failure handling solutions were developed through a similar approach: every failure — or risk of failure — identified in a system was used as a scenario for studying how the proposed information extraction technique could be used to detect the occurrence of the failure and to provide the necessary information for a recovery routine.

Four systems developed within that company were used as sources of information for the research. The process for choosing these systems aimed to form a heterogeneous set of domains that ensures a wider applicability range for the solution. The development teams for each system consisted of members with different skills and development experiences, contributing to a realistic — and, as expected, less-than-perfect — development environment. The evaluation of the solutions occurred in the context of these systems, which demonstrated that techniques developed for a given system could be migrated seamlessly to the others, even if from a different domain, with different architectures, requiring different sets of tools and frameworks, and developed by different teams.

The generality of the solution was demonstrated by the successful results obtained while applying the techniques to these four different domains, without any adaptation on the base principles, and little adaptations in the implementation — the instrumentation library was the only part of the solution submitted to these adaptations, since it is the most influenced by the domain characteristics, however, it showed a high rate of reuse while being ported between domains, thus softening even more the solution's portability. Therefore, we can attest that our evaluation evidences that the solution have a solid definition and one may expect the same result while applying to other domains.

This chapter is organized as follows: Section 7.1 describes the evaluation goals and how each part was evaluated; sections 7.2, 7.3, 7.4, and 7.5 describe each system used in the evaluation, with its main characteristics, a description of the instrumentation with measurements, the diagnosis assessment, and the failure handler assessment; finally, Section 7.6 wraps-up the chapter with a discussion on the results.

7.1 What and how to evaluate

The solution proposed by this thesis, described in previous chapters, consists of:

(1) A source code instrumentation technique to extract runtime information while the system is used in production;

(2) A policy to guide developers in using this technique;

(3) A tool to diagnose observed failures in order to determine their root cause, or at least discover their signature; and

(4) A mechanism to implement a handling routine to detect known-failure signatures and associate a recovery routine, which will be automatically applied when an occurrence is detected.

The diagnosis tool and the handling mechanism depend on a log annotated with contextual information, generated by the instrumentation technique. Evidently, if the log contains insufficient information, the solutions will be compromised. Therefore, the aim of this chapter is to demonstrate that:

(1) The instrumentation technique requires little effort to implement and implies a low overhead for the system execution, making the solution applicable in production environments.

(2) Failures can be diagnosed efficiently with the proposed diagnosis approach. Moreover, the precision of the instrumentation policy provides the necessary information for each diagnosis session.

(3) The recovery handler requires little effort to implement, avoids polluting the source-code, and imposes a low overhead on the system execution.

(4) The solution is applicable to usual system domains.

The first item is evaluated though measurements that compute the effort to instrument the code and its corresponding impact on system performance. The second and third are assessed through studies, both conducted using failure occurrences that were observed in a production environment. These failures will be separated into two groups: the first one for evaluating the diagnosis technique, and the second one for evaluating the failure handling mechanism. This separation into two groups enables a precise selection of the failures for each one, in order to explore more challenging scenarios for each solution.

Another possibility would be the application of mutant testing (DeMillo et al., 1978), by developing code to generate random failure occurrences while the system is being used, resulting in logs that are presented to maintainers who then have to diagnose the failures or generate failure handlers. The problems with evaluating the diagnosis solution by means of mutants are (1) having to insert a very large number of faults leading to a very large number of failures, so that in theory at least some of them will have some similarity with failures that could occur in a production environment; (2) without any guarantee the diagnoses will require relevant hypotheses (these related to faults in architecture, design, specification understanding, etc); and (3) the lack of a failure report that describes the reporter's point of view on each occurrence and provides a more realistic failure scenario for those doing the diagnosis. Therefore, this evaluation approach is unfeasible due to the effort required to generate and diagnose said failures with a group of maintainers. The problem considering failure handling is similar, with emphasis to the failure relevance issue: only failures with relatively serious consequences would motivate the development of a handler to mitigate future occurrences. In addition, each failure would have to be reproduced in order to test the developed failure handler. This would require great effort, since the type of inconsistency that produces the error may not be stimulated from the interface level (human or component), imposing the development of instrumentation to reproduce the failure scenario in order to test its corresponding handler.

The fourth item in the list above — the solution's wide applicability range — has been addressed by choosing systems from four different domains: web services, mobile applications integrated through a server, robotics, and embedded systems. Since their objectives, requirements, architectures, and development teams' proficiencies differ from one another, having success applying the solution in all of them is an indication that the approach has a reasonably wide range of applicability.

7.1.1 Instrumentation Measurements

The objective of these measurements is to demonstrate that the effort required to implement the instrumentation technique is very close to that of the traditional log technique, and the resulting overhead during the system's execution is acceptable for a production environment. Obviously, the result depends on the type and quality of instrumentation inserted, which must be sufficient to provide the necessary information for diagnosing and automatically handling failure occurrences.

The four systems described in the following sections were instrumented with the technique presented in this thesis (described in chapters 3 and 4). The first one, WinePad, was instrumented after having been completely implemented, and the other three, during development. Ideally, instrumentation should be inserted during development, when it is expected to require less effort than when added after this phase. Furthermore, this approach contributes to writing correct code, as the use of lightweight formal methods (Hall, 1990) stimulates developers to think about the problem at hand instead of starting to write before the solution is sufficiently well understood. Moreover, all of the four systems were instrumented by their own developers using their specific instrumentation policies, which are presented here in each of the system's descriptions.

The instrumentation effort was estimated based on the percentage of instrumented code, which was measured by counting the number of lines that use any of the instrumentation libraries' primitives (tag manipulation and event notification), then dividing it by the total number of lines in the code — after discarding comments and empty lines.

The impact on system execution was assessed based on two measurements: (1) the computing overhead when using the instrumentation, and (2) the additional space required to store the log meta-information. The first one was evaluated by subjecting each system to a controlled execution under a profiler mechanism, developed inside the instrumentation library to compute the time spent on instrumentation operations. The second measurement was evaluated by dividing the space required to store the meta-information of all events in the database — the set of tags excluding the message of each event -— for the total space required to store all events. Therefore, the result is the additional space required for applying our solution. It was also measured by a mechanism implemented into each instrumentation library. In both measurements, the system was stimulated for several minutes executing its main functionalities.

7.1.2 Diagnosis Assessment

In order to assess the effectiveness and the diagnosis effort required by the diagnosing tool (Lynx), we performed studies involving users in a controlled environment. Collaborators were asked to diagnose a set of faults purposely injected into the current version of the system, which was deployed as a clone of the instance in the production environment. All collaborators were familiar with the tool, since they had used it while developing the system.

We chose failures that had previously occurred during usage time and for which the diagnosis time was known, allowing the comparison with the time measured in these studies. As discussed before, this approach increases the assessment's validity since these faults correspond to real incidents occurred in the past. In addition, since the faults were removed in newer versions of the production environment, all faults were re-injected into the system and then stimulated to generate the log footprint. In addition to the footprint events, many other events were logged, as the instrumentation remained fully active. After that, each collaborator received access to the Lynx tool and to the corresponding failure report, containing all known information at the time the first occurrence of the failure was observed in the production environment. The objective was to recreate a scenario closest as possible to the first failure occurrence, in order to maintain the assessment's validity. We recorded the time needed to diagnose each of the reinserted failures and compared it to the time spent in the real occurrence. Moreover, all failures were selected after the instrumentation had been inserted, allowing us to assess if the events and tags defined in the instrumentation policy would be sufficient to explain the root cause of the arisen failures.

Choosing participants for this type of study was very difficult, since the selectable candidates must not have participated in the first diagnosis session, but must have some knowledge about the system's design and architecture in order to be capable of formulating hypothesis about the failure's root cause. Since the system was developed in a small company, very few candidates satisfied both requirements. We ended up selecting developers who had participated in system development, even for a short period of time, avoiding to present a failure to someone who had privileged knowledge about it — such as being the coder of the broken feature or someone who participated in the original diagnosis session when the failure was first detected. In addition, during the assessment we took care not to influence the collaborators, assisting them only in the use of the inspection interface.

7.1.3 Failure Handling Assessment

In order to evaluate the proposed solution's ability to handle known failures, we applied the Hydra mechanism (described in Chapter 6) to two systems. The objective was evaluate the impact on the system's performance and assess the solution's design — we expected seamless integration into the system. The impact on performance was measured by computing the execution time for the detection attempts inside and outside the component's process space. Moreover, the autonomic cycle framework was implemented using Python, and both the Python and C++ instrumentation libraries were extended to implement the failure handler support.

The efficacy of the solution was measured by exploring failures identified in the target-systems. Both systems presented, during development and production phases, failures that could be handled by the proposed mechanism, but most of them are too simple and would contribute little for this evaluation. Thus, we selected a few to employ effort in explaining the failure and how it was handled by the solution, in order to demonstrate the mechanism capabilities.

Sections 7.2 and 7.3, which respectively address the specific evaluation of WinePad and EMR systems, will describe these selected failures as well as their corresponding handler solutions. The assessment consisted in developing the failure handlers, installing them on the system, removing the ad-hoc approach when applicable, and then submitting the system to scenarios that stimulate the failures, in order to verify if the mechanism was capable of detecting each failure signature and proceed with the recovery routine. Each failure's scenario was generated by either real hardware or mock objects, depending on the input requirements.

7.2 WinePad

This system is a digital menu implemented as a distributed system with client-server architecture. The application that runs on each client was developed in Objective-C using the iOS platform, while the web-service runs in a cloud — Amazon Web Services (AWS, 2014) — and was implemented in Python using the Django framework.

This system (Figure 11) consists of sets of tablets used by sommeliers, waiters, sales people, and, mainly, restaurant guests. Each set of tablets interacts, over a wireless network, with a central server in a cloud, which handles all customer accounts. The server provides a web application for content management and a synchronization service to handle the tablets' content update. The customer administrator uses this application to manage the content delivered to the tablets. Moreover, any specific business may have more than one administrator, like a *chef* who defines the menu for each day of the week, a sommelier who daily updates the wine list and suggests food and wine pairings; a manager who defines the price for each item in the wine list and its availability in stock; and a marketing analyst who manages advertisement content. The synchronization service provided by the central server compiles the information for each business and publishes it in each tablet owned by the business. Finally, other tablet users may access the menu and choose what they want.

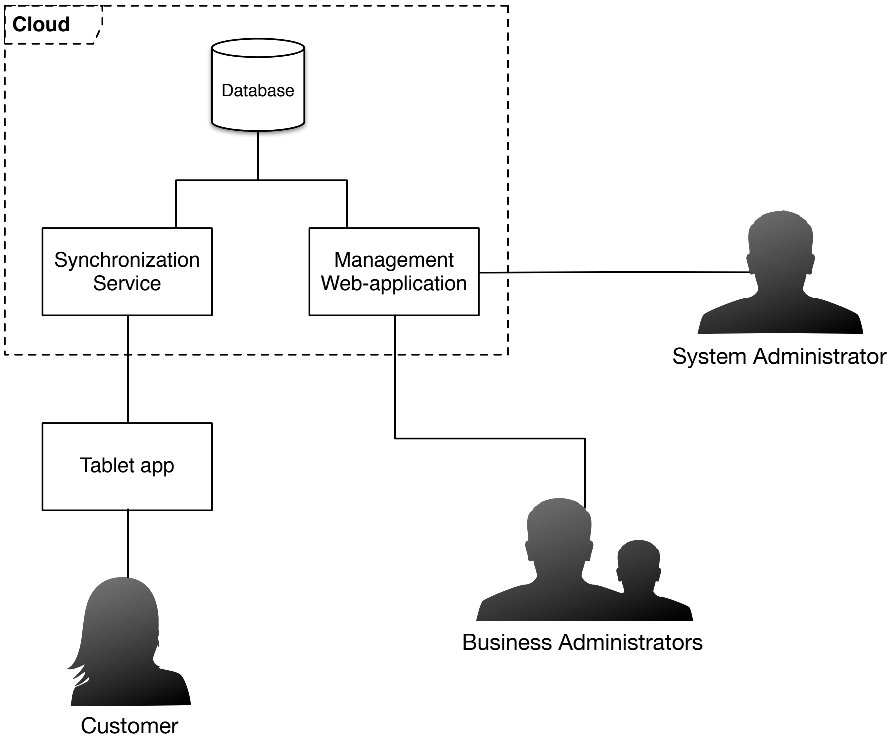


Figure 11 – WinePad Architecture's Overview.

The application that runs on the tablet is a simple Software Product Line with two layers: (1) a core asset providing hot-spots that allow the change in appearance and usability of a view, and (2) a group of features bound to these hot spots. The appearance and usability are sensitive to the choice of assets and the settings defined in configuration files. The web application content management allows the product manager to create, edit, and remove different types of features, also providing a mechanism to configure the mobile application's behavior. Finally, the synchronization service is responsible for the incremental update of the content and settings of the tablets.

This system is interesting for the evaluation of our solution due to the difficulties inherent to its usage environment, which relies on concurrent work of different kinds of actors, simultaneously using the same business account. These actors are: a system administrator configuring the product line for each business according to his/her needs; a cataloging team entering the product's data sheet (e.g. wines, dishes) required by the business administrator; a design team producing the layout according to the specifications of each business' brand; and the business administrators listed above. These actors interact not only during system implementation, but also while it is being used. In other words, the system admits an unusual number of super users, who sometimes work concurrently within the same account, frequently without being aware of it. In such a context, lack of communication between the involved actors, lack of attention and human fallibility may lead to inconsistent environment configurations. As an example, let us suppose that an administrator changes the settings in assets *A* and *B*, and inserts asset *C*. Afterwards, this administrator asks the designer to adapt the layout to these changes, forgetting to mention the inclusion of asset *C*. Furthermore, each component of the software is versioned independently, and the mobile application must maintain backward compatibility. Hence, whenever evolving the web application, the server must provide content compatible with all the versions currently in use. This system presents yet another difficulty for diagnosing failures, as end users have no interest in reporting failures and lack knowledge necessary for doing so.

7.2.1 Instrumentation

The WinePad system was developed by a team of three members: one software engineer, who wrote the most sensitive modules from each application (mobile and server), and two novice developers, one developing the mobile and the other the server application. The instrumentation was written by two of these three members, after the development phase had ended, by using the following instrumentation policy:

Rules for notification

For the server application:

- Starting and ending of functions and methods (parameterized routines representing an specified action of the system, discarding helper ones such as getters and setters)
- Every decision edge of a view (function that renders a webpage or handles a remote request from the mobile application).
- Authentication mechanism.
- Create, remove, update and delete (CRUD) of database entries, indicating the type of operation.
- Explicit database query (when not using a framework for building SQL or NoSQL queries).

- Sending emails.
- Routine tasks (ex: compute statistics based on recent data).

For the mobile application

- Create/show a ViewController (iOS class for controlling a view).
- User interaction, notified at the start of the corresponding callback.
- Every edge of a callback that handles a user interaction (except auxiliary functions).
- System parameter modification.
- Explicit database query (when not using a framework for building SQL queries).
- Every edge of the synchronization routine.

Used tags

- src_line – source-code line.
- src_file – source-code filename.
- error – error description.
- warning – suspicious function.
- exception – placed in the catch block, without value.
- stacktrace – the execution stacktrace.
- environment – the execution environment (ex: mobile).
- thread – identifier for the current thread.
- action – high level operation being executed.
- collection – database collection that is being manipulated.
- database – name of the database that is being used for the operation.
- host – computer hostname that made the access.
- ip - IP address of the user that made the request
- is_admin – indicates if the user is an administrator of the system
- user – identifier of the user that made the request
- request_id – identifier of the request
- device_id – identifier of the mobile device
- organization – identifier of the customer related to the operation
- device_status – device state (active, inactive, pending update, synchronizing).
- device_version – identifier of the client’s version.
- current_view – current view in the mobile application.

- model – typename of the record.
- id – identifier of the record.
- item_code – identifier of the product item being manipulated or exhibited.

As described in Chapter 3, some reuse patterns were applied to avoid rewriting frequent tags. The most relevant patterns are:

- The application startup pushed the tag *environment*.
- The instrumentation library automatically appends the tags in every notification: *timestamp*, *thread*, *src_line* and *src_file*.
- The instrumentation library automatically appends the tag *stacktrace* when the tags *error* or *exception* is present on the tag set received for notification.
- An interceptor in the webserver interface to automatically identify information from the request and append it to the tag stack. This implementation pushes the following tags: *host*, *ip*, *is_admin*, *user*, *request_id*, *device_id* and *organization*.
- The *action_tag* mechanism that uses the function name as the tag value. Implemented in Python with decorator, and in Objective-C with a macro.
- A listener for the database intercepts the CRUD operation and notifies events with the operation type and data.

This system's instrumentation did not present any specific difficulty to be made. As expected, developers created new tag types while instrumenting, shaping the log content to the specific abstractions of the code they were writing. Both applications were measured as explained in Section 7.1.1, with the results presented in the tables below.

	Source lines (KLOC)	**Event notification (%)**	**Tag manipulation (%)**	**Total (%)**
Mobile app	13	6.40	4.11	10.51
Server app	16	7.43	1.14	8.57

Table 1 – WinePad's implementation effort.

	Computing Overhead (%)
Mobile app	1.43
Server app	9.95

Table 2 – WinePad's computing performance overhead.

	Number of events	Data Overhead (%)
WinePad	14.748	647.17

Table 3 – WinePad's storage performance overhead.

The discussion about these results is at the end of this chapter, in Section 7.6.

7.2.2 The Diagnosis Tool Evaluation

Two studies were executed, each with different failures observed in the deployed system. The first one was executed with the first version of the WinePad, which was initially instrumented without the policy guide — this was the tool's first assessment. The second was conducted using only one failure, recently observed in the production environment. This time the system was instrumented according to the instrumentation policy. The following subsections will describe the selected failures, informing the failure observation and the correct diagnostic. After that, we present the studies' results. Observe that failures from different types were selected, forming a heterogeneous set, in order to demonstrate the generality of the solution.

7.2.2.1 Misconfiguration failure

The failure observation was: *"A new customer signed up, I configured his account and sent him his login and password by e-mail. He just called saying that although he has correctly installed the application on the tablet, he cannot activate it with his credentials. The error message says there is a problem in the account configuration"*. The correct diagnostic is that the administrator who registered the account forgot to upload the layout produced for this customer. Without having all the resources, the application in the tablet cannot start.

7.2.2.2 Logic error

The failure observation was: *"A customer just called complaining that his tablets are not synchronizing. He has modified a dish specs in the web application, but hours have passed and the tablet continues to exhibit the old information"*. The correct diagnostic is an implementation error that allows a dish to be paired with a wine that is marked as unavailable for customers. When the synchronization server compiles the data to be sent to the tablets, it lacks information and aborts the operation.

7.2.2.3 Retrocompatibility error

The failure observation was: "*A customer installed a trial version of the mobile application in his tablet three months ago and did not activate the account at that moment. Now that he became a regular customer, he activated the account, but the tablet says that there is a configuration problem. He is using the correct credentials and I am sure I uploaded his layout and configuration file correctly"*. The correct diagnostic is that the application version installed on this customer's tablet was outdated and required a previous version of the layout. As the account was not active at the time when he made the installation, a compatible configuration was not automatically installed. It is a fault in the web-service view layer, of which is meant to support previous versions.

7.2.2.4 Data inconsistency

The failure observation was: "*All tablets' menus are presenting some sections that were not defined in the web application. The dishes inside these sections actually exist, but belong in another section"*. The correct diagnosis is that a recently implemented feature that copies entries (wines and dishes) between customer's accounts produces a data inconsistency that is imperceptible in the management system, due to an anomaly design in the model that enables a dish to be associated with more than one section. In other words, a dish record has a foreign key to a category, which has a foreign key to the corresponding section; however, the dish also has a foreign key directly to the section. When the new feature was written, the developer did not know this anomaly, or forgot to correct the association in the record's clone. This failure was especially difficult to diagnose due to the root cause being far from the observation, and cloaked by a considerable amount of similar records.

7.2.2.5 The First Diagnosis Study

The first study was conducted with four people, more specifically two developers and two system administrators. The failures chosen for this study were those described in sections 7.2.2.1, 7.2.2.2, and 7.2.2.3. Observe that the faults chosen are simple to explain; however, their diagnosis based on observation is difficult when using traditional techniques. Usually, tablets' logs are inaccessible and the server presents a considerable volume of records involving several operations from different customers, making the analysis more difficult. The first fault occurred in the first months after deployment, and had an average diagnosis cost of 30 minutes per incident. It occurred several times before it was removed. The time to diagnose each failure did not vary much, even after the maintainers learned the fault's cause, because the co-evolution of the software masked it in different ways, exposing at each occurrence a different footprint. The other two faults occurred only once, the first having a diagnosis cost of one hour and the second of 2 hours and 30 minutes.

The measured times are shown in table 4, along with the times spent using the traditional approach when the failures were discovered in the production system.

	F1	F2	F3
Traditional approach	≈ 30	≈ 60	≈ 150
Developer 1	18	7	5
Developer 2	6	6	9
Administrator 1	3	2	6
Administrator 2	4	3	2

Table 4 – Time in minutes to diagnose each failure on the first study in WinePad.

The result exceeded our expectations and surprised by showing that the non-technical users had a better performance than the developers. When trying to identify the cause, we concluded that this was due to two factors: (1) these users act directly on the production system, close to these types of faults, and (2) they have a simplified view of the whole system, conceiving a smaller set of hypotheses for the possible causes of the failure.

7.2.2.6 The Second Diagnosis Study

The second study was conducted with 3 developers, one who had actually participated in the development of the system, and other two who had necessary domain expertise and knowledge about the system's design and architecture to diagnose the failure. The failure chosen for this study was the remaining one, described in Section 7.2.2.4. This failure was very difficult to diagnose when observed during production, since when the error was observed (the data inconsistency), there was no clue allowing the elaboration of hypothesis to search for the fault that produced the inconsistent record.

	F1
Traditional approach	≈ 360
Developer 1	9
Participant 1	12
Participant 2	31

Table 5 – Time in minutes to diagnose the failure on the second study in WinePad.

The measured times represent a huge discrepancy when compared with the time of the original diagnosis. Although we must consider some psychological factors, such as the absence of the stress to avoid failure consequences and its impact on business (as in the production environment), while analyzing the diagnosis strategy used by the collaborators during this study, we could identify two the key factors for their effectiveness: the record identifiers present on the events, which enabled them to quickly locate its correlations; and the request identifier, which enabled them to look for all events in the interesting request. In the original occurrence, the developer had to look for this information through trial-and-error attempts, re-executing the system and inspecting the production database — at the time of the original occurrence, the logs either did not provide this information or had it without being indexed, as opposed to the presented solution.

7.2.3 The Failure Handling Mechanism Evaluation

The WinePad system (as well as two other systems which are not in this evaluation) presented failures related to component versioning, between devices and web-services, which demanded some time to handle due to third-party services' limitations. We chose one of these failures, which proved to be the most difficult to handle at the time, to illustrate the capabilities of the proposed solution, and explain how failure handlers must be developed for the local cycle. Before this explanation, however, it is necessary to describe the system's development scenario. At the time the company that developed this system was releasing the second version of the client's software — the software embedded in the tablets —, it had only three developers. The first version of the system was just a beta, released to few costumers invited to participate on the product's evaluation. Despite the quality control effort made during the development phase, when the second version was being released the team's workload was beyond what the startup could handle, and some mistakes got through. One of them was extremely difficult to handle and, although not a critical service, harmed the product's reliability for a while, seriously threatening the company's existence. Therefore, if the failure handling mechanism proposed in this thesis were available when the failure occurred, the risk and the effort spent handling its consequences would be lower.

7.2.3.1 Mistaken retro-compatibility

The failure derived from mistakes while releasing the second version of the client's application. The person responsible for the release forgot to increment the version property of the mobile application. As a consequence, for some time the two different versions were simultaneously available and using the same version identifier. There was also a second fault, which was the *update* procedure of the server application being developed without a retro-compatibility mechanism to handle requests from the older version of the mobile application. This caused every request made from devices using the older version to fail processing the response, as the returned data was considered an unsupported format (the data structure had changed). This scenario is illustrated in Figure 12:

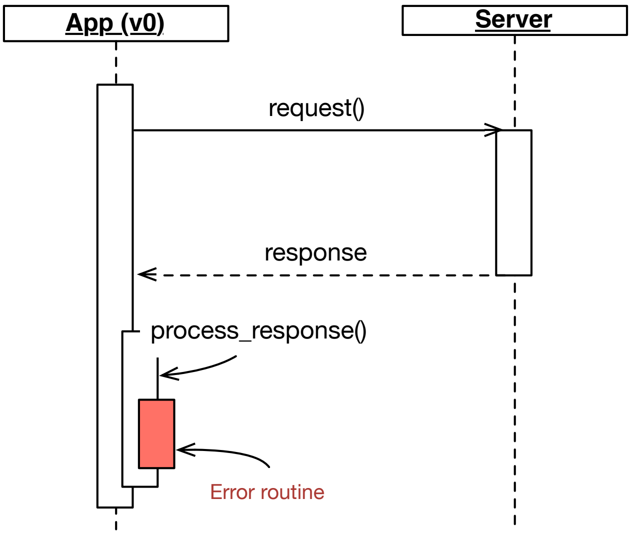


Figure 12 – WinePad Failure Example.

There was no quick process to remove these faults, since the client's application was delivered by Apple's AppStore service, which takes between two and three weeks to approve every new version. As the error could only be observed on the mobile's application, there was also no good indication on how to generate an automatic *ad-hoc* solution on the server's side to choose the appropriate format. The webserver application could be modified with less effort, but there was also no hint in the request parameters to identify whether it was generated by the newer or the older client's version. The *ad-hoc* solution was the generation of a list of customers, then manually identifying which had updated the software (and ask the others not to update until the next release), and then to use this list in the server application as a guide to decide which client application should use the newer version of the procedure. This *ad-hoc* process demanded a huge effort, involving contacting dozens of customers — and could be even worse if applied for tablet devices, of which there were hundreds at that time.

In order to demonstrate the capabilities of the proposed technique, two failure handlers were implemented to handle occurrences of the described fault. The objective was not to assess the specific recovery routine for this failure, but to demonstrate that our solution is capable of introducing an atomic software artifact to handle a given failure, then removing it from the system with little effort when no longer needed. Both handlers were developed for the local cycle (described in Chapter 6), since this type of failure needs to be handled synchronously. The first handler targeted the versioning failure, detecting when a device is running the faulty version of the software and modifying the request descriptor (in the server) to exhibit the correct one. The second handler detected when a device running the old version of the application executes an update procedure, which would trigger the failure, and then modify the content of a file, which will be sent in the response, in order to avoid the failure. Both handlers were installed in the server application.

The first handler, targeting the version failure, was implemented using the interceptor approach. The tags used to identify the vulnerable scope were [version:1][action:update], and the *explicit verification* method was implemented using the query engine to find a previous request of the given device that resulted on a failure. This can be done by looking for an event with the following six tags: (1) the device's environment (*mobile*); (2) the request name (*update*); (3) the same *device* tag as the contained in the current request; (4) the tag *error*; (5) the message explaining that fault ("*Failed while parsing field*"); and (6) the tag *field* containing the name of the field that failed to be parsed (*price*). If this event is found in recent events (temporal limit of 5 minutes), the device that made the request is using the faulty version, thus the recovery routine must be applied. Hence, the recovery routine consisted of a simple modification of the request descriptor by correcting the version parameter to the right value.

The second failure handler, targeting the absence of the retro-compatibility mechanism, was implemented using the event action approach. The tags used to identify the vulnerable approach were [version:1][action:update], and the *explicit verification* method (which receives an event as parameter) was implemented looking for an event with the message "*Data compilation complete*", which is immediately notified after the data to be sent as the response was saved on a file, which is located in the *path* tag (also present on the event). Hence, when a failure occurrence is detected the handle method opens the file and process the content modifying its structure, in order to return a compatible format with the old version of the mobile application. Hypothetically, if the content was not in a file, but compiled in memory and then returned, the recovery solution would require the interceptor approach instead of the event action approach.

Observe that this second handler must be executed after the first one, in order to avoid matching requests from the ambiguous mobile version. This sorting was made by registering the handlers explicitly on the local cycle, which was not a problem since the server application could be modified with little effort (however, it was the only modification, consisting in two lines in a single startup script of the server).

Therefore, the necessary tags to develop these handlers were: (1) the environment identifier in all events; (2) the device identifier (tablet serial number) in all events notified by the mobile application; (3) the action name in all events related to a server procedure, comprising events from mobile and server applications; (4) the error tag inserted in the specific event that notifies the failure observation when processing the response; (5) the version tag in all events, comprising events from mobile and server applications; and (6) the *path* tag in the event related to the file manipulation. It is undeniable that the device's application must use these tags in order to make the approach feasible, however when creating a policy guide for this type of system these tags are considered the basic ones, thus it is expected to be present.

This handler solution was tested on all versions of the client software: the correct first version, the faulty first version (which is the second version), and the corrected second version. It worked as expected, applying the recovery of each failure descriptor when needed. Observe that our objective was not to assess the specific recovery routine for this failure type, but to demonstrate that we can introduce a handler and remove it with little effort. Moreover, the first handler could be generalized to address similar failures on other procedures (due to similar faults), and then remain in the system until the old version becomes unsupported, since this fault cannot be removed (due to the impossibility of forcing the client's update). On the other hand, the second handler could be used as a quick solution until the proper retro-compatibility mechanism is developed and deployed, being removed thereafter.

7.3 Environment Monitoring Robot (EMR)

The environment monitoring robot (EMR) system was developed for a robotics platform that aims to provide loosely coupled mechanical and hardware components. Its goal was to adapt the solution to newer environments with little effort and without having to redesign the entire robotics solution. In order to comply with these requirements, the software system was developed as a component system using a middleware for robotic (described hereafter), which is deployed as a distributed system. The components of this system are deployed in

(1) embedded PCs, which are responsible for data acquisition and robotic control; (2) microcontrollers, responsible for hardware interfacing and low-level security; and (3) a notebook for human control and supervision. These components were implemented using C/C++ and Python languages. All PCs and notebooks execute an Ubuntu Linux environment.

The robot system can operate in automatic, semi-automatic and manual modes, being the semi-automatic a composition of components in automatic and manual modes — for example, even while the operator is controlling the robot, a component in automatic mode may automatically modify the robot's path if it identifies obstacles through ultrasound sensors.

The Robot Operating System (ROS) middleware (ROS, 2014) provides an infrastructure for developing software components that are executed independently, even when deployed in the same machine. The ROS component provides two mechanisms for message passing. The first one is the *topic*, which provides the functionality of a data producer in the producer-consumer design pattern. Every component that registers for listening to a given topic will receive generated messages as a flow. The second mechanism is the *service*, which provides client-server functionality. When a service request is generated for a given pair <component, service>, the target component processes the input and returns a response. The client is able to handle the response in a synchronous or an asynchronous way. In addition, the ROS Component provides the *property* concept, which is an externalization of an internal state that may be read and written by other components.

System components are classified into three categories: drivers, controllers, and viewers. *Drivers* are components that interact with hardware equipment, thus being responsible for providing a software interface for interacting with the physical component. This type of component also monitors the equipment's behavior through sensors, in order to provide more information for the controller layer. *Controllers* are the clients of the drivers. This type of component usually implements robotics concerns in order to evaluate the physical environment through sensors, and then plans the next steps to reach the desired target, implemented through component drivers. For example, odometer sensors for wheels, an inertial measurement unit (IMU), and a GPS may be combined by an algorithm in order to determine the vehicle's position, direction, and orientation, using the result to reason about how it can reach a desired position. These sensors are sampled by driver components; the locomotion algorithm, however, is implemented in the controller components, which use the new perspective to define the command set for each driver component. All *viewer* components are deployed in the control base software, located in the mentioned notebook. This type of component is responsible for providing (1) mechanisms to control the vehicle; (2) mechanisms to expose the internal state of the system; and (3) physical and biological information about the surrounding environment, which is the main goal for the robot's mission. These viewer components are combined in a MVC architecture and executed in the same process space, in order to facilitate the task of rendering its associated visualization in the same graphical interface. Finally, there is also a *deployer* tool, that allows the operator to select the available equipment set in the robot, in order to only deploy components related to them in the embedded PCs and in the base control application.

This system is interesting to evaluate for being a mission-critical system, which relies on complex interactions between different types of equipment in order to achieve high-level goals. Failures on this system are usually related to equipment malfunction, thus handling mechanisms must be developed to cope with them in order to keep the system running — even if this leads to reduced functionality or reduced information precision.

7.3.1 Instrumentation

The EMR software system was developed by a team of six members: one software engineer with expertise in mission-critical systems development, two experienced developers with expertise in software development for hardware equipment interaction, one developer with expertise in human computer interaction (HCI), and two experienced engineers from the robotics field, with little coding proficiency. The instrumentation was written during system development by three of these six members, by using the following instrumentation policy:

Rules for notification

- Exception notification and handling.
- Access to component properties.

- Instantiation/finalization of a component.
- Localization at startup (machine and PID).
- Component setup routine.
- Starting and ending the component's main loop.
- When sending or receiving messages from a topic or service.
- When sending or receiving data from a device (ex: probe, motors, camera, etc.).
- When a failure or suspicious activity occurs (ex: error when communicating with a device).

Used tags

- src_line – source-code line.
- src_file – source-code filename.
- error – error description.
- machine – name of the machine.
- pid – identifier of the process.
- keep-alive – presence indicator.
- warning – suspicious malfunction.
- exception – placed in the catch block, without value.
- stacktrace – the execution stacktrace.
- action – high level operation being executed.
- component – type of the component.
- node – name of the component instance.
- start_exec – indicator for the moment the component started.
- end_exec - indicator for the moment the component terminated.
- publisher – scope of a publishing routine.
- topic – name of the topic.
- service – name of the service.
- yaml – access to the parameter's file.
- param_name – parameter name.
- param_value – parameter value.
- init – scope of a node's initialization.
- loop – scope of a node's main loop.
- data – data sent/received in a communication operation.
- port – port path (ex: /dev/ttyS0).

- ip_addr – IP address (ex: 192.168.0.1).
- equipment – type of equipment.
- sensor – type of the sensor.
- measure – measurement of the sensor.
- alarm – type of alarm.

As described in Chapter 3, some reuse patterns were applied to avoid rewriting the frequent tags. The most relevant patterns are:

- Every component's main function pushes the tags: *machine*, *pid*, *component*, *node*, *start_exec* and *end_exec*.
- The setup is usually executed in a separated function, which received the scope tag *init*.
- The main loop is usually executed in a separated function, which received the scope tag *loop*, and a notification with the tag *keep-alive* in each loop.
- The instrumentation library automatically appends the tags in every notification: *timestamp*, *src_line* and *src_file*.
- The instrumentation library automatically appends the tag *stacktrace* when the tags *error* or *exception* is present on the received tag set for notification.
- The tags *publisher, topic and service* were inserted by a Python decorator that uses the function name as the tag value.
- The *action_tag* mechanism that uses the function name as the tag value. Implemented in Python with decorator, and in C++ with a macro.

The only specific difficulty encountered while instrumenting this system was in the association of the values of the tags *topic* and *services* with their real paths. Specifically for this type of system (based on ROS middleware), a mechanism was developed to push these scoped tags in a more transparent way.

Similar to the WinePad result, developers created new tags to represent specific abstractions, unforeseen by the instrumentation policy. All components were measured as explained in Section 7.1.1, the results being found in the tables below.

	Source lines (KLOC)	Event notification (%)	Tag manipulation (%)	Total (%)
C/C++ components	13.7	6.24	4.05	10.29
Python components	15.1	6.96	7.23	14.20

Table 6 – EMR's implementation effort.

	Computing Overhead (%)
C/C++ components	0.51
Python components	0.84

Table 7 – EMR's computing performance overhead.

	Number of events	Data Overhead (%)
EMR	30.889	615.51

Table 8 – EMR's storage performance overhead.

We shall discuss these results at the end of this chapter, in Section 7.6.

7.3.2 The Diagnosis Tool Evaluation

The study was executed on a failure observed while using the system in a field test. Most of the failures in the EMR field test were simple, being caused by obviously misconfiguration, such as forgetting to power on a microcontroller or configuring a wrong IP address in one of the cameras. The failure chosen for this study, however, was a misconfiguration that demanded more time than usual to diagnose, making it appropriate for this evaluation. The objective with this scenario is to demonstrate the generality of the solution to exhibit system properties as meta-information in a way that aids identifying inconsistencies, thus elaborating hypothesis earlier that would have been with the traditional approach.

The next subsection will describe the failure, informing the observation and the correct diagnosis. After that, we present the study's results.

7.3.2.1 Misconfiguration

The failure was observed due to the absence of a video streaming in the user's interface. This video streaming was produced by a component responsible for decoding video images, which are received by a camera IP that transmits data over an ethernet connection. This failure occurred in a field test and the obvious diagnosis — verifying if the camera node was running and then checking whether it was configured with the correct IP address — was executed. Surprisingly, when accessed by a browser interface, the streaming images appeared as they should. Thus, with all verifications having been made, the component's node still didn't receive images to process. This led all the personnel involved in the test (without any concrete hypothesis in mind) to start verifying physical connections and to restart the environment many times. Only much latter one tester came up with the idea of verifying the component's launch configuration, which contained the fault. The camera instance had been created in the wrong machine, so its IP was unreachable for the component, since it was configured for a different network (the robotics internal one). Therefore, the error was the launch configuration instructing the deployer tool to instantiate the camera node in the control base instead of in the embedded PC, thus preventing the camera node from accessing the streaming images.

This failure scenario was recreated in a slightly different form, as the camera equipment and the embedded network were not available at the time the study was conducted. The new scenario is the water probe component node being deployed in the wrong machine (the control base), thus being prevented from receiving data from a serial port, available in the embedded PC. Observe that this scenario keeps all of the interesting characteristics of the original one, and, hence, can be used to evaluate the same failure type.

7.3.2.2 The Diagnosis Assessment

The study was conducted with 3 developers. Two of them had actually taken part in the development of the system and the other one was a novice developer but with sufficient knowledge about the system to diagnose the failure.

	F1
Traditional approach	≈ 30
Developer 1	5
Developer 2	6
Participant 1	9

Table 9 – Time in minutes to diagnose the failure in the EMR study.

All the times observed were significantly lower than those required to diagnose the original occurrence. Despite this successful result, these times could have been even lower, since we have noticed that the failure footprint was quickly presented to the collaborators when they were filtering events from the probe node: every event had the tag [machine:base], which is the only necessary information for the diagnosis. However, visual pollution, due to other tags, prevented them from observing it.

7.3.3 The Failure Handling Mechanism's Evaluation

The EMR system was developed using strict quality control, making the application of the hydra mechanism preventive, instead of remediative as occurred in the WinePad system. During the EMR modeling phase, a study inspired on risk-based inspection (RBI, 2014) was conducted to list all the risks that could lead to possible failures in the system, in order to guide the developers during the implementation of the corresponding failure handlers. Even though, as discussed in the first chapters of this thesis, it is impossible to predict all possible failures that may occur, we managed to identify a group of recurring failures based on our previous experience. Before preparing the abovementioned list, we established a set of properties that must be defined for each one. The properties and its descriptions are:

- General
    - **Name** – Short description of the failure.
    - **High-level signature** – Observed behavior after the occurrence.

  - **Services compromised** – Description of how the failure deprecates the system.
- Detection
  - **Mechanism** – Can be (1) *global autonomic cycle* or (2) *local autonomic cycle*.
  - **Subsystem** – The machine where it should be installed. Possible options are (1) embedded PC, (2) embedded microcontroller, or (3) control base.
  - **Component** – The target's component name. Only applicable for handlers of the local cycle.
  - **Signature** – Description of how the detection strategy must be implemented.
- Recovery
  - **Routine** – Description explaining how the recovery must be applied.
  - **Policy** – Defines whether it must be applied. Possible options: (1) always; (2) N times, informing N; and (3) never. This last one is used for failures without remedy but that must at least be detected, thus notified.
  - **Check routine** – Description explaining how the recovery success must be verified.
  - **Actuators** – Component services required for applying the recovery.

All developers participated in creating the failure list, which resulted in 176 possible failures, both internal (software and hardware) and external (robotics and mechanics). The process required listing all failures for each component group or machine, then aggregating all of them in a single list. A component group is a set of components with the same objective, such as the controller of the vehicle's movement. The total number of component groups was 31 at the time of this evaluation.

The total number of failures comprises repeated ones — the failure of losing a serial connection with a device, for example, appeared nine times. Hence, the total number of unique failures is 41, which can be detailed as 20 for software, 11 for hardware, 06 for robotics and 04 for mechanics (non-software failures are those for which the root cause is external to the software environment, but that can be detected, and maybe handled, by software). Each one of these failures was evaluated in order to define if a hydra handler could be implemented. We found out that this was not possible for 07 of the failures: none in software, 05 in hardware, none in robotics, and 02 in mechanics. The hardware and mechanics failure handlers that could not be implemented were limited by the absence of sensors or equipment capable of providing necessary information to detect failure occurrences. For example, a damaged ultrasound sensor cannot be detected since others cannot be installed for monitoring the same spot (thus comparing the results). This demonstrates that solutions based on redundancy cannot always be applied, as mentioned in the introduction. Some other equipment can aid detecting collisions, but the specific failure of the damaged sensor cannot be detected in this system. Furthermore, from these 34 failures handlers feasible to be implemented, 10 only detect the occurrence, since an automatic recovery routine does not exist or cannot be developed. It is possible, for example, to detect that the underwater probe is streaming the data in a wrong format (not NMEA, as expected), but there is no programming interface to automatically configure it, thus the handler is restricted to only informing the operator. Therefore, in order to keep the operator aware of failure occurrences, an interface component was developed to exhibit the list of raised alarms, sorted by status (open and resolved), priority (indicator defined by the operator’s experience), and timestamp.

It is also worthy to discuss some aspects of applying the hydra mechanism on this system — although it is not part of the proposed technique. The designed architecture for this system (Figure 13), which is not completely implemented at the moment this thesis is being written, is divided into three logical computing units (Base Control PC, Embedded PC, and Embedded Microcontroller). These, in turn, are implemented as five physical computing units: (1) the base control PC, which is a laptop that provides a software interface with mechanisms to monitor and control the robot; (2) the primary PC, which is a x64 embedded PC that executes the robot’s software components; (3) the secondary embedded PC, which is extremely limited on resources and is used only for emergency scenarios (when the primary PC is damaged); and (4) an Arduino[3] microcontroller for hardware interfacing, with a (5) replica for redundancy (the secondary Arduino).

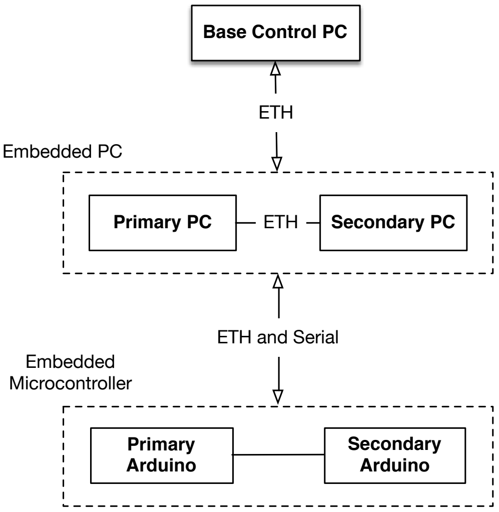


Figure 13 – EMR Architecture

With the exception of the base control PC, the other units are present in a local network inside the vehicle, thus taking advantage of an environment with higher communication reliability than when discussing an Internet system. Nevertheless, when the solution was designed, each computing unit received a global cycle agent instance, which can be recognized as an environment-local cycle instance, and then a failure handler solution was specified for each unit to address the lack of communication with all other units. For example, the microcontroller's communication failure handler was responsible for detecting when the most powerful PC is inoperative and for activating its redundancy (the weakest PC), and, if this one also becomes inoperative, for activating the disaster routine that notifies the GPS position by SMS. However, when the most powerful PC becomes operative again, the control is transferred back to it.

At the moment this thesis is being written, there already is an initial version of this system, but as this is a very large project, there are still many features awaiting implementation, and peripheral equipment awaiting integration. In fact, some failure handlers defined in the list address failures on these same equipment, and thus were not yet implemented yet. For example, the failure handler in each

[3] Arduino is an open-source platform developed for building digital devices and interactive objects that can sense and control the physical world.

computing unit to address lack of communication and proceed with a recovery is a future implementation, however the solution's design is based on the approach presented in this thesis. On the other hand, a set of different failure handlers is already implemented and has been used to assess the proposed technique. We have selected a small set of failures to demonstrate how its corresponding handler solution was designed, and present them in three groups in the following subsections.

7.3.3.1 Inoperative component

This handler targets one of the most basic failure types: the component stops working. This situation may occur due an internal error that makes the process hang or be closed by the operating system. Both detection and recovery routine implementations are generic, in order to use the same handler solution to address this kind of failure in all running components. This handler is implemented for the global cycle, presented again in Figure 14, since after the failure occurs the component's instance is terminated or become inoperative. Therefore, the handler's observation of the failure is made externally, and exclusively, by the log.

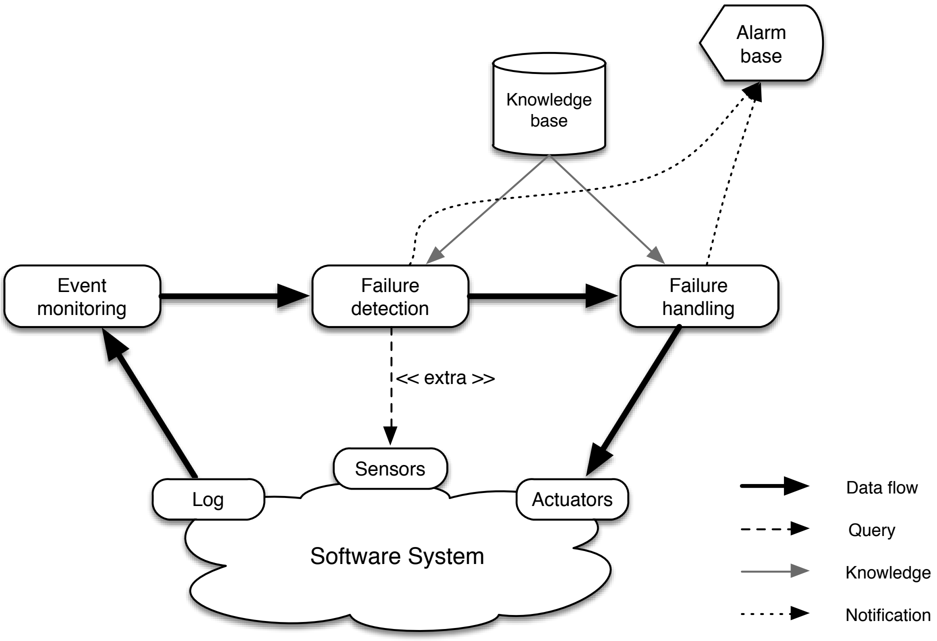


Figure 14 – Autonomic global cycle

The specific detection strategy receives in his constructor the instance name (*node* tag) that must be monitored; in each posterior cycle, the handler looks for the absence of the *keep-alive*[4] tag among the recent events from that component instance. As presented in the instrumentation, an event with the *keep-alive* tag is notified at the end of the main component's loop, in order to indicate its presence in the environment. When the tag cannot be found, the handler concludes that the instance has failed and must be restarted; the recovery routine is then called and receives as parameters the failed node id, which is used to gather the last process's id (PID), and the launcher used in the last execution. Before the re-execution, the PID value is used to kill the zombie process (if there is any) and then re-executes the component instance with the received launcher. Observe that the recovery routine is generic, and that the information needed to address the occurrence (process's id and launcher) is gathered from the footprint, using the query engine. The aim in this evaluation is to demonstrate that recovery mechanisms can be developed using the proposed solution for information extraction (the log with meta-information). The recovery solution applied to this failure is actually a traditional restart, in spite of being triggered and fed by the annotated event flow.

An evolution of this recovery handler exists but has not yet been implemented. It consists on taking advantage of the possibility of changing the startup parameters. These parameters are provided by a launch file, — a single component may have more than one of them — thus, when restarting, the recovery handler may evaluate the failure and select a different one. For example, the current and preferred launch file makes the node instance produce a better result, but also makes it unstable in some situations. In this case, when restarting, the handler may choose a more stable version.

The necessary tags and events required by this handler are: (1) the tag *node* in each event; (2) an event positioned at the end of the component's main loop, with the *keep-alive* tag; (3) an event in the startup with the *launcher* name; and (4) an event in the startup with the *pid* value. This handler was assessed by running a group of component instances and (1) forcing them to be closed using the kill

[4] Keep-alive is a signal generated within an interval in order to indicate the emitter is available.

command, and (2) activating a purposely injected actuator that hangs the execution without closing the process.

7.3.3.2 Broken information source

There are nine equipment in this system that provide information through a serial connection, and three failures that may occur in this context: (1) absence of data due to a malfunctioning hardware; (2) corrupted data due to the communication environment; and (3) unexpected format due to misconfiguration. Moreover, in this system there are five available recovery solutions for these failures: (1) restarting the connection; (2) switching to a redundant data source (if available); (3) switching to a redundant connection for the same equipment; (4) restart the equipment (physically); and (5) restart a software service inside the equipment. To illustrate, there is the possibility of an extremely accurate GPS board — that is used to obtain the vehicle's position —, presents some malfunction. In such a case, one possible solution would be changing for the IMU's GPS information, a less accurate solution but enables the system to continue relying on the position to operate. A similar failure, however presenting a different architectural solution, is being capable of establishing the connection between both PCs and the microcontroller using serial or ethernet. In this case, instead of using another equipment, the redundancy targets only the communication, providing an alternative way for interacting with the same hardware — a component in the PC is responsible for acquiring the information published by the microcontroller and propagate it through ROS topics, thus it is the one having an actuator for changing the connection type. In this specific case, the Ethernet is preferable, however if it is unavailable, the recovery may change for the serial approach; but after a recovery, the handler must monitor the environment to change back for the Ethernet connection, when available again.

This is an interesting example of the proposed approach, since the failures among different equipment drivers are very similar, although not all exhibit the same characteristics. For example, the probe could suffer from a wrong protocol configuration but does not have a recovery routine based on redundancy; on the other hand, the temperature sensor has a redundant information source, but does not need (does not make sense in this case) to monitor for a wrong protocol configuration. Therefore, the composition of detection and recovery strategies provided by the framework enables the generation of different pairs for each equipment failure, thus enabling the reuse of strategies. This conclusion is plausible while the information used by detection and recovery strategies — provided by the annotated log — are the same for every component. Therefore, such conclusion relies on the log homogeneity between component implementations.

The solution for these failure handlers is: every driver component that receives data from an external connection must notify an event with a tag *received data* for each package received from the equipment. These may assume the values *success* or *error*, depending on the result of the data processing. When an error is placed, another tag *reason* must be present in the event in order to determine if it is a data corruption error (ex: checksum) or the inability to process the package, characterizing a wrong protocol. The event must also be enriched with the tag *data*, associated with the serialized format of the values in the received data (ex: sequence of bytes in hexadecimal). Therefore, three detection strategies must be developed:

- *DataAbsenceStrategy* – Look for any event with a *keep-alive* tag from the target node, in order to guarantee the instance is executing (if not, other detector will identify the failure), and then look for any event with the *received data* tag arrived from the monitored equipment in a given time window. If none was received, the failure occurrence is characterized.
- *CorruptedDataStrategy* – Count the number of events with the *reason* tag for the *corruption error* and divide it by the count of events with the tag *received data* in the same time window, then consider a failure if the result is above a given threshold, configured by per equipment.
- *WrongFormatStrategy* – The same approach for the previous one, however looking for *suspicious wrong format* instead of *corruption error*.

When a failure is identified by one of these approaches, it informs the name of the node and the name of the topic for the recovery strategy, both received as parameters in the initialization. Hence, considering the options available in this system for these failures, we chose these three recovery strategies to present:

- *ReconnectStrategy* - Restart the connection with the external device, requiring an actuator implemented on the target component to proceed with the recovery in its process space. In this system, the actuators were implemented as a ROS service, which consists on a callback function and a statement in the main function to register the service.
- *ChangePublisherStrategy* - Activate the information source redundancy. Requires an actuator implemented on each component involved, which receives a boolean parameter indicating if it must turn on or off the ROS topic that publishes the information, and the name of the topic. Observe that in this strategy we are relying on the ROS infrastructure to transparently change the information source for the client's components, since this infra-structure enables publishing data from two sources in the same topic. The recovery strategy is initialized with the name of the topic and a list of node names that are capable of publishing the given topic; activate the topic of the first node in the list, deactivate the others; and when a failure is detected the recovery routine receives the current node name, finds it in the list and selects the next one as the new source of information (deactivate the topic in the current node and activate it in the newer). Moreover, the handler reconfigures the associated *detection strategy* indicating the new node that must be monitored (changing the node name used by the *verify* method).
- *ChangeInterfaceStrategy* - Activate the connection redundancy by executing an actuator implemented in the target component. This actuator must be implemented in a way that internally changes the information source according to a received parameter — in this case, ethernet or serial —, which can be implemented, for example, using a *strategy* design pattern. This handler strategy is used together with the *ChangeBackToEthStrategy*, which is a recovery strategy installed in the same *failure handler,* placed after the

ChangeInterfaceStrategy instance. Therefore, after the first handler changes the interface to the *serial* connection, this second handler verifies the ethernet connectivity (through a *ping* command) to the target equipment, and when succeeds executes the actuator again in order to change the connection back to ethernet, which is preferable.

The required tags for this solution are *received data*, *reason*, *topic*, *keep-alive*, and *node*. The usage of these last tags was not described, since they follow the same solution as the one explained in the last subsection.

This solution was assessed by creating nine failure handlers for four components that uses serial connections (Probe, IMU, GPS and the microcontroller's proxy), with different compositions of detection and recovery strategies — among these presented in this section. The composition of each failure handler was defined according to the characteristics of the target component and its possible failures. Therefore, for each failure of each component, an implementation was made to trigger a failure occurrence based on a random variable, generated from a configured probability. The solution was first tested with mocks (used in the development), and after that assessed in the actual driver component.

As explained in the previous failure case, the recovery strategies described are based on well-known fault tolerance solutions. The goal, however, is to demonstrate that the proposed solution for runtime information extraction is flexible enough to be used as a basis for the Hydra framework, allowing us to evaluate inconsistent or suspicious states outside the component's process space and, in this case, illustrating the potential of the handler's *strategy composition* provided by the framework, which demonstrated a high-level of reuse in this study.

7.3.3.3 Faulty third-party library

This section describes a very specific failure occurred due to a faulty library provided by the equipment supplier, which is an unstable version since the supplier does not provide support for Linux. The equipment is a stereo camera that takes two photos simultaneously from narrow lenses and computes for each pixel (X, Y) the relative distance (Z) to the camera, thus generating a depth map. Before deciding to cope with this failure, we attempted to adapt an open source component for stereo processing to this specific camera. The result, however, was far from what we wanted when compared with the supplier's proprietary algorithm. Therefore, coping with the failure was the only solution to continue with the system development until the supplier releases a stable Linux version — what is not yet scheduled to occur.

The library was a beta version, and in addition to a very rough interface, it hangs after several calls for stereo processing. Moreover, there was a serious issue related to using it in the same process space of a ROS node: due a boost library (Dawes et al., 2007) version incompatibility, that cannot be linked to the component. The immediate *ad-hoc* solution was to implement a single application with this library and create a D-BUS (inter-process communication system) service in this application, in order to use it from the ROS component — named *DisparityNode* — as a proxy. Observe that the D-BUS service with the disparity algorithm, named *DisparityEngine*, is executed in a separated process, using an inter-process communication mechanism to communicate with the client — i.e.: the ROS node.

The solution worked well. However, in addition to handling the camera's library failures, we also had to handle more complexity and possible failures from the D-BUS connection — and, thus, also the re-configuration of the proxy object when it restarts after a failure occurrence. Observe that this algorithm is guided by a set of properties intrinsic to image processing (ex: stereo mask, edge mask, min disparity, max disparity, surface validation difference, etc.), which may change independently along the execution. Hence, the *ad-hoc* solution handled the failure but polluted the logic of the component by adding a considerable amount of code to detect and handle errors. When the hydra handler was implemented, this issue was minimized, since most of the detection and recovery code was transferred to the handler. The following example demonstrates that a complex handler can be implemented with the proposed solution, avoiding blending most of the detection and recovery code with the component's functional code, and using a generic approach, since is based on the log flow with meta-information as information source.

The failure was addressed by a failure handler in the global cycle for creating and monitoring the stereo processing sub-process. The monitoring was based on a strategy similar to that described in 7.3.3.1 (inoperative component).

However, instead of monitoring a component, the detector strategy was modified to detect a D-BUS service and to reconfigure it after a restart. Figure 15 illustrates this solution. The detection strategy was implemented by looking for the absence of a *keep-alive* tag from the *DisparityEngine* in a given time window, and by looking for an *error* tag in the *DisparityNode* instance, which could also represent a *DisparityEngine* failure. Hence, whenever an occurrence is found, the recovery strategy is called, receiving the *process id* of the D-BUS service — also gathered from the footprint —, and then proceeding to kill the previous instance and launch a newer one. Moreover, it queries the log for the last value configured for each property, and then sets this configuration through actuators in the sub-process, restoring the same state as when the failure occurred. When the instance reaches stability (meaning *keep-alive* tags are detected), another actuator is called in the ROS component to restart the D-BUS connection with the sub-process. After that, the component stays reliable for a while, until a new failure occurs.

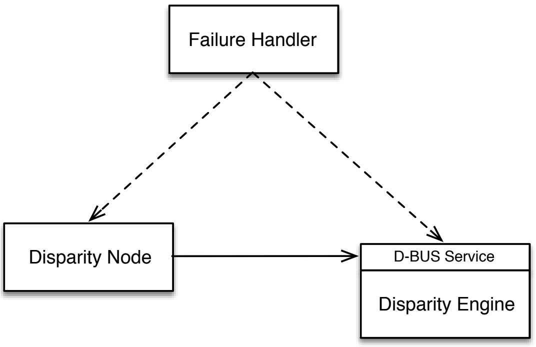


Figure 15 – Relation between all entities involved in the Disparity Failure Handler.

Therefore, the requirements for implementing this handler are: (1) an event with the tags *node*, *keep-alive*, and *pid* emitted in an internal cycle of the sub-process; (2) an event with the tags *[action:set_property], name*, and *value* for each property modification; (3) an actuator for changing the property in the D-BUS process; and (4) an actuator for reconnecting the ROS component with the new D-BUS service.

This handler was assessed by simply running the camera's component, since the failure occurs after a while due the faulty library. The failure handler behaved as expected, and the failure occurrences were almost imperceptible for the operator. This example demonstrated that a complex handler can be implemented with the proposed approach, avoiding blending the detection and recovery code with the component's functional code. In brief, this handler is capable of monitoring two component nodes for failures, and when an occurrence is detected, it restarts the *DisparityEngine*, configures it with the last valid state and after it reaches stability the other node is stimulated to re-establish connection with the new service (provided by the new node). Observe that the proposed mechanism enables developing specific detection and recovering solutions using a generic approach, which is based on the log flow with meta-information as information source.

7.4 Subsea Equipment Monitoring (SEM)

This is an embedded system developed for monitoring equipment installed in deep water, such as oil pipelines. The system's architecture is composed by Arduino microcontrollers and a satellite radio for external communication. The software that runs on the microcontrollers was written in the C++ language. However, during development, a prototype was coded in Python instead of C++ and executed on an x86 PC instead of a microcontroller, thus providing an environment that requires less effort to assess the system's specific solutions and provides mechanisms to simulate most hardware interfaces.

This type of software is extremely difficult to develop, since it requires physical hardware equipment to test and, sometimes, specific environment conditions, as described hereafter. Although software simulators have been used for a long time while testing, some critical situations can only be evaluated in a real environment and must be subjected to field tests. Field tests usually require a preparation phase and, once started, are restricted by limited resources or time. Moreover, during the test the developer usually has more concerns than when developing in a controlled environment, since it is harder to concentrate in this type of environment while debugging. Therefore, these system's field tests are interesting scenarios to evaluate the diagnosis solution proposed in this thesis. When a failure occurs in a field test, the developer does not have the necessary environment to debug the occurrence (for example, a debugger and the possibility to redeploy in order to rerun some tests), which makes the extracted information of paramount importance for further analysis. Two versions of the SEM system were evaluated: the prototype and the final version (embedded software).

As mentioned, a prototype was developed in order to evaluate the monitoring solution and to allow us to learn from the eventual problems, involving equipment limitations and third-party software failures. The main objective of the prototype was to simulate the software solution with the external equipment, in order to learn failures quicker and before implementing the final version of the solution for the microcontroller. For example, the satellite radio only works when it has a completely clear line of sight to the sky, imposing the field test to be outdoors in an open space. In this case, however, the development environment is impaired by battery limitations (for laptops and other equipment such, as the microcontrollers and the radio) and by other factors, such as clarity in laptop screen and presence of mosquitoes on the base. Hence, when a software expected to be correct presents some malfunction, such as an impossibility to transmit messages to a web server using the satellite, the developer must gather all the information he can about the environment for later inspection of the failure. Nevertheless, the execution log for this type of system is huge, including many abstraction levels — such as higher-level actions to report the equipment state, measure sensors, transfer a sample, etc.; and low-level operations such as the serial protocol implementation, the satellite message construction, the real time clock (RTC) configuration, etc. Therefore, these system's characteristics are interesting for evaluating the diagnosis tool proposed in this thesis.

7.4.1 Instrumentation

The SEM software system was developed by a team of two members: one software engineer with expertise in mission-critical system development, and an experienced developer. The instrumentation was written by both members during system development, using the following instrumentation policy:

Rules for notification

- Every decision edge of a function.
- Every code-block planned as an action (ex: sample sensors, build message, compute the sleep time, etc.).
- Read/write from the SD (the data disk).

- Read/write from the EEPROM (the non-volatile memory).
- Read from analog port.
- Result from an algorithm or complex math formula.
- State transition of a digital port.
- Access to the real time clock (RTC).
- Data sent to another device, indicating the communication method (UART, SPI, IC2, ETH, etc.).
- When entering or leaving the low consumption battery mode.
- When entering and leaving an alarm state, indicating the type (critical measurement or low battery).
- When entering and leaving the warning state, with suspicious of malfunction, indicating the type (SD, communication, EEPROM and radio).

Used tags

- src_line – source-code line.
- src_file – source-code filename.
- error – error description.
- memory – available space in the volatile memory.
- component – name of the component (CPU1, CPU2, Radio and Display).
- action – high level operation being executed.
- init – scope of the microcontroller's initialization.
- loop – scope of the microcontroller's main loop.
- data – data sent/received in a communication operation.
- interrupt – scope of an interrupt callback. The value is the port number.
- comm_type – type of communication (UART, SPI, IC2, ETH, etc).
- ip_addr – IP address (ex: 192.168.0.1).
- eeprom - address from a memory block.
- data – data sent/received by a communication; or data written/read from a SD operation.
- sensor – type of the sensor.
- measure – measurement of the sensor.
- port – port of the microcontroller.

- state – state of the related microcontroller port.
- rtc – any operation related to the external real time clock.
- alarm – type of alarm (critical measurement or low battery).
- warning – type of the suspicious malfunction (SD, comunicação, EPROM and radio).

As described in Chapter 3, some reuse patterns were applied to avoid rewriting the frequent tags. The most relevant patterns are:

- Every component's main function pushes the tag *component*.
- The setup code receives the scope tag *init*.
- The main loop code receives the scope tag *loop*.
- The instrumentation library automatically appends the tags in every notification: *src_line, src_file*, and *memory*.
- The instrumentation library automatically appends the tag *stacktrace* when the tags *error* or *exception* are present on the received tag set for notification.
- Auxiliary functions are used to encapsulate the access to hardware, such as pins (ports), communication, SD, and EEPROM. Following this pattern, we could generate a message to inform each operation, test the result, and automatically generate a notification in case of failure.

We encountered three difficulties when instrumenting this system. The first one was memory limitation: when using Arduino Uno the execution frequently reached the maximum memory available (1Kb) and rendered the logging approach unfeasible. When the hardware was changed to Arduino Mega (8Kb), the problem was solved. The second difficulty was the clock availability: this problem is intrinsic to this type of system, since the usual real time clock (RTC) used in PCs, which maintain the date and time even when the machine is turned off, is an expensive feature for a microcontroller. In this project, the CPU1 had one of these, but the CPU2 did not, preventing the instrumentation library from generating a timestamp for each event notification. The problem was solved by notifying events without timestamps, and inserting them on the remote server. Since events are sent immediately notified by each CPU, and the timestamps are generated from the same origin (the server), events from both CPUs become normalized in a single timeline with a maximum error proportional to the transmission delay, which is acceptable for our purposes. Another solution would be using a Network Time Protocol (NTP) to synchronize the clock every time the microcontroller was awoken from sleep, but it would require an NTP server and access to a network — this, however, was the third difficulty we encountered. As with the RTC, the network access for an Arduino is made through an expansion card. Observe that every new device connected requires more power, thus increases the battery consumption. In addition, microcontrollers' ports are limited, and each one of these expansion cards requires some of them to work properly. The network expansion card was used in this project only for debugging purposes during development, in order to make the Lynx usage feasible. The final version was shipped without it, since having ethernet capability makes no sense in the deployed version, as the system does not communicate with the external world after deployed. Observe that these three difficulties are related to hardware specifications, not the technique itself. The application was measured as explained in Section 7.1.1, with the results presented in the tables below.

	Source lines (KLOC)	**Event notification (%)**	**Tag manipulation (%)**	**Total (%)**
C/C++ (Arduino)	2.3	12.13	6.74	18.87
Python (PC prototype)	2.7	18.16	5.03	23.19

Table 10 – SEM's implementation effort.

	Computing Overhead (%)
C/C++ (Arduino)	9.3
Python (PC prototype)	0.21

Table 11 – SEM's computing performance overhead.

	Number of events	Data Overhead (%)
SEM	12.054	660.47

Table 12 – SEM's storage performance overhead.

These results will be discussed further ahead, in Section 7.6.

7.4.2 The Diagnosis Tool Evaluation

A study was conducted with three failures observed while using the system in distinct field test, used to evaluate the solution in a real world environment. The first two were observed in the radio's component prototype written in Python, and the third one in the final software system. The following subsections will describe the selected failures, informing the failure observation and the correct diagnosis, before presenting and discussing the results. These scenarios will demonstrate the generality of the solution to diagnose failures related to communication and resource allocation.

7.4.2.1 Escaping error

The failure observation was due to one of the messages transmitted to the satellite's modem failing to register, blaming a checksum error. In the field test, we observed that the problem was related to the dataset being transmitted, since every transmission of the exact same package produced the same result, while other messages had success in registering. However, during the field test no hypothesis was conceived regarding why the failure only occurred with that payload. The correct diagnosis was that during implementation, two of the five escape sequences had had their placements switched, causing some (very few) messages to send a payload that could not be parsed by the satellite's modem, thus generating the error. However, in order to diagnose this failure, the developer needed to review, step-by-step, the process of message transmission, studying the data buffer transformation. In a traditional log, there is a conflict between exposing this data in order to support diagnosis and avoiding it, to make the event log more legible. Before the study, the expectation was the Lynx tool to be able to filter the relevant events for the footprint, which was confirmed, thus solving this conflict.

7.4.2.2 Endianness error

Endianness[5] describes how multi-byte data is represented by a computer system and is dictated by the CPU architecture of the system (Blanc & Maaraoui, 2005). The available options are: big endian, which stores the most significant byte of a word in the smallest address, and little endian, which stores the least significant byte of a word in the smallest address.

The following failure is similar to the previous one, and is also related to the message transmission protocol. The observation was that some messages were arriving with less bytes than expected, without any obvious pattern that could be used to create the first hypothesis. The correct diagnosis was that the length field in the message was written in little endian, when it should have been in big endian. Since we do not have access to the modem's embedded logic, we assumed that it sends a number of bytes according to length field, even when more bytes are available in the data field (and it is fault tolerant to send all the bytes in the data field when the value of the length field is higher than the data field's count).

This failure's diagnosis depends heavily on the knowledge and experience of the developer, since after reaching the footprint it is necessary to carefully analyze the data being transmitted. Without comparing the buffer data with the protocol's specification, the developer would never find the root cause. However, our expectation was that the Lynx tool would quickly make the operator reach the footprint that represents the failure, which occurred for all participants during the assessment.

7.4.2.3 Limited memory

The failure observation was an unexpected behavior in the embedded software, which executed an impossible path in the program's logic: a return statement being ignored. Much time was spent following hypotheses that did not contribute to understanding the unexpected behavior, such as changing the microcontroller assuming hardware malfunction, looking for knowledge bases for a past occurrence, and distrusting the compiler's reliability. The actual cause was a dynamic array allocation with a required length that overpassed the memory limit. After that, the software made some invalid access to memory, writing in

[5] The convention used to interpret bytes in memory generating a data word.

addresses without permission, which resulted in the observed misbehavior. The memory allocation operation is usually followed by a verification code, which confirms success. In this case, however, the developer forgot to write it.

This is a very simple failure, but without a proper tool, it becomes quite hard to diagnose. The expectation is that with the Lynx, the operator will quickly identify the root cause, since the available memory is a tag notified in every event.

7.4.2.4 The Diagnosis Assessment

The study was conducted with three developers. One of them had actually participated in the development of the system, and the other two had sufficient knowledge about the system to diagnose the failure.

	F1	F2	F3
Traditional approach	≈ 40	≈ 60	≈ 240
Developer 1	5	14	2
Participant 1	10	8	4
Participant 2	9	11	4

Table 13 – Time in minutes to diagnose each failure in the SEM study.

As in the previous studies with the Lynx tool, all failures were diagnosed in less time than the original occurrence. F1 (escaping error) was quickly identified by developer 1, who started with the assumption that the checksum was correct, since the radio accepted the command. Thus, he only had to verify the log after its calculation. The other two participants reviewed the entire process of message building. As expected, the Lynx tool aided in removing unnecessary events, but F1 (escaping error) and F2 (endianness error) required external knowledge and mathematical calculations in order to verify each step of the process. The F3 failure (memory limit) had its root cause exhibited in the first query made on the tool – the diagnostic could be defined in less than a minute. Nevertheless, all collaborators needed some time to conclude the problem was related to memory allocation. As previously mentioned, the visual pollution of unnecessary tags hampers the identification of the obvious characteristics that aid in determining the failure’s root cause, or even of those that would help creating a newer hypothesis to advance the diagnosis.

7.5 Team and Equipment Management System (TEMS)

The TEMS system was developed as a modular platform that provides mechanisms to implement support applications for specific processes in an organization. For example, one of these modules is the Key-Performance-Indicators (KPI), designed to reduce the effort in managing indicators in an oil & gas company that needs to process daily reports for a large set of ships. The system was implemented as an ordinary web application that provides a user interface though desktop browsers, and was deployed in a cloud service with access to an SQL database. The web application was implemented in Python and uses the Django framework.

The main non-functional requirements in this system are security and traceability, in order to determine past interactions in case of an auditing. Hence, the instrumentation policy is heavily based on determining what occurred, when it occurred, who did it, with whose permission, from which machine, etc. The proposed logging approach enabled a different design perspective to accomplish these implementations. Our solution does not attempt to provide any additional benefit when compared to other solutions for auditing purposes. The goal here is to demonstrate that the proposed solution can be applied to systems with different levels of complexity. Moreover, this deployed system did not present any failure that was worth evaluating, thus this system will only be used for instrumentation measurements.

7.5.1 Instrumentation

The TEMS system was developed by a team of four members: one senior developer, two novice developers, and one web-designer with limited coding proficiency. The instrumentation was written during system development by two of these four members, using the following instrumentation policy:

Rules for notification

- Starting and ending a function.
- Every decision edge of a view (function that renders a webpage).
- Authentication mechanism.
- Create, remove, update, and delete (CRUD) database entries, indicating the type of operation.

- Explicit database query (when not using a framework for building SQL or NoSQL queries).
- Sending emails.

Used tags

- src_line – source-code line.
- src_file – source-code filename.
- error – error description.
- warning – suspicious malfunction.
- exception – placed in the catch block, without value.
- stacktrace – the execution stacktrace.
- environment – the execution environment (ex: mobile).
- thread – identifier for the current thread.
- module – indicates the high-level system module.
- action – high level operation being executed.
- collection – database collection that is being manipulated.
- database – name of the database that is being used for the operation.
- host – computer hostname that made the access.
- ip - IP address of the user that made the request.
- is_admin – indicates if the user is an administrator of the system.
- is_manager – indicates if the user is a manager.
- user – identifier of the user that made the request.
- request_id – identifier of the request.
- report_id – identifier of the current report.
- progress – current value of the user's progress (0-100).

Observe that this instrumentation policy is extremely similar to the one used in the WinePad system, as it is a reuse of it, with few adaptations. As described in Chapter 3, some reuse patterns were applied to avoid rewriting the frequent tags. The most relevant patterns are:

- The application startup pushed the tag *environment*.
- The instrumentation library automatically appends the tags in every notification: *timestamp*, *thread*, *src_line*, and *src_file*.

- The instrumentation library automatically appends the tag *stacktrace* when the tags *error* or *exception* are present on the received tag set for notification.
- An interceptor in the webserver interface to automatically identify information from the request and append it to the tag stack. This implementation pushes the following tags: *host*, *ip*, *is_admin*, *is_manager*, *user*, *request_id*, *device_id*, and *organization*.
- The *action_tag* mechanism that uses the function name as the tag value, implemented in Python with decorator.
- A listener for the database intercepts the CRUD operation and notifies events with the operation type and data.

This system's instrumentation did not pose any specific difficulty. The applications were measured as explained in Section 7.1.1, with the results seen in the tables below.

	Source lines (KLOC)	**Event notification (%)**	**Tag manipulation (%)**	**Total (%)**
Server app	10.63	4.56	1.52	6.08

Table 14 – TEMS's implementation effort.

	Computing Overhead (%)
Python	10.04

Table 15 – TEMS's computing performance overhead.

	Number of events	**Data Overhead (%)**
TEMS	12.143.772	463.88

Table 16 – TEMS's storage performance overhead.

Hereafter, we shall discuss such results.

7.6 Discussion

7.6.1 The Instrumentation

All systems were instrumented with the proposed technique without much effort. An instrumentation library was developed for each of them, being similar among languages but with few minor differences in order to better suit the domain's needs. For example, the auxiliary decorators were developed based on the abstractions introduced by the infrastructure where the system is executed. As explained in Chapter 4, the library must be developed with focus on the domain of the systems it will serve. Furthermore, the overall instrumentation effort was low, as presented in the following table, which aggregates all measurements discussed in the previous sections.

	Source lines (KLOC)	Event notification (%)	Tag manipulation (%)	Total (%)
WinePad (Obj-C)	13	6.40	4.11	10.51
WinePad (Python)	16	7.43	1.14	8.57
EMR (C/C++)	13.7	6.24	4.05	10.29
EMR (Python)	15.1	6.96	7.23	14.20
SEM (C/C++)	2.3	12.13	6.74	18.87
SEM (Python)	2.7	18.16	5.03	23.19
TEMS	10.63	4.56	1.52	6.08

Table 17 – Overall implementation effort.

The total instrumentation effort had a median of 10%, with maximum and minimum, respectively, as 23% and 6%, when comparing projects with different sizes. From our point of view, the application of our instrumentation technique requires little effort. Moreover, observe that event notification statements already exist in traditional software development in the form of simple log notifications, and our technique extends it with the meta-information concept. Looking at the effort overhead of our solution, we have measured that tag manipulation effort (push and pop by any mechanism) have a median of 4%, with maximum and minimum, respectively, of 7% and 1%. We attribute this effort level to the reuse techniques applied for stacking general tags, described in Chapter 4 and illustrated in the previous sections. With this approach, the tag set that must be inserted in most events is defined in a common piece of code for the entire software. The following table presents an aggregation of the measurements made to evaluate the computing overhead on each system. These values were computed by dividing the time spent processing the instrumentation by the time spent executing functional code.

	Computing Overhead (%)
WinePad (python)	9.95
WinePad (Obj-c)	1.43
EMR (C++)	0.51
EMR (python)	0.84
SEM (C++)	9.3
SEM (Python)	0.21
TEMS	10.04

Table 18 – Overall computing overhead.

From the computing overhead results, we may conclude that the impact depends on the domain and language. Web applications (WinePad and TEMS) present higher impact, what could be attributed to the automatic log generators (ex: the one installed in the database callback). Even with this higher overhead, however, the user experience regarding the productive use of the system was not affected. The SEM instrumentation also indicated a higher computing overhead, which we anticipated since the notification procedure of its library executes high cost operations for a microcontroller, such as accessing the external SD disk and handling HTTP requests[6]. The table below presents the aggregation of the storage overhead measurements.

	Storage Overhead (%)
WinePad	647.17
EMR	615.51
SEM	660.47
TEMS	463.88

Table 19 – Overall storage overhead.

[6] There is a planned evolution that consists in using the Arduino's default serial port to transmit the log content, instead of using an ethernet board.

From these measurements, we conclude that the presented log technique required more than six times more storage space than when using the traditional log technique. Since we have selected for the evaluation a heterogeneous set of system domains, and the measurements from all systems resulted the same order of magnitude, we expect a similar result when applying the technique to other domains. However, this overhead depends on the number of tags inserted on the events, which is guided by the instrumentation policy of each system. Thus, as discussed in Chapter 3, an event discard policy is a theme for future work, which may provide rules for keeping only the relevant information in the long-time storage, discarding the rest after a critical period (defined from system to system). Another technical evolution might be achieved by applying a data compressing technique. However, we must highlight that during this work there was no attempt to solve this specific problem, since in modern hardware storage space is usually very cheap. It is important to stress that even for the most limited system in this evaluation — which is the SEM system using a SD card for storage —, the calculated space for storing the log lifetime of this system is less than 1GB.

7.6.2 The Diagnosis Tool Evaluation

All failure diagnoses made in this study were much quicker than the original diagnosis made by a developer without instrumentation. However, despite the times from the original diagnoses were spent entirely to generate the diagnostic, they could not be entirely dedicated to log analysis, since some hypothesis could not be evaluated using the traditional log, as it surely did not expose some required information. Hence, the times of the original occurrence include coding and redeploying tasks, employed for hypotheses instigation.

It is important to notice that, despite the simple description, failures from all studies are difficult to diagnose when employing traditional techniques: it is hard to elaborate relevant hypotheses only by observing that a failure occurred, and to investigate them in log sets with thousands of events, mostly unrelated to the desired footprint. Thus, the traditional log does not help much, due to all limitations described in Chapter 2. The log enrichment and diagnosis techniques described in chapters 4 and 5 contributed by initially providing properties about the failed context that aided in the elaboration of hypotheses. These, in addition, were closer to the right one, effectively leading to the determination of the root cause. However, one of the first suppositions of the instrumentation policy was that critical components should have a smaller granularity. This was proven wrong by the failures in the WinePad, since some failures related to critical components required information about non-critical components, initially considered less relevant for diagnosing and given less attention in the policy. For example, in order to determine the root cause of the data inconsistency failure (described in Section 7.2.2.4), observed in the synchronization service (critical component), the instrumentation must expose events from edit operations located in the user interface, which is considered a non-critical component. The conclusion from this result is that the policy cannot be generated based only on assumptions and similar system's experiences. It must start with these assumptions, but must evolve during the system's lifetime as a live document.

We have observed that the visual pollution was one of the main difficulties while using the tool. In fact, in many scenarios the many unrelated tags obscured what otherwise would be the obvious diagnostic, which just required looking to a key-property in an event that explained the failure's root cause. The tool provides a feature to hide tags by type, which solves the problem, but the operator must remember to do it in every query result: before analyzing, wipe it by hiding irrelevant tags.

Another conclusion is that the diagnosis tool helps in investigating all hypotheses when the log contains the necessary tags to support the filtering. However, external (human) information is needed to identify the root cause of the failure. Automatic diagnosis was not a goal of this research. Nevertheless, during the study we have observed that, while diagnosing some failures, the collaborators managed to quickly locate the root cause, but needed some extra-time to fully understand it. Thus, the Lynx tool presents an efficient mechanism to investigate hypotheses, which can be further evolved by implementing semi-automated diagnosis techniques. Therefore, since the log annotated with meta-information better supports the diagnosis tools, semi-automated techniques may present better results when using this instrumentation technique, avoiding the issues discussed in Chapter 2. This, however, is a topic for future work.

Last but not least, we have observed that the Lynx tool's efficiency is not limited to finding the footprint that describes the root cause, which is produced by the correct hypothesis. The tool also aids in investigating wrong hypotheses, thus reducing the time spent looking into event sets that eventually lead to dead ends. During our studies, some of the collaborators explored the same wrong hypotheses investigated in the original occurrence, but left them much quicker, as the richer information on events avoided try-and-error attempts, as discussed before.

Furthermore, while testing the system using a debugger tool in a controlled environment, it is common to establish a breakpoint configuration to meet recurrent scenarios, which is similar to the perspective of interest of this approach. Hence, we expect that this diagnosis solution may also contribute for testing larger systems, as a complementary tool in the debugging toolset, since a breakpoint with a complex expression might be transformed into a query based on tags. However, this is a topic for future work, which needs to be further studied and experimented in order to verify its feasibility.

7.6.3 The Failure Handling Mechanism

The proposed solution for writing failure handlers was successfully applied to all failure types used in our research. As discussed before, this solution does not attempt to introduce a new fault tolerance approach. Its goal is to provide mechanisms that enable the creation of independent entities capable of detecting and recovering known failures, in a way that avoids polluting the target's functional logic. This goal was achieved by using the event flow with meta-information as a generic source of information for developing failure handlers that use the query engine to inspect the log history in order to identify suspicious activities and behavioral inconsistencies that represent a failure footprint, then using the footprint-specific data to feed the recovery routine. As expected, many of the recovery routines developed during this evaluation were inspired on usual fault tolerance solutions, such as Randell (1978), Ammann & Knight (1988), Johnson & Zwaenepoel (1990), Kazinov & Mostafa (2009), for example.

While using the hydra solution in the EMR production's environment we have observed some issues in the system's startup and shutdown operations. For example, if the hydra component is launched before others, it won't detect any instance (as there is no event with keep-alive tag), thus will conclude that all have failed, and then proceed with a restart recovery strategy for all of them, however, in a scenario that all are still initializing. Therefore, as a solution, the hydra component instance is launched by the deployer tool after all components have reached the stability (an event with keep-alive was notified). The same approach is done in the shutdown operation: the hydra instance if the first to be closed.

Another observation is the high degree of reuse achieved in the EMR system, in which failure handlers frequently consisted of common detection and recovery strategies already developed for previous failures. This result also demonstrates that, in addition to modularization, the solution may be applied with efficiency. However, since we evaluated the solution only in the WinePad and EMR systems, further investigation is necessary in order to verify this result for other domains. Moreover, we expect that the co-evolution of the EMR system will present more complex situations to apply the mechanism and identify further evolutions. There are two planned evolutions:

- Nesting failure handlers in a tree structure in order to define precedence while detecting failures. For example, the connection failure verification is executed only if the node's stability (keep-alive monitoring) was already confirmed.
- Nesting recovery strategies in a tree in order to represent alternatives handling paths.

Furthermore, there is the risk of considerably impacting the system's performance, what would prevent the solution from being installed on deployed systems. Some measurements were made in each system in order to demonstrate that the failure handler's impact on performance is actually low. The measurement in WinePad was similar to the one made for the logging instrumentation — compute the time for functional and instrumentation code using a profiler mechanism. For this measurement there were installed both failure handlers presented in Section 7.2.3.1. In EMR, since each component is executed in a different process — and the Hydra global cycle is also a component —, the measurement was made comparing the CPU allocation for each process of a running system with 9 component instances (including the Hydra component), and 19 failure handlers active (all with verification frequency configured as 500ms). Table 20 presents these results.

	Performance Overhead (%)
WinePad	0.31
EMR	0.52

Table 20 – Overall performance overhead for the failure handlers.

The overhead is actually low, however, as discussed in the Chapter 6, it may vary based on some variables: the system functional implementation, the number of failure descriptors installed, the number of tags in the tag stack, the number of tags in the vulnerable scope of each handler, etc. From our observation, in the EMR system, for example, the component instances have an impact an order of magnitude greater than the corresponding failure handlers.

Another confirmation from our case studies was that modules developed in low-level languages require much more effort to apply the solution, since they do not provide appropriate mechanisms to implement the handler in the most transparent form. When available, mechanisms based on language introspection and reflection were used to implement the solution in order to minimize the effort of in (1) instrumentation, (2) failure handling implementation and (3) failure handling installation. This last item is important, since when a failure is discovered during production use, we expect to install the recovery handler by deploying a single source file, which is integrated to the system by a class loader in the Hydra mechanism — thus without modifying the system's source code.

7.7 Concluding Remarks

In this chapter, we have evaluated the solutions proposed in this thesis. The assessments consisted on:

- Measure the effort to apply the instrumentation technique, and its impact on the system's performance.
- Verify that failures can be diagnosed more efficiently by the proposed diagnosis technique than the traditional approach.
- Assess that failure handlers can be developed without polluting the code or impacting on the system performance.

The solutions were applied to four different systems, in order to verify their validity in real world scenarios. The expectations met were: (1) applying the solution requires little effort; (2) it imposes low impact on the system's behavior;

(3) the diagnosis technique effectively aids in diagnosing failures; and (4) the proposed self-healing mechanism can assist in writing failure handlers for expected faults, in most situations without requiring changes in the faulty code. An identified weakness of the solution is its reliance on the quality of the instrumentation policy. In fact, if the instrumentation policy fails to cover a given aspect, or if the developer ignores it while developing, both diagnosis and recovery solutions will be impaired. These issues did not occur in our evaluation, but it is fair to expect them in organizations that are learning the technique. Since learning depends on the characteristics of the target system, this difficulty can possibly be attenuated using the technique and improving the instrumentation policy already during development and test phases. Therefore, the instrumentation design is a theme for future research, and must be further evolved in order to ensure the solution's effectiveness.

We also observed some limitations in components that present resource limitations, such as those implemented in microcontrollers. During the evaluation, the SEM system was evolved to use an ethernet board solely for notifying events. This is a valid solution for a lab or field test, but in a production environment, this requirement can produce a huge impact on the project, rendering the solution unfeasible due to the high costs imposed by the ethernet board and to the fact that it requires a battery with higher specifications. There is also the memory limitation problem, due to the space required for the tag stack in the latent memory. This was also not an issue during our evaluation, but it might become a critical issue in microcontrollers with less memory or with programs that handle a larger amount of data. These issues related to limited hardware will be a theme for future work.

The next chapter will present the state-of-art related to this thesis and discuss how the proposed technique is situated. In short, the hybrid instrumentation technique proposed is a strong candidate for solving the problems presented and discussed in Chapter 2, related to runtime information extraction. The proposed diagnosis technique has the capability of selecting events based on a perspective of interest — which is only feasible due to the novel instrumentation technique already mentioned —, but it does not implement semi-automated analysis, which is known to reduce the operator's effort while diagnosing. We consider these techniques complementary, and expect they may also be improved by our instrumentation technique. Finally, the proposed failure handling solution is a self-healing approach, which does not implement all guidelines of the Autonomic Computing (Murch, 2004) concept, in spite of being inspired by it. The goal was not to provide a new approach for fault tolerance, but a mechanism that enables known fault tolerance solutions to be applied with modularization and, when possible, reuse.

8
Comparing with Related Work

This chapter will present the state-of-art of the research in field of *Dynamic Analysis* (Cornelissen et al., 2009), and then discuss how it is usually combined with recovery mechanisms. The dynamic analysis approach consists on evaluating a program while executing it in order to detect failures, instead of examining the system's source-code looking for faults (known as the *Static Analysis* approach). Hence, the dynamic analysis's main objective is to aid in the comprehension of the system's behavior.

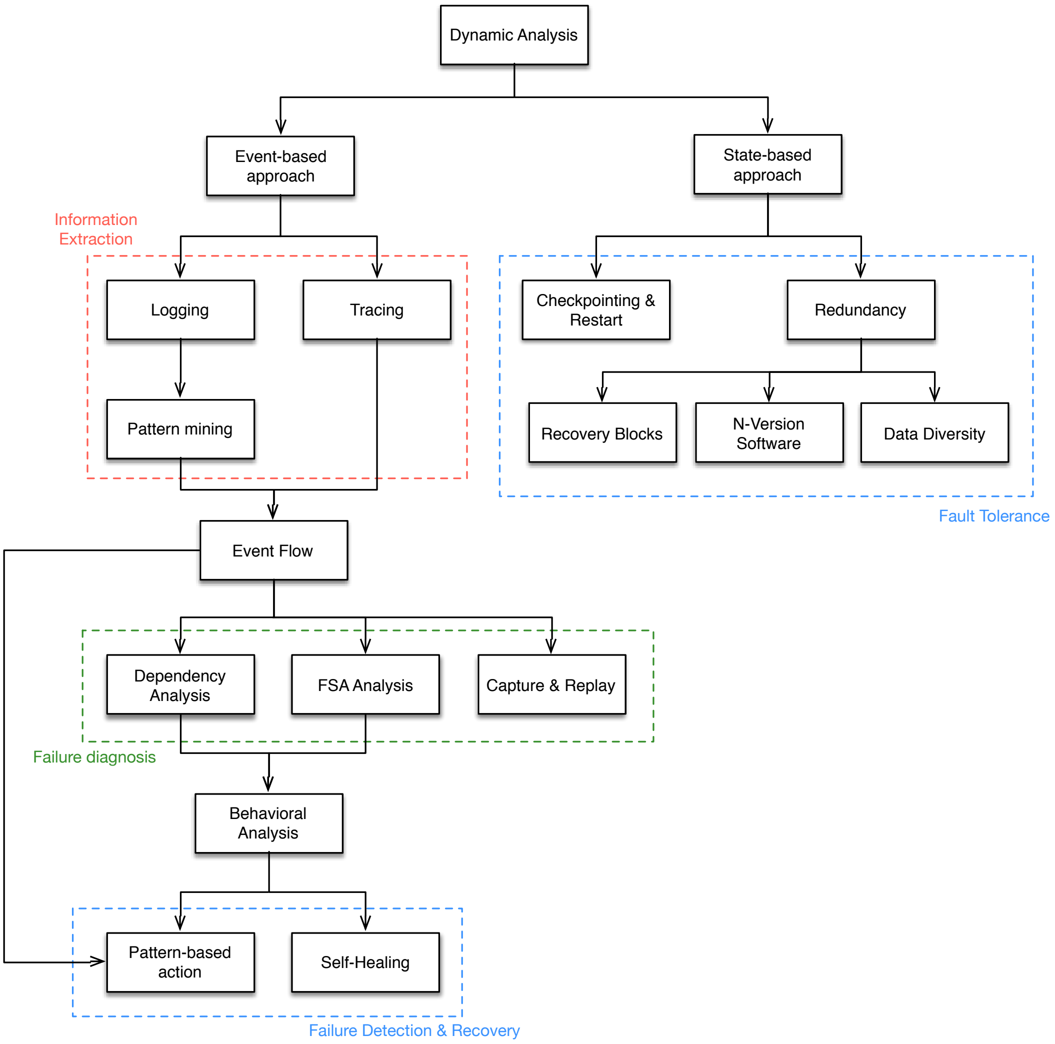


Figure 16 – Relationship between solutions in the field of dynamic analysis.

The following sections will discuss the fields of information extraction, failure diagnosis, fault tolerance, and failure detection and recovery. The relationship between techniques and their respective fields of research is represented in Figure 16.

8.1 Event-based approaches

This section will present solutions based on approaches that focus on studying system executions, in order to understand the behavior of the software and causality relationships.

8.1.1 The Most Simple Solution: Verbosity

A common mechanism to reduce the set of log events to be studied is the verbosity level (Microsoft, 2013), which enables maintainers to select the level of detail in which they desire to view past system executions: if the current level is not sufficient to explain the root cause of the failure, the detail level is raised and the set of visible events grows an order of magnitude. This type of log is created using mechanisms that enable developers to define in which level the notified event should be stored. Although this approach contributes to granularity control, it does not solve the problem of selecting only the relevant events needed to diagnose the failure, mainly because some of the key events may be at a higher level of detail that is shadowed in the log. We believe it is necessary to gather, right away, as much information about the system's execution as possible, and apply a later a filtering to distinguish which events are relevant for a given diagnosis session. This would be especially the case when trying to detect and diagnose failures without having to replicate the conditions that lead to the error (Skwire et al., 2009), something usually impossible to perform regarding multi-programmed or distributed systems. However, to support this approach, each logged event must contain more information about the context that originated it. Also, this information must be stored in an adequate way for the filtering mechanism.

8.1.2 Log Patterns, Data Mining, and Clustering

There are many works based on data mining and clustering (Manilla, 1997; Ma, 2000; Pei, 2000; Srivastava 2000; Hellerstein, 2002; Vaarandi, 2003; Vaarandi, 2004; Makanju, 2009), which are generated in a free-format, without a defined protocol. Data clustering is a data mining technique to sort events into groups called clusters, consisting of events that are similar to each other and different from events in other groups. For example, an event "The interface ETH2 failed to reply IP 192.168.2.34" would be defined in a pattern "The interface {0} failed to reply IP {1}", where the numbers represent the variable part — i.e., the context properties — from which the specific data is retrieved. Some works try to infer the property names based on empirical knowledge (Lou et al., 2010), with approaches to evaluate the text fragments nearest to the values. Hence, this approach provides execution information that helps maintainers understand the system behavior during a diagnosis session.

Arguably, this technique provides some result without impacting the development process, since it uses raw text logs as input. However, efficacy is compromised due to three major problems: (1) property names and values may have been written in different formats, making the association very difficult or even impossible. Consider, for example, these two forms of writing an IP address: "ip", "device_addr". They won't be matched. This situation can happen in an environment where events were notified from components implemented by different teams; (2) the second problem is the information in events being insufficient, as cited before. Depending on the failure under analysis, different properties from the notification scope will be needed. During development time, developers insert just the information that is relevant from that scope, discarding others that could help in some diagnosis sessions; and (3) related to problem 2, properties of the outer context will not be present in the event, making it more difficult to relate events of the same flow or to understand in which circumstances that routine was executed (for example, from which service a common asset was called). The first problem can be attenuated by implementing a rule policy that renames properties based on an incremental knowledge base. The second and third problems, however, will only be solved by changing the way events are notified.

Another output of some mining algorithms (Ma, 2000; Hellerstein et al., 2002; Vaarandi, 2002) is the pattern frequency, which helps maintainers identify performance issues when compared to failure occurrences. For instance, consider that a given routine works well for 50 executions per second, but, when it is called more than a hundred times a minute, failures starts to occur. This approach is complementary to our solution, and can be adapted in future evolutions. From the pattern frequency we can derive the periodicity, which suggests the system's normal behavior and the outliers, formed by a group of infrequent events that may or may not represent a failure occurrence and should be investigated. Despite the benefits of this output, it does not aid in locating root faults, since the root cause of the unexpected behavior may be in the events with normal periodicity. If the infrequent pattern exposes sufficient information, some of it could be used to guide the maintainer when searching for related events, but context properties are often not expressed in the event (Oliner & Stearley, 2007).

There are works that evolved this technique by post-processing patterns in order to mine dependencies between them (Hellerstein et al., 2002; Zheng et al., 2002; Zhang et al., 2009; Lou et al., 2010). A dependency is a causal relationship between events in different components of the system. This approach would help maintainers identify relations that are hard to detect without proper tools, and then diagnose the root cause of a failure. For example, this approach allows following backwards a user's request along different system services, starting from an error event produced by a generic common library. Zheng et al. (2002) works over logs composed by alarms (warnings, in the terminology adopted by this thesis), and uses the causality approach to merge related alarms into one containing all the information. This may describe the root cause of the failure and the fault location, if the necessary information is present in the final alarm.

Another approach is presented in (Xu et al., 2009), which detects patterns based on the source code instead of data mining techniques. This technique extracts the abstract syntax tree (AST) from the source-code and identifies all statements that produce events. These statements represent the set of patterns that will be used to match the events during the execution. A good side effect of this approach is the log structure being precisely defined due to the known location of the pattern in the AST. As per previous related work, the result of this approach could be further enhanced with contextual properties associated in each event. A weakness of this technique is the need for an AST, which requires a set of parsers for each supported language, limiting the tool's implementation.

There are works in failure prediction based on logs (Fulp et al., 2008; Zehng et al., 2009; Lo et al., 2009; Lou e al., 2010), which focus in failure detection by using the extracted patterns and its correlations to identify anomalous behaviors. Most of these works detect known failures, provided by a knowledge base created by developers and maintainers. Lo et al. (2009) presents a novel technique capable of detecting unknown behavior through the analysis of log histories and past failures, which are used as training data for a pattern classifier. Lo et al.'s (2009) approach uses the common execution tree to identify forgotten or out of order calls, which may represent hard-to-detect failures. Lou et al. (2010) proposes a different approach: searching for system invariants in the log history to define the set of patterns that correspond to normal behavior, and then evaluating the log flow using these patterns. This approach can help detecting problems such as wrong API usage, and is complementary to our approach. There are some works that use rule-based tools to associate patterns by describing failures with recovery routines which are executed when one of those patterns is matched in the log flow (Vaarandi, 2002; Prewett, 2003). Due to the fact that these approaches use unstructured logs, they present a high-level of false-positives (Lo et al., 2009), even with all the effort made in developing techniques and algorithms to enhance their accuracy.

Most of the works described in this sub-section target superficial failures, related to network, performance and security layers of the system. Since the information extracted from logs is limited to a few properties, its contribution to context comprehension is less than adequate, which renders these techniques not suitable for diagnosing and handling internal logic failures. Our solution follows the same trend of dynamic analysis (Cornelissen et al., 2009), by inspecting system execution through system logs, however using an enhanced instrumentation capable of extracting context properties and keeping some software abstractions. Related to the rule-based actions pattern, our solution carries the system properties from the diagnosis to the detection and recovery steps. Also, its results can reach the maximum level of accuracy due to the precision and completeness of the software context present in annotated log events, which depends on the instrumentation policy defined for the system.

8.1.3 Tracing as a solution for full state extraction

Software tracing is a technique to automatically extract runtime information based on a unified criterion. It is very similar to software logging, but the frequency of event notifications is much higher and each event is composed of much more information about the execution scope. Since this information is automatically inserted, the application of such technique demands no extra effort from the developer. The granularity and content of the event notifications depend on the insertion criterion, which may vary depending on the approach. The most common criterions, in this case, are instruction-level (Maebe et al., 2002), function-level (Mirgorodskiy et al., 2005), and remote interfaces (Hendrickson et al., 2003), registering each function call and its parameters. The higher the frequency, the greater is the impact on performance.

Trace notifications can be inserted either statically (Srivastava & Eustace, 1994; Romer et al., 1997) or dynamically (Maebe et al., 2002; Mirgorodskiy et al., 2005). The static approach can operate on source-code level (Lindlan et al., 2000) or byte-code (Eggers et al., 1990), transforming the software to be traced into an instrumented version. This approach presents a considerable overhead, which may vary depending on the granularity policy and on the volume of information inserted in the notified events. Previous works measured this overhead as being between 29% and 178% (Luk et al., 2005; Mirgorodskiy et al., 2005; Horwitz et al., 2010), considering the normal execution (discarding warm-up and inactive times), which makes this approach unfeasible for deployed systems. The dynamic approach solves this problem by inserting the instrumentation only when needed by replacing instructions (Buck & Hollingsworth, 2000; Cantrill et al., 2004), using just-in-time compiler features (Maebe et al., 2002; Bruening, 2004; Luk et al., 2005), or operational system mechanisms (Toupin, 2011). This approach enables the operator to activate the event notification only during a diagnosis session, thus reducing the impact on performance. There is also a warm-up overhead in the dynamic approach technique, which is reduced in Mirgorodskiy et al.'s (2005) work by inserting instrumentation only in the execution path that is relevant for the diagnosis session, which is similar to our vulnerable scope concept.

Undoubtedly, the trace approach produces good results, however it is usually applied for performance debugging. It is less suitable for debugging logic-specific faults, since the volume of extracted information is huge, exposing auxiliary variables and complex objects that will not present a readable description. Thus, most part of the data will be irrelevant for the operator, reducing the visibility of the relevant properties and its relations. Also, traces do not express software abstractions, since automatic approaches do not incorporate the human knowledge in the process, as discussed in Chapter 2. On the other hand, the output of a trace can be much more structured than a common log, since the properties can be indexed by name instead of blended into text messages, suggesting the technique for automated analyses.

The way the static approach impacts performance prevents the technique from be applied to a system in production. The dynamic approach, however, also does not effectively solve the problem of diagnosing faults, since no data will be saved in the first occurrence and, even if the failure is observed, maintainers may not know how to reproduce it. There are lightweight approaches designed for distributed systems (Hendrickson et al., 2003; Reynolds et al., 2006), which gather only the message calls between nodes, thus reducing the performance impact and storage requirements. These approaches, however, are prevented from diagnosing failures involving nodes' internal logic. Therefore, tracing is not suitable for system lifetime monitoring, i.e. continuous information extraction for diagnosis and automatic detection and recovery.

The approach presented in this thesis is a hybrid solution between log and traces, which enables developers to expose software abstractions in an way that part of the information is inserted automatically, thus reducing the effort in the task; and also achieves richer contextual information, since design and architecture abstractions are inserted due to the corresponding information being written by the developers that created or learned them. Also, the solution has a low impact on performance, which makes it feasible for continuous monitoring. Moreover, the tag-stack technique solves the problem of accessing properties in unreachable software state (due to encapsulation) by inserting tags of an outer scope into events notified in an inner scope, without breaking encapsulation or requiring great effort from the developer. Even the tracing technique that has access to all properties during runtime is unable to select the few properties that should be promoted to scoped tags, thus inserting them in the notifications of that scope. Human reasoning is required to make these decisions, however, current development techniques and tools do not express in the software code all abstractions created during development.

8.1.4 Tracing for failure diagnosis

There are works that use traces to automatically determine the failure's root cause (Chen et al., 2002; Barham et al., 2004; Yuan et al., 2006; Mirgorodskiy et al., 2006). With few differences between these works, the basic idea is to apply clustering and statistical analysis techniques over the runtime information extracted, in order to determine anomalous footprints based on past executions. The main limitation of this approach is the performance overhead imposed by the tracing technique. When it is coarse-grained, as in Chen et al.'s (2002) work, the solution is feasible for a production environment. However, since it monitors only the component's interface level, internal information about the component's execution is not gathered, thus preventing diagnosis of internal failures. The opposite case, using a fine-grained policy, as in Mirgorodskiy et al.'s (2006) work, makes internal failure diagnosis possible, but the impact on performance prevents the trace from being active during the full execution flow, in order to evaluate each request and detect anomalous ones. A second issue with this approach is the high rate of false positives, reported by Chen et al. (2002) as being of 40-50% of detected anomalies. Although this approach is also complementary to our work, we believe that this technique alone will not be able to precisely diagnose complex failures — even though its result, combined with human knowledge about the domain, the system, and previous experiences, could yield better results than a manual diagnosis. Moreover, the implementation of this automatic diagnosis technique, combined with our instrumentation approach, may produce a more effective result, since by using the tag concept system abstractions are better represented in the events, thus contributing to identify the failure signature, hence reducing the cognitive effort needed to understand the fault.

8.1.5 State machine inference for diagnosis support

There are works that aid in failure diagnosis by exposing the state machine of each node in the system (Lorenzoli et al., 2008; Tan et al., 2008). Lorenzoli et al. (2008) present the GK-tail algorithm that enables the extraction of Extended Finite State Machines (EFSM), which annotate each edge with property values that distinguishes it from all other edges originated from a same vertex. Tan et al. (2008) presents an FSM extraction approach that also annotates the execution time duration between vertexes to enable performance analysis. Tan et al. (2008) and Fu et al. (2009) apply data mining over traditional logs, in order to extract system properties present in the log messages. The main problem of this approach is that it relies on the information provided by the traditional log, which is usually insufficient to understand the execution context. However, the tracing option is not feasible for a production environment. Therefore, FSM extraction technique is complementary to our work since the hybrid instrumentation would improve the precision of the FSA due to the richer information used to infer it, without imposing a considerable performance overhead.

There are also works that use the FSM extraction technique to automatically detect anomalous behavior using the normal system execution as a specification (Mariani & Pastore, 2008; Fu et al., 2009). This technique works as follows: the system is executed under supervision of an operator, generating training data that outputs an FSA without failure edges (or it is expected to be). This resulting model is used as the correct FSA of the system, and after that every edge that does not match an existing edge is considered anomalous and becomes a candidate for inspection. Despite its great contribution for failure diagnosis, this technique presents a high rate of false-positives due to edges of the FSA not being exercised during the training. There is also the opposite case: failures being exercised during training without being revealed for the operator, which may cause faults to go unnoticed.

The evolution of these works produced AVA (Babenko et al., 2009; Pastore & Mariani, 2013), a tool based on FSA analysis to annotated anomalous behaviors, with tips that can lead to the cause of the failure. The goal of this tool is to reduce the operator’s effort in diagnosing. However, it focuses on behavioral aspects of the execution — such as presence, absence, anticipation, swapping, or replacement of events —, which is a great advance, and will identify failures related to event ordering and temporal dependencies. It will not, on the other hand, be able to explain failures triggered by invalid data or a complete new flow executed (set of edges not present in the FSA). Finally, there is also an evolution of these techniques that present the BCT tool (Mariani et al., 2011) and enhances diagnosis focusing on integration faults.

8.1.6 Visualization tools to assist manual diagnosis

Some works (Takada & Koide, 2002; Stearley, 2004; Bodik et al., 2005; Tan et al., 2008) invest in visualization tools to assist manual inspection. These studies aim at solving the visual pollution problem of long log files by condensing the events and generating statistical graphs. This solution gives good results when detecting and diagnosing network and security faults, but tends to be inadequate for diagnosing the application's logic, which requires detailed information about the state of execution. Our approach follows the opposite direction, seeking mechanisms to increase log details rather than simplifying it. We solved the visual pollution problem using filters that follow the maintainer's perspective of interest, showing only events related to the target failure.

8.1.7 Capture & replay

There are failures that are very hard to diagnose since their causes are usually complex, involving process timing and hardware low-level operations. Due to these characteristics, said failures cannot be easily reproduced. The attempt to reproduce concurrence faults, observed in systems with distributed behavior, using the same parameters and system state of the original execution, may not trigger the failure. This is due to the fact that such failures depend on perturbations produced by intermittent factors that lead the system, even if for a few moments, to an inconsistent global state which is sufficient to produce errors that crash the system. Therefore, these failures are diagnosed using the trace captured of the failed execution, and maintainers depend on the information in the events of this trace to determine the cause of the failure. However, it is difficult to identify temporal behaviors looking directly at the trace flow instead of at the global scenario.

There is a diagnosis approach based on tracing, called Capture & Replay (Wittie, 1988; Dunlap et al., 2008), which attempts to, faithfully and deterministically, re-execute a failed routine with the exact temporal characteristics of the original execution. This approach enables developers to study the scenario of a failed execution by using mechanisms to inspect the global state at each point of interest. The limitation of this approach is that it relies on the tracing technique, which imposes a considerable overhead for a production environment. Since the objective is to study low-level details of the execution, traces with the full system's state must be stored, occupying considerable disk space. However, this approach is appropriate for debugging and testing (Steven, 2000), and is also applicable for virtual machine migration (Liu, 2007), as long as system accepts the overhead during the migration period. Another limitation of this approach is the interaction with external services: modern systems usually use remote third-party services to execute many tasks, which would not be reproduced under the replay capabilities. This is called limited consistency, and is tolerated by some approaches (Geels et al., 2007).

8.1.8 Self-healing

Murch (2004) proposed the Autonomic Computing concept, which suggests that all system components must be aware of their actions and of changes in the environment, in order to reason about them and act accordingly to accomplish the system's high-level goals. The autonomic computing concept is divided in four facets, which are: self-management, self-configuration, self-protection, and self-healing. This last is related to the field of failure detection and recovery, which is the subject of this thesis.

This concept's objective is to allow complex systems to adapt their behaviors to unpredictable events, choosing a path that will be the most effective solution for the system's main goals. This complexity is solved by a divide to conquer approach, where each component is complemented with a local autonomic cycle to reason about its internal behavior and the external events that occur in the system. The system itself has a global autonomic cycle to gather information from all component cycles, which defines the high-level goals that are passed back to the component cycles as feedback. The Rainbow implementation (Garlan et al., 2004) proposes an architectural solution for self-adaptive systems, based on a framework that formalizes the autonomic cycle execution flow. According to it, sensors in the system provide runtime execution information, which are mapped onto the system model; some constraints are then evaluated, in pursuit of violations, which are addressed by the adaptation engine that may define which and how effectors must be called in order to bring the system to the most effective path. Sensors and effectors must be provided by the component, while the rest must be implemented in the framework, as external knowledge provided by specialists.

This concept has influenced many works in the last decade, including this thesis. However, the full implementation of the autonomic approach directly impacts software design and requires high-level expertise from all developers involved, preventing the technique from being implemented in usual software development teams, since the developers available with the needed profile being scarce and an entire team would be unfeasible for the project budget. Furthermore, systems that rely on specific frameworks or are executed on limited hardware, such as microcontrollers, sometimes lack the ability to integrate or implement a self-adaptive solution, due to resource integration incompatibility, or even to the impact caused by the approach. These are some of the problems dealt with during the last decade. Some works, however — including this one —, propose a solution towards self-healing, but without complying with all its characteristics. For example, Carzaniga et al. (2008) proposes a mechanism to reorder the sequence of operations of a routine that contains a fault, hoping that it avoids the failure. Chang et al. (2009) describes how to create healing adaptors for components that will be used as third-party software by application developers. Chang's work is very similar to ours in the way that known-faults are handled by a workaround solution. Their users, however, are the developers of the third-party components trying to minimize the impact of integration faults, while our users are the application developers trying to work with third-party faulty components (or their own components that cannot be redeployed at the moment). Denaro et al. (2009) presents an approach similar to Chang et al.'s work, by providing a self-healing layer, focused in service interoperability, which tests each new component that would like to interact with the service, by executing some test suits in order to identify interface misuses. When a misuse is detected, an adaptor is defined to handle the interactions with that specific component. Observe that these three relevant and recent works are in the same path as ours, which supports that the knowledge present in the developers' high-level abstractions is indispensable to effectively deal with logic faults.

8.2 State-based approaches

The following sections will present works in the field of fault tolerance, which aims at finding workaround for faults by providing rollback mechanisms and redundancy.

8.2.1 Checkpointing and Restart

Checkpointing technique (Johnson & Zwaenepoel, 1990; Sankaran et al., 2005; Hursey et al., 2007; Kazinov & Mostafa, 2009) is usually applied to distributed systems, and consists on saving the system's state (the checkpoint) in moments where it is believed to be consistent. When a failure is detected by any mechanism (exception, log pattern, assertion, etc.), the part of the system that has been compromised is rolled back to the last stable checkpoint, from where the execution continues normally. This rollback operation loads the previous snapshot (saved state of the software), recreating a consistent state. When available, the messages received since the loaded checkpoint are re-transmitted, in order to bring the recovered node to a synchronized state with the rest of the system. Sankaran et al. (2005) and Hursey et al. (2007) focus the technique over a message passing interface (MPI) implementation, which is a standard for many parallel and distributed applications. Sankaran et al. (2005) also discusses other uses of checkpoint/restart, such as scheduling and process migration. With the scheduler feature, it becomes possible to define when and which nodes should be restarted, in order to reduce the impact in the system's performance and availability. The major limitation to this approach is that the failure consequences are not directly handled, such as an invalid result propagated to another system or even to a physical device (ex: airplane controller).

8.2.2 Redundancy

This field is divided into execution and data redundancy. Execution redundancy is addressed by software replication, which can be designed through Recovery Blocks (Randell, 1978) and N-version programs (Avisienis, 1995). Both approaches are based on providing more than one strategy to achieve the same result. These strategies are implemented in critical areas of the code, considering that different programmers would not fail in the same code area — an assumption that has been proven wrong (Knight and Levenson, 1986; Holloway, 1997).

The recovery blocks technique is similar to checkpointing and restart, previously discussed, but applied inside a node execution. Before a critical path, the process saves its state and, if a failure occurs, the state is rolled back to the last checkpoint and an alternative implementation is executed. The cycle is repeated until the routine is successfully completed or the number of redundant implementations ends. In addition to the problem of developers tending to insert similar faults (Knight and Levenson, 1986), the rollback does not assure that the global state is consistent, since many statements in the failed execution may have changed other component's and the external service's state.

N-version software technique is a different approach, which applies an oracle at the end of the critical routine to evaluate the result and choose the most reliable one. This oracle is formed by an algorithm that votes on each result; the most voted one wins and is passed through the execution flow. There are still doubts about how to write these algorithms that will vote for the correct answer and how reliable they are. Moreover, a better question is how to precisely define areas of code that must be implemented with redundancy, since naive parts can influence complex parts to fail.

A third approach is data diversity, which is applicable to software that process some input in order to achieve an output. The technique is similar to the N-version approach, but instead of re-executing the software with another implementation, it is re-executed with another input — a logically equivalent that preserves the data semantics. There are three main formulas to generate this input: (1) changing the input data to an equivalent format, (2) changing the input data to a format that can be reversed in the output data, and (3) the decomposition of the input data and re-composition of the output.

A common problem to all approaches is how to ensure that the execution finishes without inserting a dormant failure, which will only be exercised in a different part of the software. Also, all of them present a drawback that makes their application unfeasible: implement multiple versions of the same code pieces require a much larger team, usually not affordable by small software companies.

8.2.3 Closure

A novel instrumentation technique presented in this thesis is a hybrid approach between logging and tracing. It enables developers to write events manually and to enrich them with some design abstractions, as in logging, all while requiring less effort due to automatically gathered contextual properties, as in tracing. These properties are defined by the developer and saved in a tag-stack, which is a mechanism that solves the modularity problem for accessing outer scopes and reduces the effort of describing each context. This approach produces much less data than tracing, thus imposing a lower overhead. Also, the extracted information presents a richer set of properties, since it was originally informed by developers using the system's specific abstractions. It is undeniable that the information relevance will be directly dependent on the instrumentation policy, which must be defined in a way that expresses the system's design and architecture. The end result relies, therefore, on the developers' devotion to this concern during the coding phase.

Moreover, the set of contextual properties in events are presented in a structured format, which is appropriate for further analysis. Observe that this characteristic is only made possible from the instrumentation approach, by using the tag concept. Hence, the event is structured, not only with local contextual properties, as in tracing, but also with many other properties, from higher contexts, that may reflect some architectural and design abstractions. The proposed technique follows an inverse approach when compared to dependence mining techniques, which try to gather the set of properties only after the execution, thus producing an imprecise, incomplete, and error prone result. The proposed technique may raise the effectiveness of behavioral analysis techniques, such as FSA extraction, by providing a richer input. Furthermore, these behavioral analysis techniques are complementary to our approach, and in future work we shall study how they can be adapted to our event model and evolved to take advantage of our approach.

The benefits of the recovery approach are a consequence of the event structure we just described. The recovery mechanism follows the action-pattern approach, by enabling developers and maintainers to associate recovery procedures to known failure signatures, which, due to the way the instrumentation is inserted, can be defined using system properties from the event database. The approach excels similar techniques by providing mechanisms that work directly over the abstractions created by developers and by allowing them to write temporal detectors for known-faults in a free form, taking advantage of the greater flexibility provided by the structured log format. Observe that the properties that may have defined the root cause can be directly used to create this detector, and that later some contextual properties can be passed for the recovery routine to guide them during the failure handling.

It is important to observe that this thesis does not attempt to detect unknown failures, as in Mariani & Pastore's (2008) work. The main goal is to support failure diagnosis between software components, considering their internal logic, and to provide mechanisms to implement handlers that can minimize future failures consequences. These automatic failure detection techniques can contribute by giving warnings of system malfunctions, which may represent failure occurrences, and by providing tips for determining the root cause of these failures. We do not expect them, however, to produce a final diagnosis, since the system's runtime information does not contain all the necessary information.

Fault redundancy mechanisms are not discarded in our work, but their role is limited to building blocks of recovery handlers. Mission-critical projects that can afford the cost of implementing these techniques may use this thesis' approach to bind them into a single solution, as discussed in the evaluation (Chapter 7). The proposed approach allows the technique's sophistication to be adjusted to the needs of the software under development: the simplest system would use just an action-based pattern for single events to alert about the occurrence of a failure, while mission-critical projects may detect complex behaviors and proceed with an available redundancy mechanism.

9
Conclusions and Future Work

This work addressed the problem of coping with failures observed while using deployed software. As discussed in the introduction, failures must always be expected while using a deployed version of a system, even if it was developed using strict quality control. Since software development is still a human labor intensive activity, the process is heavily influenced by whom is executing the development tasks and is obviously susceptible to mistakes. Even if we manage to develop a perfect software code, modern development relies on third-party assets, which may contain faults. Therefore, we must assume that software will fail and be prepared to, when it happens, put it back to work as quickly as possible.

Ideally, the fault should always be removed and the system redeployed. In order to do this, we must first diagnose the observed failure by understanding the failed execution — thus identifying the root cause —, then produce the modification that demonstrably removes correctly and completely the fault before finally redeploying. However, while these operations are being executed, the deployed system is in use and susceptible to new failure occurrences due to the very same fault. Sometimes, however, the consequences are critical, and it becomes necessary to find and apply solutions that avoid disasters or at least minimize said consequences. This goal may be accomplished by using either a preventive approach — such as fault tolerance techniques — or a reactive one — such as failure handling techniques — or, still, a combination of both, as described in the evaluation chapter.

In order to detect and diagnose failures, one needs information about the execution. Analyzing works from the last decade, we have identified that little attention was given to this subject, despite the large amount of work dedicated to log and tracing processing with the purpose of automated or semi-automated detection and diagnosis. As discussed in Chapter 2, both techniques present opposite benefits and limitations: logs are appropriate to expose high-level information with low impact on performance, but with few data items per event, and exposes such data in a non-indexable format, since the event notification and its content are usually written by the developer in a string format; on the other hand traces overcome these limitations by automatically generating events with indexable properties, but usually has a significant impact on system performance and additionally produces large volumes of data with little use, hence polluting the log set.

In this thesis, we have addressed the problem of runtime information extraction in order to aid failure diagnosis with a double purpose: precisely identifying and removing the causing fault; and determining how to correctly recover the execution in a very short time providing richer information to assist the development of failure handlers, which are responsible for detecting and recovering the system from known-failures. The main goals of this research were:

(1) To develop a technique to extract runtime information with an acceptable impact on performance, allowing use in deployed systems operating in a production environment;

(2) To use instrumentation restricted to traditional methods, tools, and language paradigms;

(3) To use a portable concept between languages and domains, thus imposing little effort to adapt to a new project class;

(4) To avoid influencing design decisions of the target system; and

(5) To provide appropriate information to develop diagnosing tools and failure-handling mechanisms.

The solution presented in Chapter 3 is a novel log concept based on events annotated with meta-information, which represent contextual properties about the execution and expose some abstractions that normally would not be transcribed into source code. These properties are extracted in an indexable format, and are thus appropriate for further analysis. The log is implemented following an instrumentation policy, for which there are guidelines, described in Chapter 3. The log is generated with support of an instrumentation library (described in Chapter 4), which provides the traditional methods to notify events, and methods to insert and remove contextual properties on a stack. The tag-stack is another novel concept developed during this thesis.

The solution proposed here for extracting runtime information about possible failures from the log was used as a source to develop a diagnosis tool –

Lynx –, described in Chapter 5. This tool introduces an inspection technique: several operators can restrict the event set to be displayed based on a given perspective of interest, built from the contextual information on the failure report, and evaluated by a query engine based on the meta-information found in events recorded in the log.

Another product developed using the extracted information is a failure handling mechanism – Hydra –, described in Chapter 6, which is capable of automatically detecting and recovering the system from known-failures. Due to the proposed log concept, this failure handling mechanism enables developers to implement detection strategies decoupled from the functional code, thus avoiding polluting the software's pure logic, as usually occurs when using *ad-hoc* solutions. This solution also presents a modular characteristic, allowing the handler to be installed or removed with little effort.

Therefore, the main contributions of this thesis are:

- The concept of a log annotated with contextual information to support tools and mechanisms for failure diagnosis, detection and recovery.
- The instrumentation technique that solves the abhorrence and break of encapsulation problems using the tag stack concept.
- The inspection tool capable of exhibiting events based on a given perspective of interest, thus reducing the effort in studying the system's execution, therefore the failure diagnosis.
- The failure handling mechanism capable of representing the maintainer's knowledge in a modular approach, thus avoiding corrupting the functional logic.

The solution was developed and assessed within a small software company. Four systems from different domains were chosen, in order to study the range of the proposals and their limitations. The evaluation consisted of applying the instrumentation technique to each system, observing how this was done, measuring its impact on the system's overall performance, and measuring the implementation effort. As discussed in Chapter 7, the impact on performance varies from domain to domain, but none of the systems presented significant performance decay. Moreover, the instrumentation effort was similar among all systems, and imposed a small overhead when compared to the traditional logging technique.

Both the inspection technique, aimed at diagnosis, and the failure handling mechanism, aimed at detection and recovery, were tested on actual failures, occurred in deployed versions of these systems, thus making the assessment more reliable. The overall result of the inspection technique was effective, since all collaborators have diagnosed the submitted failures in less time than the original occurrence. The result of the failure handling assessment confirmed that the proposed solution allows a developer to write short and simple modules that can be installed on the system with little effort, in order to detect and recover from a given failure occurrence, and then be removed, when no longer necessary. Limitations and ideas for future work were identified, and will be discussed below.

The main threat observed in our evaluation is the instrumentation not addressing a given concern, making it obviously unavailable for the inspection tool and the failure handler mechanism, possibly making the approach useless. Hence, learning how to further develop the instrumentation guidelines to assure that the necessary data remains available in the log in case of a failure will be the subject of future research. An assumption is that this issue can possibly be attenuated using the technique and improving the instrumentation policy already during development and test phases.

Another threat is the log volume: as our measurements have shown, using instrumentation policies similar to those described in Chapter 7 brings data overhead to around 600% of the traditional log size. Even though none of the systems (from different domains) used in the evaluation was impaired by this overhead, if limited storage space is available, the technique will be unfeasible. Therefore, it is important to further improve the instrumentation guidelines, in order to ensure the writing of complete and precise instrumentation that minimizes the amount of useless data in the log.

Two possible alternative uses for the instrumentation technique proposed in this work have been identified. They arose after the development teams were comfortable with the solution and confident that it would not require more effort than the traditional solution, but escaped the scope of this thesis and, therefore, were not addressed here. The first such use would be diagnosing or verifying execution behavior in complex *test scenarios*, as a tool to help testers and developers creating better software. For example, when the software of the microcontrollers of the SEM system were being integrated, the Lynx tool helped us understand the distributed behavior and usual failures much quicker than it would take with the traditional log output. The second possible use of the instrumentation would be to study the final-user behavior analyzing the event flow, filtered by a perspective of interest that leaves only events generated by the user interface of the system. This use was tried on a mobile app — not among the systems evaluated in Chapter 7 — that was already deployed and had more than 10 thousand users at that moment, hence motivating the study of every human-computer interaction (HCI) decision made in the application. It was a satisfaction to see initiatives like this coming from the first developers who had contact with our solution.

A relevant work derived from this thesis was the Master's dissertation of Rocha (2014), who used contracts to analyze the distributed behavior that could be extracted from the event log generated by the instrumentation. This dissertation produced the following contributions: a language to write contracts for distributed behavior considering temporal relations and a mechanism capable of evaluating these contracts over the event flow. The key-characteristic of this dissertation's solution was the usage of event tags as variables in the contracts, thus enabling a developer to represent signatures through the abstractions identifiers he used in the system's instrumentation.

Finally, here are some suggestions for future works:

- **Research how the instrumentation design can be enhanced.** If it is not possible to generate precise instrumentation, then study how to improve the guidelines presented on Chapter 3.
- **Evolve the GK-tail algorithm (Lorenzoli et al., 2008) to use the annotated log as information source**, and then verify if the state machines extracted are produced with the maximum precision, avoiding the current rate of false positives.
- **Develop a correlation algorithm to generate usual and unusual paths** considering the data flow, in order to detect suspicious activities. After that, comparing the results with similar approaches,

in order to verify if the approach based on annotated logs is more effective.

- **Assess the discarding policy proposal throughout the lifetime of a set of systems.** Study this result in order to identify how the heuristics must be defined in a way that keeps relevant information for the diagnosis moment.
- **Assess the technique using other programming languages.** The assessment in this thesis addressed C++, Python and Objective-C languages. Instrumentation libraries must be developed for languages such as Java, C#, Lua, and Ruby, in order to identify if any of them exposes limitations for the technique.
- **Research how to implement mechanisms for the recovery handler that enable modifying the state of variables** that cannot be accessed from a global reference, such as those located on functions scopes (local variables).
- **Evolve the Lynx tool interface** in order to achieve a better user experience. During the inspection technique evaluation, and in daily use, we have observed recurrent activities that may be optimized by implementing specific features. For example, the possibility of viewing multiple results in the same window or enabling custom rendering for each tag, thus reducing the cognitive effort required to understand its information.
- **Research how Aspect-oriented programming can be used to reduce the effort in instrumenting the code.** Since aspects have been successfully used for logging, we expect that some recurrent tags that are hard to insert by a reuse technique may be scoped with less effort than they currently are.

10
Bibliographic References

AMAZON web-services. Disponível em: <http://aws.amazon.com/>. Acesso em: 20 jun 2014.

AMMANN, P. E.; KNIGHT, John C. Data diversity: an approach to software fault tolerance. **Computers, IEEE Transactions on**, v. 37, n. 4, p. 418-425, abr. 1988.

ANDREWS, J. H.; ZHANG, Y. Broad-Spectrum Studies of Log File Analysis. In: International Conference on Software Engineering, 22°, 2000, Nova Iorque. **Anais…**Nova Iorque: ACM, 2000. p. 105-114.

AVIZIENIS, A.; CHEN, L.; **On the Implementation of N-Version Programming for Software Fault Tolerance During Execution**. Proc. IEEE COMPSAC 77 Conf., p. 149-155, nov. 1977.

AVIZIENIS, A.; **The Methodology of N-Version Programming**, Chapter 2 of software fault tolerance, M. R. Lyu (ed.), Wiley, 23-46, 1995.

AVIZIENIS, A.; LAPRIE, J-C.; RANDELL, B.; LANDWEHR, C.; Basic Concepts and Taxonomy of Dependable and Secure Computing; **IEEE Transactions on Dependable and Secure Computing 1(1)**; Los Alamitos, CA: IEEE Computer Society; 2004; p. 11-33.

BABENKO, A.; MARIANI, L.; PASTORE, F. AVA: automated interpretation of dynamically detected anomalies. In: Eighteenth international symposium on Software testing and analysis, 18, 2009. Nova Iorque. **Anais…**Nova Iorque: ACM, 2009. p. 237-248.

BARHAM, P. et al. Using Magpie for Request Extraction and Workload Modelling. In: Symposium on Operating Systems Design and Implementation, 6, 2004. California. **Anais…**California: OSDI, 2004. p. 259-252.

BEA. **Interim report on the accident on 1 June 2009 to the Airbus A330-203 registered F-GZCP operated by Air France flight AF 447 Rio de Janeiro** – Paris. Paris: Bureau d'Enquêtes et d'Analyses pour la sécurité de l'aviation civile (BEA), 2009.

BLANC, B.; MAARAOUI, B. **Endianness or where is byte 0**. White Paper, 2005.

BODIK, P. et al. Combining visualization and statistical analysis to improve operator confidence and efficiency for failure detection and localization. In: IEEE

International Conference on Autonomic Computing, 2, 2005. Seattle. **Anais…**Seattle: IEEE Computer Society, 2005. p. 89-100.

BROWN, A. B.; PATTERSON, D. A. To err is human. First Workshop on Evaluating and Architecting System dependability, 1, 2001.California. **Anais…** California, 2001.

BROWN, A. B.; PATTERSON, D. A. et al. **Recovery Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies**. EECS Department, University of California, Berkeley. 2002.

BRUENING, D. L. **Efficient, transparent, and comprehensive runtime code manipulation**. Massachusetts: Massachusetts Institute of Technology, 2004.

BUCK, B.; HOLLINGSWORTH, J. K. An API for runtime code patching. **International Journal of High Performance Computing Applications,** v. 14, n.4, p. 317-329. 2000.

CABRAL, B.; MARQUES, P. Exception handling: a field study in java and .net. In: European conference on Object-Oriented Programming, 21, 2007. Berlin. **Anais…** Berlin: Springer-Verlag, 2007. p. 151–175.

CANTRILL, B.; SHAPIRO,M. W.; LEVENTHAL, A. H. **Dynamic Instrumentation of Production Systems**. USENIX Annual Technical Conference, General Track. 2004.

CARZANIGA, A.; GORLA, A; PEZZÈ, M. Healing Web applications through automatic workarounds. **International Journal on Software Tools for Technology Transfer**, v. 10 n.6, p. 493-502. 2008.

CAMPBELL, D.T.; STANLEY, J.C. **Experimental and quasi-experimental designs for research**. Chicago: Rand McNally College Pub. Co.1966.

CAZZOLA, W. SmartReflection: efficient introspection in Java. **Journal of Object Technology**, v. 3, n.11, p. 117-132. 2004.

CHANG, H.; MARIANI, L.; PEZZE, M. **In-field healing of integration problems with COTS components**. Software Engineering, 2009.

CHEN, M. et al. Failure diagnosis using decision trees. In: International Conference on Autonomic Computing, 2004. Washington. **Anais…** Washington, 2004. p. 36-43.

CHEN, M. Y. et al. Path-based faliure and evolution management. In: Symposium on Networked Systems Design and Implementation, 1, 2004, California. **Anais…** California: USENIX Association, 2004. p.23-23.

CHEN, X. et al. **Auto- mating network application dependency discovery:** experiences, limitations, and new solutions. USENIX , 2008.

CHIDAMBER, S. R.; KEMERER, C. F. A metrics suite for object oriented design. Software Engineering, **IEEE Transactions on,** v. 20, n.6 , p. 476-493. 1994.

COELHO, R. et al. Assessing the impact of aspects on exception flows: An exploratory study. In In: European conference on Object-Oriented Programming, 22, 2008. Berlin. **Anais…** Berlin: Springer-Verlag, 2008. p. 207–234.

CORNELISSEN, B. et al. A systematic survey of program comprehension through dynamic analysis. Software Engineering, **IEEE Transactions on**, v. 35, n. 5, p. 684 – 702. 2009.

COTRONEO, D. et al. Investigation of failure causes in workload-driven reliability testing. In International workshop on Software quality assurance: in conjunction with the 6th ESEC/FSE joint meeting, 4, 2007. 78-85. ACM: New York, NY, USA. DOI: 10.1145/1295074.1295089.

CREŢU-CIOCÂRLIE, G. F.; MIHAI, B.; GOLDSZMIDT, M. Hunting for problems with Artemis. In: USENIX conference on Analysis of system logs, 1, 2008. **Anais…** USENIX Association, 2008.

DAWES, B.; ABRAHAMS, D.; RIVERA, R**. Boost C++ libraries**. 2007. Disponível em: <http://www.boost.org/> Acesso em: 20 jun 2014.

DE PAUW, W.; HEISIG, S. Visual and algorithmic tooling for system trace analysis: a case study. **ACM SIGOPS Operating Systems Review**, v. 44, n.1, p. 97-102. 2010.

DEMILLO, R. A.; LIPTON, R. J.; SAYWARD, F. G.; **Hints on test data selection: Help for the practicing programmer**. IEEE Computer, 11(4), p. 34-41. abr 1978.

DENARO, G.; PEZZÈ, M.; TOSI, D. Ensuring interoperable service-oriented systems through engineered self-healing. In: Joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, 4, 2009. **Anais...** ACM, 2009.

DIEHL, S. **Software Visualization—Visualizing the Structure, Behaviour, and Evolution of Software**. Springer, 2007.

DUNLAP, G. W. et al. Execution replay of multiprocessor virtual machines. In: ACM SIGPLAN/SIGOPS international conference on Virtual execution environments, 4, 2008. Nova Iorque. **Anais…**Nova Iorque: ACM, 2008. p. 121-130.

ECKHARDT, D. E. et al. An experimental evaluation of software redundancy as a strategy for improving reliability. Software Engineering, **IEEE Transactions on,** v. 17, n.7, p. 692-702. 1991.

EGGERS, S. J. et al. Techniques for efficient inline tracing on a shared-memory multiprocessor. **ACM**, v. 18, n. 1. 1990.

FU, Q. et al. Execution Anomaly Detection in Distributed Systems through Unstructured Log Analysis. In: IEEE International Conference on Data Mining, 9, 2009. Washington . **Anais...** Washington : IEEE Computer Society, 2009. p. 159-168.

FULP, E. W.; FINK, G. A.; HAACK, J. N. Predicting Computer System Failures Using Support Vector Machines. **WASL**, v. 8, p. 5. 2008.

GARLAN, D. et al. Rainbow: architecture-based self-adaptation with reusable infrastructure. **Computer**, v. 37, n.10, p. 46-54. 2004.

GEELS, D. et al. Friday: global comprehension for distributed replay. In: USENIX conference on Networked systems design; implementation, 4, 2007. California. **Anais...**California: USENIX Association, 2007. p. 21.

GEELS, D. M. et al. **Replay debugging for distributed applications**. v. 68, n. 02. 2006.

GRADECKI, J. D.; LESIECKI, N. **Mastering AspectJ**: aspect-oiented programming in Java. Wiley. p. 456.

GRAY, J. **Why do computers stop and what can be done about it**? In: BÜROAUTOMATION, 1985. p. 128–145.

BUSINESS INTERNET GROUP. **The black friday report on web application integrity**. BIG-SF, 2004.

GÜLCÜ, C. **Short introduction to log4j**. 2002. Disponível em: <http://logging.apache.org/log4j/1.2/manual.html> Acesso em: 20 jul 2014.

HALL, A.; CHAPMAN, R. Correctness by construction: Developing a commercial secure system. **Software, IEEE**, v. 19, n. 1, p. 18-25. 2002.

HALL, A. Seven Myths of Formal Methods. **IEEE Softw**, v. 7, n. 5, p. 11-19. 1990.

HANSEN, S.E.; ATKINS, E. T. Automated System Monitoring and Notification With Swatch. **Anais...** 7th USENIX conference on System administration, 1993. p. 145-152.

HANSEN, J. P.; SIEWIOREK, D. P. Models for time coalescence in event logs. **Anais...**Fault-Tolerant Computing, 1992.

HENDRICKSON, S. A.; DASHOFY, E. M.; TAYLOR, R. N. **An approach for tracing and understanding asynchronous architectures.** Automated Software Engineering, 2003.

HELLERSTEIN, J. L.; MA, S.; PERNG, C. S. Discovering actionable patterns in event data. **IBM Syst. J.** , v. 41, p. 475-493. 2002.

HOLLOWAY, C. M. Why engineers should consider formal methods. Digital **Anais…**Avionics Systems Conference, 1997. 16th DASC., AIAA/IEEE. Vol. 1. IEEE, 1997.

HORWITZ, S.; LIBLIT, B.; POLISHCHUK, M. Better debugging via output tracing and callstack-sensitive slicing. Software Engineering, **IEEE Transactions on**, v.36, n.1, p. 7-19. 2010.

HURSEY, J. et al. **The design and implementation of checkpoint/restart process fault tolerance for Open MPI**. Parallel and Distributed Processing Symposium, 2007. IPDPS 2007.

IEEE. **IEEE Standard Classification for Software Anomalies**. IEEE Std 1044. 2009

JAIDEEP, S. et al. Web Usage Mining: Discovery and Applications of Usage Patterns from Web Data. **ACM SIGKDD Explorations**, v. 1, n. 2, p. 12-23. 2000.

JIAN, P. et al. Mining Access Patterns Efficiently from Web Logs. **Anais…** 4th Pacific-Asia Conference on Knowledge Discovery and Data Mining, 2000. 396-407

JIANG, W. et al. Understanding customer problem troubleshooting from storage system logs. **Anais…** USENIX FAST'09, 2009.

JOHNSON, D. B.; ZWAENEPOEL. Recovery in distributed systems using optimistic message logging and checkpointing. **Journal of algorithms,** v. 11, n.3, p. 462-491. 1990.

KANNAN, J. et al. Semi-Automated Discovery of Application Session Structure. **Anais…** 6th ACM conf. on Internet measurement, 2006. p. 119-132.

KAZINOV, T. H.; MOSTAFA, J. S. Software fault tolerance. **Computer science and engineering: proc. of the third intern. conf. of toung scientists**. Lviv, Ukraine. 2009. p. 17-20.

KNIGHT, J. C., LEVESON N. G. An experimental evaluation of the assumption of independence in multiversion programming. **Software Engineering, IEEE Transactions on**, v. 1, p. 96-109. 1986.

KRAMER, J.; HAZZAN, O.The role of abstraction in software engineering. **Anais…** 28th international conference on Software engineering. ACM, 2006.

LAMB, D. A. IDL: Sharing intermediate representations. **ACM Transactions on Programming Languages and Systems (TOPLAS),** v. 9, n.3, p. 297-318. 1987.

LANUBILE, F.; VISAGGIO, G. **Evaluating empirical models for the detection of high-risk components: Some lessons learned**. Proc. of the Twentieth Annual Software Engineering Workshop, Goddard Space Flight Center. 1995.

LANZA, M.; MARINESCU, R. **Object-Oriented Metrics in Practice** - Using software metrics to characterize, evaluate, and improve the design of object-oriented systems. Springer, 2008.

LI, J.; HUANG, G.; ZOU, J.; MEI, H. Failure analysis of open source j2ee application servers. Quality Software, 2007. **Anais…** QSIC '07. Seventh International Conference on (2007), p. 198–208.

LIBURD, S. D. **An N-version Electronic Voting System**. Diss. Massachusetts Institute of Technology, 2004.

LINDLAN, K. A. et al. A tool framework for static and dynamic analysis of object-oriented software with templates. **Anais…**Supercomputing, ACM/IEEE 2000 Conference. IEEE, 2000.

LIU, X. **WiDS checker:** Combating bugs in distributed systems. NSDI, 2007.

LIU, H. et al. Live migration of virtual machine based on full system trace and replay. **Anais…**18th ACM international symposium on High performance distributed computing. ACM, 2009.

LO, D.; MARIANI, L.; PEZZE, M. Automatic steering of behavioral model inference. **Anais…** 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, 2009.

LONVICK, C. The BSD Syslog Protocol. **The Internet Soc**. 2001.

LORENZOLI, D.; MARIANI, L.; PEZZE, M. Automatic generation of software behavioral models. In: International conference on Software engineering, 30, 2008. Nova Iorque. **Anais…**Nova Iorque: ACM, 2008. p. 501-510.

LOU, J. G. et al. Mining dependency in distributed systems through unstructured logs analysis. **SIGOPS Oper. Syst. Rev**.v. 44, n. 1, p. 91-96. 2010.

LUK, C. et al. Pin: building customized program analysis tools with dynamic instrumentation. **ACM Sigplan Notices,** v. 40, n.6, p. 190-200. 2005.

MA, S.; HELLERSTEIN, J. L. Mining partially periodic event patterns with unknown periods. **Anais…**International Conference on Data Engineering, 2000.

MACIA, I. **On the Detection of Architecturally-Relevant Code Anomalies in Software Systems**. 2013. Tese (Doutorado em Informática) - Pontifícia Universidade Católica do Rio de Janeiro, Conselho Nacional de Desenvolvimento Científico e Tecnológico. Orientador: Arndt von Staa. 2013.

MAEBE, J.; RONSSE, M.; DE BOSSCHERE, K. DIOTA: **Dynamic instrumentation, optimization and transformation of applications**. Compendium of Workshops and Tutorials held in conjunction with PACT'02. 2002.

MAKANJU, A.; ZINCIR-HEYWOOD, A. N.; MILIOS, E. E. Extracting Message Types from BlueGene/L's Logs. In: ACM SIGOPS SOSP Workshop on the Analysis of System Logs(WASL), 2009. Nova Iorque. **Anais…** Nova Iorque: ACM, 2006.

MANNILA, H.; TOIVONEN, H.; VERKAMO, A. I. Discovery of frequent episodes in event sequences. **Data Mining and Knowledge Discovery**, v. 1, n. 3. 1997.

MARIANI, L.; PASTORE, F. Automated Identification of Failure Causes in System Logs. In: International Symposium on Software Reliability Engineering, 19, 2008. Washington. **Anais…** Washington: IEEE Computer Society . p. 117-126.

MARIANI, L.; PASTORE, F.; PEZZE, M. A toolset for automated failure analysis. In: International Conference on Software Engineering, 31, 2009. Washington. **Anais...** Washington: IEEE Computer Society. p. 563-566.

MARIANI, L.; PASTORE, F.; PEZZE, M. Dynamic Analysis for Diagnosing Integration Faults. **IEEE Trans. Softw. Eng**, v. 37, n. 4, p. 486-508. 2011.

MARIANI, L. et al. SEIM: static extraction of interaction models. In: International Workshop on Principles of Engineering Service-Oriented Systems, 2, 2010. Nova Iorque. **Anais…** Nova Iorque: ACM. p. 22-28.

MCCABE, T. J.; BUTLER, C. W. Design complexity measurement and testing. **Communications of the ACM,** v. 32, n.12, p. 1415-1425. 1989.

MENDES, C. L.; REED D.A. Monitoring Large Systems via Statistical Sampling. **Anais…** LACSI Symposium, 2002.

MENG, S. et al. Reliable state monitoring in cloud datacenters. Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on. IEEE, 2012.

MEYER, B. **Design by contract**. Prentice Hall, 2002.

MICROSOFT. **TraceLevel Enumeration**. 2013. Disponível em: <http://msdn.microsoft.com/en-us/library/system.diagnostics.tracelevel.aspx> Acesso em: 10 ago 2014.

MINISTÉRIO DE CIÊNCIA E TECNOLOGIA. **Pesquisa Nacional de Qualidade e Produtividade no Setor de Software Brasileiro**. 2001.

MIRGORODSKIY, A.V; MARUYAMA, N; MILLER, B. P. Problem diagnosis in large-scale computing environments. In: 2006 ACM/IEEE conference on Supercomputing, Nova Iorque, 2006. **Anais…**Nova Iorque: ACM, 2006.

MONGO, D.B. **The MongoDB 2.6,** 2013. Manual.

MURCH, R. **Autonomic computing**. IBM Press, 2004.

MURRAY, J.D. **Windows NT Event Logging**. O'Reilly**.** 1998.

NAGY, L.; FORD, R.; ALLEN, A. **N-version programming for the detection of zero-day exploits**, 2006.

NIST. **The Economic Impacts of Inadequate Infrastructure for Software Testing**. National Institute of Standards and Technology Program Office. 2002.

NOSEK, J.T.; PALVIA, P. Software Maintenance Management: changes in the last decade. **Software Maintenance:** research and practice, v. 2, n. 3, p. 157-174. 1990.

OLINER, A.; STEARLEY, J. What Supercomputers Say: A Study of Five System Logs. In: IEEE/IFIP International Conference on Dependable Systems and Networks, 37, 2007. Washington. **Anais…** Washington: IEEE Computer Society. p. 575-584.

PASTORE, F.; MARIANI, F. **AVA**: Supporting Debugging with Failure Interpretations. Software Testing, Verification and Validation (ICST), 2013.

PEZZE, M. **Dynamic Analysis for Diagnosing Integration Faults**. IEEE transactions on software engineering, v.37, n.4. 2011.

PORTER, A. A.; SELBY, R. W. Empirically guided software development using metric-based classification trees. **Software, IEEE,** v. 7, n. 2, p. 46-54. 1990.

PREWETT, J. E. **Analyzing cluster log files using logsurfer**. In: Proceedings of the 4th Annual Conference on Linux Clusters. 2003.

RANDELL, B. **System structure for software fault tolerance**. Springer, 1978.

REASON J. **Human error**. Cambridge University Press; 2003.

REYNOLDS, P.et al. Pip: detecting the unexpected in distributed systems. In: Conference on Networked Systems Design \& Implementation, 3, 2006. California. **Anais…** California: USENIX Association, 2006.

RBI. **Risk-Based Inspection**. Disponível em: <http://en.wikipedia.org/wiki/Risk-based_inspection> Acesso em: 10 ago 2014.

ROCHA, P. **A mechanism for contract verification in distributed-systems based on logs with meta-information**. 2014. Dissertação (Mestrado em Informática) - Pontifícia Universidade Católica do Rio de Janeiro, Conselho Nacional de Desenvolvimento Científico e Tecnológico. Orientador: Arndt von Staa. 2014.

ROMER, T. et al. Instrumentation and optimization of Win32/Intel executables using Etch. **Anais…**USENIX Windows NT Workshop, 1997.

ROS. **ROS.org | Powering the world's robots**. 2014. Disponível em: <http://www.ros.org/.> Acesso em: 10 ago 2014.

RUFFIN, M. **A survey of logging uses**. Technology report, Dept of Computer Science, University of Glasgow,1995.

SANKARAN, S. et al. The LAM/MPI checkpoint/restart framework: System-initiated checkpointing. **International Journal of High Performance Computing Applications**, v.19, n.4, p. 479-493. 2005.

SCHROEDER, B.; GIBSON, G. **A large-scale study of failures in high performance computing systems.** Proc. of DSN, 2006.

SKWIRE, D. et al. **First fault software problem solving**: a guide for engineers, managers and users. Opentask, Kindle Edition. 2009.

SMITH, B. C. **Reflection and semantics in a procedural language**. Technical Report MIT-LCS-TR-272, Massachusetts Institute of Technology, 1982.

STALLMAN, R.; PESCH, R. H.; SHEBS, S. **Debugging with GDB**. Gnu Press, 2002.

STEARLEY, J. **Towards informatic analysis of syslogs**. Washington : IEEE Computer Society, 2004. p. 309-318.

STEVEN, J. et al. **jRapture: A capture/replay tool for observation-based testing**. ACM, 2000.

STRINGER, E.T. **Action research**. Sage, 2013.

SRIVASTAVA, A.; EUSTACE, A. ATOM: A system for building customized program analysis tools. **ACM**, v. 29, n. 6. 1994.

TAN, J. et al. SALSA: analyzing logs as state machines. In: USENIX conference on Analysis of system logs, 1, 2008. California. **Anais...**California: USENIX Association, 2008.

TAKADA, T.; KOIDE, H. MieLog: A Highly Interactive Visual Log Browser Using Information Visualization and Statistical Analysis. In: USENIX conference on System administration, 16, 2002. California. **Anais...**California: USENIX Association, 2002. p. 133-144.

TASSEY, G. **The economic impacts of inadequate infrastructure for software testing**. National Institute of Standards and Technology. Planning Report, 2002.

TELEA, A.; VOINEA, L.; SASSENBURG, H. Visual tools for software architecture understanding: A stakeholder perspective. **Software IEEE**, v.27, n. 6, p. 46-53. 2010.

THOMAS, D. The Deplorable State of Class Libraries. **Journal of Object Technology**, v. 1, n. 1, p. 21-27. 2002.

THOMAS, D.; HUNT, A. **The Pragmatic Programmer: From Journeyman to Master**, 1999.

THOMSON, K. **LogSurfer - Real Time Log monitoring and Alerting**. 2012. Disponível em: <http://www.crypt.gen.nz/logsurfer/ > Acesso em: 20 jul 2014.

TOUPIN, D. Using Tracing to Diagnose or Monitor Systems. **IEEE software**, v. 28, n.1. 2011.

TIMPF, S. **Abstraction, levels of detail, and hierarchies in map series**. Spatial Information Theory. Cognitive and Computational Foundations of Geographic Information Science. Springer Berlin Heidelberg, 1999. p.125-139.

VAARANDI, R. **A breadth-first algorithm for mining frequent patterns from event logs**. Anais… v. 3283, p. 293- 308. 2004.

VAARANDI, R. **A data clustering algorithm for mining patterns from event logs**. IEEE IPOM'03, 2003. p. 119-126.

VAARANDI, R. **SEC:** a lightweight event correlation tool. IEEE Workshop on, 2002. p.111-115.

WITTIE, L. D. **Debugging distributed C programs by real time reply**. v. 24, n. 1, ACM, 1988.

XU, W. et al.. Mining console logs for large-scale system problem detection. In: Conference on Tackling computer systems problems with machine learning techniques, 3, 2008. California. **Anais...** California: USENIX Association, 2008.

XU, W. et al. Detecting large-scale system problems by mining console logs. In: ACM SIGOPS 22nd symposium on Operating systems principles, 22, 2009. Nova Iorque. **Anais…** Nova Iorque: ACM, 2009. p. 117-132.

YUAN, C. et al. Automated known problem diagnosis with event traces. **ACM SIGOPS Operating Systems Review**, v. 40, n. 4. 2006.

YUAN, D. et al. SherLog: error diagnosis by connecting clues from run-time logs. **ACM SIGARCH Computer Architecture News**, v. 38, n. 1. 2010.

ZHANG, R. et al. A **Bayesian network approach to modeling IT service availability using system logs.** USENIX workshop on WASL. 2009.

ZHENG, Q. et al. Intelligent Search of Correlated Alarms from Database Containing Noise Data **Anais…** 8th IEEE/IFIP Network Operations and Management Symposium, 2002.

ZHU, K.Q.; FISHER, K.; WALKER, D. Incremental learning of system log formats. **SIGOPS Oper. Syst. Rev,** v.44, n. 1, p. 85-90. 2010.

11
Appendix I – Failure Handler Examples

This appendix presents an example for each failure handler approach proposed in the thesis.

11.1 Keep-alive and Restart (Global Cycle)

Detection Strategy

```
class KeepAliveDetectionStrategy(DetectionStrategy):
    """
    Defines the strategy for detecting when a node crashes
    """
    def __init__(self, node, *args, **kwargs):
        super(KeepAliveDetectionStrategy, self).__init__(*args, **kwargs)
        self.node = node

    def verify(self):
        """
        Verifies if an occurrence of this fault can be detected by looking
        for the absence of a keep alive in the past seconds.

        :return None if no failure found, or the failure-specific data
        """
        result = lynx.scan_range(
            {
                'node': self.node,
                'keep-alive': {'$exists': True}
            },
            {},
            datetime.utcnow() - relativedelta(seconds=3),
            datetime.utcnow()
        )
        failed_data = None
        if result.count() == 0:
            failed_data = {'node': self.node}
        return failed_data
```

Recovery Strategy

```
class RestartHandlingStrategy(HandlingStrategy):
    """
    Defines the class for handling node restarts
    """
    def __init__(self, package):
        super(RestartHandlingStrategy, self).__init__()
        self.package = package

    def handle(self, data):
        """
        Restart the node using the associated launcher
        """
        node = data['node']
        try:
            # Find the process id
            result = lynx.scan_range(
                {'node': node, 'pid': {'$exists': True}},
                direction=SORT_DESC
            )
            if not result.count():
                raise Exception(
                    'No event [(node, {0}), pid] was found'.format(node))

            # Attempt to kill the process
            process_id = int(result[0]['pid'])
            try:
                os.kill(process_id, 0)
            except OSError:
                # Process not running
                pass
            else:
                os.kill(process_id, signal.SIGKILL)

            # Find the launcher
            result = lynx.scan_range(
                {'node': node, 'launcher': {'$exists': True}},
                direction=SORT_DESC
            )
            if not result.count():
                raise Exception('No event [(node, {0}), launcher] found'.format(node))

            # Restart the node
            launcher = result[0]['launcher']
            print 'Restarting node with launcher {0}'.format(launcher)
            result = subprocess.call(
                'roslaunch {0} {1}&'.format(self.package, launcher), shell=True)
        except Exception, e:
            print e
```

11.2 Inconsistent Version (Local Cycle)

```
class MisversioningFailureHandler(InterceptorFailureHandler):
    """
    Detects and handles the misversioning failure
    """
    def __init__(self):
        signature = [('app_version', '1'), ('action', 'update')]
        super(MisversioningFailureHandler, self).__init__(
            signature, wineserver.device.views.update')

    def explicit_verification_before(self, *args, **kwargs):
        # Check if the request was made by a faulty app version by looking for
        # an error in the mobile environment associated with the update action
        # (in the recent log).
        must_have_dict = {
            'environment': 'mobile',
            'device': logger.value_from_stack('device'),
            'action': 'update',
            'message': 'Failed while parsing field',
            'field': 'price',
            'error': {'$exists': True}
        }
        result = lynx.scan_range(
            must_have_dict,
            {},
            datetime.utcnow() - relativedelta(minutes=5),
            datetime.utcnow(),
            SORT_DESC
        )
        return result.count() > 0

    def handle_before(self, *args, **kwargs):
        print 'Fixing the version'

        # Fix the request descriptor
        request = args[0]
        request.GET['app_version'] = '2'

        # Fix the tag stack state
        logger.change_stack_value('app_version', '2')

    def explicit_verification_after(self, result):
        # Nothing to do
        pass

    def handle_after(self, result):
        # Nothing to do
        pass
```

11.3 Forgotten Retro-Compatibility (Local Cycle)

```
class RemoteUpdateRetrocompatibilityFailureHandler(EventActionFailureHandler):
    """
    Detects and handles the remote update retrocompatibility failure
    """
    def __init__(self):
        signature = [('app_version', '1'), ('action', 'update')]
        super(RemoteUpdateRetrocompatibilityFailureHandler, self).__init__(signature)

    def explicit_verification(self, event):
        return event['message'] == 'Data compilation complete'

    def handle(self, event):
        print 'Handling the retro-compatibility failure'
        f = open(event['path'], 'r+')
        data = json.load(f)
        for item in data:  # Convert to the old format
            item['price'] = item['price']['bottle']
        f.truncate(0)
        f.seek(0)
        json.dump(data, f)
        f.close()
```