ΑΛΕΞΑΝΔΡΕΙΟ Τ.Ε.Ι. ΘΕΣΣΑΛΟΝΙΚΗΣ
ΣΧΟΛΗ ΤΕΧΝΟΛΟΓΙΚΩΝ ΕΦΑΡΜΟΓΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

BSc Thesis:

Self-Healing in Internet of Things

Student

Petros Stergioulas

Reg. Number: 123904

Supervisor

Christos Ilioudis

Thessaloniki, Greece 2018

Περίληψη

Στόχος της παρούσας πτυχιακής εργασίας ήταν η μελέτη της ιστορίας, των βασικών χαρακτηριστικών, καθώς και των χρήσεων του self-healing – συγκεκριμένα, σε σχέση με το IoT. To self-healing έγινε γνωστό, αρχικά, ως μία από τις ιδιότητες των αυτόνομων συστημάτων. Πολλά χρόνια έχουν περάσει από τότε, και το self-healing έχει γίνει, πλέον, ανεξάρτητο ενώ εξακολουθεί να εξελίσσεται μέχρι και σήμερα. Για αυτό και κρίναμε σημαντικό να μελετήσουμε και να συγκρίνουμε διαφορετικές, γνωστές αρχιτεκτονικές που χρησιμοποιούνται στον τομέα του Self-healing στο IoT. Αυτό που καταλάβαμε ήταν ότι κάθε αρχιτεκτονική διαφέρει και μπορεί να είναι χρήσιμη σε διαφορετικές περιστάσεις και με διαφορετικούς τρόπους. Δεν υπάρχει μία και μοναδική «τέλεια» αρχιτεκτονική, καθώς όλες έχουν τους περιορισμούς τους, αλλά κάθε μια από αυτές είναι κατάλληλη για να εντοπίζει και να αντιμετωπίζει διαφορετικά προβλήματα. Επομένως, μπορούμε με βεβαιότητα να πούμε πως το Self-healing, ανεξαρτήτως μορφής, δύναται να να ενισχύσει το IoT, από άποψη ασφάλειας, ανθεκτικότητας και αποδοτικότητας.

Abstract

The goal of this thesis was to study the history, the main features, as well as the uses of Self-healing – specifically those in IoT. Self-healing was firstly introduced as a property of autonomic systems. Many years have passed since then, and Self-healing has become independent and is, still, making progress today. For that purpose, we deemed worthwhile to examine and compare several well-known architectures that are being used in the field of Self-healing in IoT. What we understood was that every architecture differs and can be useful in different situations and ways. There isn't a single perfect architecture, since they all show limitations, but each one is suitable for detecting and dealing with different problems. Thus, we can safely tell that Self-healing, in all of its forms, will help IoT level up in terms of security, durability and efficiency.

Index

Περίληψη....2

Abstract....3

Introduction....10

Section 1: Autonomic Systems....12

1.1 Self-Management....13

1.2 Self-Configuration....13

1.3 Self-Optimization....13

1.4 Self-Healing....14

1.5 Self-Protection....14

Section 2: A Brief History....16

Section 3: Self-healing....20

3.1 Self-healing loop....21

3.1.1 Autonomic Manager....23

3.1.2 Monitor....24

3.1.3 Analyze....24

3.1.4 Plan....24

3.1.5 Execute....25

3.1.6 Knowledge....25

3.2 Self-healing states....25

3.2.1 Maintenance of Health....26

3.2.1.1 Maintaining redundancy....27

3.2.1.2 Maintaining by probing....28

3.2.1.3 System monitoring architecture model....29

3.2.1.4 Diversity in system..32
3.2.1.5 Performance log analysis...33
3.2.2 Detection of System Failure...36
3.2.2.1 Something amiss...36
3.2.2.2 System monitoring model..37
3.2.2.3 Notification of foreign element...38
3.2.3 System Recovery..39
3.2.3.1 Redundancy techniques for healing...39
3.2.3.2 Architecture models and repair strategies..................................40
3.2.3.3 Voting methods for healing/Byzantine agreement.......................44
3.3 Self-healing policies...45
3.3.1 The Unified Framework...45
3.4 Failure classification...49
3.5 Self-healing applications...52
3.5.1 Grid computing...52
3.5.2 Software agent-based self-healing architecture...............................53
3.5.3 Distributed Wireless File Service application...................................53
3.5.4 Service discovery systems...53
3.5.5 Reflective middleware...54
3.5.5.1 dynamicTAO...54
3.5.5.2 Open ORB..55
3.5.5.3 Interceptor-based approach...55
3.5.6 GRACE approach..56
3.5.7 Clustering...56

Section 4: Self-healing in IoT....58
4.1 Autonomic Wireless Sensor Networks....59
4.2 Architectures....61
4.2.1 Service Management System For Self-healing....61
4.2.1.1 Architecture Description....61
4.2.1.2 Experiments....65
4.2.2 A Self-managing Fault Management Mechanism for Wireless Sensor Networks....74
4.2.2.1 Architecture Description....74
4.2.2.2 Experiments....79
4.2.3 A Dendritic Cell Algorithm for Security System with Self-healing property....83
4.2.3.1 Dendritic Cell Algorithm....84
4.2.3.2 Architecture Description....84
4.2.3.3 Experiments....87
4.2.4 A MAPE-K Based Self-healing Framework For Online Sensor Data....88
4.2.4.1 Architecture....89
4.2.4.2 The ClouT Case Study....92
4.2.5 Comparison of the described architectures....93
Section 5: Case study....95
Section 6: Future work....98
Conclusion....99
References....100

Index of Tables

Table 1: self-* properties....................14
Table 2: A Brief Chronology of Influential Self-Management Projects....................18
Table 3: maintaining redundancy strategies....................27
Table 4: Maintaing by probing strategies....................30
Table 5: Summarized view of the different strategies to maintain system health…35
Table 6: Failure classes....................51
Table 7: Description of each scenario....................69
Table 8: Characteristics of the simulations....................72
Table 9: Message fields....................79
Table 10: Experiment parameters....................80
Table 11: Architectures comparison....................94
Table 12: Network's major problems....................95
Table 13: Network details....................96

Figure Index

Figure 1: millions of $ revenue/hour lost....................12
Figure 2: Problem diagnosis in an autonomic system upgrade (accounting system) 15
Figure 3: Self-healing origins and properties....................21
Figure 4: Self-healing loop....................22
Figure 5: Autonomic Manager....................23
Figure 6: State diagram....................26
Figure 7: Maintenance of health....................27
Figure 8: Redundancy Policies....................27

Figure 9: Maintaining by probing..29

Figure 10: Diversity in system...32

Figure 11: Performance log analysis...33

Figure 12: System failure detection..36

Figure 13: Something amiss..37

Figure 14: Notification of foreign element...38

Figure 15: System recovery techniques...39

Figure 16: Redundancy techniques for healing...40

Figure 17: Repairing plans for healing...42

Figure 18: Component interaction based healing..43

Figure 19: Transition between policies...45

Figure 20: Relathionships between different types of policy..............................47

Figure 21: Autonomic element..65

Figure 22: Hierarchical network comprised of common nodes, cluster heads and a base-station..66

Figure 23: Delivery rate of message..66

Figure 24: Energy consumption..67

Figure 25: Detection effectiveness for centered failures (scenario 1)...................67

Figure 26: Detection effectiveness for failures near the base station (scenario 2) 70

Figure 27: Common nodes energy-consumption..72

Figure 28: Clucter-head power consuption...73

Figure 29: Fault detection and diagnosis process..76

Figure 30: Virtual grid of nodes...77

Figure 31: Average energy loss for cluster head recovery...................................82

Figure 32: Average energy loss in re-clustering..82

Figure 33: A typical RPL DODAG with one root and six other nodes....................85

Figure 34: The proposed architecture..85

Figure 35: Accuracy rate of MLPLW over the number of inputs............................88

Figure 36: The MAPE-K-based framework for self-healing of sensor data...........90

Figure 37: General architecture of the ClouT self-healing framework...................93

Figure 38: Destroyed/no energy left node flowchart..97

Introduction

Internet of Things (IoT) refers to the expansion of Internet technologies so as to include wireless sensor networks (WSNs). We will mainly talk about WSNs, since it is a sub-technology of IoT. WSNs are largely used for the military, on smartphones, for intelligent environments and ubiquitous applications. In most of the cases, WSNs are composed of hundreds of elements, which are able to collect, process, disseminate and store data. The elements perceive the environment, monitor different parameters, collect data and, afterwards, they transfer these data to the base stations. In military operations, WSNs are always placed in a hostile area, which renders them vulnerable to physical contact. Thus, it is of utmost importance to have some kind of protection.

WSNs are usually deployed to operate for a long period of time, which means that they have to be available the whole time they are “out” there. Because of this, it is important and even a requirement nowadays, to be self-managed and fault tolerant. This means that, whenever a software or hardware error occurs, they have to be automatically available. Self-healing and autonomic systems have come to solve this problem. We will now describe how autonomic systems work and which is the main purpose of self-healing.

Self-healing in Internet of Things is quite a hot topic these days, thanks to the growth rate of IoT. Bearing in mind how big the need to be self-managed and self-sustained is, we have to find ways to create a better IoT with devices that will be available for the whole time they are in production. In the present text we will give the reader the basic idea of self-managed systems and self-healing systems and how this can be implemented in IoT.

In section 1, we describe autonomic systems and their core functionalities, self-healing being one of them. Next, in section 2 we talk about the history of

autonomic systems and we see some self-management projects. Moreover, in section 3 we dive into the self-healing and its core functionalities like self-healing loop, self-healing states, self-healing policies, failure classification and we also talk a look on existing self-healing applications. In section 4 we see self-healing in internet of things and we describe in detail the architectures that are being used for self-healing networks and we compare them. In section 5 we make a case study of self-healing network and in section 6 we talk about our plans for future work and last we conclude our work.

Section 1: Autonomic Systems

Computing systems have reached a level of complexity, where humans can't interfere whenever there is a problem on the system. So, extra effort is needed in order to get the systems up and running again. This also includes increased costs to maintain such complex systems, based on data from Performance Engineering and Measurement Strategies. Figure 1 shows the millions lost per hour due to the systems being down. The main goal is to design and develop systems that will adapt to changes in their environment on their own.

Autonomic Computing was first introduced by IBM in 2001 (Horn, 2001). The autonomic concept is inspired by the human body's autonomic nervous system. By analogy, humans have good mechanisms for adapting to changing environments and repairing minor physical damages. Automatic Computing tries to integrate those properties of the human body into the computing systems.

An autonomic system is self-managing, meaning that it is self-protecting, self-configuring, self-optimizing and self-healing (Abbas, Andersson, & Loewe 2010). Those are the main properties and they are described as self-* properties.

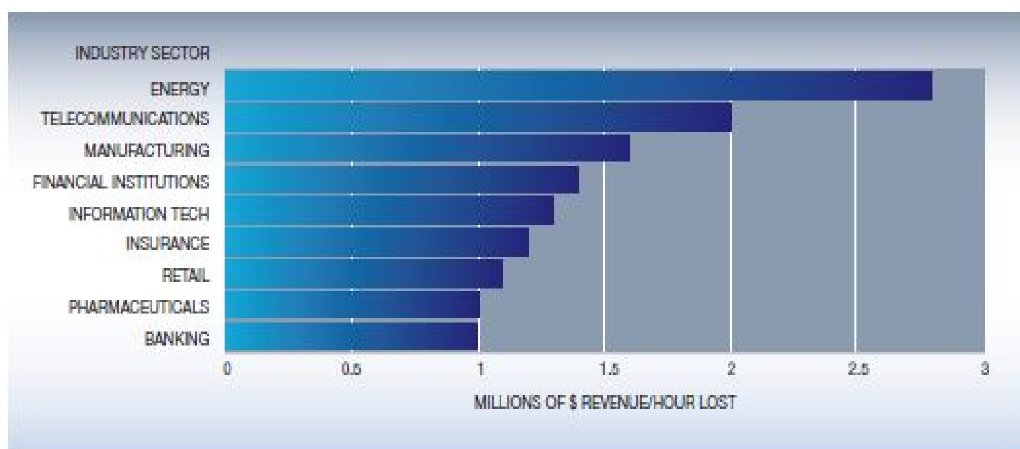

Figure 1: millions of $ revenue/hour lost

1.1 Self-Management

The general idea of self-management consists around the intention to free specific complex systems, such as databases, from system administrators and still allow them to run around the clock. The autonomic system can continually monitor its own use, and check for component upgrades, for example. If it deems the advertised features of the upgrades worthwhile, the system will install them, reconfigure itself if necessary, and run regression test to make sure all is well. When it detects errors, the system will revert to an older “image”. Figure 2 shows how this process can be done in an accounting system. The journey toward fully autonomic computing will take many years, but there are several important and valuable milestones along the path.

1.2 Self-Configuration

Autonomic systems with self-configuration property will configure themselves automatically in accordance with high-level policies, which are representing business-level objectives. When a component is introduced, it will incorporate itself seamlessly, and the rest of the system will adapt to its presence, much like a new cell in the body or a new person in a population. For example, when a new component is introduced into an autonomic accounting system, as in Figure 2, it will automatically be informed about it and take into account the composition and configuration of the system. It will register itself and its capabilities so that the other components can either use it or modify their own behavior appropriately.

1.3 Self-Optimization

Complex systems like databases may have hundreds of tunable parameters that must be set for the system to perform optimally. Yet, few people know how to tune

them. Autonomic systems will continually seek ways to improve their operation to make themselves more efficient. Just as muscles become stronger through exercise. Autonomic systems will monitor, experiment with, and tune their own parameters and will learn to make appropriate choices. They will seek to upgrade their functionality by finding, verifying and applying the latest updates.

1.4 Self-Healing

Autonomic computing systems with self-healing properties will detect, diagnose and repair localized problems resulting from bugs or failures in software and hardware, perhaps through a regression tester, as in Figure 1. Using knowledge about the system configuration, a problem-diagnosis component would analyze information from log files, possibly supplemented with data from additional monitors that it had previously requested. The system would then match the diagnosis against known software patches, install the appropriate patch, and retest.

1.5 Self-Protection

Despite the existence of firewalls and intrusion detection tools, humans must now decide how to protect the systems from malicious attacks. Autonomic systems with self-protection property will be self-protecting in two senses. On one hand, they will defend the system as a whole against large-scale, correlated problems arising from malicious attacks or cascading failures that remain uncorrected by self-healing measures. On the other hand, they will anticipate problems based on early reports from sensors and take the required steps to avoid mitigating them.

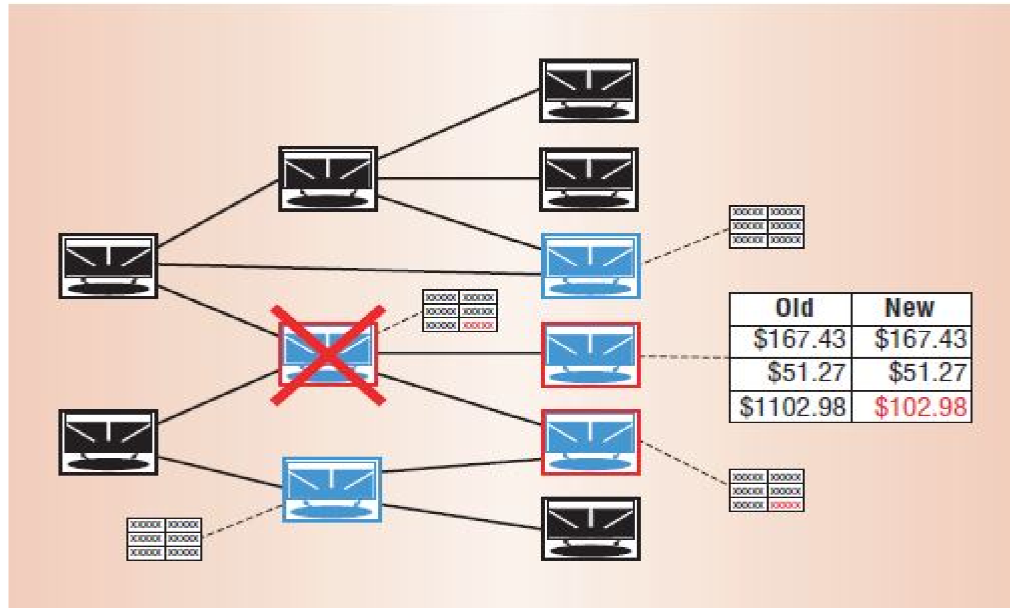

Figure 2: Problem diagnosis in an autonomic system upgrade (accounting system)

Those properties are summarized in table 1.

Property	**Description**
Self-configuration	Automated configuration of components.
Self-optimization	Components and systems continuously seeking different ways to improve their performance.
Self-healing	System automatically detects software and hardware problems.
Self-protection	System automatically defends against malicious attacks.

Table 1: self- properties*

Section 2: A Brief History

Before going into more detail about autonomic computing, self-healing, and what makes them so necessary for our modern systems, we first have to take a look at the early stages of autonomic computing and its applications.

One of the early self-managing projects was initiated by Defense Advanced Research Projects Agency (DARPA) for military application in 1997. The project was called the “Situational Awareness System” (SAS), and was part of the broader Small Units Operations (SUO) program. The purpose of this project was to provide the soldiers with better communication and location devices on the battleground. Soldiers could create status reports, for instance regarding the discovery of enemy tanks, on their personal device and have this information spread out among all the other soldiers. This kind of information could turn out extremely helpful when a soldier is entering an enemy area. Collected and transmitted data includes voice messages, as well as data from unattended ground sensors and unmanned aerial vehicles. The difficulty with these personal devices is that they have to communicate with each other under ambiguous situations, possibly while an enemy is jamming the equipment in operation, and they must at the same time minimize any enemy interception. The latter is addressed by using multihop ad-hoc routing. This is a device that sends its data only to the nearest neighbors, who, then, forward the data to their own neighbors, until, finally, all devices receive the data. This is a form of decentralized peer-to-peer mobile adaptive routing, which has been approved as a challenging self-management problem, especially because of the need to achieve a latency below 200 milliseconds from the time a soldier begins speaking until the message is received. This problem is solved by enabling the devices to transmit in a wide band of possible frequencies, 20-2,500 MHz, with bandwidths ranging from 10bps to 4 Mbps. For example, when the next soldier is many miles away, communication can only be achieved at low

frequencies, which translates in low bandwidth, which, then, may still be enough to provide a brief but crucial status report.

Another project by DARPA related to self-management, is the DASADA, which started in 2000. The objective of the DASADA program was to research and develop a technology that would enable mission critical systems to meet high assurance, dependability, and adaptability requirements. Specifically, it dealt with the complexity of large distributed software systems, a goal similar to IBM's autonomic computing initiative.

In 2001, IBM introduced the concept of autonomic computing. Horn (2001) is comparing complex computing systems with the human body, in the sense that it is an autonomic system, but has an autonomic nervous system that takes care of most of the bodily functions, thus relieving the body from the task of consciously coordinating them. IBM proposed that complex computing systems should also have autonomic properties that should be able to take care of the regular maintenance.

In 2004, DARPA started another project called "Self-Regenerative Systems" which aims to "develop technology for building military computing systems that provide critical functionality at all times, in spite of damage caused by unintentional errors or attacks" (Adger & Hughes, 2004). There are four key aspects to this project. First, the software generates a large number of versions that have similar behavior, but slightly different implementation. With this technique the software is rendered resistant to attacks and errors. Second, modifications to the binary code can be performed, such as pushing randomly sized blocks onto the memory stack that make it harder for attackers to exploit vulnerabilities. Third, a scalable wide-area intrusion-tolerant replication architecture is being worked on, which should provide accountability for authorized but malicious client updates. Fourth, technologies are being developed that supposedly allow their system to estimate the probability of a military system operator "hurting" the system and to prevent an attack on the system.

In 2005, NASA joined the game of autonomic computing with a project called "Autonomous Nanotechnology Swarm" (ANTS). As an exemplary mission, they plan to launch into an asteroid belt a swarm of 1000 small spacecrafts from a stationary factory ship in order to explore the asteroid belt in detail. Since the 60-70% of the swarm is expected to be lost, the surviving crafts must work together. This is achieved by forming small groups of workers (craft) with a coordinating ruler, which uses data gathered from workers to determine which asteroids are of interest and to issue instructions. Furthermore, messenger craft will coordinate communications between the members of the swarm and ground control. In table 2 we summarize all the projects that we discussed.

SAS Situational Awareness System	1997	DARPA	Decentralized self-adaptive (ad-hoc) wireless network of mobile nodes that adapt routing to the changing topology of nodes and adapt communication frequency and bandwidth to environmental and node topology conditions.
DASADA Dynamic Assembly for Systems Adaptability, Dependability, and Assurance	2000	DARPA	Introduction of gauges and probes in the architecture of software systems for monitoring the system. An adaptation engine then uses this monitored data to plan and trigger changes in the system.

Autonomic Computing	2001	IBM	Compares self-management to the human autonomic system, which autonomously performs unconscious biological tasks.
SPS Self-Regenerative Systems	2003	DARPA	Self-healing (military) computing systems that react to unintentional errors or attacks.
ANTS Autonomous NanoTechnology Swar	2005	NASA	Architecture consisting of miniaturized, autonomous, reconfigurable components that form structures for deep-space and planetary exploration.

Table 2: A Brief Chronology of Influential Self-Management Projects

Section 3: Self-healing

IBM, (IBM Corporation, 2005) has included self-healing as one of the main four properties that are defined in an autonomic system. Ghosh et al. (2006) provide a more recent definition of self-healing systems:

"…a self-healing system should recover from the abnormal (or "unhealthy") state and return to the normative ("healthy") state, and function as it was prior to disruption."

One might argue that self-healing systems are just subordinates of fault-tolerant systems. They are, indeed, similar and Ghosh, Sharman, Rao, Raghav, and Upadhyaya (2006) admit that self-healing systems are, in some cases, secondary. "Survivable" systems are generally handling the malicious behavior by containing failing components and securing the essential services. However, self-healing is not that simple. Those systems are implementing methods for stabilizing, replacing, securing, isolating but more essentially methods to prevent and repair faults.

The main reason to enhance a system with self-healing is to achieve *continuous availability* (Psaier & Dustdar, 2010). Currently, self-healing techniques are mostly in charge for the *maintenance of health.* *Enduring continuity* includes resilience against intended, necessary adaptations and unintentional behavior. Also, self-healing implementations use *detecting disruptions, diagnosing* failure and deriving a remedy, and then *recovery* with a sound strategy. Moreover, it is essential for the fast detection of the system misbehavior. This is only possible by continuously analyzing the sensed data. The system design leads to a *control loop,* which is a set of policies and guidelines to implement self-healing in a system. We will describe *control loop* in the next sub-section.

In figure 3 we see the origins of the self-healing property and its properties.

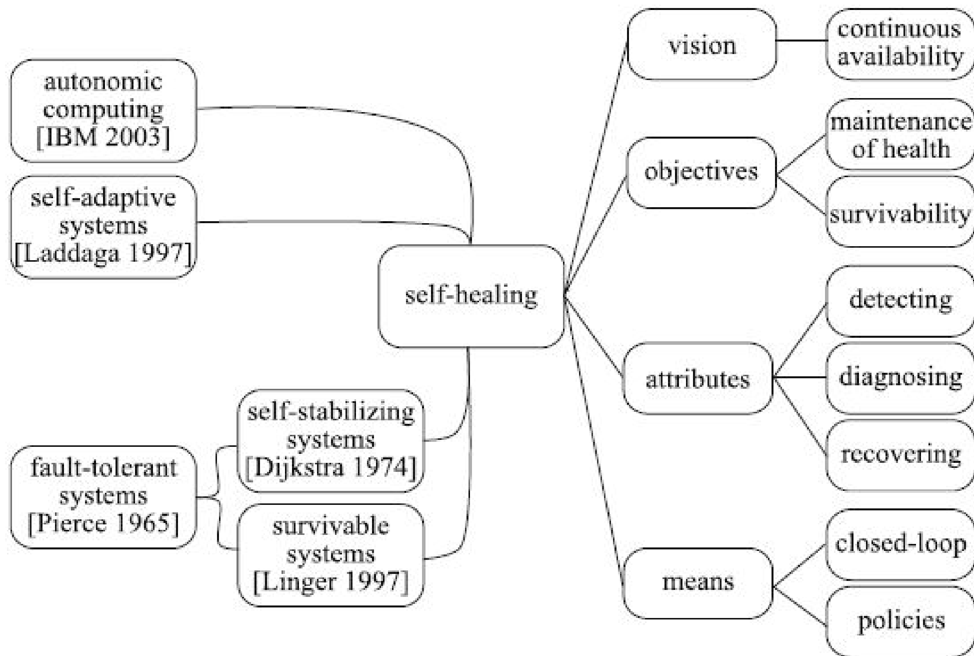

Figure 3: Self-healing origins and properties

3.1 Self-healing loop

Earlier we described the main properties of the autonomic manager. The main design of an autonomic manager is also based on a loop, which is called MAPE-K (Monitor, Analyze, Plan, Execute, Knowledge) loop and consists of five functions. The collaboration of those functions assembles the work of the autonomic manager.

The idea of a continuous multi-state processing loop is also implemented in the self-healing loop. In self-healing design, the five autonomic processes are reduced into three main stages in a loop. There are many implementations of these three stages and they have been given several different names. For example, Kephart and Chess, (2003), identify them as “detection, diagnosis, and repair”. Salehie and Tahvildari, (2009), call them “a sum of self-diagnosing and self-repairing with discovery, diagnosing, and reacting stages”. Markus, Huebscher and McCann,

(2008), define them as "detect, diagnose and fix actions". Looking at all of the researched work, it is safe to say that the first stage, detection is the most ample definition. Detection is the act of uncovering or revealing an alternating of the normal behavior. Analysis and planning functions are included in the stage of Diagnosis in the self-healing loop implementations. A set of rules and policies support Diagnosis in planning. The last stage of the loop is Recovery. Although this stage is considered just as extended, it is not always entirely successful. Figure 5 shows the form of the self-healing loop with the dataflow among the three stages.

Detecting: Filters any suspicious status information received from samples and reports detected degradations to diagnosis, (Psaier & Dustdar, 2010).

Diagnosing: Includes root cause analysis and calculates an appropriate recovery plan with the help of a policy base.

Recovery: Carefully applies the planned adaptations meeting the constraints of the system capabilities and avoids any unpredictable side effects.

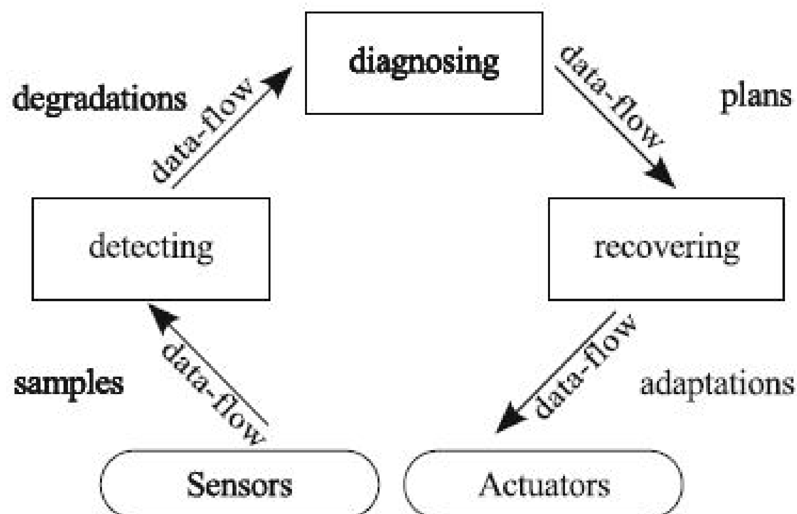

Figure 4: Self-healing loop

3.1.1 Autonomic Manager

The autonomic manager is a component that implements the MAPE-K loop, (Sterrit & Bustard, 2003). For a system component to be self-managing, it must have automated methods to collect the details it needs from the system, as well as analyze these details and, in case something has changed, it has to be able to create a plan or a series of steps/actions which it will then have to perform. When these actions are automated, a control loop is formed.

To perform all these actions, the component is separated in four parts that share information-knowledge with each other. Those four parts cooperate with each other to provide the MAPE-K loop functionality (Sterrit & Bustard, 2003).

In figure 5 we see how the autonomic manager is formed.

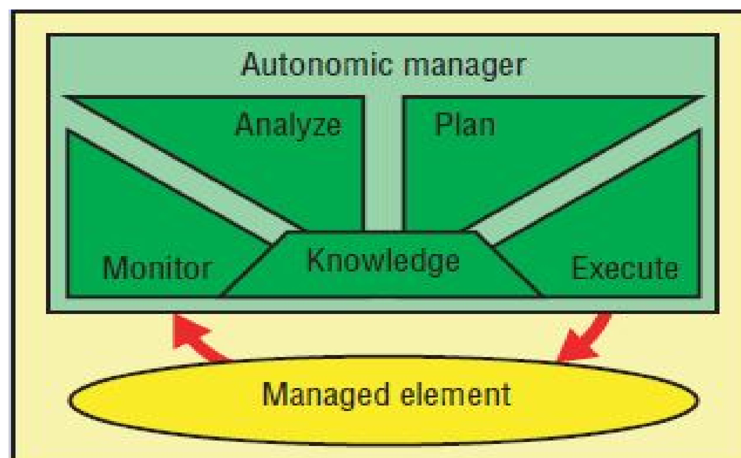

Figure 5: Autonomic Manager

3.1.2 Monitor

The monitor function is responsible for collecting the details from the managed resources. These details may include topology information, metrics, configuration property settings and so on. The kind of the data varies: some data can be static, other can be changing slowly and some may even be dynamic and change through time. The monitor function correlates and filters these details until it determines a symptom that needs to be analyzed. Afterwards, this symptom is passed onto the analyze function. The monitor function is a crucial phase of the autonomic manager because it must collect and process a large amount of data, as well as organize and make sense of those data.

3.1.3 Analyze

The analyze function provides mechanisms to correlate, observe, and analyze complex situations. Also, it tries to determine if any changes are needed to be made to the managed resource. If any changes are, in fact, required, the analyze function generates a change request and passes it to the plan function. This request describes the modification that the analyze component deems necessary or desirable.

3.1.4 Plan

The plan function creates or selects a procedure to execute a desired change in the managed resource. A change plan, which represents a set of changes for the managed resource, is created and then passed to the execute function.

3.1.5 Execute

The execute function provides the mechanisms to schedule and perform the necessary changes to the system. This function is responsible for the execution of the change plan that was generated from the plan function

3.1.6 Knowledge

The knowledge is usually stored in a knowledge source, which might be a registry, a dictionary, a database or any other repository that is capable of storing data. This Data is used by the autonomic manager’s four functions (monitor, analyze, plan, execute) and stored as a shared knowledge. Those data might be different from one another.

3.2 Self-healing states

The robustness of the self-healing must not depend on a single element, but the system as a whole should be able to recover from failures (White, Hanson, Whalley, Chess, & Kephart 2004). So, single element failures should not affect the whole system. In many cases there is no fine line clearly separating acceptable from unacceptable state. Instead, there is momentary transmission zone in between.

The most recent model is represented by Ghosh et al. (2006). Specifically, it shows a fuzzy transition zone with an unclear “Deprecated State”. This state reflects the fact that adverse conditions of a system cause self-healing systems to still be in an acceptable state, but, closer to the failure. This concept regards the fact that large systems usually do not stop all the operations when a failure occurs, keep the operations going, keep the operations performing, although more poorly. This provides the time that the system needs to apply the recovery techniques and

allows is to bring the system back on track without complete disruption. The model we described is shown in figure 6.

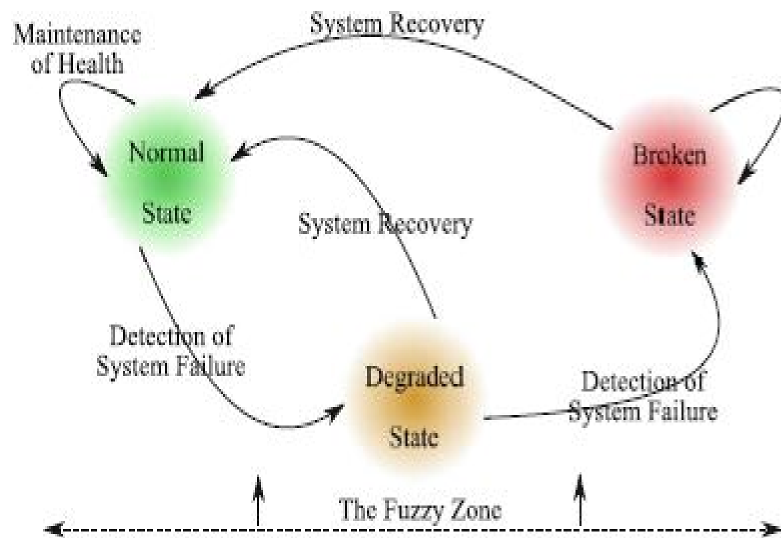

Figure 6: State diagram

3.2.1 Maintenance of Health

The system should check for faults periodically, in order to continue monitoring its health. Additionally, when a system recovers from a malfunction, different approaches may apply to keep the functionality of the system up to a normal level. There are different strategies used to maintain the health of the system, such as maintaining redundancy of components, probing into the system and assessing the state of the system, building diversity, etc. as shown in figure 7.

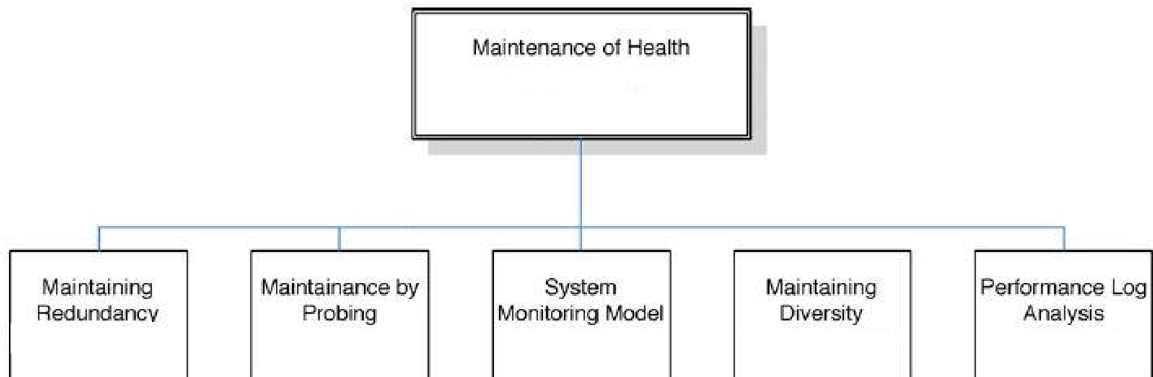

Figure 7: Maintenance of health

3.2.1.1 Maintaining redundancy

Cloning components to maintain redundancy is a standard choice for maintaining a the system health. However, there are various redundancy strategies that someone could choose from. Figure 8 shows some of those strategies and they are, also, briefly described in table 3.

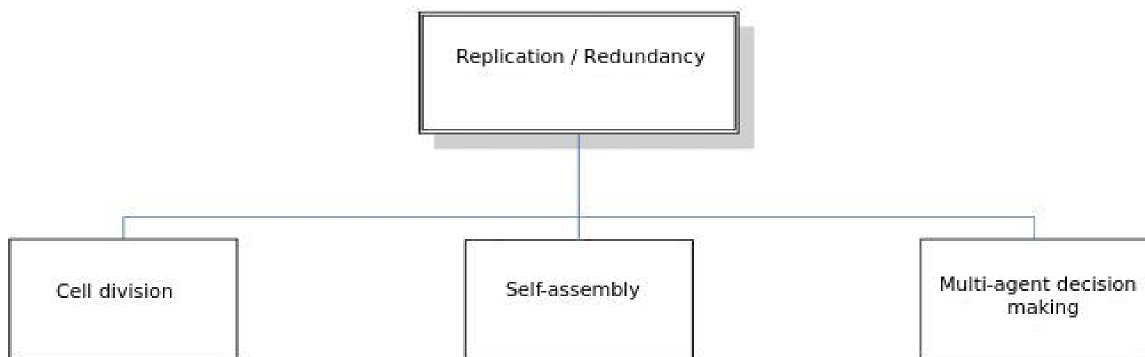

Figure 8: Redundancy Policies

Strategy	**Description**
Self-assembly, (Nagpal, Kondacs, & Chang, 2003)	Describes the biological cells and demonstrates the ability of the system to survive failures based on a cell division model.
Cell division, (George, Evans, & Marchette, 2003)	All cells start with an initial configuration and follow transition rules like a finite state machine. Self-healing provides: evidences, localization, adaptation, adequate redundancy and the unique distinction of awareness towards the environment.
Multi-agent decision making, (Huhns, Holderfield, & Gutierrez 2003)	Redundant agents are used to keep the system running with different algorithms to provide a better software solution.

Table 3: maintaining redundancy strategies

3.2.1.2 Maintaining by probing

Probing is another mechanism, which is generally used to get information from the system, in order to monitor its health. Figure 9 shows the different strategies that exist to get these information from the system. Table 4 shortly describes those strategies.

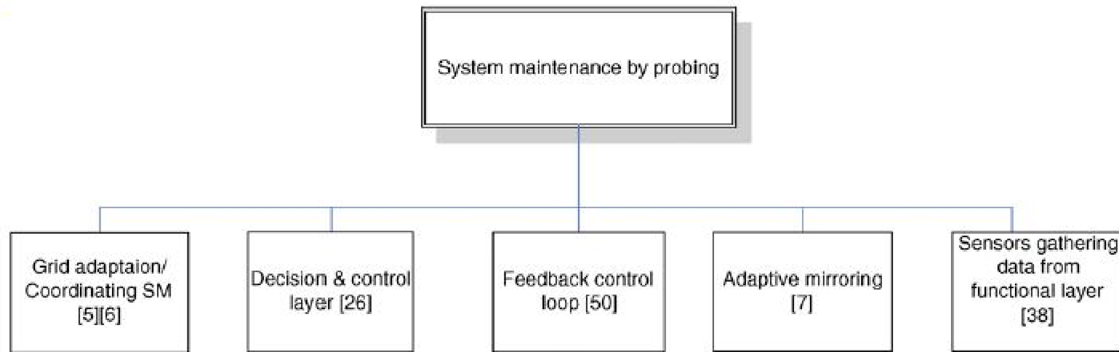

Figure 9: Maintaining by probing

3.2.1.3 System monitoring architecture model

The architecture of a system is always difficult, and self-healing is not an exception. Many steps have been taken towards the right direction but we still have a long road ahead. In this subsection we will describe some implementations of Architectural Description Language (ADL) and another implementation called “Component relation & regularities”. ADLs are being utilized to define the system architectures and components that are, comprise a necessary part of self-healing.

Georgiadis, Magee and Kramer, (2002), examine an implementation that depends on Darwin ADL and Jackson’s alloy language in association with a monitoring tool to view the runtime structure of a distributed system. This implementation retrieves the configuration from the components and tries to keep it the same when update events are (being) broadcasted. Dabrowski and Mills (2002), suggest that Java programming will make it simpler and easier for system implementers to determine the self-healing properties of their code. Java being on the background, they believe that they can design a ubiquitous architecture where components must typically self-heal in response to changes. Another approach by Dashofy, Hoek and Taylor (2002), defines the architecture of the system as represented by xADL 2.0 an extensive XML-based ADL. Dabrowski and Mills, (2002), have proposed an

architecture-based adaptation in a service discovery system. Discovery protocols in general help the components of the system to find each other over a network.

The other categories of monitoring architecture are representing regularities in a system and providing a relational model of the system. Usually, those two architectures are represented as one category called “component relation and regularities”. In De Lemos and Fiadero (2002) the strategy which is used is based on faulty components and the reconfiguration of the system. Their system architecture may apply numerous layers for doing the following:

1) Computation between the components of the system.

2) Coordination between interactions of components.

3) Configuration — which decides when and how components and connectors link up.

Every system that is likely going to change should have regularities, as pointed out by Minsky, (2003). Distributed systems often “carry” components that are built on different software and hardware, and he pointed out the desire that these systems have to include desired regularities among the components. These regularities are achieved thanks to the artificial laws. Law Governing Interaction (LGI) is one of them and is used to form those laws. LGI is a message mechanism between distributed components of a system. An LGI law has a basic function and this function is used to regulate the exchange of messages in a community. LGI law has an exceptional feature: it can be defined over events occurring only for some members in the community. Grishikashvili (2001), pointed out that appropriate control laws governing communication must exist for any system to remain functional in a changing environment. Raz, Koopman and Shaw. (2002) came up with a template design mechanism to lower the requirements of human.

Stategy	**Description**
Grid adaptation / Coording SM (Cheng, Garlan, Schmerl, Steenkiste, & Hu, 2002)	Different software architecture & monitoring infrastructure that can be maintained on runtime and can be used for system configuration. It consists of three layers: task, system-monitoring or “probes” and information. Probes incorporate the lowest level of abstraction. In the information layers meters are being used to report information via bus (also known as reporting-bus).
Decision & control layer (Inveraldi, Mancinelli, & Marinelli, 2002)	A similar architecture to the previous. It introduces a new layer called “decision & control” which receives information from the reporting bus and optimizes the metrics. Moreover, it reconfigures the system by introducing new modules.
Feedback control loop (Shaw, 2002)	Data instrumented with probes, which reports raw data into the probe bus and is used for software adaptation.
Adaptive mirroring (Cheng, Huang, Garlan, Schmerl, & Steenkiste 2004)	This approach is based on the utilization of probes to intercept a system workflow and circumnavigate data and control through a different path.
Sensors gathering data from functional layer (Merideth & Narasimhan 2003)	In this strategy, two types of sensors are used. The ones called State Sensors (SS) that gather information from functional layer and the other Analysis Sensors (AS) and those that collect messages flowing through the system.

Table 4: Maintaing by probing strategies

3.2.1.4 Diversity in system

Diversity is an important foundation of robustness in biological organisms. In comparison with computer systems, a lack of diversity is noticeable; this sameness between the systems makes them vulnerable to attacks, because the attacker may replicate the same attack in different systems and still be successful.

Sharman et al., (2004), have proposed a novel paradigm for security functionality-based systems. Healing is not performed in the system, but in the functionality that the system has. Diversity is created by designing a single function which will run on different systems. As the systems are designed independently, it becomes difficult for any intruder to break through all of them. Forrest, Somayaji and Ackley (1997), have come up with a design model where the main strategy is to limit the consistency in software, thus will make the intrusion much more difficult to replicate. For this design, Forrest et al., (1997), have given guidelines to implement it, such as adding/deleting nonfunctional codes, reordering the basic blocks of compiled codes in random order, etc. Inverardi et al. (2002), mentioned that identifying critical parts in the source code is important for providing different alternatives. Figure 10 depicts the different strategies using system diversity for maintaining the system health. The root node of the tree structure used here illustrates the concept as a whole, while the leaf nodes represent different strategies.

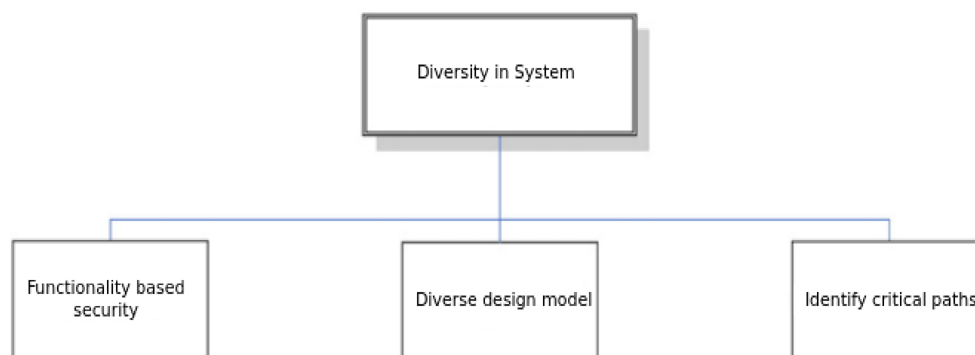

Figure 10: Diversity in system

3.2.1.5 Performance log analysis

Knop, Schopf and Dinda, (2002), propose a scheme for the analysis of data on the performance measures of the system, which can be used for the diagnosis of faults, the detection of an intrusion and the facilitation of the healing process. They built a library called "Watchtower", based on Windows NT/2000, which monitors the command lines, the console and any streaming operations. Event Log analysis is another approach, which uses different time-series techniques for predicting rare events and can also be used to predict target events (Sahoo et al., 2002), across a computer network. Hong, Chen, Li and Trivedi (2002), propose a Finite Automata scheme to describe the aging and rejuvenation states of software. This model contains elements (converters) that detect software aging by monitoring periodically for typical symptoms. Figure 11 shows all the strategies that are being used for using performance log analysis to maintain system health.

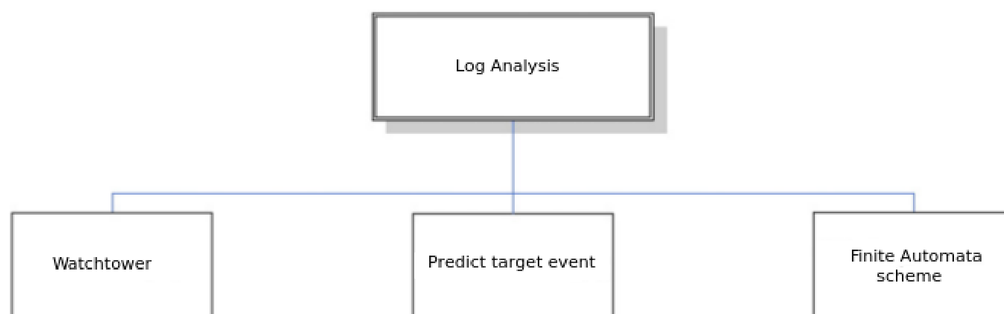

Figure 11: Performance log analysis

Below we summarize the models and the strategies.

Models	Strategy
Maintaining redundancy	Self-assembly, (Nagpal et al., 2003)
	Cell division, (George et al., 2003)
	Multi-agent decision making, (Huhns et al., 2003)
Maintaining by probing	Grid adaptation / Coording SM, (Cheng et al., 2002)
	Decision & control layer, (Inveraldi et al., 2002)
	Feedback control loop, (Shaw, 2002)
	Adaptive mirroring, (Cheng et al., 2004)
	Sensors gathering data from functional layer, (Merideth & Narasimhan, 2003)
ADL Approach	Darwin ADL and Jackson's alloy for runtime view, (Georgiadis et al., 2002)
	ArchJava language with Java for ubiquitous computing, (Dabrowski & Mills, 2002)
	XADL 2.0, (Dashofy et al., 2002)
	Rapide for service discovery, (Aldrich et al., 2002)

Component relation & regularities	Isolation of faulty component, (De Lemos & Fiadeiro, 2002)
	Regularities in architecture, (Minsky, 2003)
	Artificial law cybernetic foundation, (Grishikashvili, 2001)
	Template design, (Raz et al., 2002)
Diversity in system	Functionality-based healing, (Sharman et al., 2004)
	Diversity in vulnerable places, (Forrest et al., 1997)
	Identification of critical path, (Inveraldi et al., 2002)
Performance log analysis	Monitoring and reducing performance data, (Knop et al., 2002)
	Predicting events (Sahoo et al., 2002)
	FSA for system rejuvenation (Hong et al., 2002)

Table 5: Summarized view of the different strategies to maintain system health

3.2.2 Detection of System Failure

Failure detection is another major field of research. The expectation is that any secured system should easily find a failure or the presence of any malicious software. It must be smart enough to gauge the degree of malfunction in the system and estimate whether the system actually needs recovery or not. As in biological systems, the system should recover exactly as the body recovers from a wound leaving the other functions unaltered. Similarly, if a module in the system is under threat, the other modules should function properly, as before. In this section we will discuss about policies that are being used to encounter failure detection. Figure 12 demonstrates the different policies.

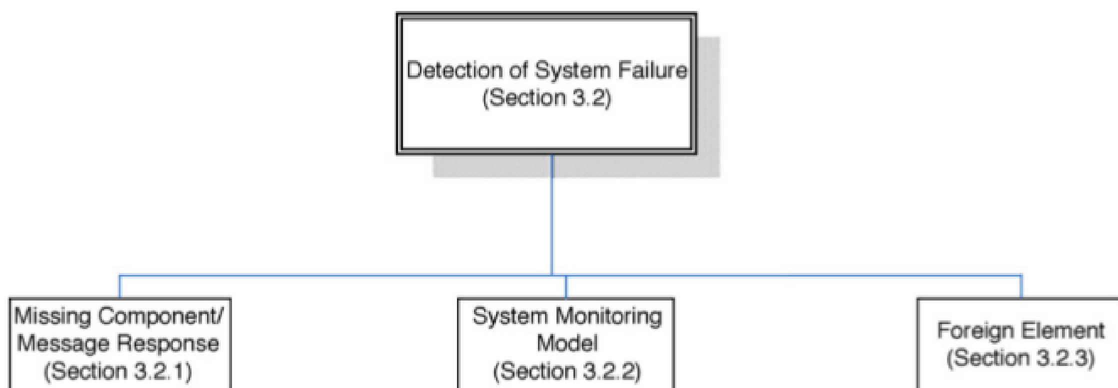

Figure 12: System failure detection

3.2.2.1 Something amiss

In this subsection, we discuss about strategies whose job is to detect if something is missing from the normal behavior of the system. Nagpal et al. (2003) proposed a strategy in which the agents are always about their neighbors. So, if they sense that any neighbor is missing, they are able to replicate him. Similarly, George et al. (2003), point out that in the DWFS paradigm, any failure is sensed by the absence

of messages. Aldrich et al. (2002), in their gardening module, use a simple policy to detect failures. It senses a defect when responses to a query are not received. Dabrowski and Mills (2002), proposed that failure in receiving scheduled announcements may reveal that the component has failed. Jini, (Apache, 2012-2015), provides broadcast messages about the accessibility of resources. Figure 13 shows the different strategies for detecting failure by sensing something amiss from regular behavior of a system.

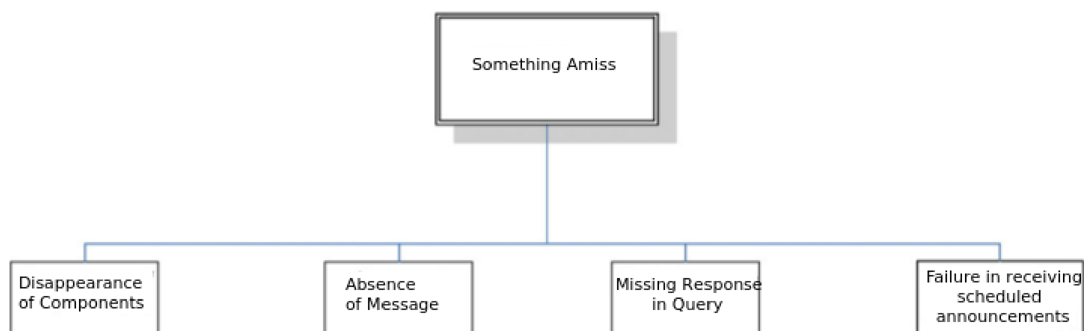

Figure 13: Something amiss

3.2.2.2 System monitoring model

This subsection deals with policies suitable for architectures where the system monitors components by probing. Garlan and Schmerl (2002), have mentioned that monitored values can be abstracted and related to these architectural properties of the model. This is the typical model representation scheme adopted by most ADLs. Cheng et al. (2002) propose that, for Grid architecture, the probes and gauges report low level monitoring information that is used for triggering events.

3.2.2.3 Notification of foreign element

In this subsection we will talk about the different existing approaches for detecting system failure by distinguishing foreign elements. Merideth and Narasimhan (2003), point out that notifying about foreign element plays a significant role in proactive containment strategy. Faulty processes generally attempt some communication; this scheme tries to limit this action whenever it is possible. Thus, this method tracks all possible avenues (secure and covert) to determine if a malice replica in a group of processes is affecting processes of other groups. Dashofy et al. (2002), seek to depict the differences between two architectures specified by xADL 2.0 in the form of an architectural difference document, called “diff”. This document shows the differences between the two architectures and, therefore, can also recognize the presence of foreign elements in the system. Figure 14 depicts those two strategies.

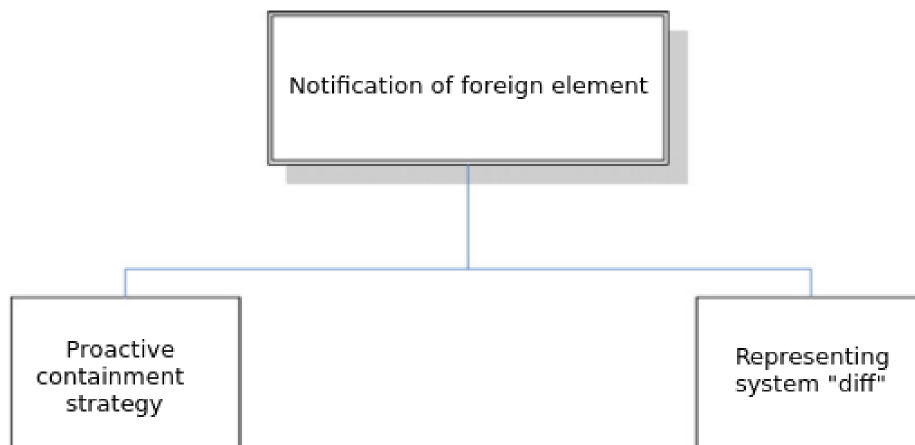

Figure 14: Notification of foreign element

3.2.3 System Recovery

In this section we look closely on the different healing techniques that are being used to return in a healthy state. The most important factor in self-healing are the components that are dealing with the actual healing. It is, therefore, when a self-healing system detects the abnormality that it should readily apply its policies for healing. Those policies may consist of redundancy techniques like producing replicating components, applying repair strategies or use the Byzantine recovery and those are the policies that we will talk about here. Figure 14 depicts these policies.

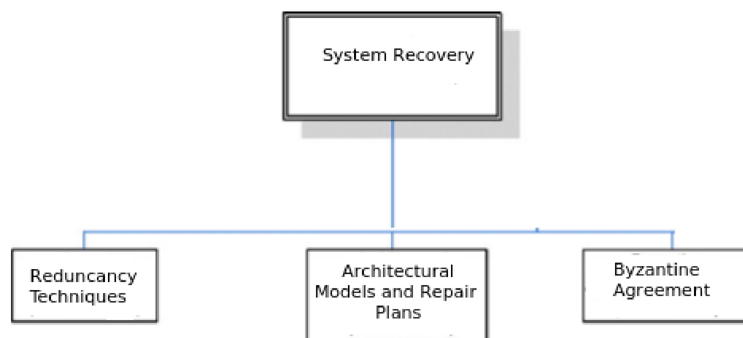

Figure 15: System recovery techniques

3.2.3.1 Redundancy techniques for healing

Nagpal et al., (2003) suggest that, for the model involving the self-assembly of the components, the system could self-repair and, also, regenerate. By regenerating, the agents can replicate components to replace the neighbors that are not available (dead neighbors) and thus recreate the entire structure. Regeneration can be used even if a large part of the system is destroyed, as long as the system has enough reference points. George et al. (2003), approach the healing process as if it was used by a biological system: the system produces as much cell as it

can, in order to combat the intrusion, so that it can survive in some way. The Recovery-Oriented Computing research project [35] is also utilizing a similar approach to affect the isolation of faulty components and provide redundancy techniques for safe recovery.

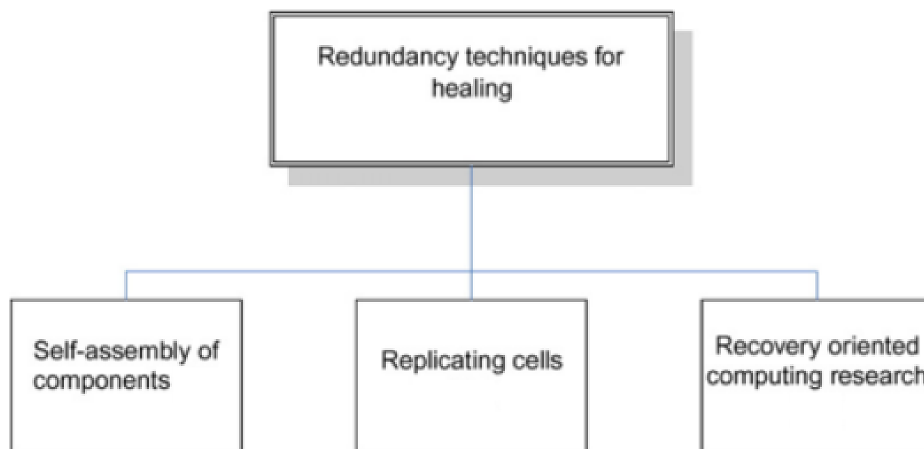

Figure 16: Redundancy techniques for healing

3.2.3.2 Architecture models and repair strategies

The usual exception catching mechanisms are often tightly integrated with the application and coupled with the source code. Those mechanisms are good for handling runtime problems, but they may not be able to find the true source of the problem. Also, these methods are not recommended if the system presents abnormalities, such as performance degradation. Garlan and Schmerl (2002), proposed that, for a system to be self-healing, it should be able to adapt in any situation that might fall outside of the system usual behavior. Different repair plans can be used so that the system can come back on its normal performance levels. In Figure 17 we demonstrate the repair plans that exist and, shortly, we will discuss them.

As we discussed above, the *repair strategies* are crucial for the system to return to its normal state again. Cheng et al. (2002), proposed an architecture which uses the system information that are provided by gauges. In this architecture, the repair strategies first try to find out how the problem occurred and, then, appraise how to repair it. In the first place, the repair strategies may form a sequence of preconditioned Repair Tactics that locate the problem and determine applicability. Then, the tactics have to choose which repair script to execute depending upon the tactic chosen. Every module that implements self-healing has to be coordinated with the other modules to maintain consistency in information acquisition. In a similar manner, regarding the architecture, the application of high level repair action plans has been suggested by Valetto and Kaiser (2002), along with a feedback control loop in a targeted system for software adaptation/for the software to be able to adapt.

Combs and Vagle, (2002), introduced an architecture and it is described as Service and Contract (S+C) protocol, which can dynamically replace a failed service with a healthy one.

Dashofy et al., (2002) have proposed an event-based configuration in which specific steps have to be followed for the repair plan to be executed. Those steps are:

1. Components and connectors that are about to be removed can do the following actions:
    a. Clean up
    b. Save their state
    c. Send a final message
1. Components that are in an unhealthy state are being prevented from sending messages.

2. Components, connectors and links are removed and added as required.
3. The components from the unhealthy state come back online again.

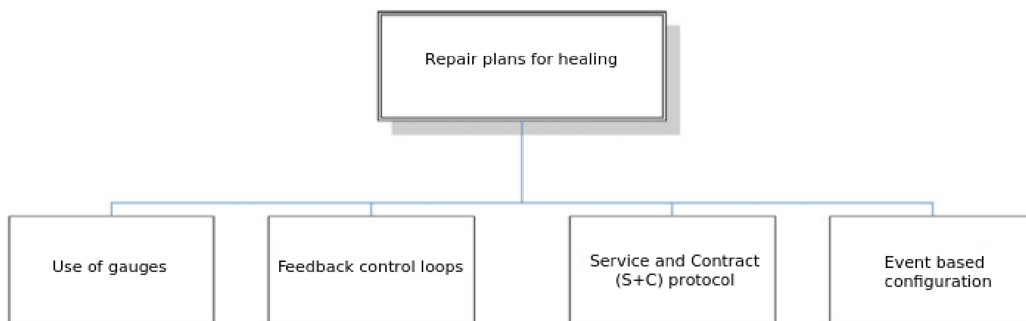

Figure 17: Repairing plans for healing

Component interaction-based healing covers a wide area of techniques and here we will discuss about:

1. Computer binding to satisfy architectural constraints,
2. soft state and application recovery,
3. isolution of faulty components.

Figure 18 depicts the above techniques and then we discuss about them.

Another proposal for a system architecture has been put forward by Georgiadis et al., (2002). Specifically, when current configuration undergoes a change, each configuration manager, not only computes the binding needed to satisfy the architectural constraints for each required port, but it/he evaluates a set of configuration rules and reevaluates the selector factors. Only after all the required

ports are bound/ secured, the system stabilizes. In short, the system is able to remain true to its specifications through the addition and removal of components.

Dabrowski and Mills (2002), proposed a service-discovery-based architecture, where the system makes use of two recovery techniques: soft-state recovery techniques and application level persistency. In the first one, the system makes periodic announcements about the current state of the system. As for the second one, it is guided by typical application level policies for recovery.

De Lemos and Fiadeiro (2002), emphasize more the isolation of the faulty components and the reconfiguration of the system. Through the externalization of all communications, the healing of faults occurring at the level of an element to the connectors through which that element interacts with the rest of the organization can be restricted. For fault treatment, this approach strongly depends on the dynamic reconfiguration of the system architecture. In our case, the proposed solution would achieve such a system reconfiguration after performing a series of atomic operations, until the final attainment of a stable system state.

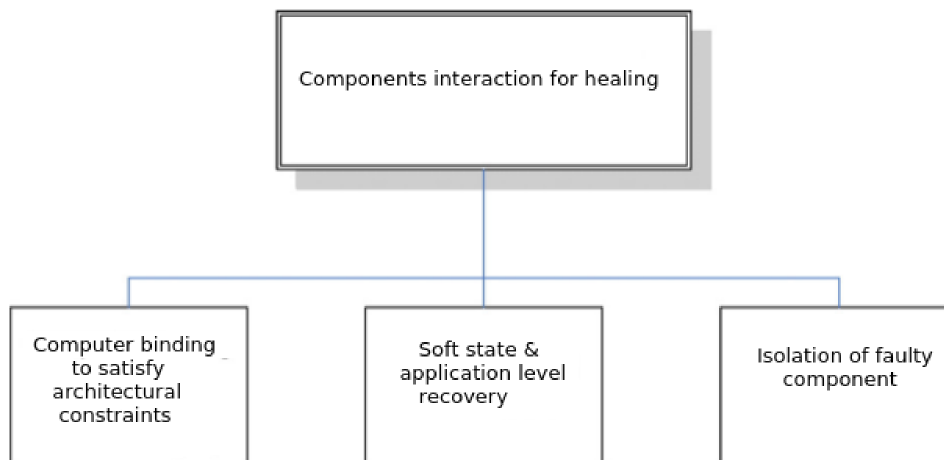

Figure 18: Component interaction based healing

Aldrich et al. (2002) and Dashofy et al., (2002), are the ones who tried to create systems that are *event-based.* Aldrich et al. (2002), created a gardening system in which any component can self-heal itself in a response of changes. A gardening module stays in a specific life cycle if it fails to receive any query within a fixed time limit. Likewise, Dashofy et al. (2002), propose another event-based architecture which used two xADL 2.0. The one will be the current architecture and the other will be the architecture that will be used after the repair plan is activated. With the help of ArchMerge we can merge two architectures, since this tool can spot the differences between the two architectures and merge them into one.

3.2.3.3 Voting methods for healing/Byzantine agreement

In this section we will talk about methods that are based on a voting mechanism. Merideth and Narasimhan (2003), proposed the restriction of a malicious fault, using active replication along with a voting technique, the system can tolerate processor and process level faults, as well as arbitrary faults. This method can find ways to figure out if a replica in a group of processes affects the other processes of other groups. Byzantine agreement[1] is used to spot a malicious processor, using active replication with majority voting. If a group of processes is infected with a malicious fault, by using the voting algorithm to test the output results that produced the processes, faults can be detected and tolerated.[2]

1. Lamport, Shostak and Pease, (1982), proposed to achive reliability among the replica of system (or component) to take the majority voting among them. To accomplish this technique two conditions have to be met: (a) all non-faulty components must have the same input value and (b) if the unit is non-faulty, then all non-faulty components use the same input.

2. The weakness with this strategy is that the a faulty component can provide correct input values and whenever its outputs are going to count on the vote, so it could avoid the Byzantine fault detection.

3.3 Self-healing policies

Influenced by AI research on human behavior, Norman, Ortony and Russell (2003), proposed a three-level model based on reaction, routine and reflection. Then, Kephart and Walsh (2004) based on Norman's research (2003), defined a framework which uses the above principles. This framework is called Unified Framework and has three different types of policies: Action, Goal and Utility Function. We will describe the three types of policies in the next sub-section.

3.3.1 The Unified Framework

The unified framework is based upon the notions of states and actions that are quite familiar in the computer science field, particularly in the realm of AI. In general, we can describe a system or its components being in a state *S* at a given moment in time. Typically, the state *S* can be described as a vector of attributes, each of which is either measured directly by a sensor, or perhaps inferred or synthesized from lower-level sensor measurements. A policy defines an action *a* to be taken, the result of which is that the system will make a transition to new state σ^2. This sequence of events is described in figure 19.

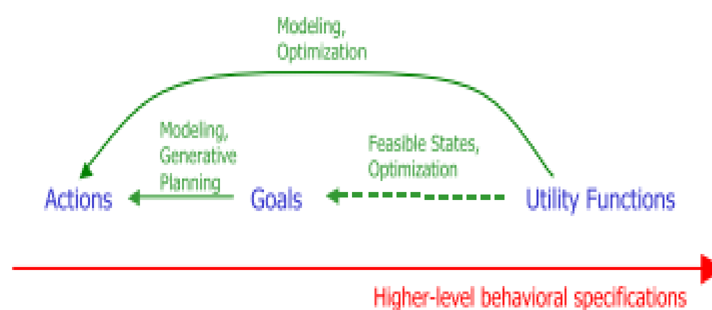

Figure 19: Transition between policies

.We present the policies from the lowest to the highest level of behavioral specifical.

1. **Action policies.** This type of policy is the most simple. This policy dictates the action that has to be taken when the system is in a given *current* state. Typically this takes the form of IF (*Condition)* THEN (*Action*), where *Condition* specifies a state or a set of possible states that satisfy the *Condition*. Note that the state that will be reached it is not specified explicitly. Probably, the author knows which state will be reached upon taking the recommended action, and deems this state more desirable than the states that would be reached via other actions.

2. **Goal policies.** Action policies are good for simple systems, but sometime the system will need stateless actions instead of predefined. Here comes the Goal policy. Rather than specifying exactly what to do in the current state *S,* Goal policies specify either a single desired state *σ*, or one or more criteria that characterize an entire set of desired states. Any member of this set is equally acceptable. Thanks to Goal policies the system does not rely on human explicitly in order to define the action to be taken when the system is in the *current* state. The system is responsible for computing an action *a* that will cause the system to make a transition *t* from the current state *S* to some desired state *σ.* This policy permits greater flexibility and frees the human policy makers of knowing low-level details of system function.

3. **Utility Function Policies.** A Utility Function policy is an objective function that expresses the value of each possible state. Utility Function policies generalize Goal policies. Goal policies are performing a binary classification between desirable and undesirable states. Meanwhile, what Utility Function policy does, is giving a real value on each state, without the desired state being specified in advance. Utility Function policies provide more fine-grained and flexible specifications of behavior than Goal and Action

policies. In situations where Goal policies would conflict, Utility Function policies allow for unambiguous, rational decision making by specifying the appropriate trade-off. In many cases, Utility Functions can require policy authors to specify multi-dimensional set of preferences.

It is instructive to compare our broad definition of policy to the standard definition used in AI, namely that a policy specifies a mapping from any state to the action that should be taken in that state, (Russel, 2003). In our definition, policy is a set of *Action* policies, in which the Conditions fully cover the state space and in which each state is mapped to a unique action. In the AI definition, there are no policy conflicts, and the policy is a single coherent, consistent mapping from state to action (Russel and Norvig, 2003). However, in any autonomic system or component an Action policy must be defined for every state and provide a single unique action for each one. Sadly, conflicts are possible in autonomic systems, because Action policies are created manually by people.

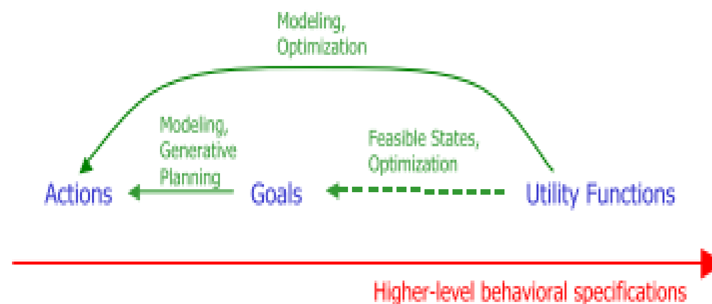

Figure 20: Relathionships between different types of policy

Alternatively, action can be automatically come from the other forms of policies. Figure 20 depicts the relationship between the different types of policies in the unified framework. Goal policies are translated into actions during the system

operation by one out of a variety of methods including generating planning, (Russel and Norvig 2003). This method of generating a sequence of actions in order to achieve the desired goal, takes into account the results of performing an action.

Implementing Utility Function policies requires optimization algorithms. Given that Utility Functions are a function of states, it might appear easy and natural to use optimization to directly identify the most desirable state as a Goal, from which actions can then be derived via planning and/or modeling. In many dynamic autonomic computing scenarios, Utility Function policies would do the optimization online to determine the best action for the current state. Usually, Utility Function policies are viewed by people as generalizations of Goal policies. Indeed, conceptually, a utility function can be defined by specifying a complete set of disjoint goals and assigning values to them. On the other hand, although Action policies are computed by optimizing Utility Function policies, there is no meaningful sense in which Utility Function policies can derive from Action policies because Action policies are defined over the current state space and Utility Function policies are defined over the desired state space.

Although not shown, there may also be self-loops in Figure 20. For example, Utility Functions may be translated into other forms of Utility Functions for usage in multiple levels of decision making. In this case, Utility Functions correspond to the same value system translated into other state spaces. For instance, a Utility Function at one level could specify the relative value of different service levels—to be used for optimizing the performance of a stream of transactions in one part of the system—while a Utility Function at another level could specify the value for obtaining different amounts of computational resources—to be used for optimally allocating resources throughout the system. Generally, to derive some Utility Function B from Utility Function A, a procedure must compute, for each state in the space of B, the optimal value that could be obtained in the space of B. This requires optimization algorithms and a model of how available actions can

transform the state space of B to the state space of A. The next section will demonstrate via a simple example how a service-level utility function by one autonomic component can be transformed into a resource-level function so as to be used by another autonomic component. One can easily imagine other examples in which Goal policies at one level can be transformed into Goal policies at another level.

3.4 Failure classification

Failure classification is always a hard task in computer networks with a complex structure. Both Classification and identification failure can help to choose the right recovery strategy for our system. Generally, a failure can occur in the whole system or in one of its components. The occurrence of a failure is defined as an event at runtime, where the system behavior is different from the expected one. Ghosh (2006), provides a comprehensive fault classification for fault-tolerant systems. Coulouris, Dollimore and Kindberg (1999) provide a classification of faults in regard to distributed systems. Table 6 represents a summary of the identified classes relevant to self-healing research.

Another fault classification is provided by Kopetz, (2011), which partitions failures into dependent on value or timing by nature. A failure can be recognized consistently by all affected parts of the system. Byzantine failure is another approach which is also called the two-faced type and can only be recognized by $3k + 1$ components (k is the number of the tolerated failures). A system can deal with the effects of a benign failure, whilst a malign failure exceeds the recovery capabilities and may cause total failure. Finally, a failure can be identified by the number of occurrences in a given time interval. Permanent failures occur only once and remain in faulty state until repair. Transient failures recover by themselves and can appear repeatedly.

While the system is getting larger and more complex, the failure detection and immediate classification is becoming harder and not such a straightforward task.

Class	**Affects**	**Description**	**Possible detection**	**Possible resolution**
Crash failure	Process	Externally undetectable interruption of a process execution	Local detection methods	State recovery and restart
Fail-stop	Process	Execution is deliberately inhibited on a failure and detected by other processes	Halt on failure property	Stable storage status reconstruction and partition of remaining work
Omission	Process or channel	Message loss generally caused by lack of buffer space	Timeout, checksum	Re-route, re-transmission
Transient	Process or channel	The instantaneous transparent presence of various self recovering faults disturbing other parts of the system	Only side effects	Recovery of side effects

Timing and Performance	Process or channel	Constrained distributed synchronous execution of tasks by a specific amount of time	Timeout	Re-assignment of task
Security	Process or channel	The system is compromised by adversary implied malicious behavior	Behavior dependent	Behavior dependent
Byzantine	Process or channel	Any type of failure may occur. A process confuses the neighbors by providing constantly individual consistent but contradicting information. A communication channel may deliver corrupted or duplicate messages	Process: redundant resend communication	Reconstruction, resend and ignore

Table 6: Failure classes

3.5 Self-healing applications

Though most of the self-healing concepts aren't still fully developed, there are a lot of ways that those concepts can be applied. We will briefly talk about the come up applications of self-healing. These applications implement the concepts of self-healing in their decision support systems, so as to increase the system capability and assist in its recovery from the broken state to the normal state. Grid computing, software agents, middleware computing are some of the promising applications that cater to the needs of the industry. The goal is for these applications to be coupled with the self-healing concepts, so that the decision-making capability will be raised. Also, by enhancing the applications with self-healing concepts, the systems would not need any human intervention and could take their own decisions for their healing. Last but not least we should note that the procedure of bringing the system back to normalcy from the broken state, will significantly decrease maintenance time.

3.5.1 Grid computing

Grid computing by its nature has to self-adapt dynamically to changing environments. The heterogeneous nature of the network and the running modules, the dynamic load balancing requirements, and the ever-changing needs of the user, make adaptation a necessity for grid applications. Cheng et al., (2002), have proposed a unique software architecture for Grid Computing. Its architecture allows the system to be maintained at runtime and can be used as a basis for system configuration adaptation. System monitoring by using “probes and gauges”, error detection and an adaptation scheme of this have already been discussed.

3.5.2 Software agent-based self-healing architecture

Huhns et al., (2003), presented the concept of multi-agent redundancy assemble software adaptation. Multi-agent systems can be used to support conventional systems or traditional software engineering techniques. Using agents as building blocks for software are able to build all the unknown components till runtime. Also agents can be added to a system at run time and software can be customized over its lifetime, even by the end-users too. This can produce more robust systems.

3.5.3 Distributed Wireless File Service application

Distributed Wireless File Service (DWFS) is an application for peer-to-peer file sharing service and it is based on a programming model inspired by biology. It's designed based on the paradigm of the cell division approach (Cheng et al., 2002). This approach is too application-dependent; if the application needs to transmit data, this model holds-up well. Though designing real life complex projects is not possible only by a system state diagram, work is continuing in the process of producing a robust self-healing system that can accomplish a complex task.

3.5.4 Service discovery systems

Service discovery system is used to “discover” the components of the system in different network conditions. Dabrowski and Mills (2002), have proposed an architecture based on adaptation in a service discovery system. The protocol of emerging service discovery systems provides the base for finding components in the system, organizing themselves, and adapting themselves to changes in system topology. This module can help in consistency maintaining mechanisms and failure detection and recovery techniques.

3.5.5 Reflective middleware

Middleware exists in between the operating system and the application level, such as the software that enables the input/output devices that are used today to interact with our personal computers. Technologies like JAVA, C++, C# and other programming languages are some examples of industry standard middleware technologies. Using these technologies, we are able to hide from the user the technical detail of network communication and other system operations. Usually the codes that run on top of the middleware are portable and the users do not bother about the operating system or network detail.

Some applications that run on top of the middleware can enhance their performance if they are aware of the facts in the underlying details. For example, in a client-server architecture, if the system is aware of the resource utilization or the middleware's request scheduling process, it can improve the load of the system or create replicas of its most used services.

Kon, Costa, Blair and Campbell (2002), have a reflective middleware model. Unlike the traditional middleware, it is represented as a compilation of various components that are reconfigurable. With this model we can change the network protocols, the security policies and other mechanisms to improve the performance of the application without "touching" the interfaces of the middleware. In the next subsections we will discuss about approaches on reflective middleware.

3.5.5.1 dynamicTAO

dynamicTAO, (Kon et al., 2002), was created to enable the reconfiguration of the TAO ORB. The Component Configurators here are the typical C++ objects, and they store the relation between ORB components and application components as a list of references. dynamicTAO supports the dynamic reconfiguration of the middleware components. Whenever a request for a component turns up, the

system inspects the Component Configurators object, in order to check the dynamic dependencies between the component, the middleware and other application components. This architecture makes the dynamic load and unload of modes easy, through changing the ORB's configuration. dynamicTAO offers support for the interceptor, which is a part of TAO. We will discuss the interceptor approach to a reliable system later.

3.5.5.2 Open ORB

The Open ORB, (Blair et al., 2002) project architecture, allowing the/its components to remain identifiable, manages to expedite runtime configuration. As for the levels of this architecture, one can clearly see the distinction between them/ between the base and the Meta levels. On the one hand, the base level is responsible for the usual middleware services, whereas, on the other hand, the Meta level deals with (the facilitation of) the reflective actions, adaptation, etc. Furthermore, the Open ORB supports/promotes behavioral reflection by supporting/promoting different meta-models, including interception and the resource meta-model.

3.5.5.3 Interceptor-based approach

The Eternal system, (Narasimhan, Moser,& Melliar-Smith, 2002), is a CORBA 2.0 compliant system, that boosts the fault tolerance nature of CORBA with replication. Narasimhan, Moser and Melliar-Smith (1997) have proposed the Eternal architecture, based on the interception approach, by capturing Internet Inter-ORB Protocol (IIOP) specific system calls made by the ORB. The interceptor calls were originally directed by ORB to TCP/IP, but are now mapped onto a reliable ordered multicast group's communication system. In the Eternal system, a replicated object

enables any client object to address the replicas of a server object as a whole, using a unique object group's identifier.

3.5.6 GRACE approach

The goal of the GRACE project is to develop an integrated cross-layer adaptive system, where the hardware and all the applications will be able to adapt to the ever-changing demands of the system resource constraints on energy, time and bandwidth, while also providing the best possible Quality of Service. The architecture proposes that all system layers will be able to adapt in response to any system or application changes. Adve et al., (2002), suggest that the various layers of a system such as hardware, the operating systems, the scheduler layer and network protocols can be coordinated to adapt when the system changes, even if change occurs in its resources or in its applications demands. This architecture is mostly used on applications running on resource constrained systems. Mobile devices are the ideal candidates for this architecture.

3.5.7 Clustering

Clustered systems are showing interest in self-healing strategies. In order to manage application inter-dependencies, in Adger and Hughes (2004), a scheme is suggested, which ensures the allocation of various resources to the running systems, as and when needed, with no service interruption. This scheme differs in many aspects from the current available clusters such as Solaris, HPUX or AIX, in that the latter are fault-tolerant but not self-healing and, most of the time, they are not secure.

In this section, we generally discussed about Self-healing. We briefly talked about the Self-healing loop and the whole idea behind it, as well as about the Self-

healing states, and what strategies are being used in order to ensure/verify whether a state is acceptable. Afterwards, we focused on Self-healing policies that specify what is to be done if a condition/situation is met. Moreover, we reviewed Classification failure and how it can be identified in the realm of Self-healing. Last but not least, we presented some already existing Self-healing applications.

In the next section we will look deeper into the world of the Internet of Things and we will talk about how self-healing can help Internet of Things become self-sustained.

Section 4: Self-healing in IoT

In the previous chapter we discussed about self-healing and all of its components (self-healing loop, self-healing policies, self-healing states, failure classification) and as well as they all work together to keep the system up and running. In this chapter we will talk about and compare different self-healing architectures that are are being developed and used in Wireless Sensor Networks (WSNs) and Internet of Things (IoT).

WSNs are usually used for the military, smart homes, intelligent environments, or other ubiquitous applications and are usually deployed to operate for a long period of time. The goal that are WSNs are trying to achieve is to sense some events and carry these sensory data to a base station. Thus, availability is very important for a long-term use of WSNs.

Moore's law, (Moore, 1965) predicted that the number of transistors in a cost effective chip and therefore, the processing or storage capacity of that chip, doubles every year. Following that pattern up until 2019 will result in making the vision of the smart dust, (Kahn, Katz, & Pister 2000), a reality. Mark Weiser has already stated back in 1991 that "In the 21st century the technology revolution will move into the everyday, the small and the invisible..." and "The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it."

Today there is a need for sensors to be fault-tolerant and it able to be deployed to inaccessible areas such as the battlefield, where maintenance would be inconvenient or even impossible. There are usually two ways, (Römer & Mattern, 2004), for those sensors to be deployed in such environments:

- Sensors can be positioned far from the actual phenomenon, i.e., something known by as sense perception. In this approach, large sensors that use complex techniques to distinguish the targets from environmental noise are required.
- Several sensors that only perform sensing can be deployed. The positions of the sensors and the communications topology are carefully engineered. They transmit time series of the sensed phenomenon to the central nodes where computations are performed and data are fused.

Normally a common sensor network is composed of a hundreds of thousands of sensor nodes that are placed very close to the area of action. The position of the nodes should be in a totally manner. This means that the algorithms that run on those nodes have to be able to possess self-organizing capabilities, because deploying and maintaining the nodes must remain inexpensive.

4.1 Autonomic Wireless Sensor Networks

In most cases, sensor nodes of a wireless sensor network are deployed on remote areas where maintenance and administration are impracticable. The size of a node varies from a box of a shoe box to a tiny particle (e.g. for military applications where sensor nodes should be almost invisible). Likely, its cost ranges from hundreds of Euros to a few cents. Each device is composed by a computational unit, a wireless communication unit, a sensing unit (one or more sensors), a logic unit (software) and a power unit. As we already mentioned, the maintenance in the most of the cases is impartible, so, changing or recharging the batteries is not possible. Depending on the application, these sensors may last a couple of hours or several years.

The design and development of energy efficient systems in environments that impose severe restrictions is not a trivial task. Considering the given

characteristics, the system should be as autonomic as possible, meaning that the wireless sensor network should manage itself with the least or no human intervention. Being autonomic is not an easy task. That's why WSNs have adapted the concept of the autonomic manager that we discussed earlier. The autonomic manager provides self-management services using monitoring, planning, analyzing and executing modules.

AWSNs have the base properties of the autonomic manager like self-management, self-configuration, self-optimization, self-healing and self-protection. However, it also introduces some new properties like self-service, self-awareness, self-knowledge and maintain (Ruiz et al., 2004).

- Self-awareness: allows the entity to know its environment and its activities context and act accordingly. It finds and generates rules to best interact with its neighbor entities.
- Self-knowledge: the management service that enables an entity to know itself. For example, an entity that governs itself should know its components, current state, capacity, and all its connections with other entities. It needs to know the extension of its resources that can be lent and borrowed.
- Self-maintain: allows an entity to monitor its components and fine-tune itself to achieve pre-determined goals.

AWSNs autonomic manager uses the same functions that already were discussed at the chapter 3.1.1. These functions are the monitor function, analyze function, plan function and execute function.

4.2 Architectures

In this section we will talk about different architectures that are being developed in the field of Self-healing in Internet of Things. We will look into each architecture as a different sub-chapter and we will try to understand the results of the experiments that have been done.

4.2.1 Service Management System For Self-healing

The system that we will describe next, tries to detect and identify failure and proposes adjusts to the network infrastructure, in order to maintain the service availability.

4.2.1.1 Architecture Description

Sensor nodes have strong hardware and software restrictions in terms of processing power, memory capability, power supply, and communication throughput. The power supply is the most critical restriction, given that it is typically not rechargeable. That is why faults are likely to occur frequently and not as isolated events. Besides, large-scale deployment of cheap individual nodes means that node failures from fabrication defects will not be uncommon.

In military applications, where these networks are deployed in open spaces or enemy territories, adversaries can manipulate the environment (so as to disrupt communication, for example by jamming), but also have physical access to the nodes. At the same time, ad-hoc wireless communication by radio frequency means that adversaries can easily put themselves in the networks and disrupt infrastructure functions (such as routing) that are performed by the individual nodes themselves. Finally, the sensor nodes are exposed to natural phenomena like rain, fire, or even falling trees since they are commonly used to monitor

external environments. Therefore, failure detection and fault management plays a crucial role in wireless sensor networks. If, in addition to detecting a failure, the management application can also determine the reasons of the failure and distinguish between malicious and non-malicious origins, it can trigger security management services or, in case it is an accidental or natural failure, activate “backup nodes”.

In certain applications such as measuring the humidity, temperature these applications are called environmental and the sensor nodes will be programmed to send back their measurements at regular intervals. These networks are called programmed and continuous. We also have the event-driven applications where the network sends back the measurements when a “special” events occurs. Event-driven networks are really attractive solution, because they reduce the transfer of the unnecessary messages. This results in reducing the energy-consumption. The drawback of event-driven networks it is, that its harder to implement, a logic that will recognize a failure. In event-driven networks, if the management application stops receiving data from certain nodes or entire regions of the network, it cannot distinguish whether a failure has occurred or whether no event has occurred.

Ruiz et al., (2004), use a homogeneous hierarchical network. The nodes are grouped into clusters, while there is a special node called cluster head. A cluster head node has more resources and, thus, is more powerful than the common nodes. Furthermore, cluster heads are responsible for sending data to a base station. The base station communicates with the observer, which is a network entity or a final user that wants to have information about the data collected from the sensor nodes. During their implementation, the management agents are execute in the cluster heads where aggregation of management and application data is performed. This mechanism decreases the information flow and energy consumption as well. A manager is located externally to the sensor network where

it has a global vision of the network and can perform complex tasks that would not be possible inside the network.

The failure detection in such a sensor network can be done in the following way:

1. In the installation phase, that occurs when the nodes are deployed in the network, each node finds out its position in the network area and reports it and its energy level to the agent located in the cluster head.

2. In the operational phase, during which the nodes are “working” - collecting and sending data, management activities take place. One of these activities is energy level monitoring that plays a central role. This information is also transmitted to the manager, which can then recalculate the energy and topology maps. Also, operations can be sent to the agents in order to execute the failure detection management service. The manager sends GET operations in order to retrieve the node state. The GET-RESPONSEs are used to build the network audit map. If an agent or a node does not answer to a GET operation, the manager consults the energy map to verify if it has residual energy. If so, the manager detects a failure and sends a notification to the observer.

Assunção, Ruiz and Loureiro (2006), proposed an extension of this model. The extension was an autonomic manager, located outside the network, and it is responsible for mapping the Service Level Agreement (SLA) into the so called “policies” for the network nodes, to monitor the service quality and availability and, if necessary, to renegotiate the SLAs. These policies are stored in the knowledge bases and the autonomic manager located in the sensor node uses this information to start the monitor, analyze, plan and execute functions. Autonomic managers on cluster head nodes guarantee that the service level is being attended inside the cluster and adjust the network components to attend these levels.

Common nodes autonomic managers are used to keep track of the resources, to optimize the nodes functioning, to detect unusual behavior, to analyze events and to adjust the nodes configuration in order to reduce the risk of faults.

Assunção et al. usee some concepts of IT Infrastructure Library (ITIL) in the definition of four autonomic managers with the purpose of creating a self-healing wireless sensor networks, namely:

- Autonomic Service Level Manager: An autonomic manager that keeps track of the SLA using manual manager and policies. This kind of manager has to guarantee the fulfillment of the SLAs. Also, it can redefine them.
- Availability Autonomic Manager: The autonomic manager that plans and manages the service availability through monitoring of the IT service availability.
- Continuity Autonomic Manager: This autonomic manager is responsible for identifying network risks and possible failures. Moreover it can create recover and risk reduction plans.
- Capacity Autonomic Manager: The autonomic manager that monitors nodes resources and identifies demands. In case of a current or future insufficient capacity this manager is responsible to reallocate resources and anticipate new resources.
- Each one of these managers employs concepts of Service Support disciplines to accomplish the monitor, analyze, plan and execute function, considering the self-healing service. See figure 21.

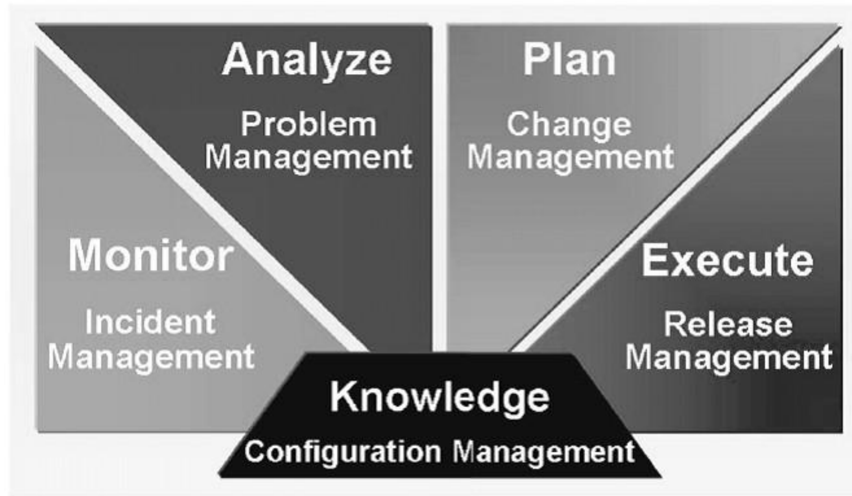

Figure 21: Autonomic element

4.2.1.2 Experiments

The experiments that are will now be presented for this architecture are the two most common problems that wireless sensor networks face. The first experiment that runs is that an unexpected event puts the nodes out of operation and the second deals with the problems caused by traffic congestion and energy loss. The aim of the experiments in the first part is to evaluate the impact of management functions over the wireless sensor network, analyzing the management costs and identify the effectiveness of the management architecture in detecting failures. The experiments in the second part are used to investigate the impact of service management regarding the power consumption and data flow.

In the first experiment, three scenarios are simulated. In scenario 1 and scenario 2 the network is simulated with all of its fault management functionality, but in the scenario 2 the failure detection function was removed. Finally, in scenario 3 the network is simulated without any fault management functionality. During the simulation, 32 nodes are located at the center of the network, when an unexpected

event causes their failure. The event occurs at 45 s of simulation (100 simulation time). In figure 22 we see network hierarchy.

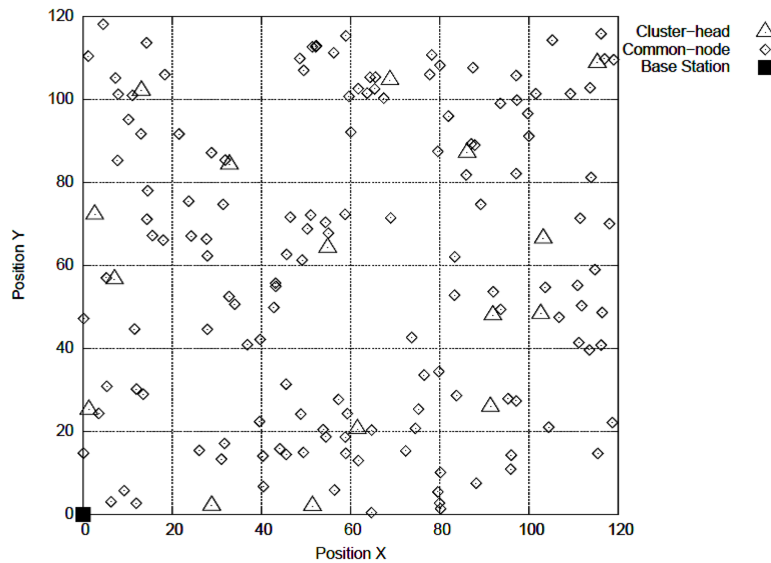

Figure 22: Hierarchical network comprised of common nodes, cluster heads and a base-station

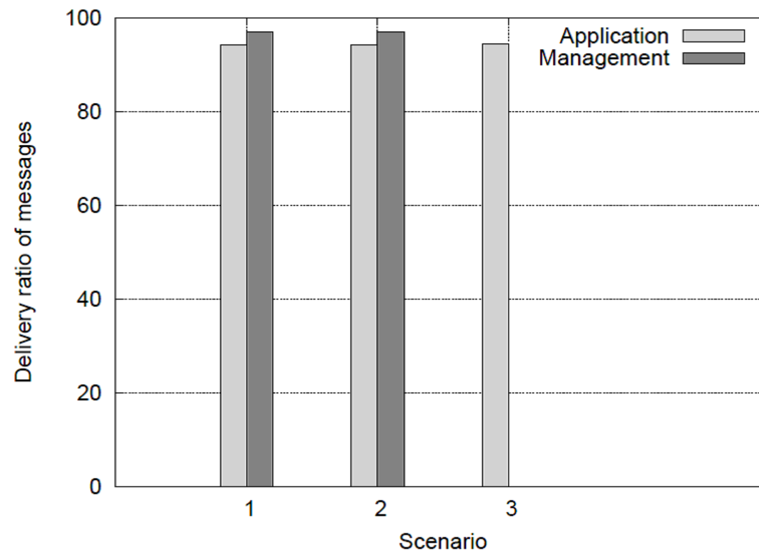

Figure 23: Delivery rate of message

As we take a closer look at figure 23, we notice the delivery rate that measures the ratio of the messages that received by the nodes in the network. As expected the delivery rate is similar in all scenarios, since the messages are transmitted over the same wireless network. Neither the limited functionality in scenario 2, nor the removal of the management has any influence.

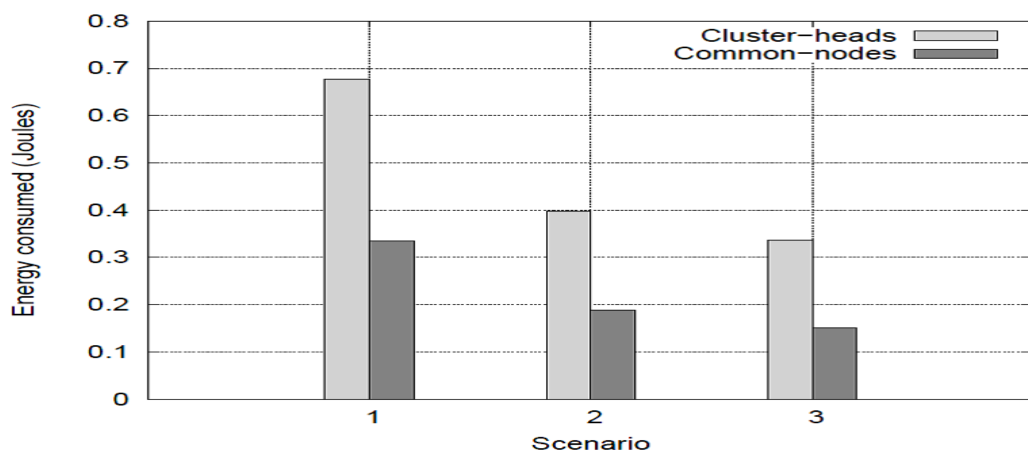

Figure 24: Energy consumption

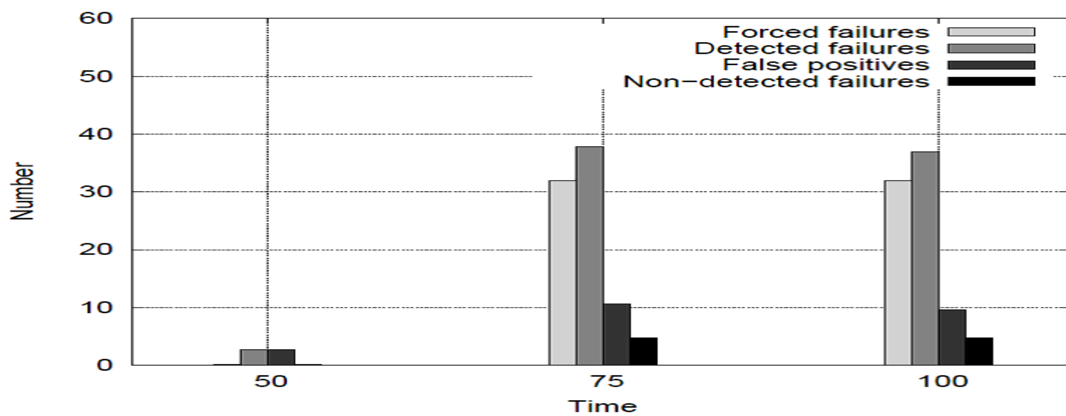

Figure 25: Detection effectiveness for centered failures (scenario 1)

Figure 24 illustrates the energy consumption of common nodes and cluster heads. It is observed that, as long as the detection of failure is deactivated, the energy-consumption of common-nodes with management appears to be increased by 18% for the cluster heads and 29.45% for the nodes. When the failure detection functionality was available the management energy consumption increased by 101.2% for the cluster heads and 129.45% for the nodes.

According to Lanthaler (2008), this is an expected result since the transmitting and receiving messages is an expensive action.

Figure 25 depicts the detection of failures in scenario 1. The x-axis is the time that the manager reports the availability of nodes. It can be observed that there were some detection failures in time 50s, although at this time the nodes code was not perceived because the drop of GET request/response messages. This caused the manager to be misled and to produce false positives. What happens is that, after the unexpected event occurs, some common nodes, which were not harmed, lost their cluster heads if they are located inside the damaged region. Consequently, these common nodes stop receiving the GETs from the manager, since they are sent to them through the agents. As a result, the manager does not receive answers from these common nodes provoking false positives. In this scenario the “orphan” nodes were 8. Scenario 5 has similar results as scenario 1.

Scenario	Description
1	32 nodes (20% of the network, composed of 3 cluster heads and 29 common nodes) located at the center of the network are harmed. These nodes have x and y coordinates between 30 and 90.
2	41 nodes (25.63% of the network, composed of 4 cluster heads and 37 common nodes) located near the base station are harmed. These nodes have x and y coordinates between 0 and 60.
3	39 nodes (24.37% of the network, composed of 4 cluster heads and

	35 common nodes) located far from the base station are harmed. These nodes have x and y coordinates between 60 and 120.
4	14 nodes (8.75% of the network, composed of 1 cluster head and 13 common nodes) located at the center of the network are harmed. These nodes have x and y coordinates between 40 and 80.
5	62 nodes (38.75% of the network, composed of 6 cluster heads and 56 common nodes) located at the center of the network are harmed. These nodes have x and y coordinates between 20 and 100.

Table 7: Description of each scenario

Figure 26 shows the results of the scenario 2. At first glance it seems that it is the same as the scenario 1. In a way it is. The results at the time of 50 s are pretty much the same. However, as far as points 75 and 100 are concerned, it is possible to observe considerable dissimilarities. The number of false positives has decreased to 10.26% (point 75) and 10.36% (point 100) of the detections. The reason is that in this experiment the number of orphan nodes is only 4, i.e., two times less than the number of orphan nodes for scenario 1. The results for the scenarios 3 and 4 are very similar to these results.

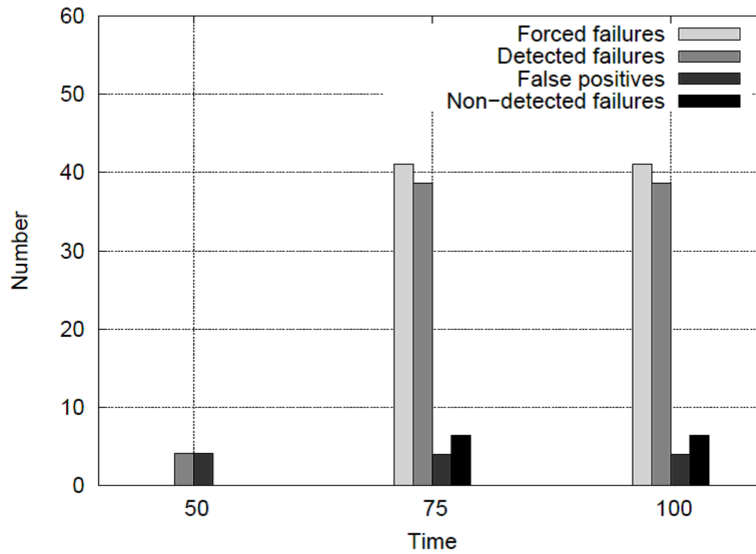

Figure 26: Detection effectiveness for failures near the base station (scenario 2)

In the second part of the experiments they evaluate the impact of the service management on restoring the network from the failure. The experiment will test the system on communication problems due to traffic congestion and energy loss. The system has to detect the root of those failures and restore them.

The knowledge source of the autonomic manager is updated whenever it receives messages or the configuration parameters are changed. The cluster heads nodes (that contains the manager) store in the knowledge source their local information and some replied information of each node of the cluster.

In this experiment two scenarios were implemented. Scenario 1 was the service management system for self-healing wireless sensor networks and all its functionality is activated and scenario 2 where all the functionality of management system is missing. All the characteristics of the simulation are represented at table 8.

Parameter	Value
No. of nodes	24 (20 common nodes and 4 cluster heads)

Cluster size	5 nodes
Simulation time	155 s
Number of simulations	33
Coverage area	50 m × 50 m
Environment conditions	Variations in the environment and noise are not considered
Initial energy available in each node	5 Joules in common nodes and 100 J in cluster head nodes
Network type	Heterogeneous
MAC protocol	IEEE 802.11
Energy spent in transmission (reception)	36 mW (24 mW) for common nodes and 600 mW (300 mW) for cluster head nodes
Transmission range	40 m for common nodes and 250 m for cluster head nodes
Processing consumption	24 mW in common nodes and 360 mW in cluster head nodes
Node capacity	Space for 10 messages in common nodes and 100 messages in cluster head nodes
Energy spent in sensing	15 mW

Sensing and disseminate type	Programmed
Node mobility	Stationary

Table 15: Characteristics of the simulations

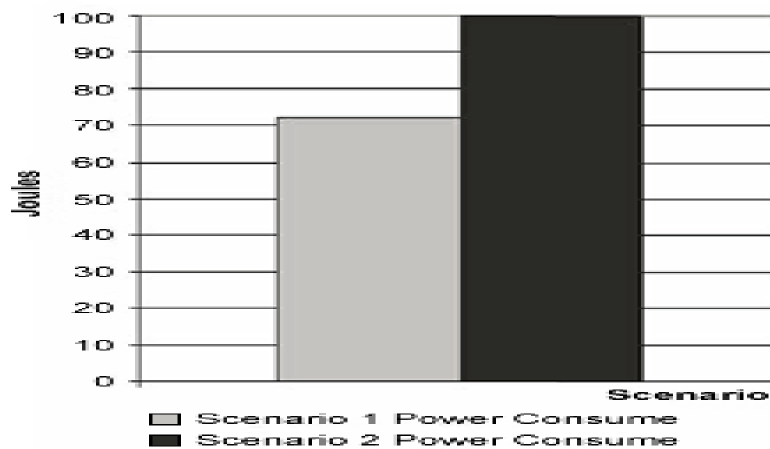

Figure 27: Common nodes energy-consumption

In every application of wireless sensor networks energy is the most crucial part of the network. If the node is not powered enough this leads to network productivity reduction.

In scenario 2 the sensor nodes exhaust their energy at 75 s and then have a communication problem for 25 s. On the other hand in scenario 1 the nodes that implement the self-management system were able to survive through the whole simulation and having support the communication for all the 60 s. From the figure 26 we see that the scenario 2 had more power consumption than the scenario 1. This happened because the nodes detected that messages are being lost and diminish their production. Also, in figure 27 we see that has similar consumption, although in scenario 1 the cluster head nodes were delivered more messages.

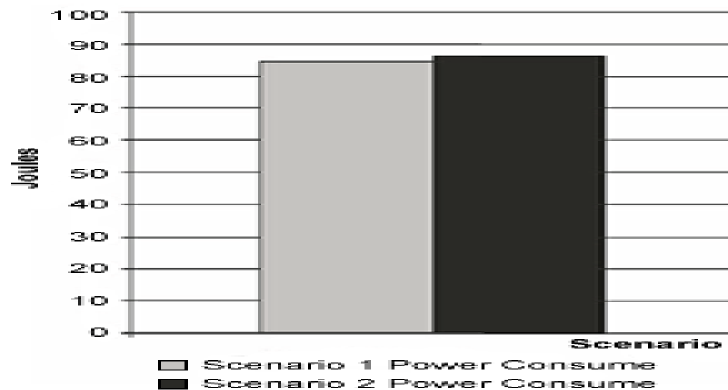

Figure 28: Clucter-head power consuption

In scenario 1 the data sent by the common nodes of scenario 1 is 10.8 times greater than the scenario 2. However, the amount of data received is only 1.3 times greater. Nevertheless, the number of messages delivered to the cluster head node is bigger if compared to the one of scenario 2. This demonstrates that the number of dropped messages in the network was reduced. The amount of data sent by the cluster head nodes to the base station in scenario 1 is 2.16 times greater than in scenario 2. Also, the amount of dropped messages in scenario 2 is 84 times greater than in scenario 1. That is obvious given the fact that, in the scenario 2 there is not any implementation of detecting the communication problems.

In scenario 1 the data sent by the common nodes of scenario 1 is 10.8 times greater than the scenario 2. However, the amount of data received is only 1.3 times greater. Nevertheless, the number of messages delivered to the cluster head node is bigger if compared to the one of scenario 2. This demonstrates that the number of dropped messages in the network was reduced. The amount of data sent by the cluster head nodes to the base station in scenario 1 is 2.16 times greater than in scenario 2. Also, the amount of dropped messages in scenario 2 is 84 times greater than in scenario 1. That is obvious given the fact that, in the

scenario 2 there is not any implementation of detecting the communication problems.

4.2.2 A Self-managing Fault Management Mechanism for Wireless Sensor Networks

Asim, Mokhtar and Merabti, (2010) previously proposed a cellular (self-healing) approach for fault-detection and recovery. Now, they are extending this approach to make use of a fault management mechanism, in order to take care of fault-detection and recovery.

4.2.2.1 Architecture Description

This architecture can be divided in two phases:

- Fault detection & diagnosis
- Fault recovery

Fault-detection can be accomplished by two mechanisms:

- self-detection (or passive-detection)
- active-detection

Fault-detection can be accomplished by two mechanisms, self-detection and active-detection as shown in figure 28. In self-detection, the nodes periodically monitor their energy and identify any potential failures. In the proposed architecture, battery depletion is considered as the main reason for the sudden death of the nodes. A node is marked as “failing” when its energy drops below a threshold value. When this happens, the “failing” node sends a message to its cell manager informing it that it is going into sleep mode due to lack of energy. It’s not

possible for the nodes to self-recover from this failure, so there are no recovery steps. Self-detection is a local computational process of sensor nodes, and does not require full network communication to preserve the nodes energy. In addition, it also reduces the response delay of the management system towards the potential failure of sensor nodes.

The proposed scheme uses an active detection mode to efficiently detect the sudden death of the node. In active detection, the cell manager asks from its cell members to send their updates on a regular basis. The cell manager sends “get” requests to the associated common nodes on regular basis and, in return, the nodes send their updates. This is called the in-cell update cycle. The update message consists of the node ID, energy and location information. Figure 27 also depicts the exchange of update messages that takes place between the cell manager and its cell members. Unless the cell manager receives the update message, it sends an instant message to the node acquiring its status. If the manager does not receive any message once again, then it declares the node as faulty and passes this information onto the remaining nodes. Cell managers concentrate only on their cell members and inform only the group manager if the network performance of its small region has reached a critical level.

A cell manager also employs the self-detection approach and regularly monitors its residual energy status. All sensor nodes start with the same residual energy. If the node becomes less than or equal to 20% of the battery life, then it is ranked as “low energy node” and it becomes liable to put to sleep. If the energy is above or equal to 50%, it is ranked as “high energy node” and it becomes a promising candidate for the cell manager. Thus, if a cell manager is ranked as a “low energy node”, it then triggers an alarm, and notices the cell members and the group manager of its low energy and appoints a new cell manager to replace it. Every cell manager sends health status information to its group manager. This is called an out-cell update cycle and is less frequent than an in-cell update cycle. If a group manager does not hear from a particular cell manager during the out-cell

update cycle, it sends a quick reminder to the cell manager and enquires about its status. If the group manager does not hear from the same cell manager again during second update cycle, it then declares the cell manager faulty and informs its cell members. This approach is used to detect the sudden death of a cell manager. A group manager also monitors its health status regularly and responds when its residual energy drops below the threshold value. It notifies its cell members and neighboring group managers of its low energy status and sends an indication to appoint a new group manager. The sudden death of a group manager can be detected by the base station. If the bases station does not receive any traffic from a particular group manager, it then consults the group manager and asks for its current status. If the base station does not receive any acknowledgment, it then considers the group manager faulty (sudden death) and forwards this information to its cell managers. The base station primarily focuses on the existence of the group managers from their sudden death. Meanwhile, the group managers and cell managers take parts mostly in passive and active detection in the network.

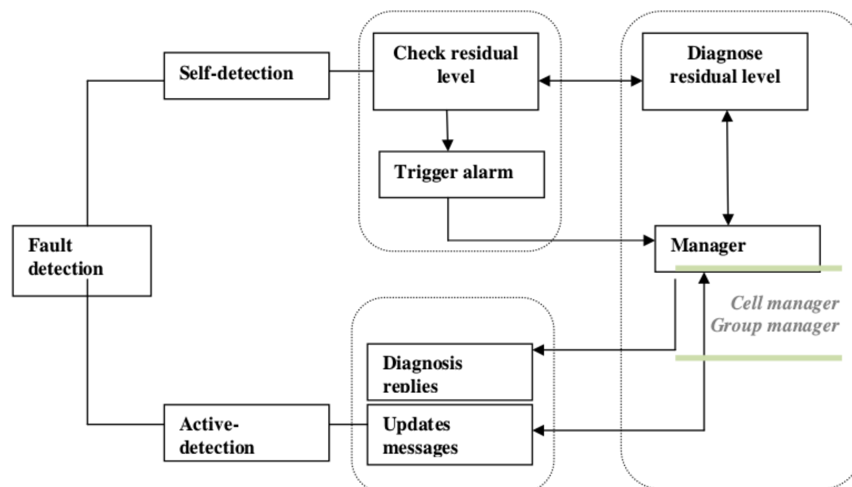

Figure 29: Fault detection and diagnosis process

Fault recovery refers to the process taking place after nodes failure detection (as a result of self-detection or active detection). Sleeping nodes can be awaken so as

to cover the required cell density or, alternatively, mobile nodes can be moved to fill the coverage-hole. Cell managers and secondary cell managers are known to their cell members. If the cell manager energy drops below the threshold value, it sends a message to its cell members and its secondary cell manager. This message also acts as a hint for the secondary manager to start operating as the main cell manager, while the existing cell manager becomes a common node and goes into a low computational mode. Common nodes will start treating the secondary manager as the main cell manager and the new cell manager, upon receiving the updates from the nodes, will choose a new secondary cell manager. The failure recovery mechanisms are performed locally by each cell. In figure 28 we see the process of replacing the cell manager. Let's assume that cell 1 cell manager is failing due to energy depletion and node 3 is chosen as a secondary cell manager. The cell manager sends a message to its cell members 1,2,3,4, which invokes the node 3 to stand up as the new cell manager.

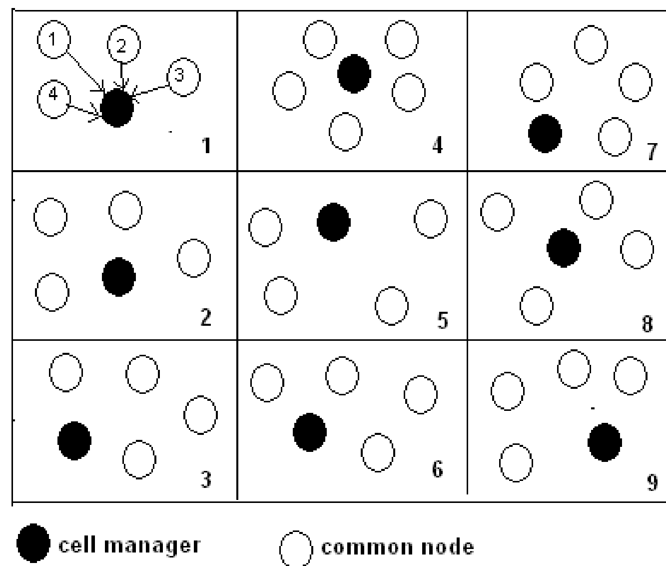

Figure 30: Virtual grid of nodes

In the proposed architecture there is also a scenario where the energy of the cell manager and the energy of the secondary cell manager are not sufficient enough

to replace the cell manager. In this case, common nodes send exchange energy messages within the cell to appoint a new cell manager, whose energy is above 50%. Moreover, if there is no candidate to replace the cell manager, the event cell manager sends a request to its group manager to merge the remaining nodes with the neighboring cells.

From the perspective of the group manager, when it detects a sudden death of a cell manager, it informs the cell members of that faulty cell manager. This is, again, an indication for the secondary cell manager to start acting as the new cell manager. If the residual energy of the group manager drops below the threshold value (i.e. greater or equal to 50% of battery life), it may downgrade itself to a common node or enter a sleep mode, and notify its backup node to replace it. The information of this change is propagated to the neighboring group managers and cell managers within the group. As a result of the group manager‘ s sudden death, the backup node will receive a message from the base station to start acting as the new group manager. Unless the backup node has enough energy to replace the group manager, cell managers within a group co-ordinate to appoint a new group manager for themselves, based on residual energy.

Each cell maintains its health status in terms of energy levels. It can be High, Medium or Low. During the out-cell update cycle the health status of a cell is reported to its associated cell manager. When the group manager receives these health statuses, it tries to predict and avoid future failures. High health status means that the group manager will always recommend using the cell in any operation, while a medium status indicates that the group manager should occasionally use this cell. Low health status means that the node doesn’t have a sufficient energy and it should stop being used for any operation. The cells with their energy being low are being joined with the neighboring cell.

The fault management that is proposed by Asim et al. (2010) relies on message broadcasting and exchanging messages among the sensors of the network. Thus, this might cause an over flooding by broadcasting messages from different sensor

nodes. To solve this problem, they created a message filtering mechanism to further reduce the redundancy of message exchange. The message format is showed in table 9.

Field	Description
Group_id	The group id
Cell_id	The cell manager id
Timestamp	The message sending out time
Curr_energy	The current node energy level

Table 16: Message fields

The Group_id field is used to determine whether the received message belongs to the same group of current node. If not, the message will be dropped to avoid unnecessary message rebroadcast. Cell_id helps a node to decide whether the message belongs to its cell manager, if not it drops the message. The Timestamp field is used to help the node to distinguish a message. If the receiving message is a new one, it will be processed and forwarded to the neighboring nodes, otherwise the message will be dropped.

4.2.2.2 Experiments

Asim et al., (2010), uses GTSNETS, (Riley, 2003), as a simulator platform and the same radio model as proposed in Gupta and Younis (2003). In table 10 we see the parameters of the experiment.

Parameter	Value
No. of sensors	40 – 80 sensors
No. of simulations	30
Coverage area	120 m × 120 m
Initial energy available in each node	2000 mJ

Table 17: Experiment parameters

In their work, Asim et al. (2010), they firstly compare it with the Venkataraman, Emmanuel and Thambipillai algorithm, (2008), which is based on failure detection and recovery due to energy exhaustion.

The failure detection in the Venkataraman algorithm starts after the cluster formation. In this algorithm, the information about the neighbor is already known by the nodes through the exchange of the hello messages. When a node fails, its parents and its children have to take an appropriate action to fill the gap that the failed node formed. The node that is about to fail reports the possibility to fail so that appropriate actions can be taken to reduce the "damage" in the network. The fail_report_message is only passed to immediate hop members and then later on, passed to the cluster head. In the proposed algorithm (Asim et al., 2010), if the nodes energy drops below a specified threshold value, then it sends a failure to its cell manager and goes to low computational mode. In Asim's work, (2010), there are only two types of nodes: common node and cell manager. Only one failure is reported to the cell manager, which reduces the energy consumption and it will "break" the network operation.

As we mentioned before, in the Venkataraman algorithm, the failure node sends a fail_report_message, and, by receiving this message, a child has to send a join_request_message to its neighbors. Afterwards, all the neighbors that have received this message have to respond with a join_reply_message or join_reject_message messages. Then, the healthy child of the failing node tries to find a suitable parent. A parent is considered suitable when its energy levels are high so it won't fail. In contrast, in the proposed architecture, common nodes (child nodes) does not require any recovery mechanism but goes on low computational mode after informing their cell manager (parent node).

In the Venkataraman algorithm, when a cluster head fails, it causes its children to exchange messages that are necessary for electing the new cluster head. Children with low energy aren't concerned among the candidates. The child with the highest energy value is the one that becomes the new cluster head and, therefore, the parent for the children of the failing cluster head. Also, if the child of the failing cluster head node is failing as well, then it also requires appropriate steps to get connected to the cluster. In the work of Asim et al., (2010), they employ the backup manager (cluster head) will replace the cell manager in case of failure. When the common nodes (child nodes) of the manager receive a failure message, the backup (secondary) manager automatically starts acting as the new cell manager and no further action is needed, since common nodes are aware of the backup manager.

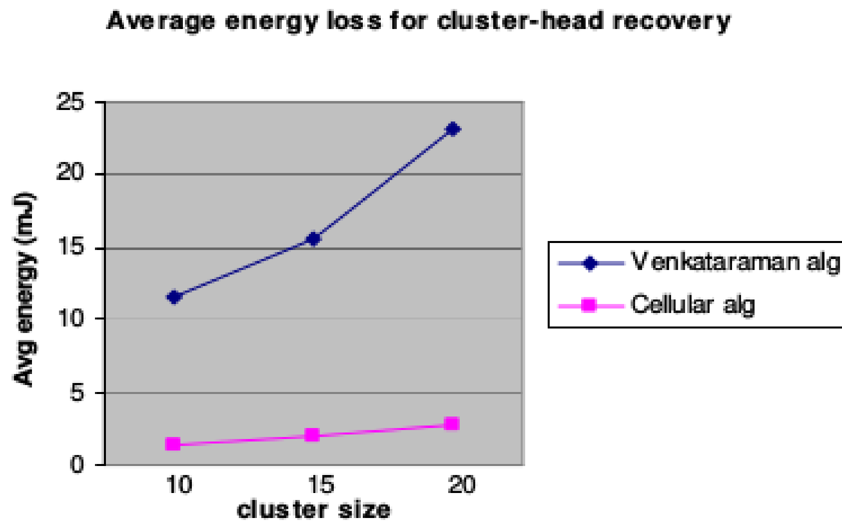

Figure 31: Average energy loss for cluster head recovery

Figure 31 depicts the average energy loss in each architecture. As we see, the Asim et al. (2010), architecture consumes less energy for cluster heads in comparison to the Venkataraman et al. (2008). In the Venkataraman algorithm, message exchange for the election of new cluster manager is both time and energy consuming. In Asim et al. (2010), the cell manager sends one message only to its members in order to recover from a failure.

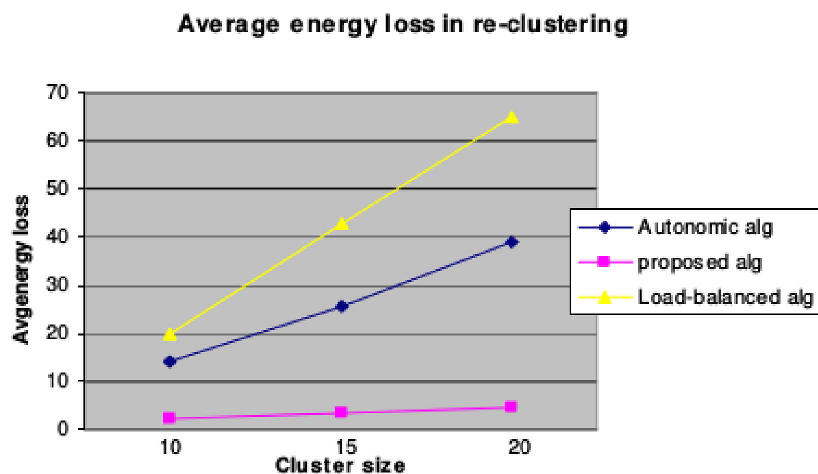

Figure 32: Average energy loss in re-clustering

Figure 32 shows the average time for cluster head recovery. The proposed algorithm has a quicker recovery compared to Verkataraman's. Asim et al. (2010) can also be compared to two other algorithms: the autonomic self-organizing architecture (Wang, 2008), and the load – balanced clustering (Gupta & Younis 2003), in the essence of energy consumption for cluster head recovery. In figure 32 it can be observed that the Asim et al. (2010) algorithm for cluster head recovery consumes less energy.

In the autonomic self-organizing algorithm, when a high level node (header) fails to operate or needs to step down due to low residual energy, all the sensor nodes from the failed header need to join other available header nodes using the same mechanism. This, once again, is not an energy efficient way to re-organize the cluster and is also time consuming compared to the Asim et. al., cellular approach. In load-balanced clustering, (Gupta & Younis, 2003), when a gateway fails, the cluster dissolves and all its nodes are re-allocated to other healthy gateways. This consumes more time and energy, as all cluster members are involved in the re-clustering process. In our proposed algorithm, only a few nodes are involved in the re-clustering.

4.2.3 A Dendritic Cell Algorithm for Security System with Self-healing property

De Almeida, Ribeiro and Ordonez (2015), propose an architecture that implements a Dendritic Cell Algorithm and is based on the essence of self-* properties that we discussed earlier. In the proposed architecture, they are trying to eliminate the security threats such as Jamming, Sinkhole, Hello Flood, Flooding.

4.2.3.1 Dendritic Cell Algorithm

Before diving in the proposed architecture, we first have to take a look on the Dendritic Cell Algorithm (DCA). The DCA was introduced by Greensmith, Aickelin and Cayzen (2005), and was inspired by the danger theory of mammalian immune system. The main elements of DCA are the following: Dendritic Cells (DC), Lymph nodes and antigens. DCs have two types of signals, the input signals that are: danger signal, safe signal, PAMP (pathogenic associated molecular patterns) and inflammatory signal and the output signals that are: Costimulatory Molecules (CSM), semi-mature signal and mature signal.

The antigens are the input of DC and they are presented iteratively to dendritic cells. Each antigen increments the CSM. When the CSM pass the migration threshold, the DC migrates to lymph node. The danger signal and PAMP increments the mature signal of DC and the safe signal increments the semi-mature signal. The inflammatory signal raises all other signal increments.

When the DC achieves the migration threshold, it will move into lymph node and the DC will be labeled as mature or semi-mature, comparing the mature and semi-mature signals. After receiving a defined number of DCs, the lymph node will calculate the Mature Context Antigen Value (MCAV) that is the percentage of mature DCs per all DCs received. The Dendritic Cell Algorithm detects an attack if the MCAV surpass a defined threshold.

4.2.3.2 Architecture Description

The proposed architecture is based on MAPE-K loop that we discussed earlier. Each phase of the MAPE-K loop (monitor, analyze, etc.) is treated as a component and is distributed between nodes. The architecture is based on the RPL Destination-Oriented Directed Acyclic Graph (DODAG). This is the default topology

of a 6LoWPAN network that uses the RPL protocol. In Figure 33 we see a typical RPL DODAG, that has a root and other nodes.

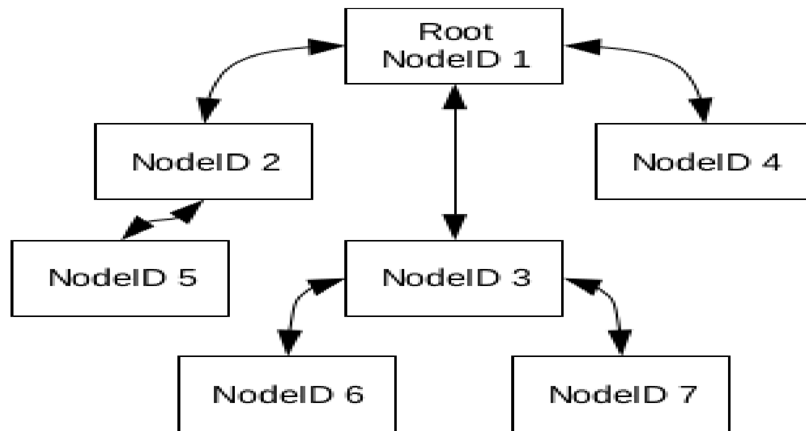

Figure 33: A typical RPL DODAG with one root and six other nodes

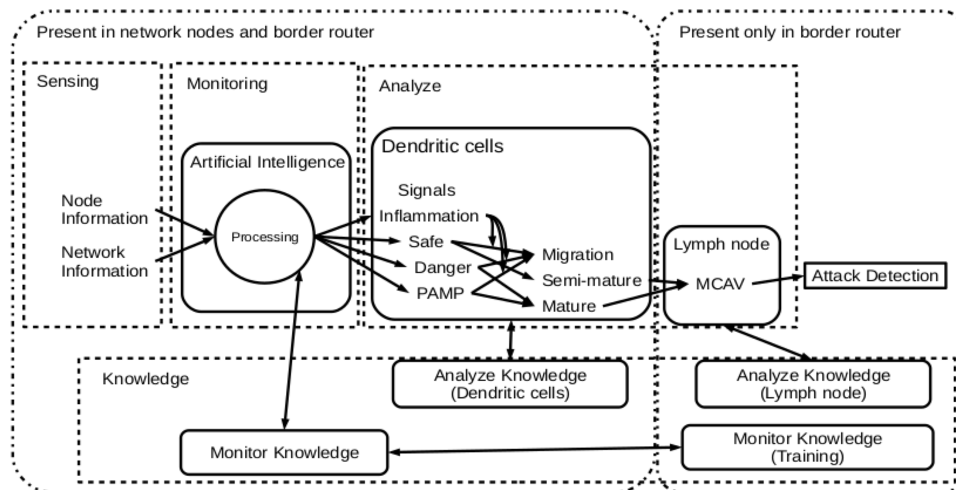

Figure 34: The proposed architecture

The architecture uses five components: Artificial Intelligence, Dendritic Cell, Lymph node, Monitor Knowledge, Analyze Knowledge. The other components (Planning,

Execution) of the MAPE-K loop are not defined yet. The current components of the proposed architecture are distributed between Sensing, Monitoring, Analyzing and Knowledge elements of MAPE-K loop. The distribution of the proposed architecture components in the MAPE-K loop elements is depicted in Figure 34. Some components are not present at all network nodes, the Lymph node, Analyze Knowledge and part of Monitor Knowledge are present only in border router. Sensing, Monitoring, Analyzing, Monitor Knowledge and Analyze Knowledge components can be present in any network node and are present in border router.

At first we start with the sensing phase, which is present in all of the nodes. In this phase, the network and node information will be sent to the monitoring phase. Node information is the rate of the successfully sent packets, total sent packets, RSSI level and more. The network gets its information from the packets information and, as long as it’s connected to the internet, it will get more information about the network.

Then, there is the monitoring phase, which is also present in all of the nodes. In this phase, De Almeida et al., (2015), use a Multi-Layer Percepton (MLP) network. The MLP is used to get useful information from the sensing phase. For example, the MLP will try to predict information from the packets, such as total time to process and respond, in order to provide this information quickly to the Analyzing phase.

Next, there is the analyzing phase, during which the proposed architecture receives the information from the monitoring phase and uses it as an input in the dendritic cells DC component. This component is also present in all the nodes of the local network. The signals of the dendritic cells are classified as safe, danger or inflammatory signals. The monitoring phase gathers this information from the sensing phase and infers the signal levels to this phase. When the DC have enough information, they will migrate their result to the lymph node, that is present only in border router. Then, the lymph node processes the DC result and detects

whether there is an attack on the network. This attack detection information is passed onto the Planning phase.

The Knowledge components described in this paper are the Analyzing and Monitor Knowledge. The Plan and Execute Knowledge components will be present in the architecture too. The monitor knowledge is split in two components: the training component and the component itself. The monitor knowledge itself is present in all the nodes of the local network while the training monitor knowledge is present only in border router. This split occurs because the training of the artificial intelligence may need more resources than the node can offer. The analyze knowledge is split in two parts, the dendritic cell part and the lymph node part. Each one is present where its counterpart component in the Analyze phase is present.

Lastly, we have the Planning, Execution and Effection Phase. Those are not present in the architecture yet. In short, the Planning phase will receive the warning from the analyze phase and plan how to mitigate the side effects of the attack. The Execution phase will receive the actions planned in the planning phase and deliver each order to the Effection phase. Then, the effectors will perform those orders and try to mitigate the side effects of the attack.

4.2.3.3 Experiments

The experiments done in this phase were about the technique that is used in the monitoring phase. The technique for the first efforts is an Artificial Neural Network, a Multi-Layer Perceptron with Limited Weights (MLPLW) based on the neural network with limited precision weights, (Bao, Chen, & Yu, 2012). The MLPLW implemented has 10 neurons in the hidden layer and each weight is represented by a byte. The technique used for training is Quantized Back-Propagation Step-by-Step (QBPSS), (Bao et al., 2012), which is a modified version of Back-Propagation for neural network with limited weight. To check the implementation there is a well-known dataset the KDD99, (Greensmith et al., 2005), which is used in Intrusion

Detection Systems. The dataset is used to train the model with a stream based training. After the first thousand inputs the MLPLW achieved 97,65% accuracy, but oscillated until the thirty-fourth thousand input. The oscillation of the accuracy rate of the MLPLW is depicted in Figure 35.

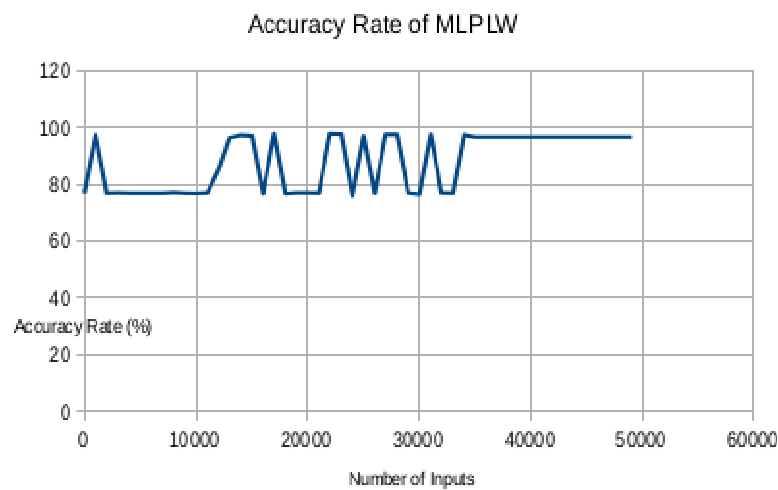

Figure 35: Accuracy rate of MLPLW over the number of inputs.

4.2.4 A MAPE-K Based Self-healing Framework For Online Sensor Data

Nguyen, Aiello, Yonezawa, and Tei (2015), proposes an architecture that fully implements the MAPE-K loop (IBM Corporation, 2005) that we discussed earlier, in Section 3.1. The goal of the proposal is to create a flexible framework, where its mechanisms could be used in different processes of the framework. Nguyen et al. (2015) also propose some mechanisms for each process. These mechanisms provide runtime capabilities for detection, classification and correction of faults that appear in sensor data.

The functionalities of the Monitor and the Analyze processes can be overlapped. For example a fault can be detected either by the Monitor, or by the Analyze functions. In the proposed framework, the Analyze function is responsible for both fault detection and classification. Figure 36 illustrates the proposed framework. As we said, the framework addresses the full cycle of the self-healing model which includes: monitoring and analysis, as well as fault detection at normal start, diagnosis and classification of faults at faulty state, resiliency and fault correction mechanisms to help system recover to normal state from a faulty one. Sensor data faults cannot be healed, fault notification transits the node to broken state, notifying other calling services and system administrators to take necessary action against detected fault and to bring the sensor node back to normal state.

4.2.4.1 Architecture

The proposed architecture, (Nguyen et al., 2015), uses a framework called "Baljak" (Baljak, Tei, & Honiden, 2013) for its fault modeling. The Baljak fault model is based on the frequency and continuity of fault occurrence and on observable and learnable patterns that faults leave on the data. This fault categorization is flexible and applicable to a wide range of sensor readings. Therefore, the cause of the error does not do this categorization, which makes it possible to handle the faults based on their patterns of occurrence on each sensor node. The Baljak framework provides a decision tree to classify data faults into four types: Bias fault, Drift fault, Malfunction fault and Random fault.

Those types of faults belong in two main categories of faults. The Intermittent and the Regular faults. Intermittent faults occur from time to time and the occurrence of faults is discrete. The Malfunction and the Random faults belong in this category of faults. The first type of faults refers to faulty readings that appear frequently, while the frequency of the occurrences of faults is higher than a threshold. A Random fault in a random manner is the frequency of the occurrences of faulty reading and

it is smaller than the threshold. During the observation period, faults occur constantly and it is possible to find a pattern in the form of a function. These are the Regular faults. Bias and Drift belong in this category of faults. In the first kind of error, the error is a constant; this can be positive or negative. Drift is the deviation of data that follows a learnable function, such as a polynomial change.

As we mentioned, the proposed architecture uses the MAPE-K loop to implement its functions. In order to build the knowledge base for the system, the system is calibrated before being deployed in a real environment. At this phase, necessary knowledge and assumptions about the environment are gathered for the system to build its knowledge base for the sake of self-healing. The knowledge base includes 1) information about the environment, 2) the database of managed sensor data elements, 3) models for fault detection and classification used at the Analyze process, 4) models for fault correction used at the Plan process, and other necessary information. The knowledge base should be automatically updated by the processes and manually by the administrators to keep it up to date.

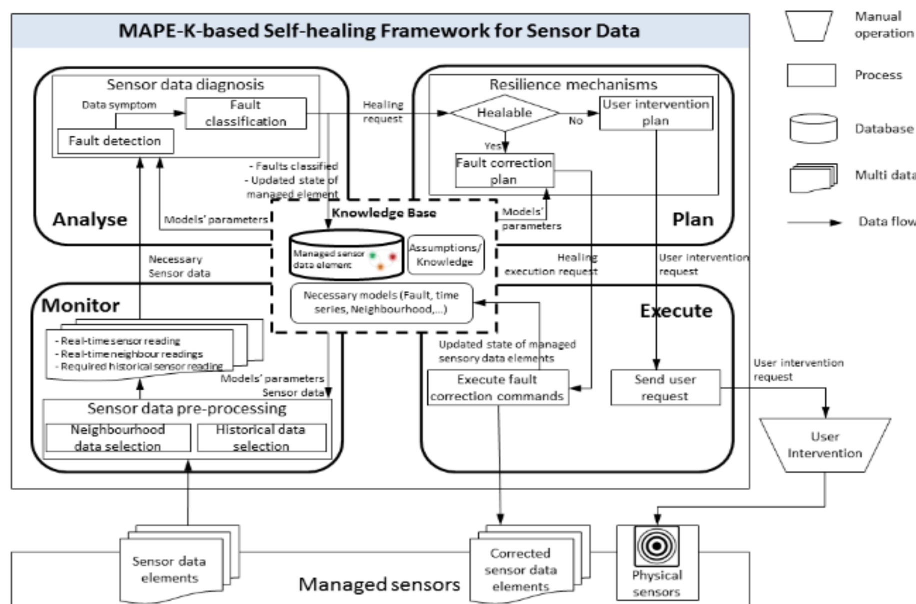

Figure 36: The MAPE-K-based framework for self-healing of sensor data

The Monitor process is responsible to 1) collect real-time reading from managed physical sensors, 2) retrieve historical data, 3) gather real-time reading from neighborhood sensor elements and 4) annotate the state of each and every sensor reading based on some assumption and information. Models, assumptions are retrieved directly from the Knowledge Base. After this pre-processing, necessary sensor data are passed in the Analyse process.

The data received from the Monitor process are analyzed actively so as to identify and detect faulty reading in sensor data. The proposed architecture applies a hybrid mechanism for fault detection using neighborhood vote together with time series data analysis. The fault classification implements the Baljak fault model discussed earlier.

At the Detection phase, each sensor data element compares its current reading with 1) the value computed by the neighborhood voting technique, and 2) the value forecast by the series data forecasting model. The result of the detection phase is the state of the reading that is examined. In case a sensor reading is detected as faulty, i.e. a symptom, it is diagnosed by the fault classification component, which is a part of the diagnosis procedure. The fault model use for classification is stored in the shared Knowledge Base. This model is, also, applied later at the Plan process, to correct readings from the respective faulty nodes. Moreover, the framework flexibly allows different sets of applicable algorithms to be implemented at each process.

After diagnosing, the Analyze process updates the faulty state of the managed sensor data elements back to the shared database in the Knowledge Base. This way, the state of all managed sensor data elements are synced and consistent among components of the framework. As an outcome, the Analyze process sends healing requests to the Plan process, providing a list of faulty sensor data elements that require the Plan process to take necessary actions: either heal -if possible- or notify users to take particular interventions.

After the Analyze process, comes the Plan process. In this kind of process the goals that we want to achieve are defined. In the case of the described model, the goals are to correct the faulty sensor reading if possible, maintaining the managed elements at their normal state as well as providing corrected sensor data to external services that are consuming the data. There is always a chance the faults cannot be healed. In those cases, the Plan process should notify the system administrators to take appropriate actions.

Last but not least, there is the Execute process. In this process, fault correction commands or sends user requests, depending on the instruction received from the Plan.

In this architecture we do not have any kind of Experiments, but we have a real case study that uses the proposed architecture.

4.2.4.2 The ClouT Case Study

The ClouT, (Tei, 2014), is a collaborative project between Europe and Japan. The overall concept of the project is to bridge the Internet of Things with Internet of People via Internet Internet of Services, establishing an efficient communication and collaboration platform. The main goal of ClouT is to design, implement, and validate a reference IoT+cloud architecture for a smart city ecosystem that helps city authorities provide the backbone for the innovation of their environments.

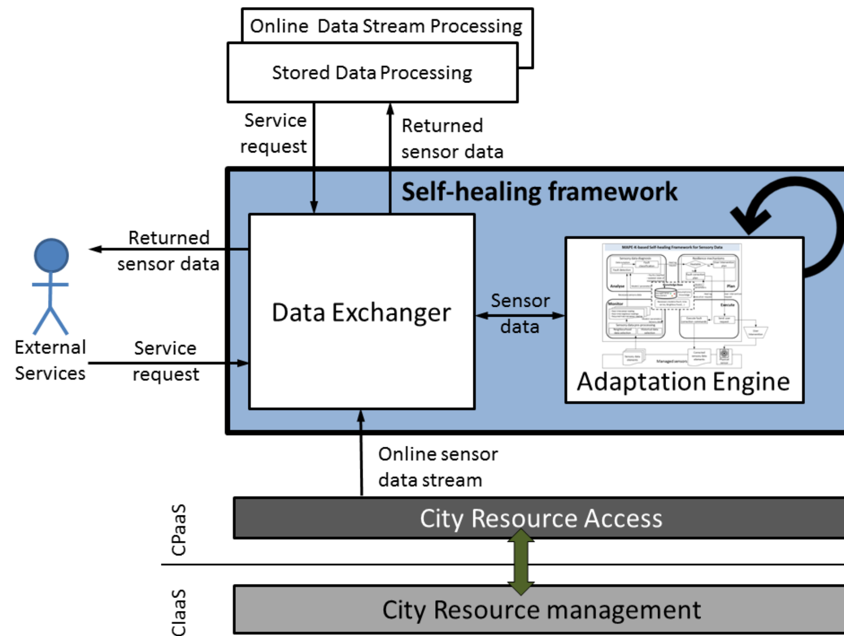

Figure 37: General architecture of the ClouT self-healing framework

The ClouT (Tei & Gurgen 2014) project uses the proposed framework as its proactive Adaptation Engine that gathers, via the Data exchanged, online sensor data from the City Resource Access components and applies it to correct faults in the data. In figure 37 we see how the architecture is implemented with the ClouT. The self-healing framework is invoked only by External services that require the self-healing service.

4.2.5 Comparison of the described architectures

In this section we compare the architectures that we described earlier.

Architecture	IOT	MAPE-K loop	Failures able to detect	Energy consumption
A Service management system for self- healing	✓	✓	Unexpected failures, Communication failures	Low
A Self-managing Fault Management Mechanism for Wireless Sensor Networks	✓	✕	Energy loss failures	Low
A Dendritic Cell Algorithm for Security System with Self-healing property	✓	✓	Jamming, Sinkhole, Hello Flood, Flooding	unknown[3]
A MAPE-K Based Self-healing Framework For Online Sensor Data	✓	✓	Intermittent faults[4], Regular faults[5]	unknown

Table 28: Architectures comparison

3 It is unknown when there are no experiments to prove it.

4 Frequent faults

5 Bias, constant error. Drift, deviation of data follows a learnable function, such as a polynomial change.

Section 5: Case study

In this section, we will talk about what could happen in a self-healing network, what are the major problems, and we will examine how this solution should be implemented to solve these problems.

Let’s assume that we have deployed our IoT devices (sensors) in a huge area such as an agricultural field. In that kind of field, the environmental phenomena are our main problem. These phenomena, such as rain, thunderstorm, wind, fire, could “break” our network. It’s easy to identify more problems. Battery exhaustion is, for instance, another big problem, thus we have to find a way to limit the energy consumption of the devices.

In table 12, we summarize the problems.

No.	Problem
1.	Environmental phenomena (rain, thunderstorm, wind, fire, earthquake, etc..)
2.	Battery exhaustion

Table 12: Network's major problems

Now that we have defined our major problems, we can create our network. We can imply that our network has 54 nodes. This network is deployed in an area of α 250 m^2. The nodes are able to collect weather data, such as weather conditions (rain, sunny, clear), temperature, wind, humidity, pressure and their battery level are high.

Parameter	Value
No. of nodes	54
Coverage area	250 m^2
Node data	Weather data
Battery level	High

Table 13: Network details

Ruiz et al., (2004), have proposed an architecture that fits our needs and that has already been discussed in section 4.2.1. Briefly, in this architecture, the nodes are grouped into clusters, while there is a special node called cluster head. This kind of nodes are more powerful and, therefore, they are responsible to send the data to a base station. Also, we will go with a programmed and continuous network because it’s easy enough to implement a logic and we need the data to be send at regular intervals.

Our failure detection works with GET requests/responses. A manager, located externally to the sensor network, sends GET requests to retrieve the nodes state. First, if the manager does not receive a response, he consults the energy map to verify if it has any energy left and, then, if he detects a failure, he reports it to the observer.

An autonomic manager located in the common nodes and in the cluster heads helps us keep truck of resources, detect unusual behaviors and much more that will help us adjust, in order to reduce the risk of faults.

The architecture that we are using helps us to efficiently arrange our major problems. First, regarding the environmental phenomena problems, when a node goes off (destroyed – e.g. from a failing tree), the node should be replaced by

another node. When the manager asks about the energy level of the node, it will not receive any feedback, then it will try to find out if it has any energy left from the energy level map. By not finding this node in the energy level map, the manager will try to replace it (activate a backup node). The same approach can be taken if there no energy left. Figure 38 depicts the above actions.

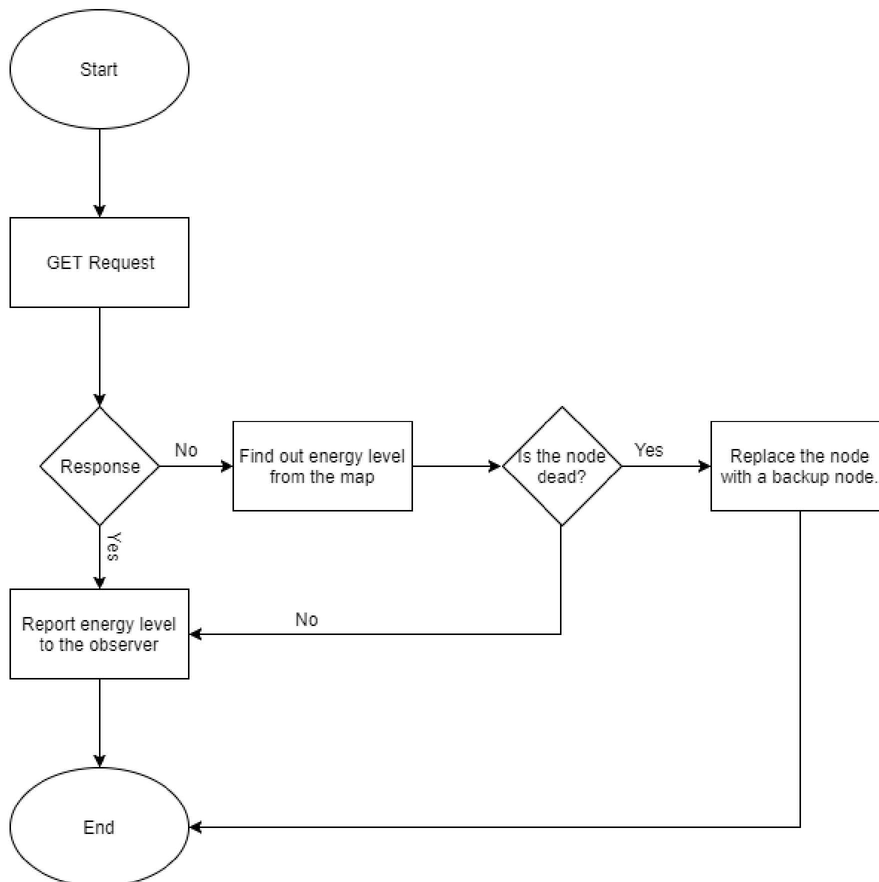

Figure 38: Destroyed/no energy left node flowchart

Another proposal could be to use an event driven network. These networks are harder to implement and it's also harder to detect a failure, because we are not sure when a node is "dead" or it just didn't sense anything. That kind of networks are more efficient in the essence of energy use.

Section 6: Future work

In this section we will discuss about what we could further do or implement, for our self-healing network to be more resistable and tested.

First, we could start by making our network tested. For instance, in our future work, we would create a simulation of our study case. That way, we would find out how our network behaves whenever a problem occurs. By simulating our network, we would have metrics over the battery usage and over what happens when a node fails, if it correctly recovers from a failure.

Moreover, in order to make our network more resistable to failures and more battery friendly, we could use machine learning techniques. For instance, if we collect data from the nodes about packages that have been successfully delivered, or if a node has failed, it is possible to use machine learning. More specifically, these data, in combination with machine learning, will help us determine better parameters for our network. Such a parameter could be when a cluster head will send a message to the node to determine if it is “online”. Using a neural network would help us use less battery when exchanging data and, thus, less failures would occur. Having a trained neural network would, also, help us recognize the cause of the failure faster, allowing the node to recover faster.

Conclusion

In conclusion, self-healing can take the IoT one step further from where it is today. We talked about its features, its uses and implementations regarding the IoT, but, what about the traditional devices like personal computers and smartphones? Even thought the use of self-healing on those traditional devices may seem purposeless, it can in fact be quite helpful, making them more fault-resistant.

However, in our opinion it would be of bigger significance to focus on the IoT devices, since, as we saw (in section 3.5), self-healing is mostly being used in military applications. Military applications are critical and should be “up” 24/7, even if a tree falls right into a node. So, baring in mind that this kind of applications should not fail under any circumstances, we have to invest more time, manpower and resources on making more research about it. Moreover, as the IoT grows, continues to expand and joins our everyday lives – even if we don’t realize it – more critical applications like the previous one will appear. For example, a system that will help disabled people pass the road should be accessible 24/7 and it should be able to recover by itself, while someone needs it. Thanks to self-healing, there won’t be the need for a person to get of his bed at 3 am to fix an error on one node among a thousand.

Moving on, in section 4 we discussed about different architectures on self-healing in IoT. Each architecture helps in different ways to solve different problems. But, all of them have the same base, that being self-healing, and is of different significance. As we mentioned in our future work, machine learning could join the game of self-healing, because the detection of faults will be more accurate.

Thus, it becomes clear why we should insist and invest on self-healing in IoT.

References

Abbas, N., Andersson, J., & Loewe, W. (2010). Autonomic Software Product Lines (ASPL). Proceeding of the 7th international conference on Autonomic computing. ACM, New York, NY, USA, 2010, pp. 324-331.

Adger, L., & Hughes, T., (2004). Self-regenerative systems (SRS), 2004.

Adve, S., Harris, A., Hughes, C. J., Jones D. L., Kravets, R. H., Nahrstedt, K., Sachs, D. G., Sasanka, R., Srinivasan, J., & Yuan, W. (2002). The Illinois GRACE Project: Global Resource Adaptation through Cooperation, Workshop on Self-Healing, Adaptive and self-MANaged Systems (SHAMAN), 2002.

Aldrich, J., Sazawal, V., Chambers, C., & Nokin, D. (2002). Architecture centric programming for adaptive systems, Proceedings of the First Workshop on Self-Healing Systems, 2002.

Apache, (2012-2015). Apache River. Retrieved from https://river.apache.org/

Assunção, P. H., Ruiz, L. B., & Loureiro, A. F. A. (2006). A Service Management Approach for Self-healing Wireless Sensor Networks, 215-228, 2006.Asim, M. I., Mokhtar, H., & Merabti, M. (2010). A self-managing fault management mechanism for wireless sensor networks. International Journal of Wireless & Mobile Networks, 2(4), 184-197.

Baljak, V., Tei, K., & Honiden, S. (2013). Fault classification and model learning from sensory Readings — Framework for fault tolerance in wireless sensor networks. 2013 IEEE Eighth International Conference on Intelligent Sensors, Sensor Networks and Information Processing.

Bao, J., Chen, Y., & Yu, J. (2012). An optimized discrete neural network in embedded systems for road recognition. Engineering Applications of Artificial Intelligence, 25(4), 775-782.

Blair, G. S., Coulson, G., Blair, L., Limon, H. D., Grace, P., Moreira, R., & Parlavantzas, N. (2002). Reflection, self-awareness and self-healing in OpenORB, Proceedings of the First Workshop on Self-Healing Systems, 2002.

Cheng, S. W., Garlan, D., Schmerl, B., Steenkiste, P., & Hu, N. (2002). Software architecture-based adaptation for grid computing, High Performance Distributed Computing, 2002. HPDC-11 2002. Proceedings. 11th IEEE International Symposium, Edinburgh, UK, UK.

Cheng, S. W., Huang, A. C., Garlan, D., Schmerl, B., & Steenkiste, P. (2004). An architecture for coordinating multiple self-management systems, Proceedings of the 4th Working IEEE/IFIP Conference on Software Architectures, 2004.

Combs, N., & Vagle, J. (2002). Adaptive mirroring of system of systems architectures, Proceedings of the First Workshop on Self-Healing Systems, 2002.

Coulouris, G. F., Dollimore, J., & Kindberg, T. (1999). Distributed systems: concepts and design. Wokingham: Addison-Wesley.

Dabrowski, C., & Mills, K. L. (2002). Understanding self-healing in service discovery systems, Proceedings of the First Workshop on Self-Healing Systems, 2002.

Dashofy, E. M., V. D. Hoek, A., & Taylor R. N. (2002). Towards architecture based self-healing systems, Proceedings of the First Workshop on Self-Healing Systems, 2002.

De Almeida, F. M., Ribeiro, A., & Ordonez, E. D. (2015). An Architecture for Self-healing in Internet of Things. Conference: UBICOMM 2015 : The Ninth International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies, 1.

De Lemos, R., & Fiadeiro, J. L. (2002). An architectural support for self adaptive software for treating faults, Proceedings of the First Workshop on Self-Healing Systems, 2002.

Forrest, S., Somayaji, A., & Ackley, D. (1997). Building diverse computer systems, In Proceedings of the Sixth Workshop on Hot Topics in Operating Systems, 1997.

Ganek, A., & Corbi, T. (2003). The dawning of the autonomic computing era. New York, USA.

Garlan, D., & Schmerl, B. (2002). Model-based adaptation for self-healing systems, Proceedings of the First Workshop on Self-Healing Systems, 2002

George, S., Evans, D., & Marchette, S. (2003). A biological programming model for self-healing, First ACM Workshop on Survivable and Self-Regenerative Systems. Washington, DC, USA.

Georgiadis, I., Magee, J., & Kramer, J. (2002). Self-organizing software architectures for distributed systems, Proceedings of the First Workshop on Self-Healing Systems, 2002.

Ghosh, S. (2006). Distributed Systems: An Algorithmic Approach, in IEEE Distributed Systems Online, vol. 9, no. 11, pp. 3-3, Nov. 2008.

Ghosh, D., Sharman, R., Raghav Rao, H., & Upadhyaya, S. (2006). Self-healing systems — survey and synthesis. New York, USA.

Greensmith, J., Aickelin, U., & Cayzer, S. (2005). Introducing Dendritic Cells as a Novel Immune-Inspired Algorithm for Anomaly Detection. SSRN Electronic Journal.

Grishikashvili, E. (2001). Investigation into Self-Adaptive Software Agents Development, Distributed Multimedia Systems Engineering Research Group Technical Report, 2001.

Gupta, G., & Younis, M. (2003). Load-balanced clustering of wireless sensor networks. IEEE International Conference on Communications, 2003.

Hong, Y., Chen, D., Li, L., & Trivedi, K. (2002). Closed loop design for software rejuvenation, Workshop on Self-Healing, Adaptive and self-MANaged Systems (SHAMAN), 2002.

Horn, P. (2001). Autonomic computing: IBM perspective on the state of information technology, IBM T.J. Watson Labs, NY. Presented at AGENDA 2001, Scotsdale, AR.

Huhns, M. N., Holderfield, V. T., & Gutierrez, R. L. (2003). Robust software via agent-based redundancy, AAMAS'03, 2003.

Huhns, M. N., Holderfield, V. T., & Gutierrez, R. L. (2003). Robust software via agent-based redundancy, Melbourne, Australia.

IBM Corporation (2005). An architectural blueprint for autonomic computing. Published in the United States of America 06-05. Third edition June 2005.

Intanagonwiwat, C., Govindan, R., & Estrin, D. (2000). Directed diffusion: a scalable and robust communication paradigm for sensor networks, Proceedings of the ACM Mobi-Com’00, Boston, MA, 56–67, 2000.

Inverardi, P., Mancinelli, F., & Marinelli, G. (2002). Correct deployment and adaptation of software application on heterogeneous (mobile) devices, Proceedings of the First Workshop on Self-Healing Systems, 2002.

Kahn, J., Katz, R. H., Pister, K. (2000). Emerging Challenges: Mobile Networking for „Smart Dust," J. Comm. Networks, 188-196, 2000.

Kephart, O. J., & Chess, M. D., (2003). The Vision Of Autonomic Computing. Published by the IEEE Computer Society.

Kephart, O. J., & Walsh, W. (2004). An artificial intelligence perspective on autonomic computing policies. In: Proceedings fifth IEEE international workshop on policies for distributed systems and networks, 2004.

Knop, M. W., Schopf, J. M., & Dinda, P.A. (2002). Windows performance monitoring and data reduction using watch tower, Workshop on Self-Healing, Adaptive and self-MANaged Systems (SHAMAN), 2002.

Kon, F., Costa, F., Blair, G., & Campbell, R. H. (2002). The case for reflective middleware, Communications of the ACM, 2002.

Kopetz, H. (2011). Real-Time Systems Design Principles for Distributed Embedded Applications. Boston, MA: Springer US.

Lamport, L., Shostak, R., & Pease, M. (1982). The Byzantine generals problem, ACM Transactions on Programming Languages and Systems 4 (3) (1982) 382–401.

Lanthaler, M. (2008). Self-Healing Wireless Sensor Networks.

Markus, C., Heubscher, A., & McCann, J. A. (2008). A survey of Autonomic Computing—Degrees, Models, and Applications. ACM Comput. Surv., 40, 3, Article 7 (August 2008).

Merideth, M. G., & Narasimhan, P. (2003). Proactive containment of malice in survivable distributed system, International Conference on Security and Management, Las Vegas, NV, 2003.

Minsky, N. H. (2003). On condition for self-healing in distributed software systems, Autonomic Computing Workshop Fifth Annual International Workshop on Active Middleware Services (AMS'03), 2003.

Moore, E. G. (1965). Cramming More Components onto Integrated Circuits Electronics, 38(8),1965.

Nagpal, R., Kondacs, A., & Chang, C. (2003). Programming methodology for biologically-inspired self-assembling systems, AAAI Symposium. Palo Alto, California, USA.

Narasimhan, P., Moser, L. E., & Melliar-Smith P. M. (1997). The interception approach to reliable distributed CORBA objects, Third USENIX Conference on Object-Oriented Technologies and Systems, Portland, Oregon, 1997.

Narasimhan, P., Moser, L. E., & Melliar-Smith, P. M. (2002). A component-based framework for transparent fault-tolerant CORBA, Software, Practice and Experience 32 (8) (2002) 771–788.

Nguyen, T. A., Aiello, M., Yonezawa, T., & Tei, K. (2015). A Self-Healing Framework for Online Sensor Data. 2015 IEEE International Conference on Autonomic Computing.

Norman, D. A., Ortony, A., & Russell, D. M., (2003). Affect and machine design: lessons for the development of autonomous machines, IBM Systems Journal, 2003.

Psaier, H., & Dustdar, S. (2010). A survey on self-healing systems: approaches and systems. Vienna, Austria.

Raz, O., Koopman, P., & Shaw, M. (2002), Enabling automatic adaptation in systems with under-specific elements, Proceedings of the First Workshop on Self-Healing Systems, 2002.

Riley, G. (2003). The Georgia Tech Network Simulator, in ACM SIGCOMM Workshop on Models, Methods and Tools for Reproducible Network Research Karlsruhe, Germany.

Römer, K., & Mattern, F. (2004). The Design Space of Wireless Sensor Networks, IEEE Wireless Communications Magazine, 11(6), 54-61, 2004.

Ruiz, L. B., Nogueira, M. J., & Loureiro A. F. A. (2003). MANNA: A Management Architecture for Wireless Sensor Network", IEEE Wireless Communications Magazine, 41(2), 116-125, 2003.

Ruiz, L. B., Siqueira, G. I., Leonardo, B., Oliveira, E., Wong, H. C., Nogueira, J. M. S., & Loureiro, A. F. A. (2004). Fault Management in Event-Driven Wireless Sensor Networks, Proceedings of the 7th ACM international symposium on Modeling, analysis and simulation of wireless and mobile systems MSWiM"04, Venice, Italy, 2004.

Russell, S., & Norvig, P. (2003). Artificial Intelligence: A Modern Approach, 2nd ed. Prentice Hall, 2003.

Sahoo, R. K., Bae, M., Vilalta, R., Moreira, J., Ma, S., & Gupta, M. (2002). Providing persistent and consistent resources through event log analysis and predictions for large-scale computing systems, Workshop on Self-Healing, Adaptive and self-MANaged Systems (SHAMAN), 2002.

Selehie, M., & Tahvildari, L. (2009). Self-adaptive software: Landscape and research challenges. Waterloo, Canada.

Sharman, R., Ragvar Rao, H., Upadhyaya, S., Khot, P., Manocha, S., & Ganguly, S. (2004). Functionality defense by heterogeneity: a new paradigm for securing systems, 37th Hawaii International Conference on System Sciences, 2004.

Shaw, M. (2002). Self-healing: softening precision to avoid brittleness, Position Paper for WOSS'02: Workshop on Self-Healing Systems, 2002.

Sterrit, R., & Bustard, D. (2003). Towards an Autonomic Computing Environment. Proceedings of the 14th International Workshop on Database and Expert Systems Applications (DEXA’03).

Tei, K., & Gurgen, L. (2014). ClouT : Cloud of things for empowering the citizen clout in smart cities. 2014 IEEE World Forum on Internet of Things (WF-IoT).

Valetto, G., & Kaiser, G. E. (2002). Case study in software adaptation, Proceedings of the First Workshop on Self-Healing Systems, 2002.

Venkataraman, G., Emmanuel, S., & Thambipillai, S. (2008). Energy-efficient cluster-based scheme for failure management in sensor networks. IET Communications, 2(4), 528.

Wang, Y. (2008). Study on Model and Architecture of Self-Organization Wireless Sensor Network. 2008 4th International Conference on Wireless Communications, Networking and Mobile Computing.

White, R. S., Hanson, E. J., Whalley, I., Chess, M. D., & Kephart, O. J., (2004). An Architectural Approach to Autonomic Computing,1st International Conference on Autonomic Computing (ICAC 2004), 17-19 May 2004, New York, NY, USA.