UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

PEDRO ARTHUR PINHEIRO ROSA DUARTE

A P2P-based Self-Healing Service for Computer Networks Maintenance

Thesis presented in partial fulfillment of the requirements for the degree of Master of Computer Science

Advisor: Prof.
Liane Margarida Rockenbach Tarouco

Porto Alegre
July 2015

CIP – CATALOGING-IN-PUBLICATION

Duarte, Pedro Arthur Pinheiro Rosa

A P2P-based Self-Healing Service for Computer Networks Maintenance / Pedro Arthur Pinheiro Rosa Duarte. – Porto Alegre: PPGC da UFRGS, 2015.

82 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2015. Advisor:

Liane Margarida Rockenbach Tarouco.

1. Autonomic Network Management. 2. P2P-Based Network Management. 3. Self-Healing Systems. I. Liane Margarida Rockenbach Tarouco,

. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos Alexandre Netto
Vice-Reitor: Prof. Rui Vicente Oppermann
Pró-Reitor de Pós-Graduação: Prof. Vladimir Pinheiro do Nascimento
Diretor do Instituto de Informática: Prof. Luis da Cunha Lamb
Coordenador do PPGC: Prof. Luigi Carro
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

LIST OF FIGURES

2.1 Autonomic Element's reference structure. 19
2.2 Relation among self-* properties, secondary autonomic systems' characteristics and systems' quality factors. 21
2.3 Systems' states diagram and Self-healing basic execution flow. 24
2.4 Self-healing mechanisms' basic functional components and their interactions. 25
2.5 Model-based Modeling. 27
2.6 DieselNet topology at two different time instants of the same day. . . . 35

3.1 Self-healing service biding. 38
3.2 Monitoring workplan execution. 42
3.3 An unhealthy notification arriving at the healing service. 43
3.4 Healing execution and monitoring service notification for *a*) unsuccessful workplan execution; and *b*) successful workplan execution and monitoring resuming. 43

4.1 P2P-Based Network Management System. 48
4.2 ManP2P-ng high-level architecture. 49
4.3 Service and Resource Discovery organization. 52
4.4 monitoring service data organization. 54
4.5 healing service data organization. 54

5.1 Evaluation Scenario for conventional networks. 59
5.2 Total management traffic during the Self-Healing process. 60
5.3 Average duration time of the Self-Healing Process 61
5.4 Evaluation Scenario for Disruption-Tolerant Networks. 62
5.5 Monitoring and Healing plans distribution 64
5.6 Management traffic during the self-healing process. 65
5.7 Total management traffic during the Self-Healing process. 66
5.8 Words per Command. 69
5.9 Characters per Word. 70
5.10 Plot of Keystroke-level Model of Equation 5.1 and experimental data. . 72

5.11 Plot of Keystroke-level Model of Equation 5.2 and experimental data. . 73

LISTINGS

2.1 Interface-based 28
2.2 Annotation-based 28
2.3 Sample implementation 29
3.1 Monitoring workplan developed using Python Programming Language . . . 44
3.2 Monitoring workplan developed using Ponder2's PonderTalk 45
3.3 Healing workplan developed using Python Programming Language 45

CONTENTS

ABSTRACT 9
RESUMO 11
1 INTRODUCTION 13
2 BACKGROUND 17
2.1 Autonomic Network Management Essentials 18
2.1.1 Autonomic Control Loop 19
2.1.2 Self-* Properties 20
2.1.3 Secondary Characteristics 20
2.2 Self-healing Property 22
2.2.1 Normal and Faulty states 22
2.2.2 Autonomy 23
2.3 The Healing Process 23
2.3.1 A Functional Component Analysis 24
2.4 Self-healing mechanism proposals 25
2.4.1 Model-based 26
2.4.2 Development framework-supported 26
2.4.3 Runtime instrumentation-based 30
2.5 Self-healing's Future perspectives 31
2.6 Delay and Disruption Tolerant Networks 31
2.6.1 Networking Architectures 32
2.6.2 Management of Delay and Disruption Tolerant Networks 34
2.7 Perspectives on Delay and Disruption Tolerant Management 36
3 A SELF-HEALING SERVICE FOR P2P-BASED NETWORK MANAGEMENT SYSTEMS 37
3.1 Self-Healing Service overview 37
3.1.1 Healing Service Request 38
3.1.2 Monitoring Service Request 39
3.1.3 Monitoring-healing Binding Request 40

3.2 Monitoring Service . . . 40
3.2.1 Monitoring of Managed Entities . . . 41
3.3 Unhealthy Notifications . . . 41
3.4 Healing Service . . . 42
3.5 Workplans . . . 44
3.6 Deployment Considerations . . . 44
3.6.1 Service Request . . . 45
3.6.2 Monitoring Workplan Execution . . . 45

4 IMPLEMENTATION . . . 47
4.1 P2P-Based Network Management foundations . . . 47
4.2 ManP2P-ng implementation . . . 49
4.2.1 Low-level Interfaces Layer . . . 49
4.2.2 P2P Overlay Maintenance Layer . . . 50
4.2.3 Peer Grouping and Discovery Facilities Layer . . . 51
4.2.4 Components and plug-ins layer . . . 52
4.3 Self-healing Service implementation . . . 53
4.3.1 monitoring service . . . 53
4.3.2 healing service . . . 54
4.4 DTN Management Component implementation . . . 55

5 EXPERIMENTAL EVALUATION . . . 57
5.1 Description of the Case Study . . . 57
5.1.1 Failure Scenarios . . . 58
5.1.2 Self-healing of HIDS Manager and its Sensors . . . 58
5.2 First evaluation: conventional networks . . . 59
5.3 Second evaluation: disruption tolerant networks . . . 62
5.4 Keystroke-level Model discussion . . . 66
5.4.1 Modeling naïve management scenario . . . 67
5.4.2 Modeling common management scenarios . . . 68
5.4.3 Characterization of Users' Behaviour . . . 68
5.4.4 Comparison of naïve, common, and self-healing scenario . . . 71

6 CONCLUSIONS . . . 75

REFERENCES . . . 77

ABSTRACT

In recent years, a huge raise in networks' complexity was witnessed. Along the raise in complexity, many management challenges also arose. For instance, managed entities' heterogeneity demands administrators and managers to deal with cumbersome implementation and deployment specificities. Moreover, infrastructures' current size and growth-trends show that it is becoming infeasible to rely on human-in-the-loop management techniques. Inside the problem domain of network management, Fault Management is appealing because of its impact in operational costs. Researches estimate that more than 33% of operational costs are related to preventing and recovering faults, where about 40% of this investment is directed to solve human-caused operational errors. Hence, addressing human interaction is mostly unarguably a need. Among different approaches, Self-Healing, a property of Autonomic Network Management's proposal, targets to avoid humans' interactions and decisions on Fault Management loops, thereupon unburden administrators and managers from performing Fault Management-related tasks. Some researches on Self-Healing enabling approaches suppose that Fault Management capabilities should be planned in design-time. These approaches are impossible to apply on legacy systems. Other researches suggest runtime analysis and instrumentation of applications' bytecode. Albeit applicable to some legacy systems, these last proposals are tightly-coupled to implementation's issues of underlaying technologies. For this reason, it is hard to apply such proposals end-to-end, for example, in a scenario encompassing many managed entities implemented through different technologies. However, it is possible to offer to administrators and managers facilities to express they knowledge about networks' anomalies and faults, and facilities to leverage this knowledge. This master dissertation has as objective to present and evaluate a solution to imbue network management systems with self-healing capabilities. The solution relies on *workplans* as a mean to gather administrators and managers' knowledge on how to diagnose and heal networks' anomalies and faults. Besides that, the design and implementation of a standard framework for fault detection and notification customization is discussed while considering a P2P-Based Network Management System as its foundations. At last, an experimental evaluation renders clear the proposal's feasibility from the operational point of view.

Keywords: Autonomic Network Management, P2P-Based Network Management, Self-Healing Systems.

Um serviço de *Self-Healing* baseado em P2P para Manutenção de Redes de Computadores

RESUMO

Observou-se nos últimos anos um grande aumetno na complexidade das redes. Surgiram também novos desafios para gerenciamento dessas redes. A dimensão atual e as tendências de crescimento das infraestruturas tem inviabilizado as técnicas de gerencimento de redes atuais, baseadas na intervenção humana. Por exemplo, a heterogeneidade dos elementos gerenciados obrigam que administradores e gerentes lidem com especificidades de implantação que vão além dos objetivos gerenciais. Considerando as áreas funcionais da gerência de redes, a gerência de falhas apresenta impactos operacionais interessantes. Estima-se que 33% dos custos operacionais estão relacionados com a prevenção e recuperação de falhas e que aproximadamente 44% desse custo visa à resolução de problemas causados por erros humanos. Dentre as abordagens de gerência de falhas, o *Self-Healing* objetiva minimizar as interações humanas nas rotinas de gerenciamento de falhas, diminuindo dessa forma erros e demandas operacionais. Algumas propostas sugerem que o *Self-Healing* seja planejado no momento do projeto das aplicações. Tais propostas são inviáveis de aplicação em sistemas legados. Otras pesquisas sugerem à análise e instrumentação das aplicações em tempo de execução. Embora aplicáveis a sistemas legados, análise e instrumentação em tempo de execução estão fortemente acopladas as tecnologias e detalhes de implementação das aplicações. Por esse motivo, é difícil aplicar tais propostas, por exemplo, em um ambiente de rede que abrange muitas entidades gerenciadas implantadas através de diferentes tecnologias. Porém, parece plausível oferecer aos adminitradores e gerentes facilidades através das quais eles possam expressar seus conhecimentos sobre anoamlias e falhas de aplicações, bem como mecanismos através dos quais esses conhecimentos possam ser utilizado no gerenciamento de sistemas. Essa dissertação de mestrado tem como objetivo apresentar e avaliar uma solução comum que introduza nas redes capacidades de *self-healing*. A solução apresentada utiliza-se de *workplans* para capturar o conhecimento dos administradores em como diagnosticar e recuperar anomalias e falhas em redes. Além disso, o projeto e implementação de um *framework* padrão para detecção e notificação de falhas é discutido no âmbito de um sistema de gerenciamento baseado em P2P. Por último, uma avaliação experimental clarifica a viabilidade do ponto de vista operacional.

Palavras-chave: Gerenciamento Autonômico de Redes, Gerenciamento Baseado em P2P, *Self-Healing*.

1 INTRODUCTION

The growth of data communication networks is pushing classic network management techniques to their limits. Also, the requirements of services provided through these networks changed dramatically, becoming more diverse and resource demanding. Then, because of the high complexity, heterogeneity, and low reliability arising from that growth and diversification, it is becoming infeasible to effectively manage networks when the management in place highly depends on human intervention. Furthermore, traditional network management techniques usually rely on premises, such as low-latency data exchange and permanent connectivity, that are not always met in novel networking environments. Thus, new network management techniques are needed in order to fulfill the requirements of these novel environments and to tame networks' increasing complexity.

Among other proposals, the Autonomic Network Management – or ANM for short – is emerging as a promising approach to handle the increasing complexity of modern networks (JENNINGS et al., 2007). The ANM approach is inspired in the behavior of the human's autonomic nervous system, in which multiple biological elements dynamically interact in order to monitor, evaluate, and react, at cellular level, to environmental changes, unburdening the somatic nervous system[1] from primitive but essential tasks, such as keeping the heart and lung functions. Alike the humans' autonomic nervous system, the ANM approach aims to free administrators from low-level tasks, such as devices configuration and monitoring, enabling them to focus on high-level systems goals.

The ANM approach envision the self-management of networks through complementary properties, which are commonly referred the self-* properties. From all ANM properties, four of them are considered the core of its approach. These properties are self-configuration, self-optimization, self-protection and self-healing.

Conceptually, the self-healing property concerns the autonomic systems' ability to discover, diagnose, and react transparently to failures or errors in system components, without requiring further decisions from users or administrators (GHOSH et al., 2007a). Self-healing-enabled systems shall, on a best effort way, diagnose and readjust their run-time parameters or replace faulty elements by spare ones on the occurrence of failures or defects, requesting administrators attention only on exceptional or unknown cases.

[1] The somatic nervous system is responsible for conscious and rational tasks, like moving body parts, evaluating mathematical problems, etc.

The self-healing property has called researches' attention because a total cost of ownership's reduction opportunity may arise from its development. Among 33% to 50% of nowadays networks' total cost of ownership is spent in fault prevention and recovering (FOX; KICIMAN; PATTERSON, 2004). Besides, about 40% of the investment on fault prevention and recovery is related to human administrators' injected faults. In other words, almost a third of the total cost of ownership is concerned with operational errors. Along with the total cost of ownership reduction opportunity, the self-healing property may also positively impact the dependability of the communication infrastructures and systems.

Primordial solutions to achieve self-healing capacities are based on additional engineering efforts during systems' development. Garlam & Schmerl suggest that systems shall be instrumented with monitoring and configuring interfaces (GARLAN; SCHMERL, 2002). PANACEA framework's design proposes the use of code annotations to imbue systems with entry points for healing agents, entities responsible for monitoring and healing the system in run-time (BREITGAND et al., 2007). Despite showing satisfying results, these solutions are tightly coupled to the applications, requiring reengineering efforts to adapt to new scenarios. Moreover, these solutions do not present standardized communication and synchronization primitives, thus, demanding application developers to implement them.

Recently, peer-to-peer (P2P) overlays have been figuring as a basis for developing self-healing-enabled systems. Marquezan *et al.* proposed a mechanism embedded in the monitoring overlay to monitor and heal a Network Access Control (NAC) infrastructure (MARQUEZAN et al., 2007). Considering the problem of Service Level Agreement's Quality of Service maintenance on virtualized infrastructures deployments, Marquezan *et al.* endeavor the use of a self-healing enabled self-organizing scheme (MARQUEZAN et al., 2010). Relying on the management overlay features, Marquezan *et al.* presents a standard communication and synchronization mechanism for the management entities. However, both solutions lack an interface to change or adapt self-healing strategies to new scenarios and applications. Beyond their specific limitations, none of the solutions so far addresses connection intermittence or high delays in data links, which are common problems in modern network environments.

Considering the limitations of the previously presented solutions, this master dissertation has as objective to propose a solution to imbue networking environments with self-healing capabilities. This solution is then based on the fulfillment of two main requirements. The first requirement to be fulfilled is to provide a mechanism to abstract management elements' monitoring and healing process. The second requirement is to provide a framework through which administrators may apply those abstractions to monitor and heal their infrastructures. This framework must consist in an event notification bus and a standard distributed fault detection mechanism with a customization interface so administrators may specialize its functionality to aid them in the monitoring of the services provided in their infrastructures.

In order to address the first requirement, this master dissertation presents the *workplan* concept. Workplans are descriptions written in a high-level language that gather the knowledge of system administrators on how to maintain network devices and systems. To address the second requirement, this master dissertation proposes the use of P2P manage-

ment overlays and also the split of self-healing into two complementary mechanisms, which are provided as management services through the management overlay. The first one is the monitoring service, responsible for executing fault and anomaly detection procedures; the second one is the healing service, responsible for treating the faults and anomalies detected by the monitoring service.

The self-healing mechanism was then implemented as a management component for the prototype P2P-Based Network Management (P2PBNM) System ManP2P-ng (DUARTE et al., 2011), exploiting the overlay's intrinsic characteristics and using its communication infrastructure as basis. ManP2P-ng is briefly described in this master dissertation. The Ponder2 toolkit was used for the interpretation and execution of the workplans (TWIDLE et al., 2009).

An experimental evaluation is presented in order to assess the proposal's design principles. This evaluation consists on the use of the self-healing mechanism as a support for a Distributed Host-Based Intrusion Detection System deployed in two scenarios. The first evaluation scenario is a conventional TCP/IP network encompassing three different administrative domains. The second evaluation scenario is a Delay-Tolerant Network that follows the connectivity island model. Besides these evaluations, this dissertation also features a Keystroke-Level Model analysis in order to set forth proposal's productivity leveraging capabilities.

The experimental evaluation results are reassuring and the main insights achieved through these results show that the proposal is promising. First, the management traffic overhead is minimal when compared to the manual execution of the procedures described in the workplans. Second, the self-healing mechanism performs better as the managed environment grows when compared to the manual healing procedure execution or when compared to the manual execution assisted by standard management tools. Third, the results show that the proposal is able to assist modern network set-ups, such as delay and connection disruption tolerant environments.

The remaining of this dissertation is organized as follow. Chapter 2 presents the state-of-the-art and also the main challenges of developing self-healing-enabled systems. Chapter 3 details dissertation's proposal and also outlines components' organization and behaviour. Chapter 4 exposes ManP2P-ng and self-healing service internals. Chapter 5 describes a Host-based Intrusion Detection System, the environment-of-choice for evaluation. Moreover, Chapter 5 details experiments and presents their results. At last, chapter 6 discusses the conclusions of this dissertation.

2 BACKGROUND

The current trends on the growth of data transmission networks and networked systems has been pushing classical network management techniques to their limits. Given the high complexity, dynamics, heterogeneity, and low reliability that arose from this growth, it is becoming more and more difficult to maintain effective management policies when these policies depend on human intervention for their consolidations (JENNINGS et al., 2007). Therefore, tools and technologies that minimize human intervention during network management related task are been pursued as outcome in many research efforts.

Among many new and novel proposals, the *Autonomic Network Management* – or ANM, for short – gained much attention because its ambitious objective: to build network management architectures that are capable of managing most of its operation without human intervention. The ANM proposal has its basis on the homeostasis process, mostly performed by the Autonomic Nervous System, where multiple organic elements dynamically interact to monitor and react to environmental changes in order to maintain one's stability irregardless of the harmfulness of the environmental changes involved.

Autonomic Network Management proposals are composed of four basic properties, which are maintained through control loops (HUEBSCHER; MCCANN, 2008). Largely known as *self-* properties*, the basic ANM's properties are *self-configuration*, *self-optimization*, *self-protection*, and *self-healing*.

Conceptually, the *self-healing* property concerns the ability of autonomic systems to transparently discover, diagnose, and react to anomalies and faults on its components. Therefore, given the occurrence of any anomaly or fault in any network's managed entity, a self-healing enabled system shall diagnose and solve them by readjusting specific parameters, inserting a new service provider, or triggering any other set of actions that may seem plausible.

The self-healing property presents a high potential to reduce systems' Total Cost of Ownership. Some researches show that 33−50% of networking systems' ownership costs are related to the maintenance and replacement of defective managed entities (PATTERSON et al., 2002). Besides that, the deployment of a *self-healing* mechanism would also improve systems' availability since most diagnosis and recovering tasks finish faster when executed by computers than when executed by human staff.

This chapter presents the state-of-the-art and the main challenges faced when designing and developing self-healing-enabled systems. Although starting with a broad discussion of the topic, its main focus is on proposals applicable to networking and distributed systems. It is organized as follows.

Section 2.1 presents ANM's basing concepts, from which self-healing concepts came originally. Section 2.2 discourses about self-healing proposals, their mechanics, components, development methodologies and how they adapt to a legacy-aware environment. At last, section 2.5 presents current work's and conclusions about the discussed proposals.

2.1 Autonomic Network Management Essentials

Nowadays, networking and distributed systems are becoming more and more dynamic and heterogeneous (JENNINGS et al., 2007). Therewith, these systems are also becoming harder to integrate, deploy, tune, and maintain by applying only legacy network management techniques and tools (KEPHART, 2005).

Legacy network management techniques and tools mainly rely on human intervention and supervision during systems' deployment, monitoring, and maintenance tasks. Then, by observing current trends on systems' complexity growth, it is projected that the systems' deployment will become unfeasible because of the extremely high quantity of human staff needed to manage them, alike what happened with 1920's telephone switching networks (HUEBSCHER; MCCANN, 2008).

Foreseeing a crisis, IBM launched in 2001 the *Autonomic Manifesto*, proposing to academy and industry to pursue a new approach to design complex computing systems. *Autonomic Computing* – the terminology by which this approach became known – has as main objective to imbue computing systems with the necessary elements required to these systems to auto manage (PARASHAR; HARIRI, 2005). The concepts behind Autonomic Computing were specialized to the network management research field, giving birth to *Autonomic Network Management* research.

As early stated, Autonomic Computing, and therefore Autonomic Network Management, is inspired in the autonomic nervous system's behaviour. The autonomic nervous system constantly monitors body's perception of external environment's variables like temperature and luminosity, to cite a few, and then autonomously adjusts body's behaviour in order to maintain its homeostasis. However, autonomic nervous system and autonomic computing diverge in the sense that, while the former behaves totally unaware of individuals' purposes and goals, the latter bases its decisions on high level systems' administrators' decisions on how to deal with changes (STERRITT et al., 2005).

Autonomic Elements are autonomic systems' building blocks. These elements must guide systems to self-manage by constantly reading and reacting to relevant environmental variables and changes (KEPHART, 2005). Autonomic elements must have embedded mechanism to implement their functionalities, export execution restrictions, manage their behaviour according to administrative decisions, and to interact and collaborate with other autonomic elements (PARASHAR; HARIRI, 2005).

Figure 2.1: Autonomic Element's reference structure.

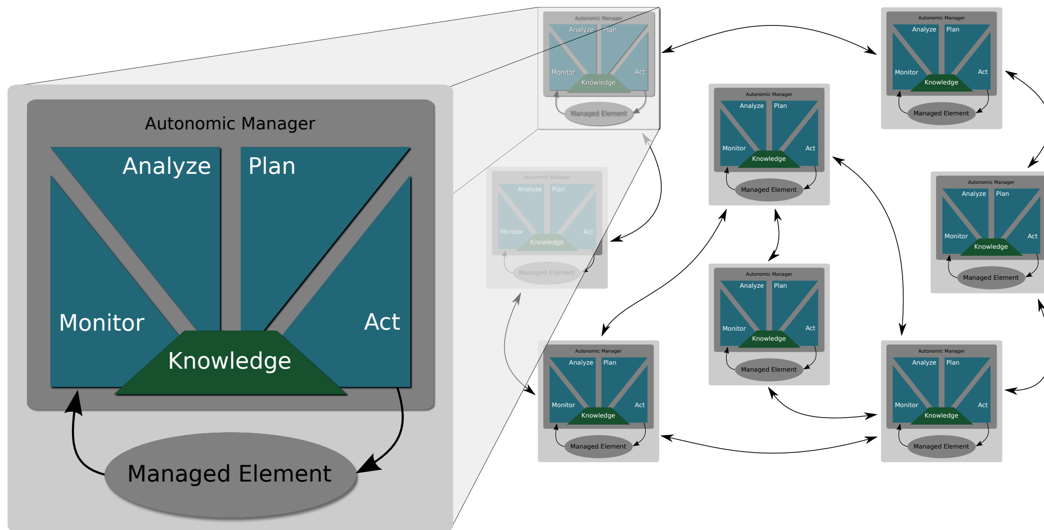


Figure 2.1 depicts a reference structure of an Autonomic Element (KEPHART; CHESS, 2003). As seen in Figure 2.1, an autonomic element is composed of at least one *Managed Element* assisted by at least one *Autonomic Manager*. In the context of autonomic computing, an *Managed Element* may consist of any kind of system, from a single HTTP server instance to a whole content distribution infrastructure (STERRITT et al., 2005).

On its turn, an *Autonomic Manager* is a software entity responsible to collect managed entities' runtime data through different sensors and to execute management actions through different actuators (HUEBSCHER; MCCANN, 2008). In Figure 2.1, autonomic manager's sensors and actuators are respectively abstracted by incoming and outgoing edges of the managed entity.

2.1.1 Autonomic Control Loop

Autonomic manager's basic behaviour consists in four actions: to *monitor*, to *analyse*, to *plan*, and to *execute* (KEPHART; CHESS, 2003). The continuous repetitions of these four actions is denoted as the *autonomic control loop*.

Monitoring consists in periodically collecting environmental attributes and properties that are relevant to system's policy maintenance. For example, in a system that administrators stated that memory usage and CPU utilization must stay bellow a given threshold, memory usage and CPU utilization would be constantly monitored by autonomic managers.

Given that data is collected, it is intuitive that an *analysis* takes place. This analysis shall verify administrative invariants, transform data to human interpretable formats, and any other task that may assist the autonomic manager itself and systems' administrators.

Besides that, an autonomic manager may also figure that anything must be done, been it to optimize resource usage or systems' performance, or to try to keep administrative

invariants. However, before taking any action, an autonomic manager must *plan* its actions so these given actions do not reflect on invariants breaking on other managed entities or even in its own managed entity. At last, the planned actions are executed and then the autonomic element repeats these four actions once and again, closing the control loop.

Autonomic computing research has identified a subset of properties that are present in virtually any computer's systems. Therefore, the main purpose of autonomic managers presented at most proposals is to maintain invariants over these properties (PARASHAR; HARIRI, 2005). These properties are presented in the section that follows.

2.1.2 Self-* Properties

As mentioned before, autonomic managers' control loops are usually engineered to maintain invariants on systems' common properties. In autonomic computing research, these properties are grouped in what is called *Self-* Properties*[1]. Self-* properties are *self-configuring*, *self-optimization*, *self-protecting*, *self-healing.*

Self-configuration is the ability of a system to adapt to configuration-related changes in order to obey administrative decisions (KEPHART; CHESS, 2003). These changes may consist, for example, in the availability of new services instances, network routes, or Quality of Service requirements.

Self-optimization expresses the aptitude of autonomic systems to adjust themselves to maximize their efficiency at runtime. Then, based on their initial configuration and environmental readings, autonomic systems seek to maintain the maximal of the function that represents their global state (BERNS; GHOSH, 2009).

Given that many current systems have rigorous dependability requirements, autonomic computing encompasses security-related issues in the *self-protecting* property. This property has its foundations in the implementation of mechanisms that enable systems to pro-actively resist to malicious usage attempts or even erroneous usage patterns (HUEBSCHER; MCCANN, 2008).

Self-Healing is the property that enables systems to monitor their own state and react to changes that put them in faulty or anomalous states (GHOSH et al., 2007b). This property presents some cross-cutting concerns with other self-star properties since to perform some repairing actions it might be necessary, for example, to change entities' configuration.

2.1.3 Secondary Characteristics

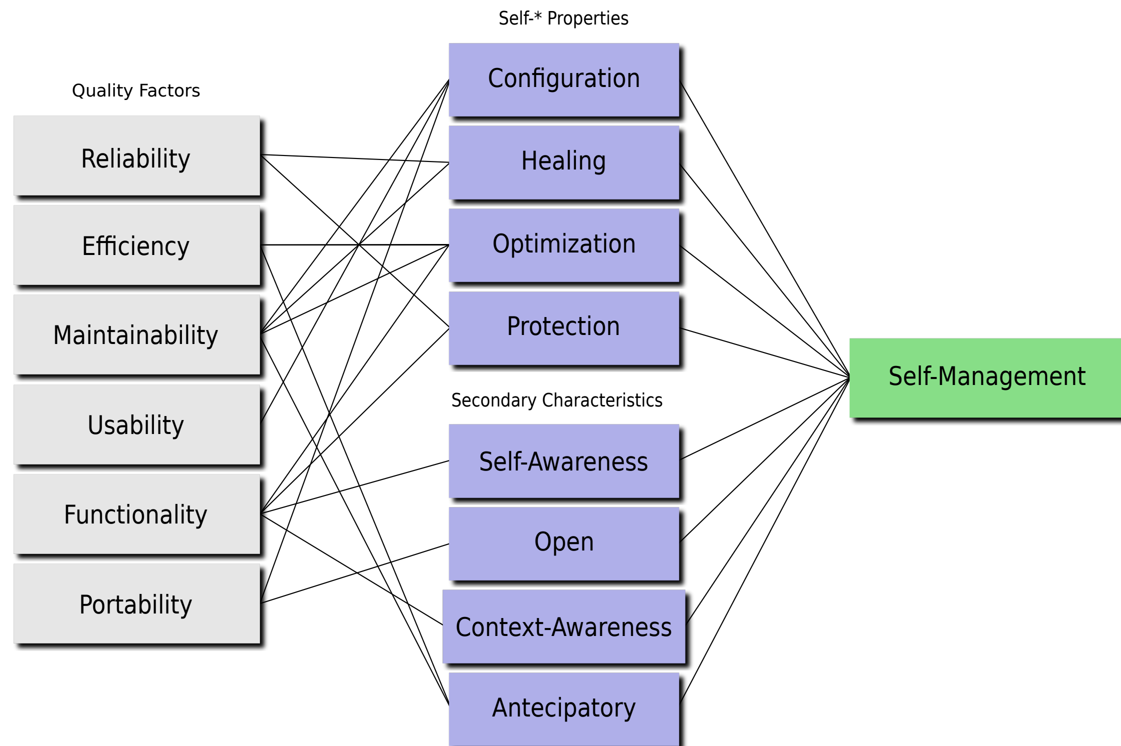
Besides the main self-* properties, literature also defines some extra properties that largely impact autonomic behaviour. Often referred as secondary properties or characteristics, these properties are *self-awareness*, *openness*, *context-awareness*, and *anticipatory behaviour* (SALEHIE; TAHVILDARI, 2005; HUEBSCHER; MCCANN, 2008).

Self-awareness denotes the autonomic elements' need to be always aware of their internal state and behaviour (JENNINGS et al., 2007). Although knowing and reasoning about the internal state may lead to concise and thoughtful local choices, greedy behaviours would mostly lead system to a chaotic or even collapsing state. For this reason, *context-awareness*

---

[1]It is said "self star properties"

Figure 2.2: Relation among self-* properties, secondary autonomic systems' characteristics and systems' quality factors.



is largely regarded given that through it autonomic elements may choose a course of actions that may lead systems to a global maximum that considers the whole infrastructure and service providing in its calculations. Besides that, it is not desired that applications stand still waiting to react to changes at the system's deployment environment. It is desired that the autonomic system can show an *anticipatory behaviour*, where the runtime is quietly tuned while environment changes.

At last, networking environments are generally based on standardized technologies so multi-vendor solutions may coexist and cooperate, consisting virtually all of them in heterogeneous environments. Therefore, it is vital that open and vendor-agnostic interfaces exist. Also, it is essential that these interfaces concisely and uniformly abstract the low-level details and specificities of managed entities.

Figure 2.2 relates self-* properties and secondary systems characteristics with systems' quality factors embraced in software engineering literature (SALEHIE; TAHVILDARI, 2005). Through the previously mentioned figure it is possible to see that, although the distinctions between self-* properties and the secondary characteristics, there is an intersection among the software engineering's quality factors impacted by them. Moreover, it is quite clear that they all contribute to an even end, the self-management.

2.2 Self-healing Property

The self-healing property consists in the ability of autonomic systems to discover, diagnose, and react to disruptions caused by anomalies or faults in a transparently manner, when considering users' and administrators' perspectives (SALEHIE; TAHVILDARI, 2005). To this end, self-healing enabling mechanisms somewhat mimic administrators' behaviour by constantly monitoring the managed infrastructure and taking specific overcoming actions when they are necessary.

Self-healing has gathered researchers' attention because of the Infrastructures' Total Cost of Ownership reduce that would arise from its development. Some results have shown that 33% up to 50% of network infrastructures' Total Cost of Ownership is devoted to prevent and recover failures (PATTERSON et al., 2002). Also, the results have shown that 40% of investment on fault diagnosis and recovery is related to faults inserted by systems' administrators themselves. Besides its impacts in the Total Cost of Ownership, self-healing may also severely impact network systems' dependability, since this later is intimately related to the overall monitoring and diagnosing process duration; duration which would be greatly reduced by a standardized anomalies and faults' monitoring and diagnosing framework.

Next sections discourse about self-healing mechanisms' proposals. However, first Section 2.2.1 discusses the challenges related to diagnosing systems' normal, faulty, and anomalous states. Thereafter, Section 2.3 exposes a common abstract self-healing process while Section 2.3.1 enumerates and describes the essential components of a self-healing mechanism.

2.2.1 Normal and Faulty states

Self-healing mechanism is triggered by the occurrence of anomalies, faults, or even defects. Then, the first effort to an effective self-healing mechanism development is to figure – according to systems' behaviour – which parameters characterize normal operation and which parameters characterize anomalies or faults.

Nevertheless, the definition of normal system's behaviour is related to temporal and subjective questions such as users' expectations on performance (SHAW, 2002). For instance, it might be considered that an application server hosting a real-time data streaming application is anomalous if there is high delay between itself and its client applications, but normal while considering it is hosting of a less demanding application.

Exposed the aforementioned, drawing a discrete distinction among normal, anomalous, or event faulty state is not a trivial task (SHAW, 2002), although most scenarios for faults may seem clear and thinkable instinctively. In most cases, the transition among states may be thought as gray area that represents a gradual degradation of systems' performance until an unacceptable threshold is reached. Therefore, it is important that self-healing mechanisms observe this gradual degradation and also identify threshold reaching so they can initiate rectifications in order to maintain systems' healthy (GHOSH et al., 2007b).

2.2.2 Autonomy

When viewed from human intervention needs, self-healing mechanisms are classified into two categories (GHOSH et al., 2007b). In an ideal scenario, in which all self-healing related tasks are performed automatically without human intervention other than high-level systems' objective settings, a self-healing mechanism is said to be *non-assisted*. Self-healing mechanisms that require humans' decisions on matters not related to high-level management's objectives are said to be *assisted*.

Although a self-healing enabled system's objective is to show complete autonomy while detecting and recovering faults and anomalies, this objective is currently too pretentious. Many anomaly and fault's detection and recovery scenarios demand techniques that are beyond current knowledge gathering, interpretation, and analysis capabilities. In these scenarios, mechanisms require human administrators' guidance to diagnose and recover systems.

Notwithstanding the previously exposed limitation, some proposals still suggest that administrators must only define high-level policies (CHENG et al., 2002). Though remarkable from a theoretical point of view, most of these proposals are unrealizable in practice given the state-of-the-art of cross cutting technologies. Some other proposals suggest that besides defining systems' policies, human administrators must also aid systems to derive new related policies. In the *NxGrantt-Hacker*, the self-healing constantly presents administrators new correlations rules. Administrators then must decide whether these rules shall or shall not be integrated into systems' policies set (STERRITT, 2004).

2.3 The Healing Process

From a functional point of view, the healing process implemented by self-healing mechanisms try to mimic human administrators while they are performing maintenance and recovery tasks (ROTT, 2007). Thus, the expected self-healing mechanism's behaviour shall arise from the observation of human administrators acting.

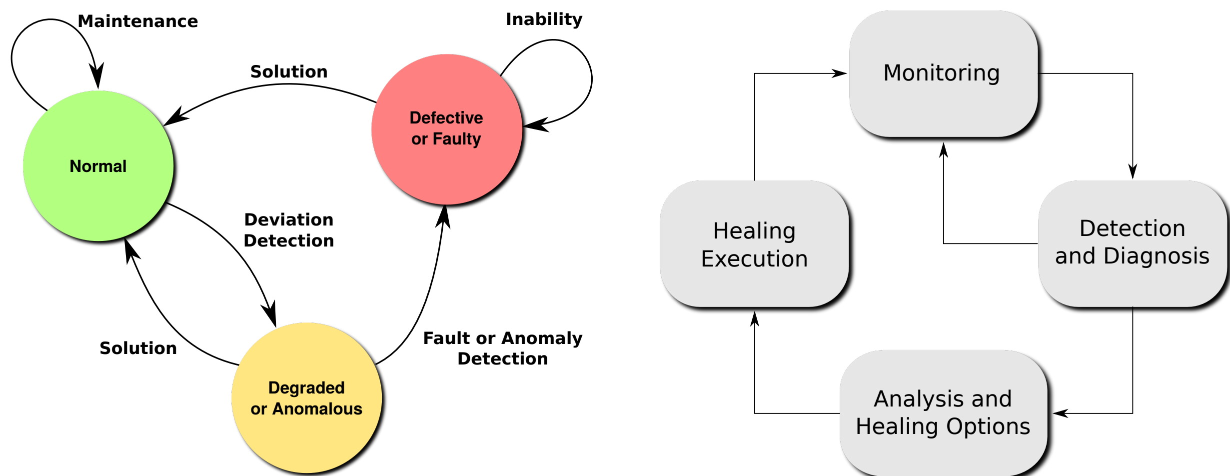
When implemented and deployed, systems are instrumented with tools that enable administrators to collect runtime performance data, offering therefore a monitoring interface, even primitive ones, such as raw logs. The collected data is then manually or automatically processed and summarized so they can be easily understood and reasoned. From their understanding and reasoning, administrators evaluate systems' state. If the evaluation reveals a deviation of what is considered its normal state, administrators *i*) diagnose its root-cause, *ii*) evaluate possible remedies, *iii*) apply normalizing actions and feedback systems' deployment so the very same problem does not occur once again.

The diagram in Figure 2.4a is based on the behaviour previously described and presents systems' states while considering self-healing mechanisms' point of view. For these mechanisms, systems transit from normal to anomalous state when any deviation occur. At this moment, the self-healing mechanism must evaluate and choose an action that solves the anomaly and that puts the systems back to normality. Thereafter, the self-healing mechanism must apply these actions and verify if systems returned to their normality.

Given quality thresholds, the continuous service degradation takes systems to faulty state. Although in place, it is possible that the self-healing mechanism is unable to find suitable actions to restore systems' normality. This inability is represent in Figure 2.4a as Faulty State's self-transition (GHOSH et al., 2007b).

Lastly, if the self-healing mechanism is able to restore the system, it transit to normal state. The processing flow executed by the self-healing mechanism is presented at figure 2.4b.

Figure 2.3: Systems' states diagram and Self-healing basic execution flow.



(a) Systems' state diagram when considered by self-healing mechanisms.

(b) Self-healing mechanism' basic execution flow.

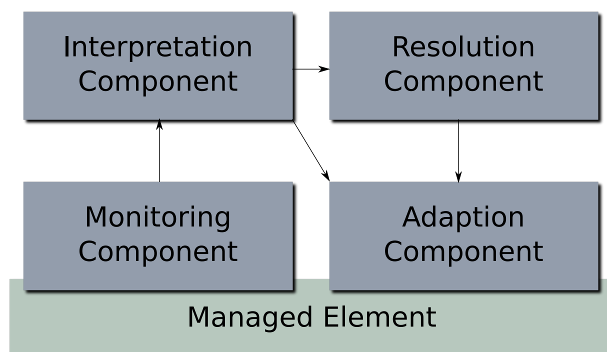
2.3.1 A Functional Component Analysis

Self-healing mechanisms can be dissected in four main functional components. These components are *monitoring*, *adaption*, *interpretation*, and *resolution* components (ROTT, 2007). Figure 2.4 depicts these components and their interactions (ROTT, 2007); these will be detailed at subsequent paragraphs.

Monitoring components are responsible for collecting the data required to analyse systems' state. Because of the high variability of operating environments and their high heterogeneity, many aspects of these components are tightly coupled and devoted to deal with operating environment's details. Therefore, it is recommended that monitoring component's implementations employ a software layer that may easily adapt and abstract environments specificities and also translate these specificities into globally known primitives, just like device drivers in an operating system.

Interpretation components' main objective is to analyse monitoring component's collected data aiming to determine if the systems is running according to high-level objectives. Upon anomalies or faults detection, this component notifies the resolution component so

Figure 2.4: Self-healing mechanisms' basic functional components and their interactions.



this last may find suitable amendments to get systems back to their normal operation. It is suggested that this component may contact the adaption component directly, so simple premeditative actions may be taken, such as restarting simple services or asking applications' servers new instances.

Resolution components are responsible for defining appropriate *treatments* given the data sent by the interpretation component in place. In this context, a treatment denotes a high-level set of actions necessary to get systems back to their normal operation state. Generality/effectiveness is the main trade-off that implementations need to address, since a too general approach may lose many cases and a too specific one would burden systems administrators.

Adaption components translate treatment descriptions into instructions that managed elements can execute. Therefore, any high-level action must be submitted to the adaption component in place so all specificities are decoded to locally concise operations.

As Figure 2.4 implies, monitoring and adaption components are more likely to be coupled with the management element since these are the only components theoretically truly required to know systems' specificities. However this coupling cannot be literally fulfilled since many legacy systems lack the appropriate interfaces to effectively implement the required functions, amendments are possible having as basis legacy interfaces like SSH and Telnet based terminals.

2.4 Self-healing mechanism proposals

Faults detection and recovering methods and mechanisms have existed for a while. Their most well known instances are Object Orientation's exceptions flow support, protocols' retransmissions, and algorithms such as byzantine fault tolerant ones. However, these techniques are limited to detect and recover problems that were envisioned at projects' design. This section is devoted to present and discuss some proposals to overcome current limitations.

2.4.1 Model-based

Model-based proposals suggest that running concurrently the actual systems and models based on these systems can leverage the overall self-healing process (GARLAN; SCHMERL, 2002). In these proposals, running models are used to feed external monitoring and interpretation components so they may compare models' runtime data and actual systems' runtime data. Then, comparing these data would guide self-healing mechanism to the actual adaption needed to restore systems to their normal operation.

Systems models are created through *components interactions graphs*, which are formally specified through *Architecture Description Languages* – or ADL, for short – such as Acme, xADL, and SADL (MILLER; VANDOME; MCBREWSTER, 2010). In components interactions graphs, nodes represent systems' components such as database management systems, web servers, users interfaces, among others management-aware entities, whilst edges, referred as *connectors*, represent systems' interactions and data paths. Figure 2.6a depict a system that was modeled as a components interactions graph.

Applying systems models, the monitoring occurs by means of annotations that guide monitoring components to collect systems' runtime data pertinent to high-levels objectives in place. Besides guiding data collecting, annotations may also contain constraints over this data. These constraints then enable gauging, permitting models to alert self-healing components about actual applications' misbehaviours. Figure 2.6b depicts a components interactions graph as it would be described in Acme language.

With models and annotations in place, administrators must then set-up *repair plans*. These plans describe the actions needed to take systems back to normal operation. Their execution is triggered by constraints trespassing. Figure 2.6c presents a repair plan that will be executed when the constraint $Avg_Latency <= maxLatency$ is trespassed.

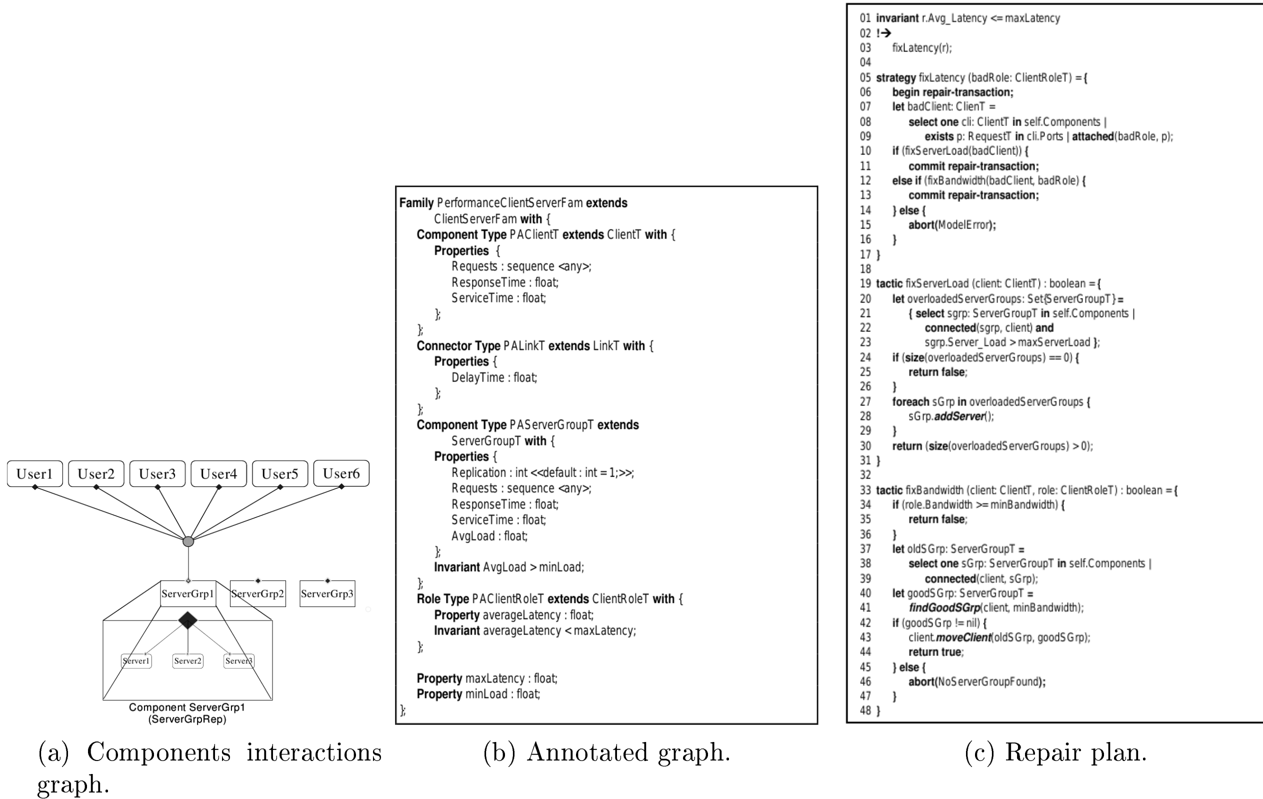
The main advantage of the application of models as self-healing assistance mechanism is to enable the use of different models in different scenarios. In this way, no further adaptation to the self-healing mechanism is required in order to conform to applications' deployment scenarios. Also, using different models permits one to apply various analytical methods and adaption/recovering technique while managing one single scenario.

The main disadvantage of model-based proposals is its poor suitability to legacy applications, since it requires embedded monitoring and adaption interfaces. Besides that, it is difficult for administrators to maintain systems' models in face of constantly changing scenarios, which are more likely to be the case for the application of autonomic management techniques.

2.4.2 Development framework-supported

PANACEA is a development framework for systems with embedded self-healing mechanisms. This framework main proposal is to instrument systems' code at development time with self-healing enabling annotations. At runtime, *healing agents* will use the annotations to guide themselves to monitor, configure, heal, and, utterly, manage systems (BREITGAND et al., 2007).

Figure 2.5: Model-based Modeling.



(a) Components interactions graph. (b) Annotated graph. (c) Repair plan.

Two development facilities are exposed to programmers. The fist one consists in standardized interfaces – as defined by the Java Programming Language – that developers may implement to benefit at runtime of PANACEA's functionalities. This approach is seen at Listing 2.1. The second facility is the use of annotations, as seen in Listing 2.2. *@Manageable* interface marks annotated class as an managed entity. In both facilities, PANACEA-manageable classes are loaded through a special loader, which keeps track of instances and state. Both Manageable interface and *@Manageable* annotation provide three basic primitives for self-healing. These primitives are usable through *@ExecutionCategory*, *@HealerAgent*, and *@DefinedLatencyMonitor* interfaces.

@ExecutionCategory purposes to categorize classes to a known category relevant to self-healing. For example, annotating a class with *@ExecutionCategory(IO)* indicates to that the developer thinks that I/O will be the major component of the class or method annotated. PANACEA defines four major categories, I/O, CPU, memory, and network, but developers are given the interfaces to implement their own categories.

@HealerAgent associates the annotated component to a set of *Healer Agents* (explained afterwards) and *Metric Types*, stating then which healers and metrics must be invoked and monitored, respectively, in order to manage a given component. At runtime, annotations result in the instantiation of healer agents, *i.e.*, overhead in objects' construction phase.

@DefineLatencyMonitor is used to transparently instrument a method to produce measurements of its performance. Therefore, this annotation is fed with an implementer of PanaceaPredicate, an observer-like interface that defines methods to state if an assertion about a metric was transposed while executing a monitored class method. These annotations may be fed with healer agents other than declared in class *@ExecutionCategory* in order to specialize healing actions for the annotated method. Also, this annotation can take a data collecting frequency parameter.

Listing 2.3 presents a sample component which is annotated with all three aforementioned annotations. By loading the sample component, PANACEA will then automatically instantiate and manage instances of *MyCPUHealer* and *MyLatencyHealer* classes.

Through PANACEA facilities, application development occurs twofold. First, the application is formerly functionally developed. In other words, its is submitted to all the engineering process, such as design, development, verification, and validation, as any application would.

Second, a self-healing overlay is applied. This overlay identifies systems' components to monitor by asking the PANACEA's class loader artifacts annotated with *@Manageable* annotation or implementers of the actual interface. The overlay then notifies *healing agents* about these components so the actual monitoring can start.

Listing 2.1: Interface-based

```
public class PseudoClass implements Manageable {
        /* And the code flows */
}
```

Listing 2.2: Annotation-based

```
@Manageable
public class PseudoClass {
        /* And the code flows */
}
```

As early stated, PANACEA's healer agents are special objects managed by PANACEA's runtime components. These special objects have systems' monitoring and healing as task. To easy the use of both monitoring and healing provided by its framework, PANACEA divides healer agents in two categories: *generic* and *application specific* healers. Generic healers encompass agents that can be reused in many context. Moreover, PANACEA further categorized generic healer agents based on the tradeoff that exits between the

Listing 2.3: Sample implementation

```
@ExecutionCategory({CPU,IO})
@HealerAgent(
        type={MetricType.CPU, MetricType.LATENCY}
        valyes={MyCPUHealer.class , MyLatencyHealer.class}
)
public class SampleComponent implements Manageable {

        @DefineLatencyMonitor(
                healer=IOLatencyHealer.class ,
                predicate=MyPredicate.class ,
                frquency=5, enable=true
        )
        public void IOMethod(int time){
                DoIOBenchmark(100);
        }

        /* And the code flows */
}
```

improvement it gives on the healing-targeted system's property and the deterioration its actions will present on other system's properties. Application Specific Healers are mostly not reusable since their tasks are very related to their application domain. However, proposers state that they can provide fine-grained healing and optimization properties to developers and operators.

When compared to alternative frameworks, *e.g.*, *Glassbox*, PANACEA imposes lower impact in systems' performance. For instance, the processing overhead of PANACEA's runtime components is responsible for less than 2.5% of systems' total processing demands.

Besides that, PANACEA proposals' has the advantage of been built atop of well known programming interfaces. Code annotations are ubiquitous in modern programming paradigms, such as Inversion of Control and Aspect Oriented Programming, and then would demand little effort from developers. Moreover, since annotations transparently couple with systems' components, these components can be progressively adapted as self-healing supporting code is developed.

Despites its performance characteristics, PANACEA demands legacy applications to be redesigned so they may benefit of its self-healing primitives. Therefore, PANACEA may be unfeasible in most legacy deployment scenarios. Besides that, PANACEA lacks a proper standardized interface to coordinate distributed agents. Given current trends on computer systems, distribution is essential to many applications if these applications are expected to scale. Thus, healers' interactions and the interfaces used to coordinate these

interactions must be developed in-house. Lastly, Java annotation mechanism requires most of the code to make use of static instances and fields, making it difficult to operators to adjust parameters at runtime and also to use many Aspect Oriented Techniques. However, it is understood that this is a implementation-specific problem and that it might not exist on alternative implementations of the framework.

2.4.3 Runtime instrumentation-based

Some proposals target directly applications developed using interpreted programming languages (FUAD; OUDSHOORN, 2007; SCHANNE; GELHAUSEN; TICHY, 2003; HAYDARLOU; OVEREINDER; BRAZIER, 2005), such as Python, Perl, and many others, or target to virtual machines, such as Java, Scala, Haskel, and many others. In this approach, applications' runtime is analysed in order to determine partition blocks, all of which are considered distinct managed entities, and self-healing interfaces are introduced on the boundaries of these blocks. This technique targets CPU-bound parallel applications that can be modeled as a collection of independent resources.

These proposals show some intrusiveness since they are based on runtime code instrumentation that add proxies (as defined by the proxy design pattern) to methods calls in order to implement self-healing interfaces. Therefore, proxies are created for every class of the system. However, proposed proxies are much more than the classical design pattern proxies as they are aware of object migration and are thus able to delegate calls even if the object is now residing in another machine. Proxies then monitor systems' state before and after each partition blocks' boundary cross in order to assert about performance anomalies and transient faults.

In some proposals (FUAD; OUDSHOORN, 2007), self-healing actions are then expressed as statements in *Autonomic Computing Policy Language*. According to authors, by using this language it is possible to address policy management consistency across the system and provide an user friendly form of policy definition and an API to work with. Other proposals (SCHANNE; GELHAUSEN; TICHY, 2003; HAYDARLOU; OVEREINDER; BRAZIER, 2005), use standard programming interfaces to implement self-healing behaviour.

Best experimental results show that there is an average 51% growth in applications' original bytecode, regardless the complexity of the healing behaviour associated with its monitoring and healing (FUAD; OUDSHOORN, 2007). Besides that, average memory and processing requirements increase in 15% and 45%, respectively. Despites its overhead, all of which is mostly solved with better machinery, these approaches may give satisfiable results when dealing with legacy applications and also applications for which code is unavailable. Anyway, all proposals lack standardized interfaces that enable proper communication and synchronization for monitoring and healing tasks that involve complex scenarios.

2.5 Self-healing's Future perspectives

It was proposed the utilization of Model-Driven and Aspect-Oriented programming to address the lack of self-healing behaviour in systems (LIU et al., 2008a,b). In this proposal, an analysis framework is used to model and inspect UML diagrams of applications in order to identify anomalies and faults detection and recovering opportunities. After identified, *pointcuts* are placed so Aspec-Oriented frameworks can plug-in monitoring and healing *advices* (*i.e.*, self-healing additional behaviour).

Although no experimental result has been presented till date, the utilization of Aspect-Oriented techniques seems promising since most frameworks will natively enable self-healing techniques that can adapt to systems' execution contexts. Moreover, Aspect-Oriented enables even dynamically exchanging execution contexts.

Some proposals explore artificial intelligence techniques. For instance, intelligent agents are applied in a technique known as *roles* (FUNIKA et al., 2010). In order to achieve self-healing, roles communicate and synchronize through an overlay network. This classifies this proposal as the first one to address standardized communications issues. Efforts to design a fully operational management system based on roles are running (FUNIKA; PEEGIEL, 2010).

Neural networks are also considered (AL-ZAWI et al., 2009). Initially, a neural network has its weights tuned through supervised learning. After initialization, the neural network becomes able to suggest new parameters for reconsideration in its adaption step. A case-study (AL-ZAWI et al., 2009) shows proposal's feasibleness even in scenarios that lack of a good training data set.

Although techniques and proposals that merge insightful and most advanced technologies in fields like software engineering and artificial intelligence are arriving, a *de facto* solution for self-healing stills missing. Challenges such as autonomic elements' interaction, environmental heterogeneity, self-learning, data gathering and representation, and scalability will demand many efforts.

2.6 Delay and Disruption Tolerant Networks

It is well known that most Internet protocols fail if inherent networking assumptions unmet (VOYIATZIS, 2012); assumptions such as the existence of continuous and bidirectional end-to-end paths, short round-trip time, almost symmetrical data rates, and low errors. Although mostly met in current widespread networking scenarios, low delay, continuous end-to-end connectivity is absent in deployment scenarios such as Interplanetary networks, mobile *ad-hoc* networks, and connectivity islands, to name a few.

In the context of this research area, *delay* and *disruption* are expressed as a function of the *Transmission Latency*, *Link Disruption*, and the *Action Latency Limit* (BIRRANE III; BURLEIGH; CERF, 2011). The *Transmission Latency* is defined as the latency between placing the data on a transmission queue at one node and retrieving it from a receiving queue at another node, if even the last node is not the final destination of the communication (*i.e.*, is a path section to the destination such as a router or switch). A *Link Disruption*

consists in a loss of communication across the data link. For purpose of engineering link-layer and network-layer protocols, minor link degradations such as transient high error rates are not considered disruption if data exchange may still progress. The *Action Latency Limit* is a requirement associated with a maximum latency permitted between the occurrence of an event and the application of a proper response to such an event.

Using the aforementioned terms, *Delay and Disruption Tolerant Networks* (DTN) are the networks that transmission latencies and link disruption frequencies demands extra techniques to comply with applications' action latency limits and to proper operate in face of an unstable and constantly partitioned topology. The tolerance level a DTN must commit to may call for achieving traditional networking functions by using alternative enabling techniques.

2.6.1 Networking Architectures

Delay and Disruption Tolerant deployment scenarios require new protocols or complete architectures (BURLEIGH et al., 2003), given that is a common consensus and knowledge that only new protocols are unable to overcome all challenges that arose. This section presents proposal of architectures and protocols to aid the deployment of such challenged networks.

2.6.1.1 Delay Tolerant Networking Architecture

The Delay Tolerant Networking Architecture emerged as a generalization of Interplanetary Internet design (FARRELL; CAHILL, 2006). This architecture suggests the usage of *message bundles* and the separation of networks in *regions*. In this proposal, each region is characterized by high homogeneity in their communication capabilities (FALL, 2003).

Given that each region presents different communication constraints, delay and disruption tolerant communication is then handled by *DTN Gateways*, specially crafted nodes that are able to understand and translate different regions' protocols. These gateways are expected to interoperate with current technologies and also to offer reliable communication capabilities instead of the best-effort approaches that Internet's routers provide. To provide reliable communication capabilities, DTN Gateways are suggested to store messages in a non-volatile way, thereby employing a storage and forward message delivery approach.

For routing messages among regions, gateways must be enable to translate technology-specific addressing schemes into globally valid *named-tuples*. These name-tuples are composed of an unique and hierarchically structured region name, and region-unique name used to identify a resource inside that region. The proposal suggests standardization of region identifiers while suggesting resource identifiers to stay flexible enough to easily adapt to different delay and disruption tolerant technologies.

Besides specific addressing schemes, the Delay Tolerant Networking Architecture considers path and hop selection to be too critical to region's functionalities so it does not propose any *one size fits all* selection strategy. Instead, the proposal suggests the use of region-specific routing algorithms given that algorithms pose high influence in region's functionalities and data delivery performance.

Although not requiring standardized routing schemes, Delay Tolerant Networking Architecture does propose that gateways support *custody transfers* of message bundles (CERF et al., 2007). In this context, custody transfer refers to a special communication operation that permits a gateway to dispose of its responsibility over a message bundle and give it to another gateway. After transferring the custody of a message bundle to another gateway, the transferrer may forget any state-related data it had to maintain.

2.6.1.2 RFC 5050: The Bundle Protocol

RFC 5050 describes the protocol, block formats, and abstract service description for the exchange of messages bundles in Delay Tolerant Networking (SCOTT; BURLEIGH, 2007). Aiming to interoperate with existing deployments, the Bundle Protocol defines *Bundle Protocol Agents*, the *Convergence Layer*, and *Convergence Layer Adapters*.

The *Convergence Layer* is defined as the interface between the Bundle Protocol and a specific internetwork protocol suite. *Convergence Layer Adapters* therefore are the interfaces through which communication agents – or *Bundle Protocol Agents* – send and receive bundles utilizing the services provided by the technologies supported at a node.

Bundle Protocol capable nodes are then identified through *Endpoint IDs*. In the context of the Bundle Protocol, an endpoint is a set of zero or more nodes *registered* with a single Endpoint ID. For flexibility, Endpoint IDs are specified as Uniform Resource Identifiers, consisting in a scheme definition and a scheme-specific part. The scheme definition abstracts a set of syntactic and semantic rules applied by Bundle Protocol Agents to interpret the scheme-specific part. Therefore, Endpoint IDs are expected to adapt to many existing and novel deployment scenarios. The only reserved scheme and scheme-specific parts reserved are *dtn* and *none*, used to represent `dtn:none`, the *Null Endpoint ID*.

Given that an Endpoint ID may contain an arbitrary number of registered nodes, the Bundle Protocol discusses extra semantics for communications. Hence, Bundle Protocol defines the *Minimal Reception Group*, which may be any of the following: ALL, where all nodes registered for a Endpoint ID will perceive the communication; $ANY(n)$, where n nodes registered for Endpoint ID will perceive the communication; and, $SOLE$ where a singleton node will perceive the communication. The Bundle Protocol states that Minimal Reception Group characteristics might be inherent to the Endpoint ID's scheme or lexically expressed within the scheme-specific parts.

Following Delay Tolerant Architecture recommendations, the Bundle Protocol says that implementers must provide some level of store-and-forward capabilities to overcome delays and constant disruptions. For calculating the undeliverability of a given bundle, the Bundle Protocol Agent must associate a *lifetime*, hence requiring partial clock synchronization among agents. Nevertheless, the clock source is considered a convergence layer's implementation issue.

2.6.1.3 Internet Draft: HTTP-DTN

Many criticisms on the Bundle Protocol arose. First, authors argue that the protocol alone is not well suited or mature enough to address the problems it proposes to tackle

(WOOD; EDDY; HOLLIDAY, 2009). For example, Bundle Protocol does not standardize common convergence layer capabilities neither endpoint resolution methods, therefore opening road to the existence of many incompatible delay and disruption tolerant networking technologies. Experimentation with the Bundle Protocol (WOOD et al., 2008) has demonstrated that Bundle Protocol is affected by physical concerns and effects that trespass convergence layers' boundaries. For example, Bundle Protocol has no native error detection or correcting neither has checksumming mechanisms.

As an alternative to the Bundle Protocol, HTTP-DTN has been proposed (WOOD; EDDY; HOLLIDAY, 2009). This proposal suggests the Hypertext Transfer Protocol as a session layer so independent transport layers may communicate. The main argument is that HTTP is a well understood and widely deployed protocol and thereupon of easily and straightforward adoption.

For this matter, HTTP-DTN benefits from the special handling that current complying implementations must do with *Content-** headers: implementations must deny the processing of any unknown *Content-** header containing request. Then, *Content-source* and *Content-destination* headers are proposed as end-to-end routing information providing mechanisms, proposal which creates a separated overlay for DTN purposes. Along with other already standardized *Content-** headers, such as *Content-type* and *Content-MD5*, HTTP-DTN would leverage development and deployment of delay and disruption tolerant applications.

2.6.2 Management of Delay and Disruption Tolerant Networks

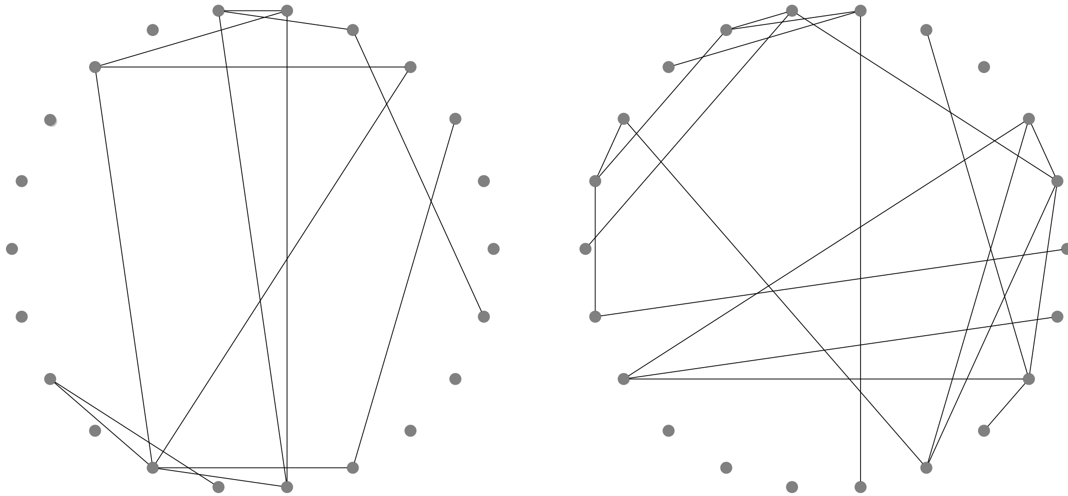
Communication requirements of societies have grown to surprising levels. To meet these demands, networking requirements have grown likewise. Although mainly targeting users, many requirements aim networking infrastructure itself. Manageability, for instance, is a requirement that mostly concern infrastructures and service providers.

Despite the fact that Delay and Disruption Tolerant Networks arose from specific applications requirements such as interplanetary communications, these networks still demand manageability if they are to be seamlessly deployed. Regardless, just a few investigations have explored management capabilities of DTNs. For example, the Delay Tolerant Network Research Group (DTNRG), a special interest group inside the Internet Research Task Force (IRTF), have produced many investigations related to DTN's concepts and practical implementation issues, however producing just a little few investigations targeting management of such networks.

In a glance, most management approaches do not fit for Delay and Disruption Tolerant Networks since they rely on the establishment of end-to-end control loops; *i.e.*, the continuous fetching, processing, and acting upon managed entities' data. Besides that, management solutions must provide some degree of self-management because the larger propagations delay of sending operational commands prevents managers to promptly perform remote actions.

Furthermore, novel management solutions will have to deal with the instability of Delay and Disruption Tolerant Networks' topologies. For instance, the network topology may

Figure 2.6: DieselNet topology at two different time instants of the same day.



partition itself in uncommon and unpredictable ways. To illustrate, Figure 2.6 presents the topology of a subset of DieselNet (BURGESS et al., 2006) nodes at different snapshots of its operation.

Given all these facts about challenges of Delay and Disruption Tolerant Networks, it is thereupon straightforward to argue that management solutions must inherently support distributed deployment of management entities. Indeed, some authors are already investigating towards applying Distributed Network Management (DNM) techniques to manage such challenged networks.

The Internet Draft *DTN - Network Management Requirements* (IVANCIC, 2009) represents a major step of DTNRG because it was the first document discussing Delay and Disruption Tolerant Networks' management issues. This work describes general requirements and properties, and also suggests technical challenges that implementers must consider specially while addressing configuration and monitoring tasks.

Another Internet Draft (CLARK; KRUSE; OSTERMANN, 2010) presents the Diagnostic Interplanetary Network Gateway (DING). As its name imply, DING aims to provide manageability to Interplanetary Networks. In Interplanetary Networks delay tolerance must achieve new levels since the expected transmission latency range from some hours to months. For this reason, DING relies on a subscription model to gather management data. This model is implemented using two concepts, *schema* and *schedule*.

The *schema* is a static data model that defines management data to which managers would need to subscribe to. Roughly, schemes represents *Object Identifiers* (OIDs) data that follows standardized representations. In their turn, *schedules* describe time intervals and conditions management data must comply in order to be useful to management.

The Internet Draft *Initial Requirements for Remote Network Management in Delay-Tolerant Networks* presents initial considerations and requirements for designing management protocols for DTNs. Authors discuss current protocols' deployments challenges and

propose several distinct architectures, as do comment on their strong and weak points. Authors propose data gathering mechanism similar to the subscription model envisioned by DING and also propose to apply proxies and caches to achieve better performance.

Although expired, these Internet Drafts represent major steps into the development of an architecture for Delay and Disruption Tolerant Network Management. They are the first attempts of managing challenged networks. They present knowledge mostly acquired trough experimentation and also point trade-off, do and don't, and the question one should query while trying to implement and deploy management solutions for these networks.

2.7 Perspectives on Delay and Disruption Tolerant Management

The infeasibility of maintaining control loops over Delay and Disruption Tolerant Networks' deployments have pushed researchers to explore the application of Autonomic Network Management for such scenarios. Context-Aware Broker (PEOPLES et al., 2010) proposes a policy-based framework for DTN management. Context-Aware Brokers (CAB) incorporates network environment data and applications requirements to configure data transmission automatically. CABs would then enable resource saving and adaptive management.

Moni4VDTN (ISENTO et al., 2012) proposes an application-layer approach where a dedicated server is deployed to collect load-related data from managed entities. This proposal targets mainly Vehicular Delay-Tolerant Networks, remaining an open issue how the interaction between managed and management elements would occur when considering different deployment scenarios.

Peer-to-peer based Distributed Network Management has also figured in proposals. Nobre et al (NOBRE et al., 2014, 2013) present experiments that extend Peer-to-Peer based Network Management Systems to operate atop challenged networks. Their proposal encompass a full implementation of the HTTP-DTN Internet Draft. In these proposal, authors show that is possible to rework current management tools to run over DTNs, obviously observing the coherence of tools' purposes and DTN limitations.

It is clear that Delay and Disruption Tolerant Network Management research is in its beginning and there is much efforts yet to be spend in this area. However, it seems feasible to think of a common substrate for managing such networks given that their main challenge is the availability of data and the maintenance of control loops, challenges to which already exists proposals.

3 A SELF-HEALING SERVICE FOR P2P-BASED NETWORK MANAGEMENT SYSTEMS

Previous investigations have show that merging network management systems and P2P communication models introduces interesting characteristics. For example, Application-Layer Routing provides extra flexibility for administrator since this routing technique easily adapts to administrative domains boundaries; Gossip-based peer membership management algorithms, on their turn, improve management systems' connectivity robustness, avoiding network partition by exploiting random graph-like structures' properties.

In this dissertation we exploit P2P-Based Network Management systems capabilities to propose, implement, and evaluate a self-healing service aimed to aid administrators in their daily network maintenance tasks. The self-healing service is then described in this chapter. Besides that, the last section of this chapter presents considerations about the deployment of the proposed solution in conventional and challenged networks.

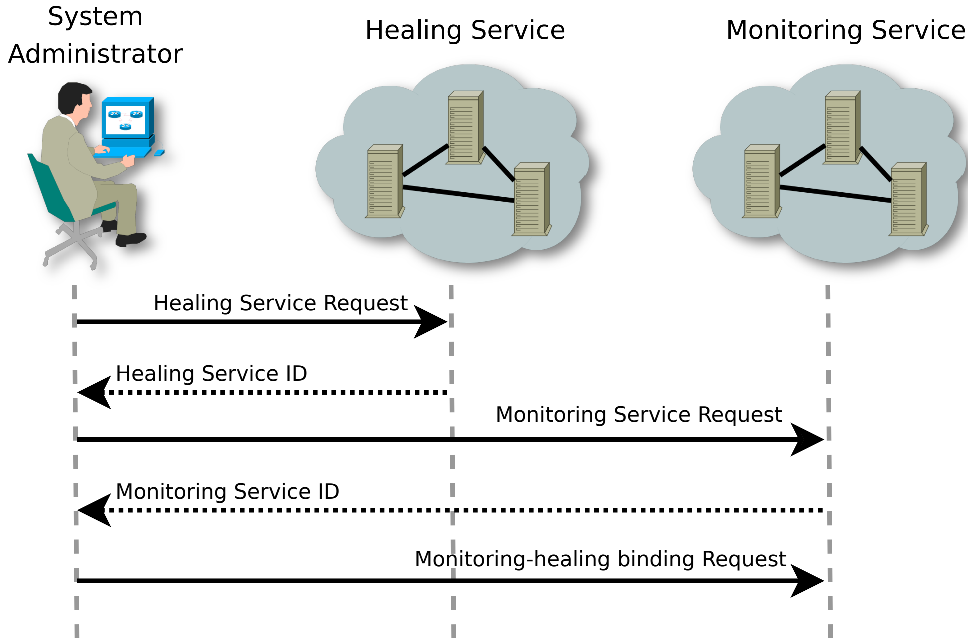
3.1 Self-Healing Service overview

The self-healing service proposed in this dissertation divides the self-healing into two different services, both of them implemented as management components for a P2PBNM system. The first service, the *monitoring service*, is responsible to periodically poll and evaluate managed entities[1] searching for anomalies and faults on these entities. The monitoring service is also responsible to notify anomalies and faults to the healing service, the second service proposed in this dissertation. The *healing service*, is responsible to execute healing procedures in order to recover faulty managed entities. Healing service healing procedure execution is triggered upon receiving special notifications from the monitoring service. These notifications will be referred from now on by *unhealthy notifications*.

Before the self-healing can take place, system administrators must develop *monitoring* and *healing workplans*. Briefly, workplans are high-level descriptions that gather administrators' knowledge on how to evaluate and fix anomalies or faults in order to reestablish system's operation. The step in which workplans are assigned to managed elements is called *service binding*, and it is described as follows.

[1] In the context of the present work, *managed entities* may refer either to physical entities, such as routers and switches, or to software entities, such as HTTP and SSH daemons.

Figure 3.1: Self-healing service biding.



The binding of the self-healing service and its managed entities is done at run time by system administrators and consists of three major steps, depicted in Figure 3.1. First, the system administrator issues a *healing service request* to the healing service's peer group, which will then return a *healing service identifier*. Second, the system administrator issues a *monitoring service request* to the monitoring service's peer group, which will then return a *monitoring service identifier*. Finally, the system administrator issues a *monitoring-healing binding request* to the monitoring service's peer group, which binds a *monitoring service identifier* to a *healing service identifier*. This binding tells the monitoring service which healing service instance to activate when anomalies or faults are noticed in managed entities.

3.1.1 Healing Service Request

A *healing service request* is issued by system administrators to request the healing service's peer group to handle anomalies and faults in managed entities and consists in a tuple:

$$((target_0,\ \ldots\ ,\ target_n),\ workplan,\ unhealthy\ threshold,\ candidate\ peers)$$

The first attribute of the tuple is a *target list*. This list enumerates the managed entities managed by this service instance. The target list shall bundle management-relevant information, like transport layer protocol and service port, or system-specific parameters, such as operating system and daemons implementations, etc.

The *workplan* attribute of the tuple is the managed entities *healing workplan*, a high-level description that defines how anomalies and faults shall be treated. The extra information bundled in the target list is passed to the healing workplan upon its execution in order to assist the healing peers to deal with managed entity specificities and to provide more flexibility and reuse possibilities for the healing workplan.

The *unhealthy threshold* specifies how many *unhealthy notifications* must be received before executing a healing workplan. In short, unhealthy notifications are messages that signal that anomalies or faults were diagnosed in a managed entity.

As will be explained on the next sections, workplans are replicated among a subset of management peers in order to provide load balance and fault tolerance. In most management scenarios, these peers may be picked at random. However, in scenarios such as challenged networks with poor bandwidth or connectivity, it may be required to assign management tasks to specific managed entities, given that optimizing locality is mostly advantageous in such scenarios (BENAMAR et al., 2014). Therefore, requests may encompass the *candidate peers* parameter. This parameter will hint the healing service about peers that should host and execute the workplan.

As a response to a healing service request the healing group sends a *healing service identifier* (*HsId*). This identifier is used to globally identify the healing workplan, to bind it to a monitoring service instance in the services binding phase, and for future modification or update of the plan.

3.1.2 Monitoring Service Request

A *monitoring service request* is issued by the system administrators to request a monitoring service for a managed entity. A monitoring service request consists of a tuple:

$$((target_0,\ \dots\ ,\ target_n),\ workplan,\ schedule,\ confirmations,\ candidate\ peers)$$

The *target* attribute of the tuple specifies the target managed entity and any other relevant information, just like in a monitoring service request.

The *workplan* attribute is the management element's *monitoring workplan*, a high-level description that defines how the managed entity shall be monitored and which parameters identify its normal and anomalous state. Alike the healing service request, the extra information passed through the *target* attribute is used to aid dealing with the specificities of the managed entities and for reuse purposes.

The third attribute of the monitoring service request tuple is the *schedule* of a monitoring service instance. This schedule tells the monitoring group when peers shall trigger the evaluations process for the managed entities targeted on a request.

It is known that many networking related problems are due partitions or transient connectivity. Considering novel deployment scenarios, partitions and transient connectivity may mislead monitoring service's diagnosis. To lessen this issue, the monitoring service request encompass the $confirmations$ attribute. Acting as a mechanism to provide consensus, this attribute forces $confirmations$ number of peers to reevaluate the managed entity

before issuing a unhealthy notification. Through this mechanism, many peers may be consulted about anomalies or faults before contacting the healing group to heal a managed entity.

Analogously to the healing service request, the last attribute, *candidate peers*, serves primarily to deal with requirements such as static mappings of mangement elements to managed entities. The use of *candidate peers* serves as a mechanism for topology-awareness. For example, an external topology service or administrators expertise would aid the service to avoid poor management performance due to bad choices of monitoring service's peers.

As response for a monitoring service request, the monitoring service group issues back a *monitoring service reply*. The content of this reply is a *monitoring service identifier* (*MsID*), which globally identifies a service request and may be used for reference updating purposes. As a reference, a *MsID* can be used to apply its monitoring workplan to another managed entity (*i.e.*, sending the *MsID* instead of the actual workplan during the monitoring service request).

3.1.3 Monitoring-healing Binding Request

The last request, the *monitoring-healing binding request*, ties a monitoring service instance to some healing service instances (in other words, a monitoring service may trigger several healing services). This request consists in a tuple ($MsID$, [$HsID_0$, $HsID_1$, ..., $HsID_n$]), where the first parameter specifies the monitoring service instance and the second parameter lists the healing service instances that shall be activated on anomalies or faults detection.

In scenarios where point-to-point communication can severely impact in the biding process, the biding may be issued with hash values based on the contents of the healing service request and the monitoring service request. Given that no acknowledgement is required, it becomes possible to issue requests in parallel.

3.2 Monitoring Service

The monitoring service is the peer group responsible for monitoring-related tasks. Then, the assignment of this group is to diagnose anomalies and faults in managed entities based on the administrators knowledge contained in monitoring workplans.

The output of a diagnose is the managed entities' evaluation. The evaluation is *healthy* when no anomalies or faults are diagnosed. The evaluation is *unhealthy* when anomalies or faults are diagnosed. Upon an unhealthy evaluation, the monitoring service contacts the healing service to activate the healing service's instances previously binded to the faulty monitoring service instance.

As early stated, a monitoring service instance is started just after the monitoring service request and its execution is activated based on the schedule sent along the request. Besides the schedule, the request also encompass detailed data about the managed element targeted in a service instance and the workplans which defines how the managed element shall be monitored.

After processed and accepted, monitoring workplans are distributed among a subset of monitoring service's peers. Previous results shown that this subset must grow logarithmically to the size of the entire managed network in order to reliably monitor it (MARQUEZAN et al., 2010).

3.2.1 Monitoring of Managed Entities

The monitoring of managed entities is mandated by the value of the $confirmations$ attribute of the monitoring service request. Then, if $confirmations$ is equal to zero, the monitoring workplan will be concurrently executed by the monitoring service peers following the schedule.

However, if $confirmations$ is greater or equal to one, the subset of peers chosen for a monitoring service instance organize themselves in a logical ring. The monitoring workplan is then executed in a *token-signalized round-robin* fashion, where the peer currently holding the token is the one responsible for the next scheduled evaluation.

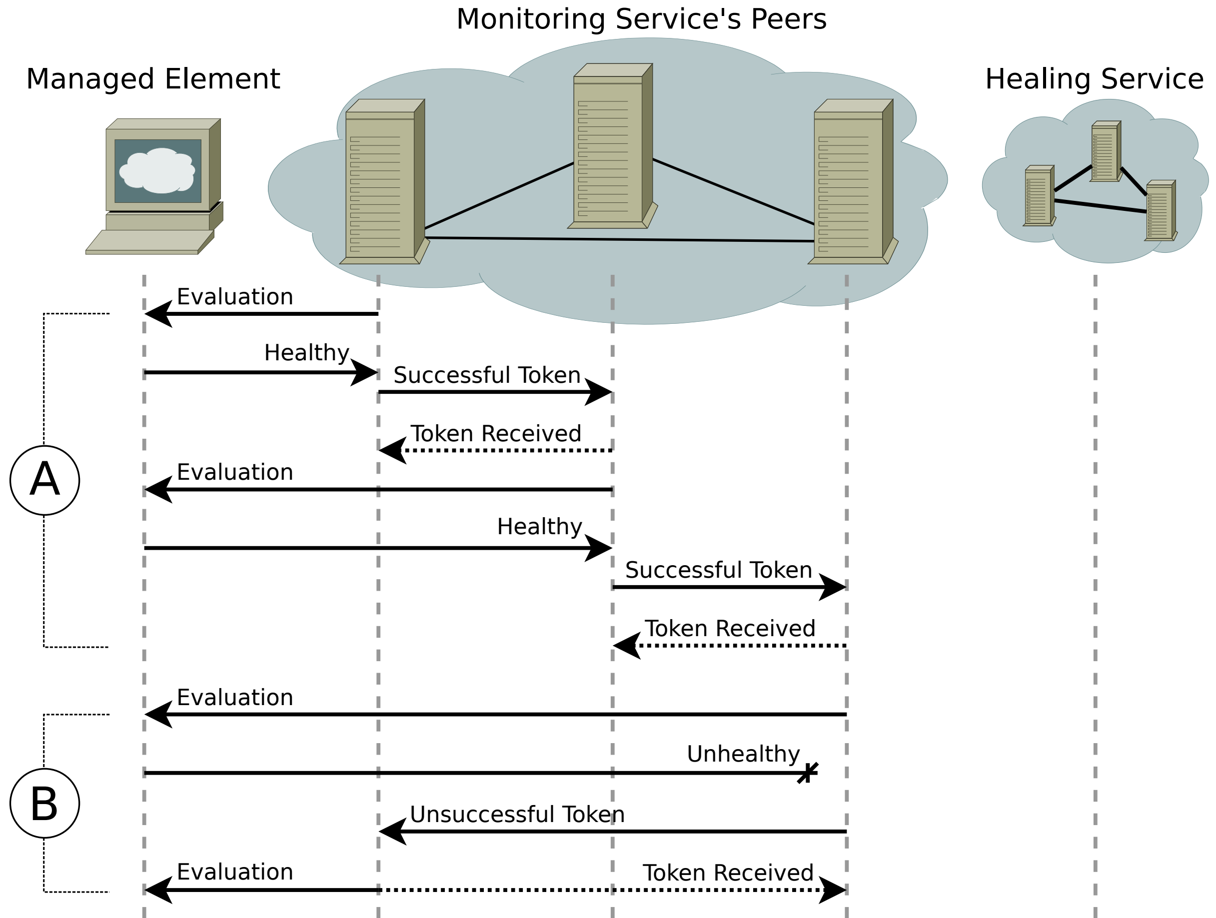
The monitoring service's peer holding the token executes the monitoring workplan against its target and *i*) if the results indicates that the target managed entity is healthy, that is, no anomalies or faults were noticed during the diagnosis, the token is passed in the logical ring flagged as *successful*; *ii*) if the workplan indicates that the managed entity is unhealthy, the token is passed in the logical ring flagged as *unsuccessful*. Figure 3.2 depicts these interactions.

Figure 3.2A, depicts service interactions for subsequents healthy evaluations, while Figure 3.2B, depicts an unhealthy evaluation in a scenario where diagnose confirmation was requested. As shown in Figure 3.2B, when confirmations are needed, a monitoring service peer immediately executes its diagnosis when an unsuccessful flagged token is received, ignoring, then, the workplan schedule. This measure is taken in order to assure that the anomalies or faults are not related to connectivity problems between the targeted management entity and the previous evaluation executor. In the case that $confirmations$ peers also identify anomalies or faults, an *unhealthy notification* is sent to the healing service to start the healing service instance associated with the current monitoring service instance, thus executing the healing workplan. The monitoring service's peers associated with this service instance will stop monitoring the ill managed entity until the healing service notify them if the workplan shall be resumed, that is, that they successfully healed the managed entity, or if it shall be dropped, when the healing process failed to recover the managed element.

3.3 Unhealthy Notifications

The unhealthy notifications are the triggers of healing workplans executions. These notifications feed the healing service with helpful information for the proper execution of the healing workplan. To achieve that, monitoring workplan execution's output is appended to the notification, hence enabling the healing service to consume it and to make decision with current state data.

Figure 3.2: Monitoring workplan execution.



3.4 Healing Service

Considering a scenario with low faults and anomalies rate, the healing service will stand still most of the time. The healing service main task is to wait for *unhealthy notifications* sent by the monitoring service. This is shown in Figure 3.3.

At the arrival of an unhealthy notification, the healing service extracts from it the healing service identifier and instantiate it. The instance is then fed with the data specified at the service request and also the data sent by the monitoring service through the unhealthy notification's payload. At last, the healing process is started.

After executing the workplan, the healing service may take two courses of actions. If the healing workplan was executed as expected, the healing service will notify the monitoring service to continue monitoring the managed entity.

However, if the healing service readily state that the healing workplan failed (for example, when the managed entity is steadily unreachable), the healing service will notify monitoring service to stop monitoring the anomalous or faulty managed entity. Both situations can be seen in Figure 3.4.

Figure 3.3: An unhealthy notification arriving at the healing service.

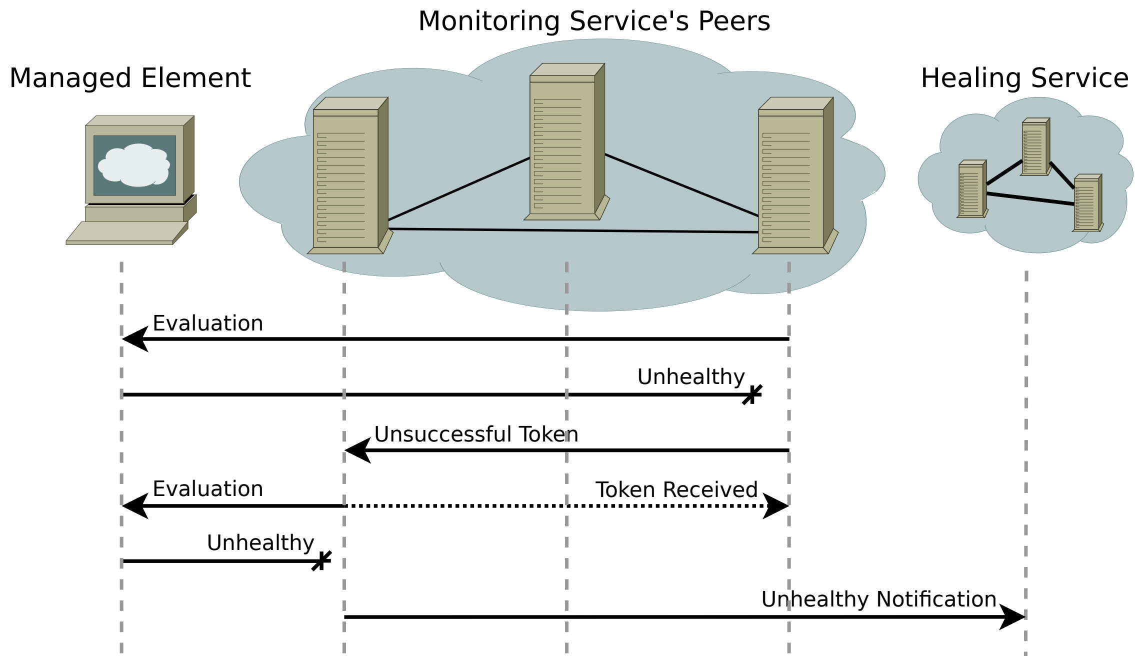
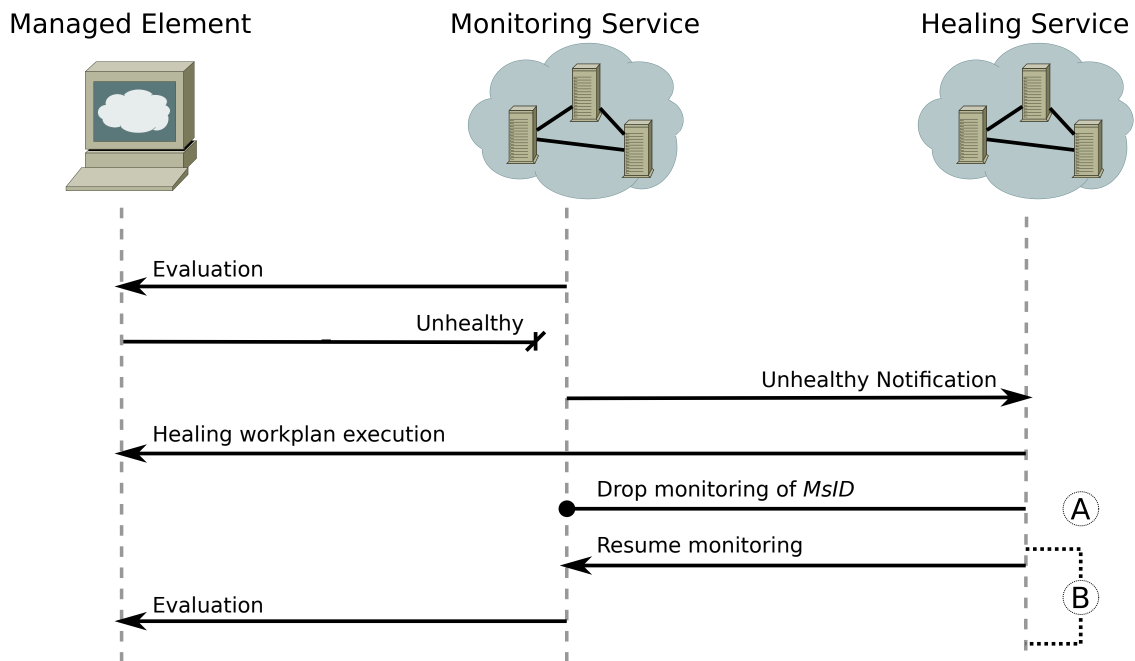


Figure 3.4: Healing execution and monitoring service notification for a) unsuccessful workplan execution; and b) successful workplan execution and monitoring resuming.



3.5 Workplans

As early stated, *workplans* are the abstractions through which administrators' knowledge about network maintenance is gathered. Initially, workplans were designed to be represented only as Ponder2 policies (TWIDLE et al., 2009). However, current trends on the adoption of networking maintenance tools (HAIGHT, 2011) reveals that high-level, general-purpose programming languages such as Python and Ruby are feasible choices. For instance, Metasploit Framework (RAMIREZ-SILVA; DACIER, 2007) and w3af (KE; YANG; AHN, 2009) leverage the power of the aforementioned languages to provide system administrators basic tooling and abstractions to automate their networking-related tasks. Then, Python-like workplans are also considered in the context of this research. This empowers the monitoring and healing capabilities by enabling administrators to employ an increasing number of tools, APIs, and yet to come technologies to represent their thinking while managing infrastructures. Listings 3.1 and 3.2 show the same monitoring workplan, which notifies the healing service when the processor usage is above a threshold (90% in the examples).

Listing 3.3 shows a healing workplan suitable to be deployed along the monitoring workplans at Listings 3.1 and 3.2. This healing workplan uses a third-party library to request through a REST API the instantiation of a new server at a famous cloud-computing service provider. This workplan presents the flexibility and simplicity that can be achieved by the means of this proposal.

Listing 3.1: Monitoring workplan developed using Python Programming Language

```
from system import cpu
from service.healing.client import notifyHealingGroup

if cpu.usage > .9:
  notifyHealingGroup(cpu.highUsage)
```

3.6 Deployment Considerations

Virtually any technology – ranging from algorithms up to application-level software – will need adaption to cope with Disruption Tolerant Networks singularities. This is due to the premise that most of the current technology base on: low delay, partial reliable end-to-end connectivity. Following subsections will discuss how to deploy this proposal in conventional and challenged networks.

Listing 3.2: Monitoring workplan developed using Ponder2's PonderTalk

```
cpuHigh := newpolicy create.
cpuHigh
  event: root/event/cpuUsage;
  condition: [ :cpuUsage |
    cpuUsage > .9 ];
  action: [
    root/service/healing/notify highCpuUsage ];
  setActive: true.
```

Listing 3.3: Healing workplan developed using Python Programming Language

```
from requests import post

payload={
  'ImageId': 'server-template',
  'MinCount': 1,
  'MaxCount': 2,
  # here goes authentication data
}

post('https://ec2.amazonaws.com/?Action=RunInstances',
      data=payload)
```

3.6.1 Service Request

As previously discussed, when dealing with challenged scenarios, managers and management systems cannot rely on low latencies or the actual message ordering. Because of that, services request shall preferably occur without confirmations. In order to attain a non-confirmation service request, hash-based requests must be used. The hashes must be based on workplans content since the same workplan may be applied to many managed entities.

3.6.2 Monitoring Workplan Execution

Section 3.2.1 explains how the monitoring service performs its job. The same section exposes the purpose of the *confirmations* parameter of the monitoring service request. Deepening on this matter, this subsection discusses when to apply confirmations and when not to apply them.

Informally, it is arguable that a monitoring service peer would diagnose a managed entity as unhealthy if some of the diagnosing messages get lost (*e.g.*, if some of path's router or switch fails). For conventional deployment scenarios, such as low delay TCP/IP networks, confirmations may be used as a reliability mechanism that would largely prevent such mistaken diagnose. For example, setting *confirmations* during service request to one would require another monitoring service peer to confirm the diagnosis before contacting the healing service to heal the managed entity.

Although the informal argument before mentioned holds for challenged networks, using confirmation and the logical ring topology can be infeasible in many challenged networks scenarios. For instance, two monitoring service peers present a delay high enough to unacceptably retard further actions. Deployments on these scenarios should consider to concurrently execute workplans (*i.e.*, *confirmations* equal to zero during request). This leads to the following property.

Let $g(w)$ be the projection of the monitoring group responsible for a workplan w, $e(x, y)$ be the event of peer x having an encounter with peer y, and $P(e)$ a function that returns the probability of an event e, it is trivial to show that the concurrent workplan execution renders the workplan w a reachability r to its targeted managed element t equal to

$$r(w, t) = 1 - \prod P(e(i, t)^c),\ \forall\ i \in g(w) \tag{3.1}$$

In other words, concurrent execution guarantees a workplan a probability to reach its target equal to the probability of any of monitoring peer responsible for its execution to have an encounter with the targeted managed element. In Equation 3.1, the probability is expressed through the opposite probabilities of events $e(x, y)$.

4 IMPLEMENTATION

This chapter presents the design decisions of the implementation of the ManP2P-ng and the self-healing service. It is organized as follow. Section 4.1 presents the basic concepts behind ManP2P-ng design and implementation. Section 4.2 presents the details of the management overlay. Section 4.3 presents the details of the self-healing service.

4.1 P2P-Based Network Management foundations

In face of the constant growth of networking infrastructures, researchers and practitioners have envisioned that classical centralized management approaches would not stand against the scalability issues that would arise. For instance, centralized approaches would poorly perform when dealing with big infrastructures, since a single manager will be unable to poll device data, keep up-to-date views of devices' configuration and status, process asynchronous network events such as traps and notifications, among other challenges.

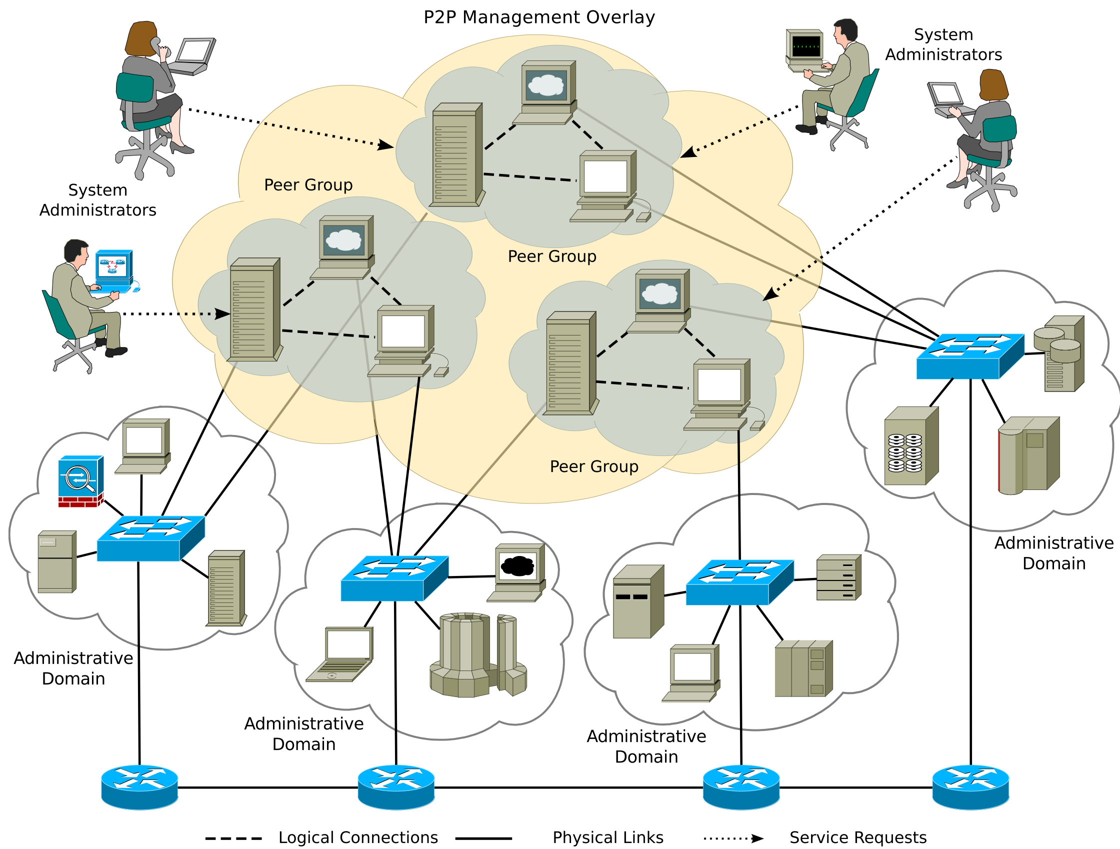
To deal with scalability issues, researchers suggest the application of some kind of distribution strategy (SCHONWALDER; QUITTEK; KAPPLER, 2000). Management by Delegation (MdB) proposes the delegation of management tasks to management entities distributed onto network locations (GOLDSZMIDT; YEMINI, 1995). This enables, for example, placing management entities and elements at the same physical substrate, then benefiting from lower transmission delays and higher throughput. In MdB, the management entities are classified as Top-Level Managers and Mid-Level Managers. Top-Level Managers are the entities responsible to monitor and coordinate delegated management tasks. Mid-Level managers, in their turn, are the entities responsible to execute the management tasks themselves.

Other proposals suggest the use of Peer-to-Peer technologies to scale management infrastructures (GRANVILLE et al., 2005). In a Peer-to-Peer Based Network Management System – or P2PBNMS – peers are categorized into groups. The categorization is based totally on the services provided by each peer. For illustrating the concept, all peers providing a Secure Shell Transport Layer (SSH) Service would be grouped together in the *SSH* peer group. Besides logically disposing peers, the grouping mechanism is conceptually thought as to provide transparent balancing and fault-tolerance. However, complex services would need custom service dispatching policies and coordination strategies.

In such an organization, clients (*i.e.*, system administrators and network operators) would not interact directly with *management entities* – the term through which *peers* are going to be referred from now on. Instead, users use the overlay client protocol to generate management service's requests which shall be transparently dispatched to peers providing the service. For example, an administrator wishing to establish an interactive shell session with some managed entity only needs to request the service through the overlay client protocol and the overlay itself shall find a peer to serve it or signaling the unavailability of the service.

Besides that, Peer-to-peer Based Network Management Systems better adapt to administrative domains boundaries given that these management systems employ Application Layer Routing (ALR). through ALR it is possible to create different transport protocols by defining a single high-level substrate to abstract addressing issues. Therefore, it is possible to fully isolate management elements, management entities and clients. Also, inter-domain management is made easy since it is possible to plug-in protocol adapters for allowing management entities to operate over services and protocols that are already in place. Figure 4.1 illustrates the concepts presented in this section.

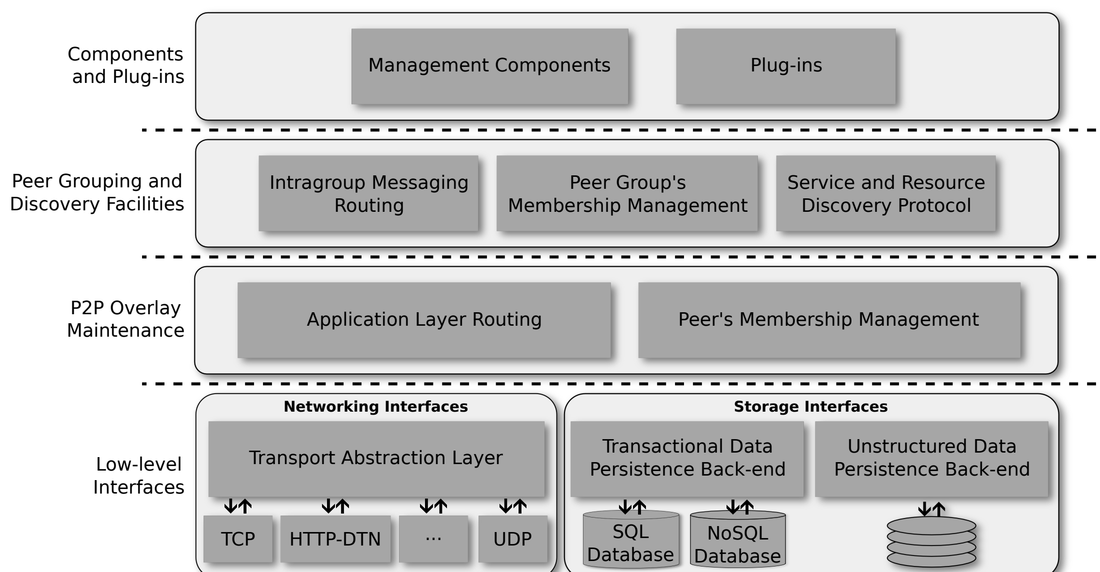
Figure 4.1: P2P-Based Network Management System.



4.2 ManP2P-ng implementation

The main purpose of ManP2P-ng is to provide a platform to ease the implementation of novel management applications by developers and network administrators. An *Application Programming Interface* (API) is provided for extending its capabilities. The API enables the implementation of different management applications as well as modifications in the overlay basic mechanisms (*e.g.*, for better adequacy to a specific scenario such as delay and disruption tolerant networks). Likewise, the API itself can be extended. The basic building blocks of ManP2P-ng are the Python Programming Language (VANROSSUM; DRAKE, 2010) and Twisted Framework (LEFKOWITZ; SHTULL-TRAURING, 2003), an event-driven networking engine licensed under the open source MIT license. Figure 4.2 presents the high-level architecture of the ManP2P-ng and will guide the discussion in this section.

Figure 4.2: ManP2P-ng high-level architecture.



The ManP2P-ng architecture is composed of four layers, depicted in Figure 4.2. Each layer groups related interfaces, which are exposed as new abstractions for upper layers. This design eases the development of management applications without restricting complex modifications such as the introduction of new transport protocols or distributed algorithms. The principal line of thought was to provide a simple and convenient framework yet powerful and flexible. The features of each layer are described in the next subsections.

4.2.1 Low-level Interfaces Layer

This layer provides the basic building blocks for the management overlay. It encompasses interfaces related to Input/Output of data: abstraction for networking and storage. These interfaces are separately explained bellow.

4.2.1.1 Networking Interfaces

The low-level networking interfaces have two objectives. The first objective is to deliver a networking-technology agnostic message sending and receiving interface. This interface acts as a mean to develop high-level primitives, enabling the implementation of common overlay maintenance algorithms, regardless of low-level details. Therefore, upper layers must deal with a compact set of standardized operations since all the specificities are hidden by networking low-level interfaces.

The second objective is to provide developers a set of ready to use networking protocols, such as SSH, HTTP, FTP, among others. This was achieved mainly by exposing Twisted Framework interfaces.

4.2.1.2 Storage Interfaces

The low-level storage interfaces provide abstractions for common storage options and it is divided in two categories. The first category encompasses transactional persistence engines, like SQL and Not Only SQL (NoSQL) back-ends. Through transactional storage interfaces developers may leverage their management components by using relational or document-oriented storage. These interfaces consist in wrappers over Python's *PEP 249: Python Database API Specification v2.0*.

The second category is the file system-like abstraction. They are used for local and distributed storage. When concerning local storage, Python's portable file-systems abstractions and persistent dictionaries-like structures are exposed. However, when concerning distributed and reliable storage, Zookeeper/Kazoo's (HUNT et al., 2010) wrappers are provided. These wrappers simply auto-configure Zookeeper's interfaces so any node that joins the *distributed.storage* peer group may join the service infrastructure.

Zookeeper offers distributed execution of atomic operations *create*, *delete*, *setData*, *getData*, and *getNode*. These operations are available for creating an Unix-like distributed directory structure. For example, calling `create("/data")` would add a *node* called *data* as a *sub-node* of the *root* node.

By the means of Zookeeper's operations, the management overlay is enhanced with distributed reliable and secure storage. Besides that, the distributed storage may be used for coordination purposes (*e.g.*, distributed locking) since there exists guarantees about the atomicity of operations and the consistency of the data stored in a node.

4.2.2 P2P Overlay Maintenance Layer

The P2P overlay maintenance layer is responsible for application layer message routing and peers' membership maintenance. Natively, the architecture supports the utilization of different Application Layer Routing strategies, however only two are implemented. The first strategy uses the Cyclon protocol (VOULGARIS; GAVIDIA; STEEN, 2005) to create a gossip-based overlay for scenarios where connectivity is unreliable. Although primarily following Cyclon protocol, the implementation also follows best practices on gossip-based algorithms implementation (RIVIèRE et al., 2007), enabling this way the experimentation of different gossip-based protocols.

The second strategy uses the ZeroMQ protocol to maintain a structured however brokerless multi-protocol overlay. This way, a subset of more robust nodes may form a logical backbone for the management overlay, thus enabling more flexibility while defining network boundaries (*e.g.*, fewer connections would need to be maintained on firewalls and other devices). This ALR strategy applies when structured topologies are feasible choices for message passing.

Peer's Membership Management is highly coupled with the ARL strategy mainly due reliability issues. When the gossip-based ARL strategy is applied, the membership is achieved though exchange operations and its properties (STAVROU; RUBENSTEIN; SAHU, 2002). When structured ARL strategy is applied, membership is maintained at infrastructure's nodes, as it is stated in the protocol.

4.2.3 Peer Grouping and Discovery Facilities Layer

As early stated, Peer Grouping is the basis for implementing management services in a P2P-Based Network Management System. Besides that, nothing would work without nodes knowing where and how to reach services. Supporting primitives for group communications and discovery is then a must.

4.2.3.1 Group Membership Management

Two algorithms are provided for grouping. The first one, suitable for unstructured deployment scenarios, keeps a copy of the group membership information on every node that composes the group. Joins, departures and faults are managed gossip-like: a node may actively join a peer group; other peers may inform the joining; and departures and faults detections are broadcast.

The second algorithm uses the distributed storage to keep peers' grouping information. This algorithm reserves `/groups` as base node. Then, each group is registered as a sub-node of `/group` (*e.g.*, `/groups/ssh/`) and memberships are registered as sub-nodes inside `/peers` sub-node (*e.g.*, `/groups/ssh/peers/sdn_dmz_manager`).

4.2.3.2 Service and Resource Discovery

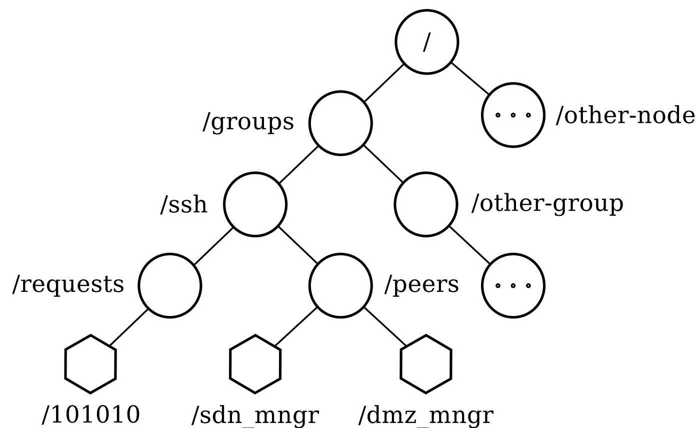
Service discovery is essential to any overlay because it is the facility that enables administrators and services to reach themselves. Since discovery is closely related to deployment scenarios, the latter must dictate the discovery algorithm to use. Therefore, there are two discovery strategies that may be applied at run-time: iterative deepening with checking, and directory based.

Iterative deepening works by asking neighbours if they provide a service (MESHKOVA et al., 2008). If they do provide, the search is over and the provider is contacted with actual service request data. However, if all of its neighbours are unaware of the service, the requester now asks its neighbours' neighbours about it. The request still deepening a level until a provider is found or fails if there is no new level to deep.

For structured overlays, service discovery is already provided by the *Group Membership Management* facility. Finding services consists only in checking the existence of the corre-

sponding /group sub-node. For example, if one is interested in a SSH service, one must check the existence of /group/ssh node. For brevity, the tilde character (~) will be used to shortly refer to a previously mentioned path. Requesting services consist in creating a sequential leaf-node at the child-node named /requests (*e.g.*, ~/requests/101010) with the necessary data. Figure 4.3 presents the services' organization as a tree-like hierarchy.

Figure 4.3: Service and Resource Discovery organization.



If the service was once provided by any peer, its /requests node will exist regardless if there is any peer capable to serve the request. Hence, requests may be later served when some peer can serve it. Also, this opens the risk of starving. For this reason, requests that must be served respecting some deadline shall specify this deadline during the service request. Requests are asynchronously dispatched for execution.

4.2.4 Components and plug-ins layer

The Components and Plug-ins layer provides the interfaces through which new services are implemented and provided. This layer deals with two kinds of software entities, *plug-ins* and *components*.

4.2.4.1 Plug-ins

In ManP2P-ng, plug-ins have the same semantics that they have in other domains: a software entity that builds up new functionalities atop of another software entity. Therefore, plug-ins are meant to extend ManP2P-ng with capabilities that are too specific to be implemented at the overlay's core. Plug-ins are reached through usual import statements.

4.2.4.2 Management Components

Management Components are how new services are implemented. Thus, management component's functionalities are reachable throughout the overlay. Besides that, management components are also provided with messaging facilities, enabling different entities that provide the same service to coordinate their behaviour.

Management Components are given two communication channels. The first one is used to internal communications between peers providing the service. Messages sent to this channel are broadcast to all peers providing the service. This channel is also used by the overlay to send control messages, such as join, departure, and failures.

The second channel is used by other peers to request services. An interface is given so specific service providing peers may chose to listen to these messages while other peers do dedicate to service providing.

4.3 Self-healing Service implementation

The self-healing service was implemented using the management component interfaces provided by the ManP2P-ng. Analogously the proposal, the self-healing service was implemented as two different management components: the monitoring service management component and the healing service management component.

4.3.1 monitoring service

The monitoring service is responsible for monitoring managed elements and contacting the healing service when any anomalies or faults are detected. Aiming to avoid classifying transient or locality-related network errors as anomalies or faults, the monitoring service assigns a subset of its peers to execute a monitoring workplan. Then, a decision about the anomalies or faults are based on the result of a given number of monitoring attempts, all of them executed by different peers or at different moments.

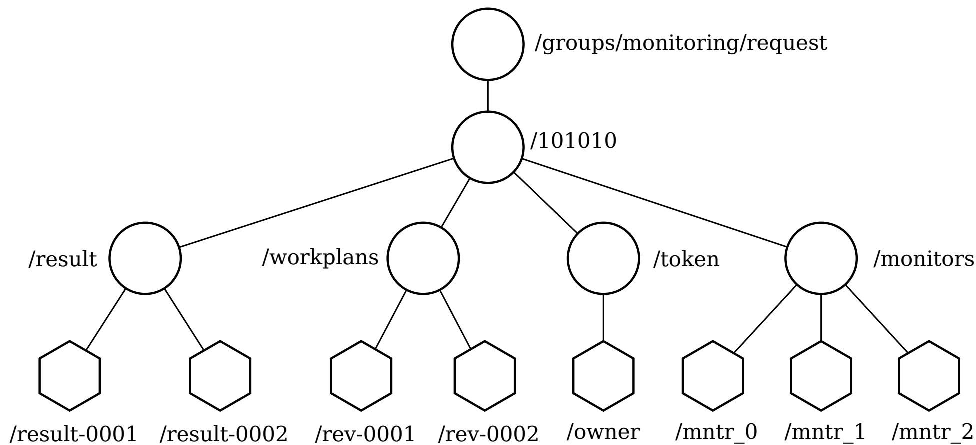
The subset of peers responsible for executing a workplan is composed of any peer that were listed as a candidate peers (refer to section 3.1.2) plus the quantity of peers necessary to reach a total of $log(size(monitoring\ group))$ monitoring peers. The peers chosen are then stored in the `/monitors` node, which is a sub-node at the request node. No criteria is used for choosing non-candidate peers, albeit is plausible to argue that some peers may get overloaded given bad choices.

The monitoring starts afterwards the request serving peer chooses all the monitoring peers responsible for executing the workplan. The execution follows the order that the peers were added to the `/monitors` node. The next scheduled monitoring executor is pointed by the node `/token/owner`.

For storing the monitoring results, sequential nodes are created at `/results`, which in turn resides into the request node. Therefore administrators, managers, and other peers may check the contents of a result anytime, given that no other process has deleted the results. Results may be erased periodically or on demand (*i.e.*, administrators request).

Given that anytime administrators and managers are expected to create new revisions of workplans, a sub-node `/workplans` is used to store sequential nodes that represent the revisions of the workplan. The overall picture of `/groups/monitoring` sub-node is presented in Figure 4.4.

Figure 4.4: monitoring service data organization.

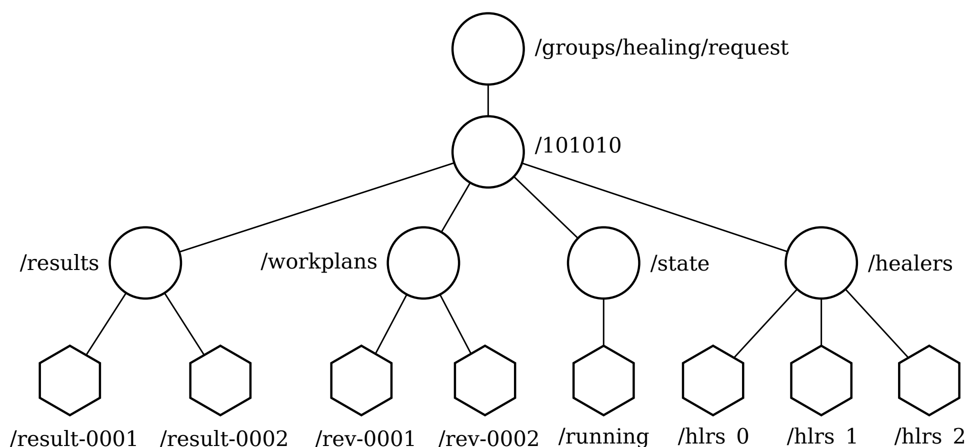


4.3.2 healing service

The healing service internals shares most of the decisions taken for the monitoring service. Therefore, the various worplans' revisions are also stored at requests' `/workplans` sub-node. Also, $log(size(healing\ group))$ peers are chosen to serve the request and this subset is again places at the storage facilities, this time in the `/healers` sub-node.

Different from the monitoring service, the healing service does not need synchronization of execution aside from asserting that only one peer will be executing a healing workplan in a given moment. Thereupon, no sub-node is used to store synchronization information, however an ephemeral `/state/running` is used to signalize that some peer is currently executing the workplan. The result of healing workplan execution is then stored at the `/result` sub-node. The overall picture of `/groups/healing/` sub-nodes is presented in Figure 4.5.

Figure 4.5: healing service data organization.



4.4 DTN Management Component implementation

The Delay and Disruption Tolerant Management Component is implemented as a protocol adapter for the operation of the P2P management overlay in DTN environments. The protocol adapter defines mainly interfaces and, hence, is not tied to any specific DTN protocol implementation, although these protocols strongly influenced its interfaces.

Since untied of any specific protocol, the management component enables flexible deployments: managers and administrators are free to choose the protocol – or even a full stack – that best suits their needs. As a consequence, the management component is unconcerned with IP or IP-related protocols (*e.g.*, TCP or DNS), and because of that overlay namespace is used for addressing and routing purposes. The design choice of keeping the management component technology agnostic also supports the transport layer independence which is necessary in long delay environments.

This research is primary interested in opportunistic connections without too long delays. For this reason, TCP is used as the Proof of Concept transport protocol implementation. HTTP-DTN draft, in turn, is used to deal with network disruptions. Despite using TCP and HTTP-DTN in this current research, the management component can easily handle long delays and TCP limitations through the use of alternative transport protocols, such as Saratoga (WOOD et al., 2012). For store-and-forward purposes, this implementation uses overlay's transactional persistence engines. To this end, HTTP-DTN packages features were modeled using Entity-relationship Model and accessed according the Object-Relational Model.

The HTTP-DTN protocol is a superset of HTTP/1.1 and, in this context, HTTP/1.1 pipelining and persistence allows multiple transmissions to be made in sequence. Addressing information is provided through DTN specific header fields, all of which inter-operate transparently with current HTTP/1.1-compliant implementations. For addressing purposes, `Content-Destination` and `Content-Source` fields identify the destination and the source of the data, which are filled with peers' identifiers.

The current management component implementation works considering the premise that all peers in a peer group have this extension enabled. This premise may not be true in some scenarios, which decreases network efficiency. However, it is feasible to develop a DTN gateway-like feature (in order to interconnect networks with different characteristics) through adaptations in the management component.

5 EXPERIMENTAL EVALUATION

In order to analyse and assert the feasibility the self-healing service proposed in chapter 3, two experiments and their experimental set-ups are presented and discussed in this chapter. Both experiments encompass the maintenance of a Distributed Host-based Intrusion Detection System's Infrastructure. Moreover, the maintenance of this infrastructure is considered when operating over both conventional and challenged networks. When applying the proposal to a challenged network scenario, the analysis focuses on Internet Access for Remote Villages, an instance of a Delay and Disruption Tolerant Network.

At last, given the automation purpose of the self-healing service, this chapter compares the performance of human-in-the-loop fault management and the performance of the self-healing service proposed at chapter 3. To realize this comparison, this chapter presents a Keystroke-Level Model analysis of the self-healing service

This chapter follows the forthcoming organization. Section 5.1 discusses and presents the managed environment and also its management infrastructure. Sections 5.2 and 5.3 shows results gathered through experiments and their analysis. Section 5.4 discusses the Keystroke-level analysis and shows the differences between human-in-the-loop approaches and the proposed self-healing service.

5.1 Description of the Case Study

The case study is based on the use of a self-healing service to assist a Host-based Intrusion Detection System (HIDS). A HIDS monitors and analyzes hosts in order to determine whether they are being attacked or compromised. Ideally, a HIDS must employ a correlation engine able to detect patterns in misuse scenarios (DEBAR; DACIER; WESPI, 1999). Moreover, HIDS also must produce human-readable outputs so system administrator may diagnose threatening events and respond to these events. However, HIDS still subjected to new or unknown attacks (FUNG et al., 2010), or even non-malicious faults.

Though most HIDS employ manager-agent approaches, they devise from standard management-agent approaches since they rely on *data pulls*, where each intrusion detection agent periodically collects data from its hosting system and sends this data to its manager. In the context of HIDS, agents are usually referred as *sensors*, and so they will be referred along this case study.

5.1.1 Failure Scenarios

There are two scenarios for faults in an HIDS deployment. The first scenario is when a sensor node fails. A well-designed HIDS must continue to function properly without the failing node. It is not essential, but it is strongly recommended that administrators get informed about these faults. The second and most critical scenario is when the manager node fails. In this scenario, an HIDS must notify the administrator about the manager fault and graciously halt the sensor nodes to prevent misuses.

Given the primary role that Information Technology (IT) security plays in nowadays communication networks infrastructures, both scenarios described in this section are undesirable as they partially or completely stop the intrusion detection facility. In the first scenario, the sensor's host may be compromised and used for malicious purposes without knowledge by the systems administrators. In the second scenario, the misuse would have more comprehensive scales. For example, an *0-day* worm-like threat would indiscriminately spread itself through the infrastructure.

5.1.2 Self-healing of HIDS Manager and its Sensors

The self-healing of HIDS can be performed in different ways and may involve different failures scenarios, reasons and recovery procedures. In this subsection, we discuss a common procedure to recover sensors and managers nodes.

Fault monitoring is traditionally performed through periodical polling by a network management system. When faults occur, human administrators have to manually perform the healing procedure. The traditional procedure to heal a HIDS manager or its sensors consists in *i*) remotely access the machine; *ii*) verify its latest entries in the system log files; *iii*) determine the cause of the failure or degradation; *iv*) readjust its parameters or develop a new set-up; and finally, *v*) put it on-line. In HIDS manager failure scenario, this is more critical, as some modifications needs to be propagated to all the sensors managed by the faulty manager.

The utilization of a self-healing service brings advantages as the traditional procedures for monitoring and healing HIDS have concerns related to scalability and robustness. First, in a large HIDS system, it would be infeasible for human administrators to deal with a faulty manager that has a high number of sensor nodes associated with itself. Second, the decisions related to how to deal with most faults, usually, do not involve complex analysis and action performing, thus, these faults would be easily healed though simple healing workplans. As minor faults are the most frequent, this would greatly reduce the demands for the attention of the system administrators. Finally, the repeated execution of the same tasks by human resources is proved as error prone.

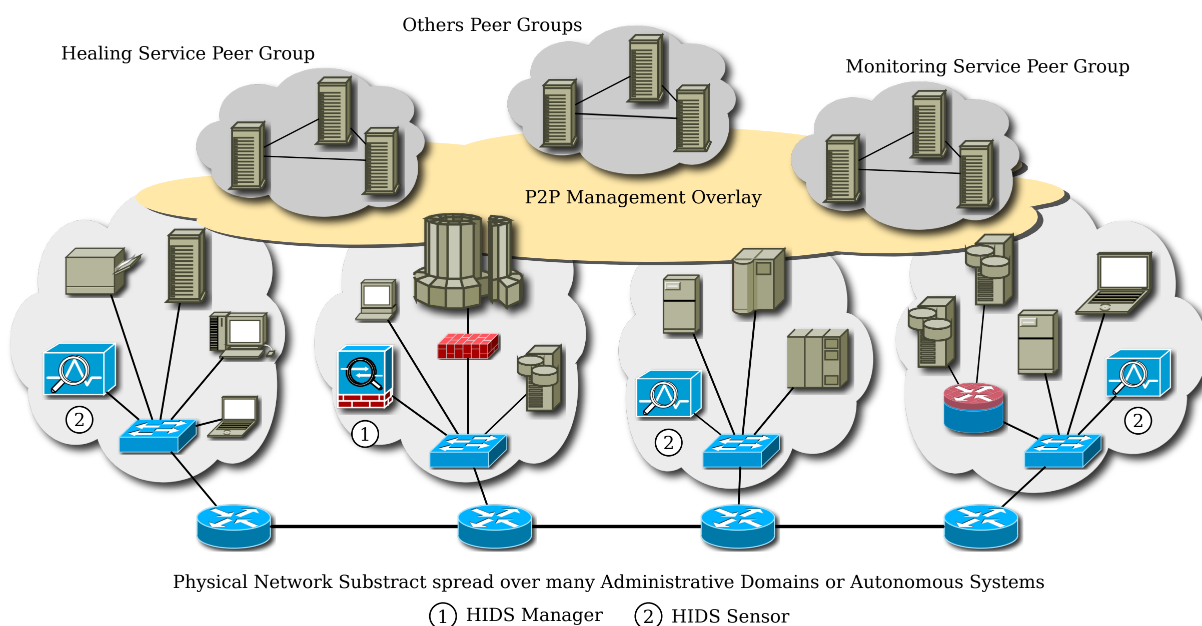
Challenged networks require characteristics other than scalability. However, most of their challenges, *i.e.*, frequent disconnections and network partitioning, can be overcame by P2P approaches. For instance, self-healing service implemented over P2PBNM premises most transparently deals with partitioning by redistributing tasks among remaining peers. Alike, moderate churn rates (*i.e.*, arrival and departure of peers) is gracefully handled by overlay maintenance algorithms.

5.2 First evaluation: conventional networks

In this section, assuming the previous statements about self-healing of HIDS infrastructures, experimental measurements are presented to show the feasibility of the proposed self-healing service when considering its deployment in a conventional network. These experiments aim to measure the *total management traffic* generated throughout the self-healing process and its *average duration time*. The results of the first experiment shows the service impact in the infrastructure while the second presents its performance.

Both experiments were conducted in a management overlay composed of 32 peers, evenly divided into monitoring service peer group and healing service peer group. These peers were instantiated as virtual machines running over two bare-metal hosts. In order to evaluate the relation of the size of the HIDS infrastructure and the parameters observed, the number of sensor nodes vary as 1, 2, 4, and 8. The HIDS deployment was based on OSSEC[1]. All the HIDS infrastructure was instantiated as virtual machines running over a single bare-metal host. The overall evaluation scenario is depicted in Figure 5.1.

Figure 5.1: Evaluation Scenario for conventional networks.



The experiments considered two healing workplan implementations. In the first implementation, the healing workplan is completely executed by the healing service peer that received the unhealthy notification from the monitoring service. In the second implementation, the healing workplan makes heavy use of the management overlay capabilities. Hence, some of workplan's operations involve the use of services provided by other management components deployed at the management overlay. For simplicity, we refer to the former im-

[1]OSSEC - http://www.ossec.net/

plementation as *independent healing workplan execution*, and the last as *cooperative healing workplan execution*. From a practical point of view, the workplans can be considered equal since their operational purposes are identical.

For the cooperative workplan implementation, two peers of each service were also used to implement and provide a Remote Procedure Call (RPC) service used by the healing workplan. Besides these services, overlays' basic services where running.

In the first experiment, the total amount of management traffic generated during the healing process is measured. The objective of this experiment is to show the impact of the self-healing related traffic in the communication network as the number of managed entities grows. Figure 5.2 shows the results of this experiment.

Figure 5.2: Total management traffic during the Self-Healing process.

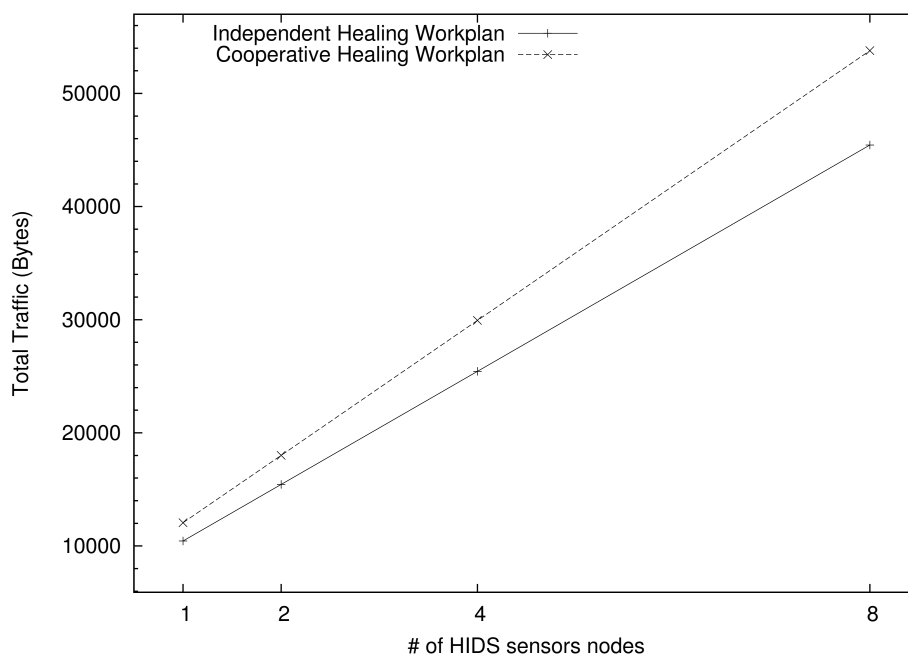


Figure 5.2 shows that, although essentially the same tasks are performed in both scenarios, as both workplans are functionally the same, the independent workplan execution requires less network traffic than the cooperative workplan execution. Inspection of network traces reveals that the differences presented are due to extra data exchange required for the cooperative workplan execution.

During the independent workplan execution, the monitoring service's peer who notes the fault and the healing service's peer who executes the healing workplan are the only peers to exchange data. On the other hand, when collaborative execution is in place, in addition to the regular messages' flow, the healing service's peer and the RPC service peer's must exchange extra messages for procedure invoking and result gathering. This difference was intuitively expected given that synchronization primitives are in place for interprocess coordination.

In the next experiment, the average duration time of the healing process is measured considering both workplan implementations. The motivation of this experiment is to evaluate the impact of the cooperation among the overlay's peers in the time needed to heal managed entities. The results are shown in Figure 5.3.

Figure 5.3: Average duration time of the Self-Healing Process

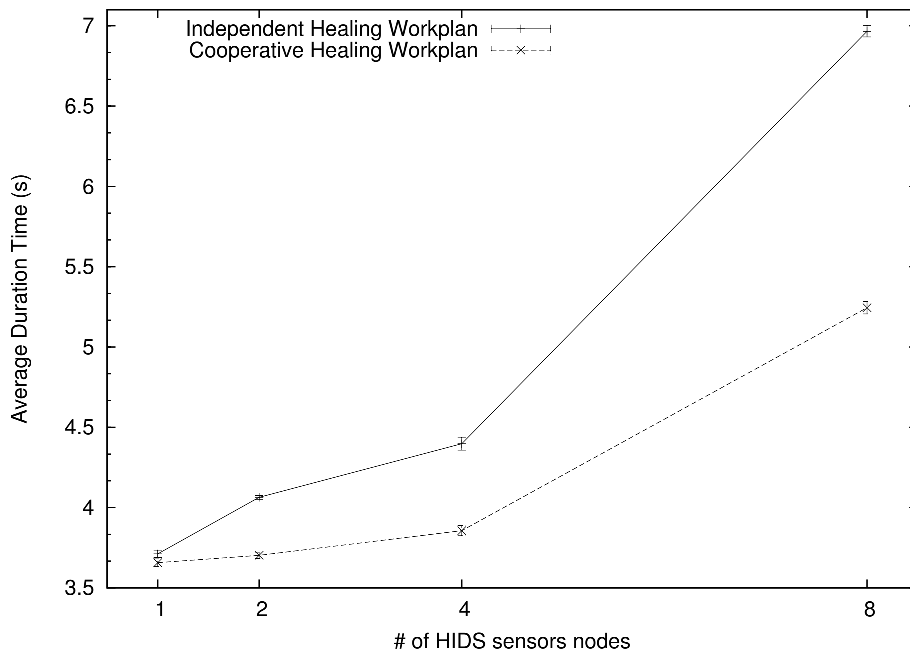


Figure 5.3 shows that the cooperation among peers of the management overlay beneficially impacts the healing duration time. Further analysis shows that this impact is mostly related to the distribution of management tasks and better utilization of management elements' resources.

The independent workplan execution demands resources from a single peer of the healing service. Then, although some tasks run in parallel, the peer responsible for the workplan gets overloaded as the number of HIDS sensors grows. Collaborative workplan execution permits a more fine grained distribution of resource usage, allowing peers to share tasks' workload. Thus, as the management overlay grows and peers implement services necessary to a specific healing procedure, the duration time of this procedure would mostly reduce.

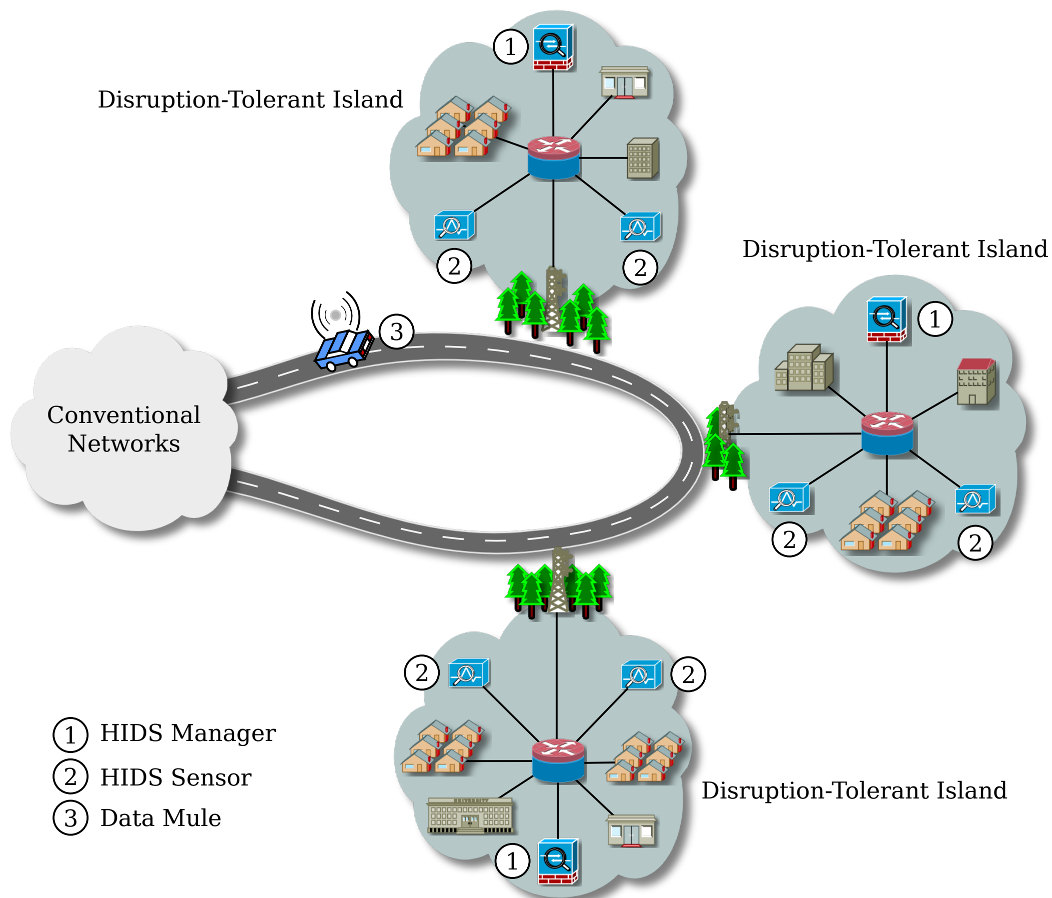
The results shown in Figures 5.2 and 5.3 make explicit the *trade-off* between the amount of traffic generated by the execution of a healing workplan, and the total time it takes. While an independent healing workplan execution is less resource demanding, a cooperative healing workplan execution is most suited for time-critical situations, as the workload it imposes might be shared among the peers of the management overlay.

5.3 Second evaluation: disruption tolerant networks

For this second evaluation, which considers the employment of the proposal in Delay and Disruption Tolerant management scenario, the experimental setup was composed of three DTN islands interconnected by a data mule. The data mule travels through each island always in the same order, from *island #1* to *#2* and then *#3*. A *visit* is a whole round of the mule arriving and leaving all islands to deliver and collect DTN traffic.

Each island hosts its own HIDS infrastructure, which encompass one HIDS manager and eight sensors nodes. Besides, the islands host eight management elements able to both monitor and heal managed entities. Figure 5.4 depicts the overall scenario. The Figure omits the management overlay and the actual HIDS for presentation purposes.

Figure 5.4: Evaluation Scenario for Disruption-Tolerant Networks.



Inside an island, three peers are chosen to monitor the local HIDS infrastructure. Six other peers are picked up to monitor other islands' infrastructures, three peers for each island. Since there are eight peers per island, but nine peers are required to monitor local and remote HIDS, one of the eight original peers ends up responsible to monitor two HIDS

infrastructures. The same strategy of using nine peers is employed for healing purposes. The motivation for performing intra-island monitoring is to provide a location agnostic diagnosis of the problem. For example, a monitoring peer in the very same island would incorrectly state the system as healthy even if it is unreachable by HIDS managers outside the island. Besides that, given the stochastic nature of challenged networks' communications, a monitoring service peer might have missed workplans updates. Therefore, this peer would fail to verify new issues that might be affecting managed entities.

The experiments were performed using a set of two virtual machine (VM) servers and one commodity computer. The first VM server hosted nine VMs (three VMs per DTN island), each one executing one instance of the ManP2P-ng. The second VM server hosted another nine VMs: one for the HIDS manager and other eight for the HIDS sensors, all of them positioned in the third DTN island. Similarly the previous case study, OSSEC was used to realize HIDS deployment. Finally, the third computer was used to simulate the data mule. Basically, this computer runs a ManP2P-ng instance and simulates the travels by performing the bootstrap process with the peers of each island. Moreover, the mule and peers exchange pending data through HTTP-DTN protocol (Section 2.6.1.3).

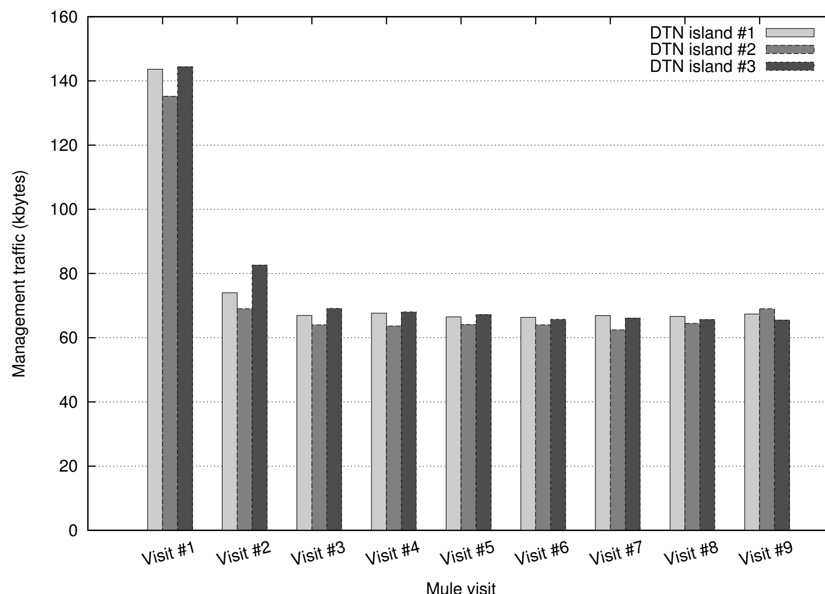
Aiming to understand behaviour and impacts of the proposal in a Delay and Disruption Tolerant Network, three experiments have been carried out. The first experiment measures the management traffic generated while delivering monitoring and healing workplans. The second experiment emulates anomalies and faults at HIDS managers and measures the management traffic related to the detection and notification of these events. The third experiment evaluates the total management traffic required to employ the self-healing service on challenged networks. In order to compare the impact of using HTTP-DTN as the underlying protocol, conventional networks' results are used as baseline.

In the first experiment, the total generated traffic from the data mule to deliver nine different monitoring and nine different healing workplans for nine peers (three in each island) is measured. Given that it is impossible to optimize communication as payloads and destinations completely differ, this task characterizes the worst case scenario for the data mule when concerning workplan delivery. This experiment aims to show the impact of the workplans' distribution in the data mule workload and, consequently, in the network infrastructure.

Figure 5.5 shows the traffic inside each island during the first nine visits in the managed DTN concerning a typical experimental run with no transmission failures among the data mule and island's gateways. In *visit #1*, the management overhead is higher because of the workplans distribution across the management overlay. In *visit #2*, workplan acceptance notifications are still carried around the managed DTN, so the traffic is still considerable. From *visit #3* and on, it is possible to observe that the management traffic reduces significantly (less than 70 *kb* per visit). This remaining traffic consists of regular overlay maintenance and monitoring service's monitoring messages.

In the second experiment, the HIDS manager from *island #3* was forced to go down, emulating, for example, an attack or a non-malicious fault. Thus, following the previous statements about HIDS infrastructures, aside fixing the broken manager node, the eight sensors from *island #3* must be reconfigured also.

Figure 5.5: Monitoring and Healing plans distribution



When a fault emulation occurs, three monitoring service's peers from each island detect the faulty HIDS manager and issue unhealthy notifications to report it. This way, the *unhealthy threshold* is crossed - since it was setted to 6 and then 9 notifications are sent - and the healing service starts executing the healing workplans. Because intra-island monitoring was setted up, a healing service's peer of each island is activated. Although less efficient, permitting many activations avoids the poor timing synchronization presents in challenged environments.

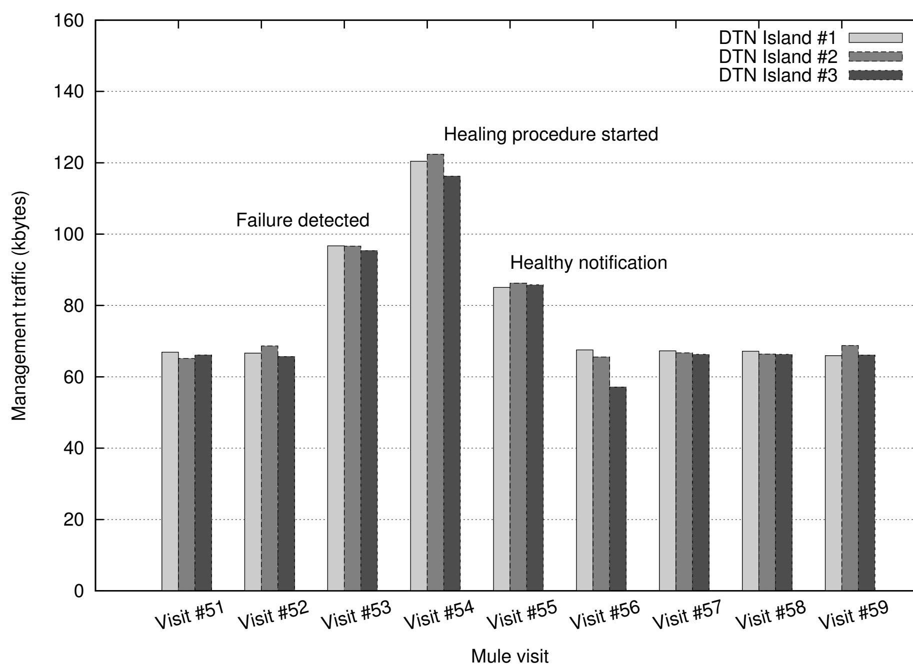
Figure 5.6 presents the impact of the entire self-healing procedure while concerning a typical fault injection experiment. The failure in the HIDS manager is emulated during the mule's *visit #52* and its effects are more perceptible in *visit #53*. The increased traffic at these visits is mainly due to unhealthy notification issuing by the monitoring service's peers of all three islands.

During *visit #54*, the management traffic increases even more because the mule carries remaining unhealthy notifications from the previous visit in addition of carrying action requests resulting from the execution of the healing workplans. In *visit #54* the faulty HIDS manager is recovered and the first *resume monitoring* messages are issued by the healing service's peers from *island #3*. That contributes to the increased traffic of *visit #54* too.

In *visit #55*, the last *resume monitoring* messages are issued. The monitoring service's peers from all three islands then resume their monitoring workplans. From *visit #56* on the healing procedure is complete and then the management traffic returns to normal levels.

In the last experiment, in order to evaluate the relationship between the size of the HIDS infrastructure and the generated management traffic, the number of sensor nodes

Figure 5.6: Management traffic during the self-healing process.



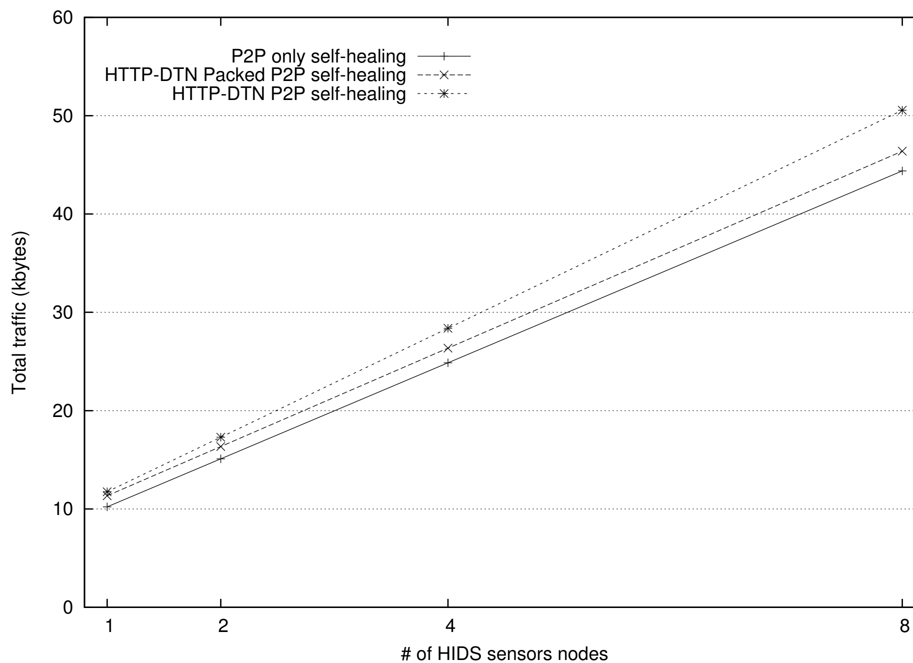
in the HIDS infrastructure is varied from one, two, four, and eight, although only one manager node is still in place. Different than the previous two experiments, the focus is to measure the traffic generated by a single healing service's peer while executing a healing workplan.

This experiment considers two message passing strategies. The first strategy consists in sending individual messages through single HTTP-DTN packages. In the second strategy, HTTP-DTN message packaging is employed; in this case, management messages are grouped into a HTTP-DTN package that is constantly reassembled until the mule visits the DTN island that hosts the healing service's peer. When the mule finally arrives at the island, the message grouping stops and the HTTP-DTN package is forwarded.

Figure 5.7 presents the traffic generated by the healing workplan executed by a single healing service's peer considering a varying number of sensor nodes, as mentioned before. Three curves are depicted: one for P2P healing traffic without employing HTTP-DTN (included just as a baseline since it does not support delay-tolerant management), a second one for HTTP-DTN packages carrying a single management message, and a last one using HTTP-DTN packages carrying several management messages, as discussed in the previous paragraph.

The results presented in Figure 5.7 show that transporting management messages through HTTP-DTN in P2P overlay is feasible since the overhead is not significantly higher than when the same management tasks are performed in a non-DTN environment. Moreover, HTTP-DTN packaging features smooth the overhead growth, which can be seen in the similar angular coefficient of both curves (P2P only self-healing and HTTP-DTN Packed P2P self-healing).

Figure 5.7: Total management traffic during the Self-Healing process.



The results present in this Section show important features of our proposal. First, the traffic produced during workplan distribution and regular monitoring does not impact too much, considering our deployment scenario. Second, despite an increase in management traffic, the healing procedure still keeps an acceptable traffic load. Third, it is also shown that the introduction of HTTP-DTN in P2P overlay does not increases significantly network overhead, specially when using HTTP-DTN message packaging.

5.4 Keystroke-level Model discussion

The Keystroke-level Model, or KLM, is a quantitative tool to predict the time it takes a user to perform a task with a given method on an interactive computer system (CARD; MORAN; NEWELL, 1980). Then, the application of KLM techniques hence enable user interface designers to compare and quantitatively evaluate different implementation proposals or actual implementation themselves. A concise discussion besides with examples of applications of this technique are presented in (KIERAS, 2001).

KLM's basic building-block are operators, standard interactions performed by users on control interfaces. Operators represent actions such as single key presses, mouse's point-and-clicks, and word typing. Using KLM, an interface is then modeled by the juxtaposition of the Operators required to interact with that interface. The output of the model is the average interaction time. The average interaction time is based on the average time users take while performing operators. The average time users take while performing the operation an Operator defines was set trough actual applications' usage trace gathering.

In the context of this discussion, the Keystroke-level Model is used to argue that the proposed self-healing service has advantages over operators reacting to failures on demand. We therefore consider two operations scenarios. The first scenario encompasses a primitive management deployment, where no automatized tool assists managers and administrators. The second scenario encompass most management deployments, where management elements able to collect and process data are in place. For each scenario, two user classes are considered: an experienced system administrator, a user which is mostly familiar with faults in HIDS and is able to rapidly respond to these faults; and a newbie administrator, a user which is getting familiar with HIDS management but is not yet an expert. The evaluation scenarios are illustrated in the next subsections.

The behaviour of systems administrators was derived using real data. Several history of several autonomous systems deployed Brazil-wide were collected and characterized. This characterization then was used to feed developed models. By this mean the analysis presented in this section attempts to better comply reality.

5.4.1 Modeling naïve management scenario

The first operation scenario considers that only basic monitoring facilities are available, such as a central monitoring entity able to contact all hosts in the network and gather information like running processes, and basic performance data, while, however, being completely unable to perform any inference on this data. Besides that, this entity has no capability to react to any event.

In this scenario the monitoring of a Distributed HIDS is considered to be composed of the following tasks:

1. Logging into the system
   - Typing the host name: $T(n)$
   - Typing its username: $T(n)$
   - Typing its password: $T(n)$
2. Gathering root-cause-related data
   - Typing commands: $T(n) \times c$
3. Analyse the cause of missing processes
   - Mental act of routine thinking or perception: $M(m)$

Therefore,

$$monitoring = T(n) \times (3 + c) + M(n) \tag{5.1}$$

Considering then a simple failure scenario where the process died for a trivial misconfiguration reason, the reaction to the failure would be composed of the following actions:

1. Open the configuration file
   - Typing editor and file name: $T(n)$
2. Edit the configuration file:
   - Navigating through lines on modal interface: $K(l)$
   - Editing lines: $T(n)$
3. Run process:
   - Typing commands: $T(n) \times c$

Therefore,

$$reacting = T(n) \times (c + 2) + K(l) \tag{5.2}$$

5.4.2 Modeling common management scenarios

The second scenario, the most conceivable on medium to large network deployment scenarios, a centralized general-purpose monitoring facility is deployed and is able to collect and perform basic inferences, like figuring missing processes, and trespassed performance thresholds. Also, this second entity is able to actively alert administrators by sending e-mails, SMSs, playing sounds, or any other alerting medium it seems plausible. Alike the entity in the naïve scenario, this second one is unable to react to events besides alerting administrators. At this scenario, then, only the reaction to the failure would be performed. The reaction for this scenario follows the same pattern described for the naïve scenario at Equation 5.2.

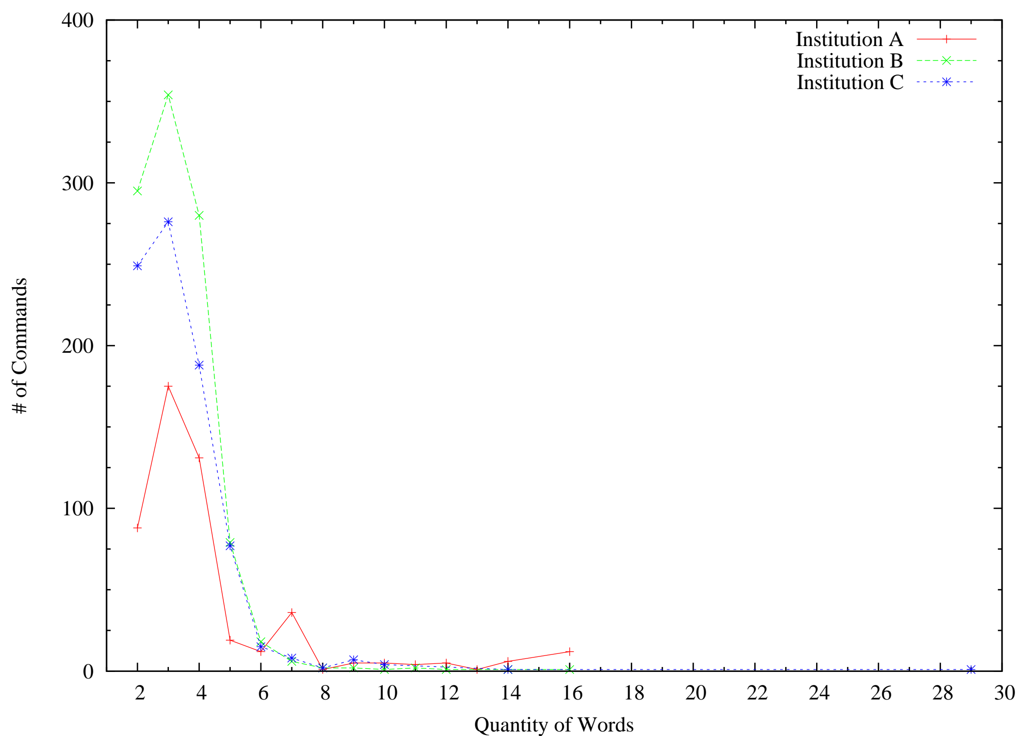
5.4.3 Characterization of Users' Behaviour

To better estimate users' behaviour, CLI traces were collected and analysed. This analysis then gives support for choosing concise values for variables required by KLM analysis. The traces were collected at Federal University of Rio Grande do Sul, State University of Rio Grande do Norte, and Federal Technology Institute of Ceará, besides other autonomous systems that prefer to remain anonymous. The traces contained 8551 commands, which then encompassed 26388 words, and 207142 characters.

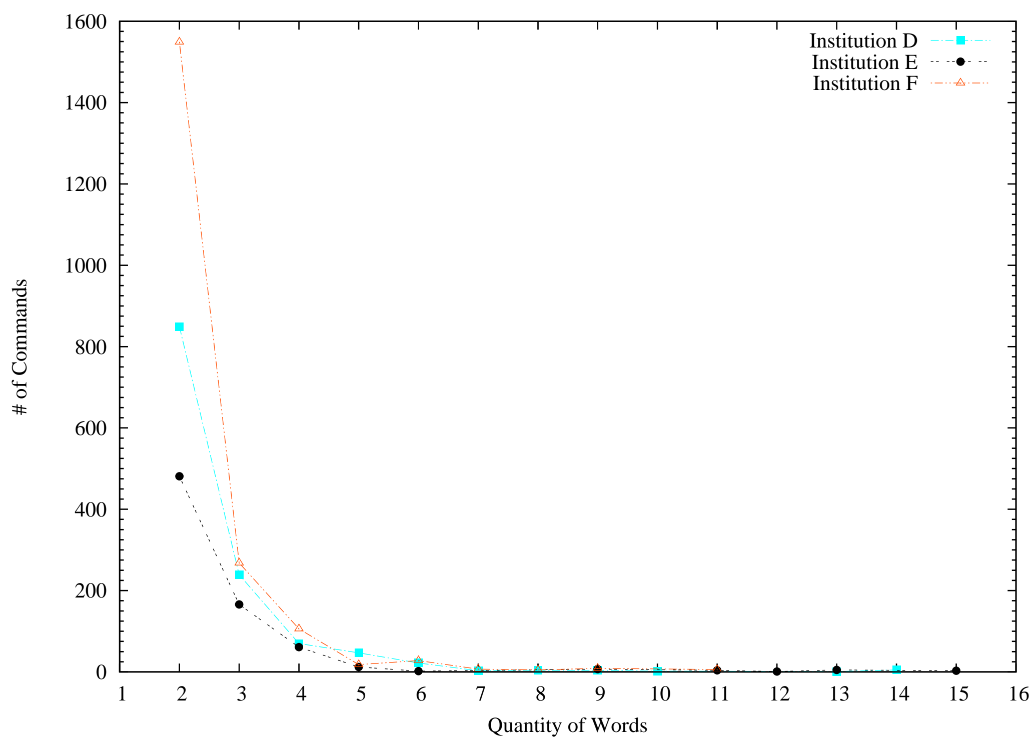
The first step was to verify if users presented similar behaviour. To this end, the quantity of words of each sample were calculated and used to plot the graphs at Figure 5.8 and 5.9. The graphs at Figure 5.8 presents the relation between issued commands and the quantity of words they contain. The graphs at Figure 5.9 presents the relation between the typed words and the quantity of characters they contain.

In both figures, data analysis revealed two patterns. When concerning the quantity of words in a single issued command, Figure 5.9a shows that most commands contains 3 words. Whereas, in Figure 5.9b most commands contains 2 words. All data traces shows similar behaviour when considering 4-words-length commands and onwards.

Figure 5.8: Words per Command.

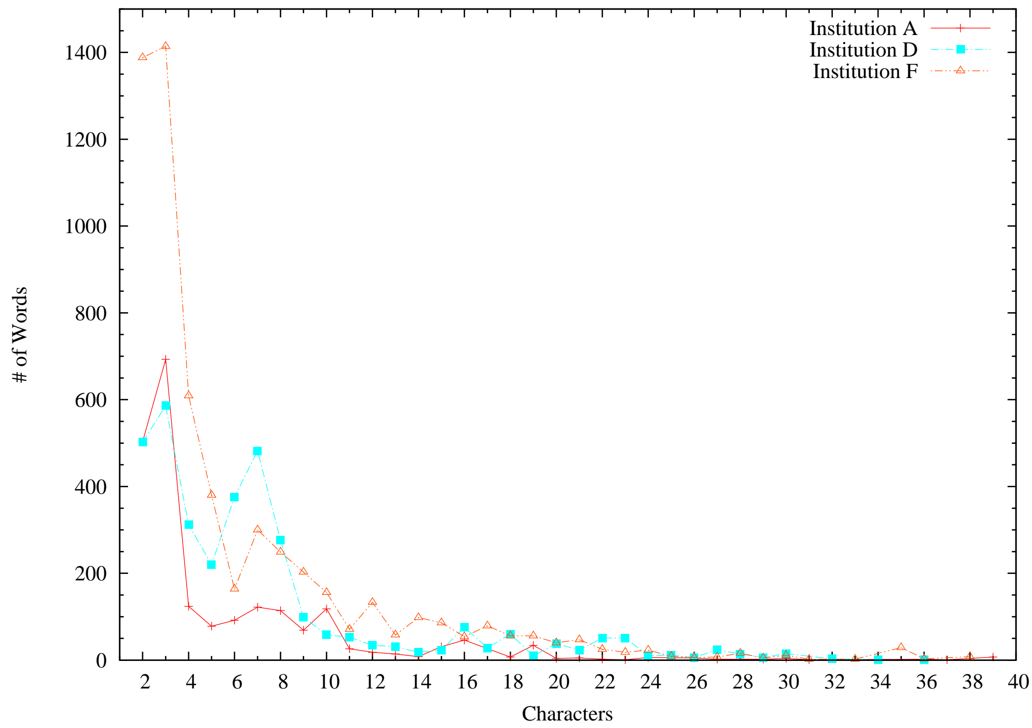


(a) First User-class

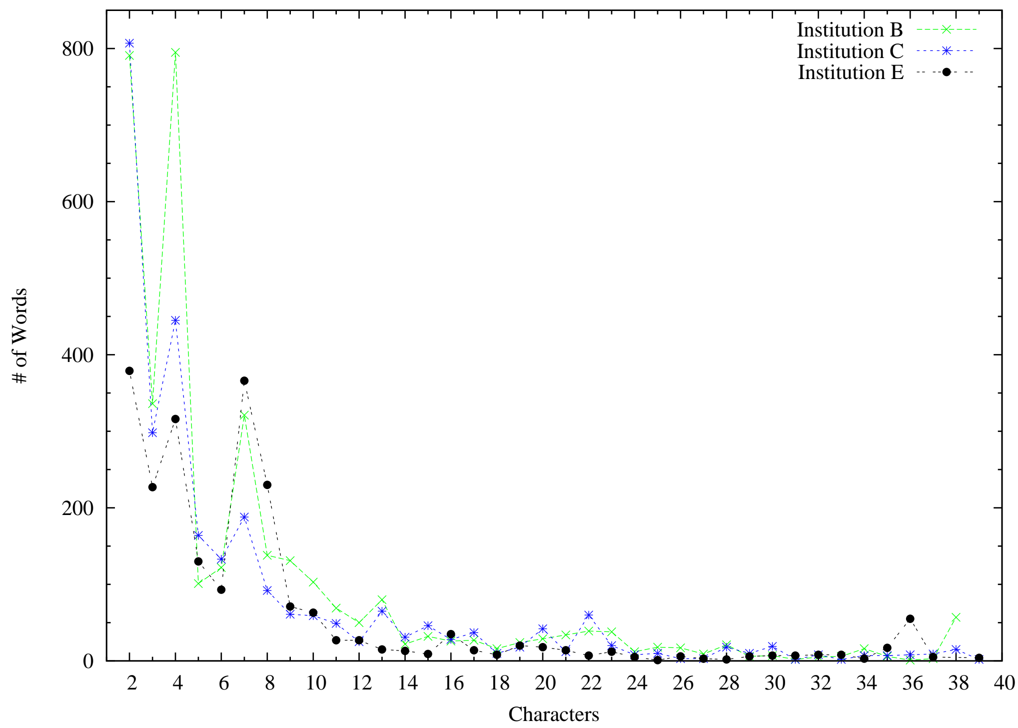


(b) Second User-class

Figure 5.9: Characters per Word.



(a) First User-class



(b) Second User-class

As early stated, when concerning characters per word, there are also two patterns. In Figure 5.10a most words are composed of 2 or 3 characters. Whereas, in Figure 5.10b most words are composed of 2 or 4 characters. From 5 characters onward both graphs behave mostly equal.

The second step was to summarize the data. To this end, the arithmetic mean of values was chosen. This choice seems appropriate for the ongoing analysis given that Figures 5.8 and 5.9 show that differences occurs mostly at the beginning of the series and that these differences are unable to bias the result in a misleading way. Thereupon, the *average word length* will be 6 characters and the *average command length* will be 3 words.

While plotting Figure 5.8, 1-word-length commands were discarded since they are commonly used to assert the successfulness of previous commands. However, 1-word-length commands can be interpreted as thinking time. The traces contain 8551 commands, of which 1714 were 1-word-length. Therefore, by calculating the quotient of these quantities, the *average thinking* will be considered 4.

5.4.4 Comparison of naïve, common, and self-healing scenario

To perform the comparison of the scenarios, an experimental environment was set-up. The experimental environment was composed of eleven machines. Ten machines were 2.22 GHz Pentium Core 2 Quad with Intel VT-x extensions enabled. These ten machines were used to virtualize ten other machines each, which then served as the managed entities. The managed entities were virtualized through User-mode Linux. The remaining machine, a 2.3 GHz Pentium Core 2 Duo, served as the self-healing service instance hosting machine. The network environment was a switched Ethernet network presenting approximately 0.225 milliseconds of round trip time with 0.009 standard deviation.

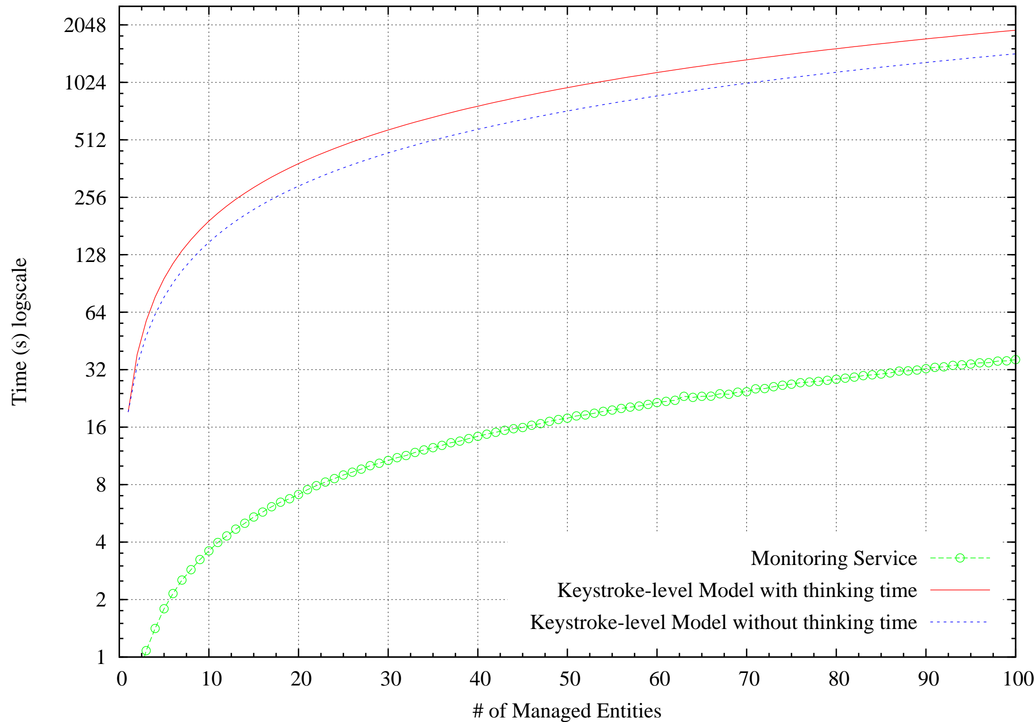
Figure 5.10 presents the plotting of the time a theoretical administrator needs to flawlessly perform the procedure denoted by Equation 5.1 and the plotting of the time that the monitoring service executing an actual implementation of the same equation needs. This figure intends to contrast the time needed by an administrator to monitor a given number of managed entities and the time needed by the proposed monitoring service to achieve the same goal.

To plot the Keystroke-level Model's curve in Figure 5.10, the standard values for typing and thinking (KIERAS, 2001) were applied. Besides that, resting time and network-related delays were ignored. In other words, context switches between tasks and delays presented by most networks are unaccounted, characterizing then the best-case scenario.

However, to plot monitoring service results, raw experimental data was considered and, therefore, all delay is accounted. For this reason, it is clear that using the monitoring service yields productivity given that, while an administrator handles 5 managed entities, the monitoring service has already handled 100 managed entities.

By regarding real network management scenarios, it might be arguable that the thinking time present at Equation 5.1 may disappear or mostly fade since, after some executions, administrators will be very familiar with tasks they are performing, becoming hence less thoughtful. This is the expected behaviour that emerges with experience.

Figure 5.10: Plot of Keystroke-level Model of Equation 5.1 and experimental data.



On the other hand, it is also arguable that the thinking time might return because of inconsistencies among managed entities configurations, or as eventual errors that occur during recurring tasks. Notwithstanding both arguments, Figure 5.10 already encompass these assumptions and shows that differences are insignificant when compared to the monitoring service performance.

Figure 5.11 presents the plotting of the time a theoretical administrator needs to flawlessly perform the procedure denoted by Equation 5.2 and the plotting of the time that the healing service executing an actual implementation of the same equation needs. This figure aims to explicit disparities of the time an administrator takes to recover a growing number of managed entities and the time that the healing service takes to achieve the same goal.

To plot the Keystroke-level Model's curve in Figure 5.11, standard values for an experienced typist (KIERAS, 2001) were applied. Alike the previous analysis, resting time and network-related delays were ignored. Also, Equation 5.2 considers that the administrator already knows exactly what to do, since administrator had it all analyzed during the monitoring, and hence exhibits the best-case behaviour.

Once more, healing service's plots are raw experimental data and therefore encompass all delays presented by the implementation and by the network environment it interfaces. Besides encompassing the serial execution of tasks, Figure 5.11 also encompasses parallel execution-related data.

Figure 5.11: Plot of Keystroke-level Model of Equation 5.2 and experimental data.

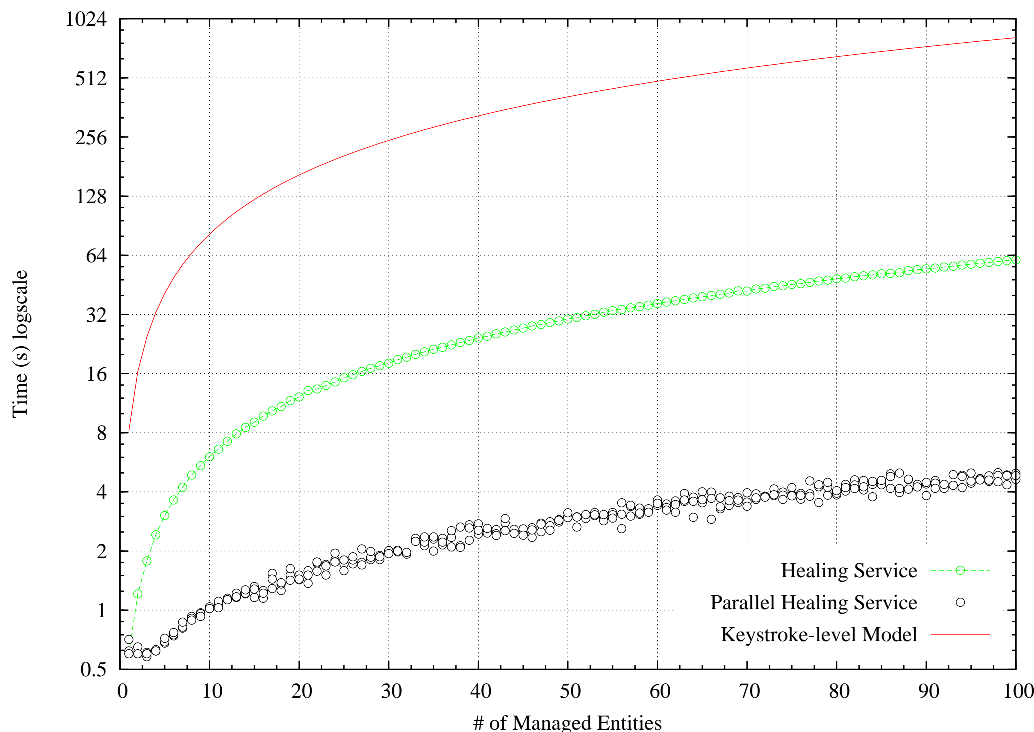


Figure 5.11 shows the same discrepancies previously observed when comparing administrators behaviour and the monitoring service. Moreover, this same figure also appraises insights primarily gathered at Figure 5.3: management resources availability largely impacts self-healing related tasks because of the workload sharing of management tasks that are inherently present in P2P-based Network Management solutions.

Figures 5.10 and 5.11 mostly explicit that human-based or human-assisted management is unfeasible for even small networking scenario. For instance, considering the naïve scenario, results show that a theoretical flawless administrator, working in a theoretically perfect computer networking environment would required at least 3 minutes to manage a network composed of 10 managed elements. Awhile, results show the self-healing service required approximately 10 seconds to manage such a management infrastructure.

By considering monitoring infrastructures present in most common medium to large scale networks, human-assisted management improves. Still, the self-healing service continues to exhibit advantages. For example, 100 management elements are handled in approximately 1 minute and 40 seconds when tasks are serially executed and less than 45 seconds when workload is shared among services instances. These results, however, consider that the monitoring service will analyse all managed elements before contacting the healing service. An administrator would need at least 13 minutes given that interpreting the monitoring infrastructure would be necessary and then take time.

6 CONCLUSIONS

Networking applications are changing and so must do management techniques if the latter are to be up to date with the former. Time has come to invest in techniques and tools that unburden administrators from repetitive and recurrent tasks, hence enabling then to focus on intelligently applying resources and efforts in order to achieve high-levels business' goals and objectives.

Among other proposals, Autonomic Computing and Autonomic Network Management highlight themselves by arguing that only by combining cutting edge and novel monitoring, analysis, planning, acting and knowledge gathering strategies will any management technique overcome challenges and truly leverage human resources' potential.

Autonomic Network management proposals are based on four properties, widespread known as self-* properties. These properties are self-configuration, the ability of a system to configure itself on changes; self-optimization, the ability of a system to optimize its own operation to cope to environmental specificities; self-protection, the ability of a system to deal with known and unknown threats; and self-healing, the ability of a system to diagnose and react to anomalies and faults.

The last of the aforementioned properties, self-healing property, gained researchers attention due to the known impact that the fully development of this property would show when concerning systems' maintenance costs and overall dependability. Therefore, many proposals were developed in order to fulfil this gap.

Garlan & Schmerl (2002) proposed modeling systems through interactions graphs and using code annotations so systems would be monitored and anomalies would be eventually healed. Although adaptable – since it allows total swapping of its models in runtime, thus adjusting itself to new scenario – this proposal is mostly unfeasible given the difficulty to model current systems' complexities. Besides that, model-based proposals suit themselves only when applications provide runtime monitoring and tunning interfaces. Although monitoring can be achieved through *ad-hoc* mechanisms, runtime tunning is not so straightforward.

Breitgand *et al.* (2007) presents PANACEA, a development framework devoted to self-healing enabled applications' development. PANACEA proposes developers to instrument their code with self-healing primitives. These primitives then unfold into runtime monitoring and tuning capabilities. Albeit providing novel capabilities, PANACEA bases

itself over well-known programming primitives, code annotations. PANACEA main drawback is its lack of coordination interfaces, which are essential while considering distributed applications deployment scenarios. Moreover, PANACEA requires access to source code. Therefore, it might be impossible to apply it in many networking scenarios.

Fuad & Oudshoorn (2007) propose to analyse applications bytecode at runtime in order to identify autonomic entities and to attach monitoring and healing capabilities to these entities. Therefore, this proposal is applicable to interpreted languages-based systems for which no source code is available or no further development effort is feasible. Regardless, a distributed coordination mechanism still missing besides proposal's impact on systems' performance.

Notwithstanding all current development and their foundations, it is clear that most applications that target at solving conventional networking-related problems will need adaption in order to cope with novel and challenging network scenarios. As early stated, many networking scenarios do not share the characteristics and soft constraints that current applications are built upon, like high connectivity and low delays. Known as Delay and Disruption Tolerant Networks, these networks have gather wide attention due to interest of players like NASA and Google (BURLEIGH et al., 2013) and due to the possibility to cover areas where current approaches seems to fail to cover. Given the aforementioned, it is essential that any network management solution that expects to cope current challenges and to softly adapt to technological shifts addresses issues like frequent connection disruption and communications with high delays and high fail rates.

Considering classical and novel network management challenges related to the maintenance of infrastructures and services, this dissertation proposes a distributed self-healing service with generic interfaces to enable developers and systems administrators to customize healing activities according to the services their infrastructures provides. Besides that, the self-healing service proposed provides standardized communication interfaces so distributed management elements may coordinate their execution.

Also, this dissertation proposes, executes, and discusses the results of an experimental evaluation of the self-healing service it propose. The evaluations aimed to show the feasibility of the proposal when considered from communication overhead and mean time to recover perspectives. The evaluation encompassed both classical and challenged networks.

Albeit the results, many self-healing-related challenges still need further investigation. For instance, efficient autonomic elements' coordination is not directly tackled in the present work and therefore the mechanism proposed can be said suited only in the environments addressed by the experimental evaluation.

Besides that, workplans are proposed as gatherers for Event-Rule-Action scenarios and also as a mean to abstract coarse-grained and commonplace concepts such as device accessing, process handling, and configuration editing. Environments' specificities are expected to demand more representativeness and abstraction power than what is proposed here. Consequently, the suitability of Ponder2-based workplans must be delve into a greater degree.

REFERENCES

AL-ZAWI, M. et al. Using Adaptive Neural Networks in Self-Healing Systems. In: SECOND INTERNATIONAL CONFERENCE ON DEVELOPMENTS IN ESYSTEMS ENGINEERING (DESE) 2009, 2009. **Proceedings...** [S.l.: s.n.], 2009. p.227 –232.

BENAMAR, N. et al. Routing protocols in Vehicular Delay Tolerant Networks: a comprehensive survey. **Computer Communications**, [S.l.], Apr. 2014.

BERNS, A.; GHOSH, S. Dissecting Self-* Properties. In: THIRD IEEE INTERNATIONAL CONFERENCE ON SELF-ADAPTIVE AND SELF-ORGANIZING SYSTEMS, 2009., 2009, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2009. p.10–19.

BIRRANE III, E. J.; BURLEIGH, S. C.; CERF, V. Defining tolerance: impacts of delay and disruption when managing challenged networks. In: AIAA INFOTECH@ AEROSPACE CONFERENCE, 2011. **Proceedings...** [S.l.: s.n.], 2011.

BREITGAND, D. et al. PANACEA Towards a Self-healing Development Framework. In: IFIP/IEEE SYMPOSIUM ON INTEGRATED MANAGEMENT, 10., 2007. **Proceedings...** [S.l.: s.n.], 2007. p.169–178.

BURGESS, J. et al. MaxProp: routing for vehicle-based disruption-tolerant networks. In: IEEE INTERNATIONAL CONFERENCE ON COMPUTER COMMUNICATIONS, 25., 2006. **Proceedings...** [S.l.: s.n.], 2006. p.1–11.

BURLEIGH, S. C. et al. **NASA's Disruption Tolerant Networking Challenge Series**. [S.l.]: NASA Tournament Lab, 2013.

BURLEIGH, S. et al. Delay-tolerant networking: an approach to interplanetary internet. **Communications Magazine, IEEE**, [S.l.], v.41, n.6, p.128–136, June 2003.

CARD, S. K.; MORAN, T. P.; NEWELL, A. The Keystroke-level Model for User Performance Time with Interactive Systems. **Commun. ACM**, New York, NY, USA, v.23, n.7, p.396–410, July 1980.

CERF, V. et al. **Delay-Tolerant Networking Architecture**. [S.l.]: IETF, 2007. n.4838. (Request for Comments).

CHENG, S.-W. et al. Software Architecture-Based Adaptation for Grid Computing. In: IEEE INTERNATIONAL SYMPOSIUM ON HIGH PERFORMANCE DISTRIBUTED COMPUTING, 11., 2002, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2002. p.389.

CLARK, G.; KRUSE, H.; OSTERMANN, S. **DING Protocol – A Protocol For Network Management**. [S.l.]: IETF, 2010. n.draft-irtf-dtnrg-ding-network-management-02. (Internet-Draft).

DEBAR, H.; DACIER, M.; WESPI, A. Towards a taxonomy of intrusion-detection systems. **Computer Networks**, [S.l.], v.31, n.8, p.805 – 822, 1999.

DUARTE, P. A. P. R. et al. A P2P-Based Self-Healing Service for Network Maintenance. In: IFIP/IEEE INTERNATIONAL SYMPOSIUM ON INTEGRATED NETWORK MANAGEMENT (IM 2011), DUBLIN, IRELAND. MAY 2011., 12., 2011. **Proceedings...** [S.l.: s.n.], 2011.

FALL, K. A Delay-tolerant Network Architecture for Challenged Internets. In: CONFERENCE ON APPLICATIONS, TECHNOLOGIES, ARCHITECTURES, AND PROTOCOLS FOR COMPUTER COMMUNICATIONS, 2003., 2003, New York, NY, USA. **Proceedings...** ACM, 2003. p.27–34. (SIGCOMM '03).

FARRELL, S.; CAHILL, V. **Delay- and Disruption-Tolerant Networking**. Norwood, MA, USA: Artech House, Inc., 2006.

FOX, A.; KICIMAN, E.; PATTERSON, D. Combining statistical monitoring and predictable recovery for self-management. In: ACM SIGSOFT WORKSHOP ON SELF-MANAGED SYSTEMS, 1., 2004, New York, NY, USA. **Proceedings...** ACM, 2004. p.49–53.

FUAD, M. M.; OUDSHOORN, M. J. Transformation of Existing Programs into Autonomic and Self-healing Entities. In: ANNUAL IEEE INTERNATIONAL CONFERENCE AND WORKSHOPS ON THE ENGINEERING OF COMPUTER-BASED SYSTEMS, 14., 2007, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2007. p.133–144.

FUNG, C. J. et al. Bayesian decision aggregation in collaborative intrusion detection networks. In: IEEE NETWORK OPERATIONS AND MANAGEMENT SYMPOSIUM (NOMS), 2010 IEEE, 2010. **Proceedings...** [S.l.: s.n.], 2010. p.349 –356.

FUNIKA, W. et al. Towards Role-Based Self-healing in Autonomous Monitoring Systems. In: INTERNATIONAL CONFERENCE ON COMPLEX, INTELLIGENT AND SOFTWARE INTENSIVE SYSTEMS, 2010., 2010, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2010. p.1063–1068.

FUNIKA, W.; PEEGIEL, P. A Role-Based Approach to Self-healing in Autonomous Monitoring Systems. In: WYRZYKOWSKI, R. et al. (Ed.). **Parallel Processing and Applied Mathematics**. [S.l.]: Springer Berlin / Heidelberg, 2010. p.125–134. (Lecture Notes in Computer Science, v.6068).

GARLAN, D.; SCHMERL, B. Model-based adaptation for self-healing systems. In: SELF-HEALING SYSTEMS, 2002, New York, NY, USA. **Proceedings...** ACM, 2002. p.27–32.

GHOSH, D. et al. Self-healing systems - survey and synthesis. **Decis. Support Syst.**, [S.l.], v.42, n.4, p.2164–2185, Jan. 2007.

GHOSH, D. et al. Self-healing systems - survey and synthesis. **Decis. Support Syst.**, Amsterdam, The Netherlands, The Netherlands, v.42, n.4, p.2164–2185, 2007.

GOLDSZMIDT, G.; YEMINI, Y. Distributed management by delegation. In: INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTING SYSTEMS, 15., 1995. **Proceedings...** [S.l.: s.n.], 1995. p.333–340.

GRANVILLE, L. Z. et al. Managing computer networks using peer-to-peer technologies. **Communications Magazine, IEEE**, [S.l.], v.43, n.10, p.62–68, 2005.

HAIGHT, C. DevOps: the changing nature of system administration. **Gartner Research, ID**, [S.l.], n.G00211703, 2011.

HAYDARLOU, A. R.; OVEREINDER, B.; BRAZIER, F. M. T. A Self-Healing Approach for Object-Oriented Applications. In: INTERNATIONAL WORKSHOP ON DATABASE AND EXPERT SYSTEMS APPLICATIONS, 16., 2005. **Proceedings...** [S.l.: s.n.], 2005. p.191–195.

HUEBSCHER, M. C.; MCCANN, J. A. A survey of autonomic computing—degrees, models, and applications. **ACM Comput. Surv.**, New York, NY, USA, v.40, n.3, p.1–28, 2008.

HUNT, P. et al. ZooKeeper: wait-free coordination for internet-scale systems. In: USENIX CONFERENCE ON USENIX ANNUAL TECHNICAL CONFERENCE, 2010., 2010. **Proceedings...** [S.l.: s.n.], 2010. v.8, p.11–11.

ISENTO, J. N. et al. Moni4VDTN: a monitoring system for vehicular delay-tolerant networks. In: IEEE INTERNATIONAL CONFERENCE ON COMMUNICATIONS (ICC), 2012., 2012. **Proceedings...** [S.l.: s.n.], 2012. p.1188–1192.

IVANCIC, W. **DTN - Network Management Requirements**. [S.l.]: IETF, 2009. n.draft-ivancic-dtnrg-network-management-reqs-00. (Internet Draft).

JENNINGS, B. et al. Towards autonomic management of communications networks. **Communications Magazine, IEEE**, [S.l.], v.45, n.10, p.112 –121, october 2007.

KE, J.-K.; YANG, C.-H.; AHN, T.-N. Using w3af to achieve automated penetration testing by live DVD/live USB. In: INTERNATIONAL CONFERENCE ON HYBRID INFORMATION TECHNOLOGY, 2009., 2009, New York, NY, USA. **Proceedings...** ACM, 2009. p.460–464. (ICHIT '09).

KEPHART, J. O. Research challenges of autonomic computing. In: SOFTWARE ENGINEERING, 27., 2005, New York, NY, USA. **Proceedings...** ACM, 2005. p.15–22.

KEPHART, J. O.; CHESS, D. M. The Vision of Autonomic Computing. **Computer**, Los Alamitos, CA, USA, v.36, n.1, p.41–50, 2003.

KIERAS, D. **Using the keystroke-level model to estimate execution times**. [S.l.]: University of Michigan, 2001.

LEFKOWITZ, G.; SHTULL-TRAURING, I. Network Programming for the Rest of Us. In: USENIX ANNUAL TECHNICAL CONFERENCE, FREENIX TRACK, 2003., 2003. **Proceedings...** [S.l.: s.n.], 2003. p.77–89.

LIU, Y. et al. A model-based approach to adding autonomic capabilities to network fault management system. In: IEEE NETWORK OPERATIONS AND MANAGEMENT SYMPOSIUM, 2008. **Proceedings...** [S.l.: s.n.], 2008. p.859 –862.

LIU, Y. et al. A Case Study: a model-based approach to retrofit a network fault management system with self-healing functionality. In: ANNUAL IEEE INTERNATIONAL CONFERENCE AND WORKSHOP ON THE ENGINEERING OF COMPUTER BASED SYSTEMS, 15., 2008. **Proceedings...** [S.l.: s.n.], 2008. p.9 –18.

MARQUEZAN, C. C. et al. Self-managed Services over a P2P-based Network Management Overlay. In: LATIN AMERICAN AUTONOMIC COMPUTING SYMPOSIUM (LAACS 2007), 2., 2007. **Proceedings...** [S.l.: s.n.], 2007.

MARQUEZAN, C. C. et al. Distributed autonomic resource management for network virtualization. In: IEEE NETWORK OPERATIONS AND MANAGEMENT SYMPOSIUM, 2010., 2010. **Proceedings...** [S.l.: s.n.], 2010. p.463 –470.

MESHKOVA, E. et al. A survey on resource discovery mechanisms, peer-to-peer and service discovery frameworks. **Computer networks**, [S.l.], v.52, n.11, p.2097–2128, 2008.

MILLER, F. P.; VANDOME, A. F.; MCBREWSTER, J. **Architecture Description Language**: software engineering, enterprise modelling, programming language, software architecture, systems architecture, technical architecture, carnegie mellon university. [S.l.]: Alpha Press, 2010.

NOBRE, J. C. et al. Self-* Properties and P2P Technology on Disruption-Tolerant Management. In: IEEE SYMPOSIUM ON COMPUTERS AND COMMUNICATIONS (ISCC 2013), SPLIT, CROATIA. JULY 2013., 18., 2013. **Proceedings...** [S.l.: s.n.], 2013.

NOBRE, J. C. et al. On using P2P Technology to Enable Opportunistic Management in DTNs through Statistical Estimation. In: IEEE INTERNATIONAL CONFERENCE ON COMMUNICATIONS (ICC 2014), 2014. **Proceedings...** [S.l.: s.n.], 2014.

PARASHAR, M.; HARIRI, S. Autonomic computing: an overview. In: INTERNATIONAL WORKSHOP ON UNCONVENTIONAL PROGRAMMING PARADIGMS, 2004., 2005. **Proceedings...** Springer Verlag, 2005. p.247–259.

PATTERSON, D. et al. **Recovery-oriented computing (ROC)**: motivation, definition, techniques, and case studies. [S.l.]: Citeseer, 2002.

PEOPLES, C. et al. Context-aware policy-based framework for self-management in delay-tolerant networks: a case study for deep space exploration. **Communications Magazine, IEEE**, [S.l.], v.48, n.7, p.102–109, 2010.

RAMIREZ-SILVA, E.; DACIER, M. Empirical Study of the Impact of Metasploit-Related Attacks in 4 Years of Attack Traces. In: CERVESATO, I. (Ed.). **Advances in Computer Science – ASIAN 2007. Computer and Network Security**. [S.l.]: Springer Berlin Heidelberg, 2007. p.198–211. (Lecture Notes in Computer Science, v.4846).

RIVIèRE, E. et al. Compositional gossip: a conceptual architecture for designing gossip-based applications. **SIGOPS Oper. Syst. Rev.**, New York, NY, USA, v.41, n.5, p.43–50, Oct. 2007.

ROTT, A. Self-Healing in Distributed Network Environments. In: INTERNATIONAL CONFERENCE ON ADVANCED INFORMATION NETWORKING AND APPLICATIONS WORKSHOPS, 21., 2007, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2007. p.73–78.

SALEHIE, M.; TAHVILDARI, L. Autonomic computing: emerging trends and open problems. **SIGSOFT Softw. Eng. Notes**, New York, NY, USA, v.30, n.4, p.1–7, 2005.

SCHANNE, M.; GELHAUSEN, T.; TICHY, W. Adding autonomic functionality to object-oriented applications. In: INTERNATIONAL WORKSHOP ON DATABASE AND EXPERT SYSTEMS APPLICATIONS, 14., 2003. **Proceedings...** [S.l.: s.n.], 2003. p.725–730.

SCHONWALDER, J.; QUITTEK, J.; KAPPLER, C. Building distributed management applications with the IETF Script MIB. **Selected Areas in Communications, IEEE Journal on**, [S.l.], v.18, n.5, p.702–714, May 2000.

SCOTT, K.; BURLEIGH, S. **Bundle Protocol Specification**. [S.l.]: IETF, 2007. n.5050. (Request for Comments).

SHAW, M. "Self-healing": softening precision to avoid brittleness: position paper for woss '02: workshop on self-healing systems. In: SELF-HEALING SYSTEMS, 2002, New York, NY, USA. **Proceedings...** ACM, 2002. p.111–114.

STAVROU, A.; RUBENSTEIN, D.; SAHU, S. A lightweight, robust p2p system to handle flash crowds. In: IEEE INTERNATIONAL CONFERENCE ON NETWORK PROTOCOLS, 10., 2002. **Proceedings...** [S.l.: s.n.], 2002. p.226–235.

STERRITT, R. Autonomic networks: engineering the self-healing property. **Eng. Appl. Artif. Intell.**, Tarrytown, NY, USA, v.17, n.7, p.727–739, 2004.

STERRITT, R. et al. A concise introduction to autonomic computing. **Adv. Eng. Inform.**, Amsterdam, The Netherlands, The Netherlands, v.19, n.3, p.181–187, 2005.

TWIDLE, K. et al. Ponder2: a policy system for autonomous pervasive environments. In: INTERNATIONAL CONFERENCE ON AUTONOMIC AND AUTONOMOUS SYSTEMS, 5., 2009. **Proceedings...** [S.l.: s.n.], 2009. p.330–335.

VANROSSUM, G.; DRAKE, F. L. **The Python Language Reference**. [S.l.]: Python Software Foundation, 2010.

VOULGARIS, S.; GAVIDIA, D.; STEEN, M. CYCLON: inexpensive membership management for unstructured p2p overlays. **Journal of Network and Systems Management**, [S.l.], v.13, n.2, p.197–217, 2005.

VOYIATZIS, A. A survey of delay-and disruption-tolerant networking applications. **Journal of Internet engineering**, [S.l.], v.5, n.1, 2012.

WOOD, L.; EDDY, W. M.; HOLLIDAY, P. A bundle of problems. In: IEEE AEROSPACE CONFERENCE, 2009., 2009. **Proceedings...** IEEE, 2009. p.1–17.

WOOD, L. et al. Use of the delay-tolerant networking bundle protocol from space. In: ASTRONAUTICAL CONGRESS, GLASGOW. IAC, 59., 2008. **Proceedings...** [S.l.: s.n.], 2008.

WOOD, L. et al. Saratoga: a scalable file transfer protocol. **work in progress as an internet-draft, draft-wood-tsvwg-saratoga-11**, [S.l.], 2012.