Jesús Martín Talavera Portocarrero

RAMSES: REFERENCE ARCHITECTURE OF A SELF-ADAPTIVE MIDDLEWARE FOR WIRELESS SENSOR NETWORKS

TESE DE DOUTORADO

Rio de Janeiro
2016

UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE MATEMÁTICA
INSTITUTO TÉRCIO PACITTI DE APLICAÇÕES E PESQUISAS COMPUTACIONAIS
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

Jesús Martín Talavera Portocarrero

RAMSES: REFERENCE ARCHITECTURE OF A SELF-ADAPTIVE MIDDLEWARE FOR WIRELESS SENSOR NETWORKS

Tese de Doutorado apresentada ao Programa de Pós-Graduação em Informática, Instituto de Matemática e Instituto Tércio Pacciti, Universidade Federal do Rio de Janeiro, como requisito parcial à obtenção do título de Doutor em Informática.

Orientador: Prof. Flávia Coimbra Delicato, DSc

Co-orientador: Prof. Paulo de Figueiredo Pires, DSc

Rio de Janeiro
2016

Jesús Martín Talavera Portocarrero

RAMSES: REFERENCE ARCHITECTURE OF A SELF-ADAPTIVE MIDDLEWARE FOR WIRELESS SENSOR NETWORKS

Tese de D [illegible] ada ao Programa de Pós-Gr [illegible] rmática, Instituto de Mat [illegible] ituto Tércio Pacciti, U [illegible] do Rio de Janeiro, como [illegible] obtenção do título de Doutor

TROCAR PELA FOLHA DE ROSTO ASSINADA PELOS PROFESSORES

Aprovada em 30 de março de 2016.

I dedicate it to my parents Lily Portocarrero and Rufino Talavera

Acknowledgment

Firstly, I would like to express my sincere gratitude to my advisor Prof. Flávia Coimbra Delicato and my co-advisor Prof. Paulo de Figueiredo Pires, for the continuous support of my Ph.D. study and related research, for his patience, motivation, and immense knowledge. Their guidance helped me in all the time of research and writing of this thesis. I could not have imagined having better mentors for my Ph.D. study.

Besides my advisors, I would like to thank the rest of my thesis committee: Prof. Luci Primez, Prof. Markus Endler, and Prof. Elisa Yumi Nakgawa, for their insightful comments and encouragement to improve this research.

My sincere thanks also go to my friends and fellow UBICOMP labmates, Mohammadreza Iman, Bruno Costa, Thomaz Barros, Ricardo Caldeira, Jose Renato Silva, Wagner Lopez, Taniro Rodrigues, Marcelo Pitanga and Marcus Ravelo. Their support helped me overcome setbacks on my Ph.D. research. A special thanks go to Jorge Mauricio Fernandes and his family, for receiving me in their home and hearts, I sincerely appreciate their belief in me.

This research would not have been possible without the financial assistance of CAPES (Brazilian Funding Agency), the Federal University of Rio de Janeiro (UFRJ), and the Department of PPGI at Federal University of Rio de Janeiro (Research Scholarships). I express my gratitude to those agencies.

Last but not the least, I would like to thank my family: my parents and to my brothers and sister for supporting me spiritually throughout writing this thesis and my life in general.

“The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it”
- Mark Weiser

Abstract

TALAVERA PORTOCARRERO, Jesús Martín. **RAMSES: Reference Architecture of a self-adaptive Middleware for Wireless Sensor Networks**. 2016. 198. Tese (Doutorado em Informática) – Instituto de Matemática, Instituto Tércio Pacciti, Universidade Federal do Rio de Janeiro, Rio de Janeiro, 2016.

Wireless Sensor Networks (WSN) consist of networks composed of tiny devices equipped with sensing, processing, storage, and wireless communication capabilities. WSN nodes have limited computing resources and are usually powered by batteries. Typically, WSNs were designed to attend requirements of a unique target application usually of a single user, who was also the infrastructure owner. However, the rapid evolution in this area and the increasing of the complexity of sensors and applications involved, the need for specific middleware platforms for these networks have raised. Middleware systems developed until today represent useful instruments for defining the high-level application logic and dealing with heterogeneity and distribution issues, but most of them do not provide an explicit way of describing the underlying autonomic behavior. In this perspective, Autonomic Computing (AC) is presented as an attractive option to meet basic requirements in the WSN design. Autonomic computing principles can be applied to WSN to optimize network resources, facilitate their operations in the vast field of sensing based applications and providing conditions for this type of network manage itself without involving human operators. The application of these AC principles into WSN would be facilitated by the development of a system at the middleware level. Thus, in this Ph.D. thesis, we introduce RAMSES, a reference architecture of a self-adaptive middleware for WSNs. RAMSES follows the autonomic computing model MAPE-K, for making decisions aiming to attend self-adaptive WSN requirements. We present an implementation of this reference architecture using a formal Architecture Description Language (pi-ADL).

Keywords: Reference Architecture, Wireless Sensor Network, Autonomic Computing, Middleware, Pi-ADL

List of Figures

List of Tables

List of Achronyms

AC	Autonomic Computing
ADL	Architecture Description Language
CBR	Context-based Reasoning
DoS	Denial of Service
FCL	Feedback Control Loop
FERA	Framework for Evaluation of Reference Architectures
FM	Feature Model
GML	Goal Management Layer
ISO	International Organization for Standardization
IT	Information Technology
MA	Mobile agents
MAPE-K	Monitor, Analyzer, Planner, Executing and Knowledge Base
MDD	Model-Driven Development
MT/CG	Model transformation/Code Generation
NML	Network MAPE-K Layer
OMG	Object Management Group
PBR	Policy-based Reasoning
QC	Quality Criteria
QS	Quality of Services
RA	Reference Architecture
RQ	Research Question
SLR	Systematic Literature Review
SML	Sensor MAPE-K Layer
SNRA	Sensor Network Reference Architecture
SPL	Software Product Lines
SysML	Systems Modeling Language
UML	Unified Modeling Language
WSN	Wireless Sensor Network

Sumary

CHAPTER 1: Introduction

1.1 Contextualization

Wireless Sensor Networks (WSNs) consist of small devices connected through wireless links and equipped with sensing, processing, and storage capabilities. Each node of a WSN is often endowed with several sensing units, which are able to perform measurements of physical variables, such as temperature, luminosity, humidity, vibration, among others (Potdar, Sharif, & Chang, 2009). Sensor nodes have limited computing resources (such as RAM and processing power) and are usually powered by small batteries; thus, energy saving is a key issue for such networks in order to provide a long operational lifetime and increase potential profits of the deployed infrastructure. WSN contain hundreds or thousands of sensor nodes. These sensor nodes operate collaboratively by extracting environmental data, performing same simple processing, and transmitting such data to one or more exit points of the network (often called gateway) to be analyzed and further processed. Gateways act as a bridge between the WSN and the networks and/or external systems.

WSNs are typically used in highly dynamic, sometimes remote and even hostile environments, and should operate with minimal human intervention. Therefore, such networks should tolerate several types of failures, such as faulty nodes or hardware physical malfunction (e.g., failures in the sensor units or battery), lack of sensing coverage and radio connectivity, among others. Hence, WSNs should have self-management capabilities to deal with failures and unpredictable circumstances, and dynamically adapt to environment changes without human intervention and with minimum disruption of the provided service (Puccinelli & Haenggi, 2005).

WSNs were usually designed to meet the requirements of a single target application. However, with the evolving of the field, there is a trend to share the sensing data produced by a single deployed WSN among different applications. Such applications have different functional and non-functional (QoS) requirements that should be met by the WSN. Moreover, the applications have to face environments in which operation conditions change very often, requiring that the system be able to adapt and reconfigure itself according to these dynamic scenarios while satisfying the application requirements.

Managing the dynamic and context-aware adaptation is not an easy task, as the nodes in a WSN can be heterogeneous, having different manufacturers and operating systems, sensing devices, and communication constraints (Sohraby, Minoli, & Znati, 2007). In addition, most of current WSN applications use the same fixed software architecture for each node in the network. By architecture we mean a set of system components, the externally properties of those components and the relationship between them. For simple applications, the choice of a static software architecture might not be a problem. However, when we consider more complex applications, with dynamic requirements that may change at runtime, and assume the WSN infrastructure is shared by multiple applications, the need for a more advanced, dynamically reconfigurable architecture, arises. The idea behind node reconfigurability is that the network can adapt its functionality to the current situation (context), in order to reduce the use of the scarce energy and memory resources of nodes, while maintaining the integrity of its operation.

All the above-mentioned characteristics require the use of specific middleware systems to WSN (Wang, Cao, Li, 2008). A middleware is a layered software that lies between application code and the communication infrastructure providing, via well-defined interfaces, a set of services that may be configured to facilitate the development and execution of distributed applications in an efficient way (Hadim & Mohamed, 2006). A middleware for WSN should provide generic services for sensing-based applications. Such services should consider the application's specific needs, the inherent features of the WSN nodes (such as the limited resources of energy, memory and CPU) and the dynamic execution context. These services may also include mechanisms to formulate complex high-level sensing tasks, communicate these tasks to the WSN, coordinate the nodes to collaboratively perform the tasks, merging the sensor readings of individual sensor nodes into a high-level result, and report the result back to the task issuer (in a timely way and meeting defined QoS requirements).

1.2 Motivation

The authors in (Portocarrero et al., 2014) argued that middleware systems developed until today are valuable tools for aiding developers to build distributed applications by presenting a unified interface, and for dealing with heterogeneity and distribution issues. However, the majority of WSN middleware does not provide an explicit way for defining the network autonomic behavior from an architectural point of view. Thus, we noticed a lack of well-defined architecture designs that fully support the autonomy of sensor networking. In other words, although WSNs are inherently dynamic and autonomous systems, there is still a lack of proposals for software architectures and middleware platforms for such networks that recognize autonomic behavior as a first-class citizen and provide mechanisms to address adaptation as basic building blocks. Such identified research gap may come from the fact that the Software Engineering community perceives WSNs as too low-level (Picco, 2010). However, WSNs are a key element of the grand vision of a physical world augmented by a myriad of computing devices (such as envisaged in the paradigms of Internet of Things, Cyber-Physical Systems, Ubiquitous computing, Ambient intelligence, etc.). As a consequence, sooner or later, as one of the enablers of this vision, WSNs are going to become mainstream (Picco, 2010).

According to (Salehie & Tahvildari, 2009), Autonomic Computing (AC), also known as self-adaptive computing, is a capacity of an infrastructure for adapting itself according to policies and business goals. Autonomic computing tries to help IT professionals to focus in higher value tasks, turning technological work more intelligent, with rules oriented to self-management. These rules, also known as self-* properties, are (IBM, 2005): (i) Self-Configuration, denoting the ability to adapt itself to the environment changes according to high-level policies, aligned with business goals and defined by system administrators; (ii) Self-Healing, the ability of recovering after a system disturbance and minimizing interruptions to maintain the software available for the user, even in the presence of individual failure of components; (iii) Self-Optimization, the system ability to improve its operation continuously and (iv) Self-Protection, the ability to predict, detect, recognize and protect from malicious attacks and unplanned cascade failures. Examples of self-adaptive systems are those that optimize their performance under changing operating conditions, and systems that heal themselves when certain components fail (Lemos, 2013).

A highlighted approach to develop autonomic systems is the autonomic computing architecture proposed by IBM (IBM, 2005) that defines an abstract framework for self-managing IT systems. In this framework, an autonomic system is a collection of autonomic elements. Each element consists of an autonomic manager and a managed resource. In the context of WSN, an autonomic manager can be a middleware system and the sensor network represents the managed resource. The autonomic manager allows adaptation through four activities: monitoring, analyzing, planning and executing, with support from a knowledge base. In the monitoring activity, elements collect relevant data via sensors to reflect the current state of the system (the managed resource) and thus, grant it context awareness. In analyzing activity, the collected data are analyzed in search of symptoms relating the current and desired behaviors and a diagnosis is performed to decide whether it is necessary to adapt the system to attend the previously defined goals. In planning activity, an action plan with all the required adaptation actions is built in order to perform certain changes in the target system and to reach the desired state of the managed resource. In execution activity, actuators or effectors instrument the desired adaptation acts.

Considering its goals, AC appears as an interesting option to meet basic requirements in WSN design. Autonomic computing principles can be applied to WSN in order to optimize network resources, facilitate the network operations and achieve the desired functionality in the wide field of sensing-based applications, providing conditions for this type of network manage itself without involving human operators (Portocarrero et al., 2014). In this context, we claim that the applying of these AC principles into WSN would be facilitated by the development of a system at the middleware level. So, with the support of a middleware system that leverages the AC principles, a WSN becomes an autonomous WSN *by design*.

In order to implement the AC principles thus allowing self-adaptation of software, feedback loops are required, with explicit functional elements and interactions between them for managing the dynamic adaptation. These elements are known as MAPE-K model (Monitor, Analyze, Plan, Execute and Knowledge Base).

Feedback control loops are considered a key issue in pursuing self-adaption for any system, because they support the four above-mentioned activities. They play an integral role in adaptation decisions. Thus, key decisions about a self-adaptive system's control depend on the structure of the system and the complexity of the adaptation goals. The MAPE-K model provides conceptual guidelines about the autonomic systems conception; in practice, this reference model needs to be mapped to an implementable reference architecture for managing and controlling of autonomic systems, such as WSNs. While a reference model consists in a set of minimal unified concepts, axioms and relationship of a particular domain, independently illustrated of specific patterns, technologies and other concrete concepts (MacKenzie, 2006), reference architectures are defined as mapping of reference models into software elements that cooperatively implement functionalities defined by this model (Bass, 2012). Thus, a reference architecture consists of software components and the relationship between them, that implements functionalities related to parts defined in the respective reference model.

PROBLEM: Most of middleware systems for WSN does not provide an explicit way for defining the underlying autonomic behavior of WSNs. There is a lack of reference methods for modeling the dynamic adaptation in WSNs.

1.3 Objectives

The main objective of this work is to propose RAMSES, a Reference Architecture (RA) for contributing with the development of self-adaptive WSN middleware systems. The main purpose of a RA is to facilitate and guide (E. Y. Nakagawa, F. Oquendo, & M. Becker, 2012) (i) the design of concrete architectures for new systems; (ii) the extensions of systems of neighbor domains of a RA (iii) the evolution of systems that were derived from the RA and (iv) the improvement in the standardization and interoperability of different systems. RAs play a dual role in relation to specific software architectures, the first role generalizes and extracts common functions and configurations, and the second role provides a base for instantiating target systems. In other words, RAs can be seen as a repository of a given knowledge area, contributing towards software development, since the reuse of knowledge and improvements of productivity are promoted. Thus, the proposed RA for a WSN middleware aims to satisfy this dual role in the WSN domain.

In order to create such reference architecture, the ProSA-RA (Nakagawa, Guessi, Maldonado, Feitosa, & Oquendo, 2014) process was used. ProSA-RA is a process that systematizes the design, representation, and evaluation of reference architectures. In short, four steps compose the ProSA-RA process. Firstly, information sources are selected and investigated (Step RA-1). Secondly, architectural requirements of the reference architecture are identified (Step RA-2), describing the common functionalities and configurations presented in systems of the target domain. Following, the architectural description of the architecture is built (Step RA-3) and its evaluation is conducted (Step RA-4).

RAMSES is based on autonomic computing principles and describes the mechanisms capable of managing the autonomic behavior of WSNs. RAMSES follows the autonomic computing model (MAPE-K (IBM, 2005)) proposed by IBM and maps its elements to a set of software components to be implemented and deployed in WSNs. We used a formal description language (pi-ADL (F Oquendo, 2004)) for describing the RAMSES architecture. Pi-ADL describes the dynamic software architecture of RAMSES under structural and behavioral perspectives. A Pi-ADL specification enables to explicitly define the dynamic behavior of WSN networks, a key issue in this type of systems.

The main idea behind RAMSES design is to provide a common structure and guidelines for dealing with core aspects of developing and using self-adaptive WSN middleware systems. One of the main benefits to be achieved is the use of the proposed RA for generating compliant architectures for specific, concrete WSN middleware. In order to support such generation, as part of this work we also propose the definition of a model-based instantiation process of RAMSES, responsible for translating the RA elements (specified using pi-ADL) into a concrete middleware instance and implementation.

Designing WSN systems using a reference architecture is a task typically performed by a software architect. Since such specialist often does not have the knowledge on computer networks, specifically on the details for code implementation in WSN platforms, it is necessary to close the gap between the reference architecture specification and the implementation code. In this context, model-driven development approaches (MDA) have become an increasingly important solution due to their power of mapping information from a higher abstraction level into source code for a target platform. Therefore, to enable the software architect to remain focused on his/her expertise field, we propose a model-driven architecture (MDA) solution, which is a particular vision of MDD proposed by OMG (Object Management Group), to support our instantiation process by automatically generating the implementation code of the middleware instance derived from RAMSES specification.

In order to achieve the general goal, the specific objectives of this PhD thesis are:

- **First**. To establish a set of information sources that allows identifying processes, activities, and tasks that must be supported by self-adaptive middleware systems for WSN.
- **Second**. To establish a set of architectural requirements and quality attributes for self-adaptive middleware systems for WSN.
- **Third**. To design RAMSES for guiding future development of self-adaptive middleware systems for WSN, based on the identified user requirements and information sources, and on a set of established quality attributes.
- **Fourth**. To evaluate RAMSES, by conducting a proof of concept aiming to instantiate RAMSES into a concrete middleware for WSN, and by conducting a survey with researchers and specialists in the fields of software architecture and WSN, in order to know about the viability and relevance of the proposed RA.

1.4 Contributions

The main contributions of this work are:

- A reference architecture (RA) that meets the efficiency and flexibility requirements for self-adaptive middleware systems for WSNs, based on autonomic computing principles; such RA was designed by mapping the MAPE-K model proposed by IBM to a set of software modules encompassing a WSN middleware.

- A specification of the reference architecture using pi-ADL Architecture Description Language.
- A concrete middleware instance built from the reference architecture.
- The definition of the instantiation process of RAMSES to generate a middleware instance from a pi-ADL architecture specification.
- A set of model to text transformations (M2T) for mapping RAMSES components (specified with pi-ADL) into C code for Contiki OS, a lightweight and flexible operating system for tiny-networked sensors, with support for dynamic loading;
- The complete implementation of the proposed adaptive mechanism, based on MAPE-K, for dynamically adapting the WSN behavior according to policies and application requirements, where components deployed on sensor nodes and gateways were defined to evaluate the middleware mechanisms for extending nodes lifetime and for reacting to contextual changes.

1.5 Organization

This PhD thesis is organized in 6 Chapters.

Chapter 2 brings an overview of the background information that supports the topics investigated in this thesis. Initially, the main concepts related to Wireless Sensor Networks are discussed. Then, an introduction to autonomic computing and its main properties are detailed. The Chapter also addresses terminology and theory associated with reference architectures, as well as methods for their evaluation and representation.

Chapter 3 describes and conducts a methodology for building reference architectures (ProSA-RA). Four steps compose ProSA-RA: in the first step we describe related works and the main information sources to be used in the design of the proposed RA. In the second step, the main requirements and quality attributes are identified. In the third step, we built our RA based on RAModel (Reference Architecture Model), and the outcome of the architectural description comprises a set of architectural views. In the fourth step, we evaluate our RA by conducting FERA (Framework for Evaluation of Reference Architecture).

Chapter 4 presents the evaluation of an instance of the RA through a Proof of Concept. We assess through a series of adaptation scenarios the execution of a RAMSES-based middleware for WSN, named SAMSON (Self-Adaptive Middleware for wireless SensOr Networks).

Chapter 5 presents a comparative analysis between RAMSES and related reference architectures, and we also detail an evaluation of the completeness of those RAs.

Finally, Chapter 6 concludes this thesis, revisiting the achieved contributions, summarizing limitations, and presenting perspectives of future research. The list of publications resulting from this work is also presented in this Chapter.

CHAPTER 2: Background Concepts and Technologies

2.1 Initial Considerations

This chapter provides an overview of the subjects that underlie the research developed in this thesis. The organization of the chapter is as follows. Section 2.2 presents the Wireless Sensor Networks (WSNs), their characteristics and the main applications and middleware systems. Section 2.3 describes the Autonomic Computing and the MAPE-K process. Section 2.4 introduces to Reference Architecture concept, where its definition, classification, evaluation and representation methods are detailed.

2.2 Wireless Sensor Networks

Wireless Sensor Networks (WSNs) are an emerging technology that promotes an unprecedented functionality to monitor, orchestrate, and control the physical world. WSNs consist in a huge number of wireless devices (sensor nodes or simply sensors) densely distributed in a target area. Sensor nodes have wireless connectivity and are connected to a network, such as Internet. They are typically powered by batteries and have limited communication and computing functions. Each node may be equipped with a variety of sensory modalities such as temperature, humidity, acoustic, seismic and infrared.

WSNs can operate by periods of time, from weeks to years, in an autonomic way. Basically, this depends on the amount of available energy for each node in the network. In many applications, the sensor nodes cannot be easily accessible because of the location they are deployed on or due to the network scale. In both cases, network maintenance for power supply becomes impractical. Frequently replacing the battery of sensors would waste the main advantages of WSNs.

WSNs are composed of sensor nodes and sink nodes (nodes to interface with other networks). Sensor nodes are autonomic devices, equipped with capabilities of sensing, computing and communication. When they are deployed conforming an ad hoc network, they form the sensor networks. Nodes collect data via sensors; they process it locally or coordinately among neighbors in order to send information to users or, in general, to the sink nodes. Essentially, each node in the network may have different tasks: sensing of the environment, processing of information and tasks related to traffic retransmission in a multi-hop way.

The main components of a sensor node are: transceiver, memory, processor, sensor and battery. The downsizing of a sensor implies the reduction in capacity and size of its components. Therefore, a sensor is a device that produces a measurable response for a change in a physical condition. In addition, the node of the network presents resources of processing, storing of information, source of energy and communication interface. Figure 1 depicts the basic hardware of a sensor node.

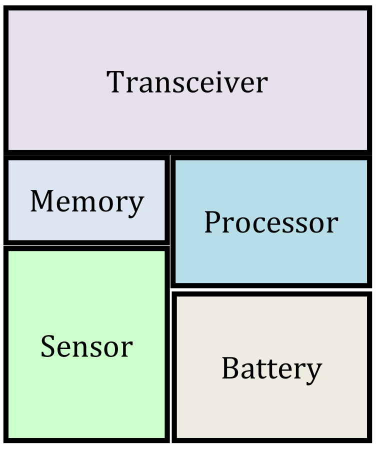


Figure 1. Components of sensor nodes

Network communication with other networks occurs through sinks nodes. Messages traverse the network towards a gateway that will route to other network, such as Internet. Figure 2 depicts the communication architecture of a WSN.

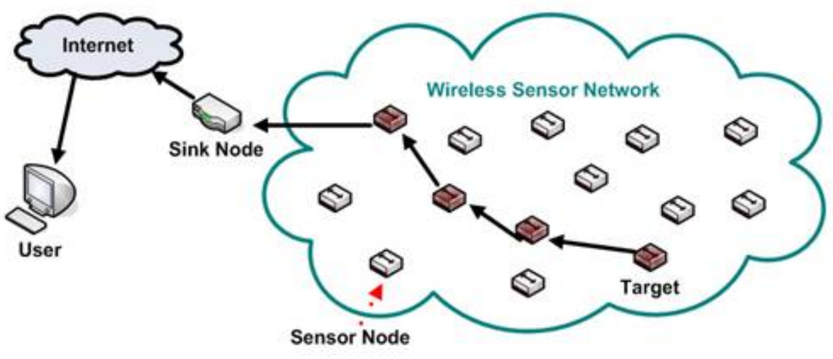


Figure 2. Communication architecture of Wireless Sensor Networks

2.2.1 Characteristics of Wireless Sensor Networks

WSNs have special characteristics according to areas in which they are applied. This causes specific issues that have to be addressed by them. Some of these characteristics and issues discussed by (Loureiro, Gonzàlez, & Mini, 2010) are as follows:

- **Addressing of Sensors or Nodes**. Depending of application, each sensor can be uniquely addressed or not. For instance, sensors embedded on an assembly line or placed on the human body must be uniquely addressed, if it is necessary to know the exact local where data is collected, but sensors monitoring the environment in an external region may not need to be identified individually because the important issue is to know the value of a given variable in this region.
- **Data Aggregation**. This indicates the ability of a WSN to aggregate or summarize collected data by sensors. If network has this functionality, it is possible to reduce the amount of messages transmitted by it.
- **Mobility of sensors**. This indicates if sensors are able to move according to a given system. For instance, sensors deployed in a forest aiming to collect humidity and temperature data are typically static. However, sensors deployed on the ocean surface aiming to measure the level of water pollution are mobile.
- **Restrictions of collected data**. This suggest if collected data by sensors have some type of restriction, such as a max time interval to disseminate their values to a given supervision entity.
- **Amount of sensors.** WSNs composed of 10 to 100 thousand of sensors are indicated for environmental applications such as ocean/forest monitoring.
- **Limitation of available energy.** In many applications, sensors are deployed in remote areas. This difficult the accessibility to these elements for maintenance. In this scenario, the sensor lifetime depends on the amount of available energy.
- **Self-organization of network.** Due to physical destruction or power failure, sensors in a WSN may be lost. Also, due to trouble in communication channel or by the decision of a network management algorithm, sensors can be isolated. This can happen for several reasons, for instance, to save energy or because other sensor in the same region is already collecting the desired data.

- **Collaborative task.** The main objective of WSNs is to execute some collaborative task where it is important to detect and to estimate events of interest and not only to provide communication mechanisms. Due to WSN restrictions, normally the data is fused or summarized in order to improve the performance on the process of event detection.
- **Query response ability**. Queries regarding to collecting information, in a given region, can be submitted to an individual node or group of nodes. Depending on aggregation mechanism, transmitting data through network towards sink node may become unviable. Thus, defining more sink nodes to collect data from a given area can be necessary. That would bring reply queries related to nodes under their competence.

2.2.2 Applications of Wireless Sensor Networks

Traditionally, sensor networks have been used in the context of high-end applications such as biomedical applications, habitat sensing, and seismic monitoring. Existing and potential applications of sensor networks include, among others, military sensing, physical security, air traffic control, traffic surveillance, video surveillance, industrial and manufacturing automation, process control, inventory management, distributed robotics, weather sensing, environment monitoring, national border monitoring, and building and structures monitoring (Chong & Kumar, Aug, 2003). Some applications for WSN are listed below:

- **Military applications**. Monitoring inimical forces, monitoring friendly forces and equipment, military-theater or battlefield surveillance, targeting, battle damage assessment, nuclear, biological, and chemical attack detection.
- **Environmental applications**. Microclimates, forest fire detection, flood detection, precision agriculture.
- **Health applications**. Remote monitoring of physiological data, tracking and monitoring doctors and patients inside a hospital, drug administration, elderly assistance.
- **Home applications**. Home automation, instrumented environment, automated meter reading.

- **Commercial applications**. Environmental control in industrial and office buildings, inventory control, vehicle tracking and detection, traffic flow surveillance.

Basically, wherever one wants to instrument, observe, and react to events and phenomena in a specified environment, one can use WSNs; the environment can be the physical world, a biological system, or an IT framework. Also, Wireless sensors can be used where wire line systems cannot be deployed, for instance, a dangerous location or an area that might be contaminated with toxins or be subject to high temperatures, in these places the sensor nodes may also be damaged.

2.2.3 Middleware for WSNs

According to (Blair, 2004), middleware is a software artifact residing between an application and operating system, providing the reuse of services through interfaces, by easing the development of more efficient applications for distributed environments such as WSNs. The main goal of a middleware is enabling communication between distributed components, by hiding from applications the complexity of underlying network environment, releasing them for explicit manipulation of protocols and services of infrastructure.

In (Macolo, Capra, & Emmerich, 2002) the authors present a reference model classifying middleware systems among fixed and ad hoc. This classification takes into account three aspects: the type of computational load, paradigm of communication and the context representation.

Middleware for traditional distributed systems are a category of fixed middleware, and they have some limitations that prevent their use in WSNs. Those systems demand computational resources (computational overload), hide context information from applications as much as possible (transparency), and support synchronous communication between components. Synchronous communication is unsuitable for environments with many disconnections. WSN have limited resources and the computational load of traditional middleware platforms becomes a problem. Transparency presented in traditional middleware platforms is not always appropriate for WSN applications; they need some information about executing context for better adaptability (Macolo et al., 2002). In this context, it is necessary the conception of a middleware platform specific for WSN.

Middleware for WSNs are a category of ad hoc middleware, and they must be designed for attending some middleware requirements for ad hoc networks. The main requirements include computational light load for supporting tiny devices, provide asynchronous communication for controlling troubles caused by usual disconnections and become context-aware applications. Thus, new type of middleware have been designed and developed in order to attend several types of WSN applications (Macolo et al., 2002).

The main purpose of middleware for WSN is to support the development, maintenance, distribution and execution of sensing applications. This include some mechanisms for sensing, communication and coordination among sensor nodes for distributing tasks, aggregation/fusion of data in order to join sensor readings in a high level result and to report these results of tasks back to emitter (Kay, Kaste, & Mattern, 2008).

In (Wang et al., May, 2008) three ways for supporting application developers in order to build middleware systems specific for WSN are described. First, the middleware may provide appropriate abstractions of systems, so that the application programmer can focus on logic of application without worrying about low-level implementation details. Second, the middleware may provide the reuse of code and services, so application programmers can distribute and execute applications without worrying about complex and tiresome functions. Third, the middleware may support programmers on the management and adaption of the network infrastructure, by providing efficient resources of services, such as energy management, and support integration of systems, monitoring and security.

In (Yu, Krishnamachari, & Prasanna, 2004) the authors propose principles to be adopted by a middleware design for WSN:

- The middleware must provide mechanism based on data for the processing and consulting of data inside the network;
- Local algorithms must be used to achieve a desired goal while providing good scalability and robustness to system;

- Whereas traditional middleware platforms are designed to support a wide range of applications on network, middleware for WSNs cannot be generalized that way, due to available limited resources
- Considering that the availability of node resources is low, the middleware must be light in terms of communication and processing requirements;
- Due to available limited resources (in terms of memory, communication, processing and energy), it is probable that resources of performance of all executing applications cannot be simultaneously satisfied. Therefore, it is necessary that the middleware negotiates the quality of services (QoS) in an intelligent way among several applications.

The design and development of middleware impose many challenges due to characteristics of WSNs, for instance, resource constraints, availability and diversity of sensor hardware. In (Molla & Ahamed, 2006) a description of several challenges associated to middleware for WSNs was elaborated as follows:

- **Abstraction support**. WSNs consist in a huge number of heterogeneous sensors. Sensors are developed with different hardware platforms. One of the main challenges of middleware platforms for WSNs is to hide the underlying hardware platform in order to offer an overview of network.
- **Fusion/Aggregation of data.** Sensors are used to collect data from environment. Data collection from several sensors, joining data, aggregating data and presenting them at a high level is another important challenge.
- **Resource Constraints.** A middleware for WSNs must be light for working over limited-resource hardware.
- **Dynamic topology.** Due to mobility, failure of nodes, and communication failure among nodes.
- **Application knowledge.** Network optimizations may be achieved with knowledge at the level of applications. For instance, characteristics of applications must influence both in network infrastructure and in used protocols. The application knowledge may be availed by network in order to achieve a better efficiency in terms of energy consumption. Thus, the network lifetime is extended.

- **Adaptability.** A middleware for WSNs must support algorithms with adaptive performance to deal with unexpected events.
- **Scalability.** In terms of number of nodes and number of users in order to operate for long periods of time.
- **Security.** Sensor nodes have to address security issues in data processing and communication. Due to limited resources and computational low power, most of existing algorithms and security models are not suitable for WSNs.
- **QoS support.** A middleware for WSNs must tackle many QoS issues, for instance, response time, availability and bandwidth.

Middleware systems, such as the studies presented by (Costa et al., 2007; F. C. Delicato, Pirmez, Rust, & Pires, 2003; Khedo & Subramanian, 2009; T. Liu & Martonosi, 2003; Rahman, April, 2009), represent good software instruments for defining the high-level application logic and to deal with heterogeneity and distribution issues, but most of them does not provide an explicit way for defining the underlying autonomic behavior of WSNs. Therefore, novel studies are presented in section 3.3.1. These studies propose approaches aiming to provide adaptive mechanisms to the middleware system.

Existing middleware platforms for WSNs can be classified into four categories (Aslam, 2009) i.e. Traditional or Classic, Data Centric, Virtual Machines, and Adaptive. Classic middleware aims to encapsulate the complexity of the underlying communication and sensing system by providing interfaces to ease application development for WSNs. However, distributed processing across multiple nodes is not an inherent feature of this middleware category but needs to be included in applications. Data Centric middleware technology enforces the concept of dealing with sensor as data sources where data can be extracted using SQL like queries. However, this technology only allows query-based applications and therefore limits the scope of the sensor network. Furthermore, dynamic retasking of the WSNs is not possible with data centric middleware. Virtual machine (VM) based middleware abstracts the node or network into an execution layer that executes multiple instances of application programs or scripts. VM based middleware offers great flexibility in terms of the range of applications it supports and the ability to retask the sensor network in a dynamic fashion. However, current VMs have limited memory and code execution performance for instruction sets. Finally, adaptive middleware approaches use fidelity algorithms focusing on the adaptability. This classification of middleware provides an approach for the reconfiguration of system running on sensor devices. This type of middleware platforms has specific deficits as scalability, QoS limitations, resource constraints and lack of domain knowledge.

2.3 Autonomic Computing

Autonomic Computing refers to the self-managing characteristics of distributed computing resources, adapting to unpredictable changes while hiding intrinsic complexity to operators and users. It is inspired by the autonomic nervous system of the human body. This nervous system controls important bodily functions (For instance, respiration, heart rate, and blood pressure) without any conscious intervention.

In a self-managing autonomic system, the human operator takes on a new role: instead of controlling the system directly, a general policies and rules that guide the self-management process are defined. For this process, IBM defined the following four types of property referred to as self-* properties (IBM, 2005). These properties are detailed in Section 2.3.1 and Section 2.3.2 describes the adaptation mechanism based on MAPE-K model, also proposed by (IBM, 2005).

2.3.1 Autonomic Computing Self-* Properties

- **Self-Configuration.** The ability to adapt itself to the environment changing according to high-level policies aligned with business goals and defined by system administrators. Self-Configuration is not limited to the ability of a system for configuring each device singly, it is related to provide the ability to adjust the device configuration dynamically (on-the-fly) in a global way for the well being of the environment as a whole.
- **Self-Healing.** The ability to recover after a system disturbance and minimize interruptions to maintain the software available for the user, even in the presence of individual failure of components. For that, the system must be able to isolate

devices or malfunction components in order to minimize the impact in services, by continuously maximizing the availability and reliability of the managed environment.

- **Self-Optimization.** The system ability to improve its operation continuously, by identifying new opportunities for performing same operations with better performance or minor cost, using the same resources. For that, it is necessary to identify, verify and perform changes in configuration to maximize the resource usage without human intervention.
- **Self-Protection.** The ability to predict, detect, recognize and protect itself from malicious attacks and unplanned cascade failures. For such, it is necessary to anticipate problems based on data correction and the study of previous states. Usually, this requirement is related to unauthorized access, denial-of-services (DoS) or failures in general.

2.3.2 IBM MAPE-K Process

Components of Autonomic Computing systems are derived from the original IBM MAPE-K proposal (Castañeda, 2012). They are in charge of the adaptation process. This set of components process data information about the context into adaptation decisions and therefore is the heart of this reference architecture.

- **Sensor.** This component is the starting point of the MAPE-K loop and it is configured to make measurements about the context with the purpose of sensing a specific variable of interest during runtime. A sensor acts as a watcher and interacts with the monitor component. The sensor provides a representation of the target system's state in a specific variable of interest and other external systems information, in the form of context entities needed for the adaptation mechanism.
- **Monitor.** As described by IBM, this element is responsible to monitor the sensed context. The data required to perform this monitoring is called context data and is provided by the Sensor component (IBM, 2005). However, this context data is every information about the context that the Sensor Component can sense, either because something happened in the context or because the Monitor asked it to gather information in a specific time. Despite this, not all the context data that the

Sensor Component is providing are required to be monitored. The Monitor is responsible for filtering all that context data, keeping only the one that is relevant.

- **Analyzer.** According to IBM proposal (IBM, 2005), this element is responsible for studying the Context Event Information given by the Monitor in order to evaluate if the system objectives are being fulfilled. Moreover, the Analyzer has to decide whether an adaptation is required. As an income, context events are received from the monitor and then the Analyzer must evaluate them against the adaptation policies configured in the system.
- **Planner.** As described in the IBM MAPE-K loop proposal, the Planner element is responsible for building a plan with all the adaptation actions needed to perform certain changes in the target system, reaching the desired state of the context entity provided by the Analyzer in the diagnosis (IBM, 2005). To build this plan, the Planner must follow some Adaptation Policies provided by the Knowledge Base Component.
- **Executer.** This is the last element in the IBM intelligent Loop and is responsible for executing the adaptation plan (IBM, 2005). This component translates every step of the plan to commands and guarantees the correct execution over the target system. This component is connected to the Effector, which directly instruments the target system.
- **Effector.** This element is the finishing point of the MAPE-K loop interfacing the adaptation mechanism and the target system. The Effector component is configured to effect the changes needed to alter the target system's behavior according to the adaptation needs. The Effector can either modify parameters of system operations or execute operations. This component is connected to the Executor component and receives the commands to execute over the target system from it.
- **Knowledge Base.** In the IBM's proposal, the Knowledge element is connected to each of the four components in the MAPE-K loop (IBM, 2005). This component's main purpose is the provisioning of adaptation policies for other components in the MAPE-K loop. The knowledge base can be used to manage the historical behavior of this adaptation mechanism during time.

In order to implement MAPE-K components allowing self-adaptation of software, feedback loops are required with explicit functional elements and interactions between them for managing the dynamic adaptation. The Feedback control loop approach is considered essential for understanding not only the model of adaptation and collaboration, but also the types of adaptive systems. It can be considered the most dynamic adaptive approach (Castañeda, 2012).

The MAPE-K model described in Figure 3 provides conceptual guidelines about the autonomic systems conception; in practice, this information model needs to be mapped to an implementable architecture for managing and controlling autonomic WSNs.

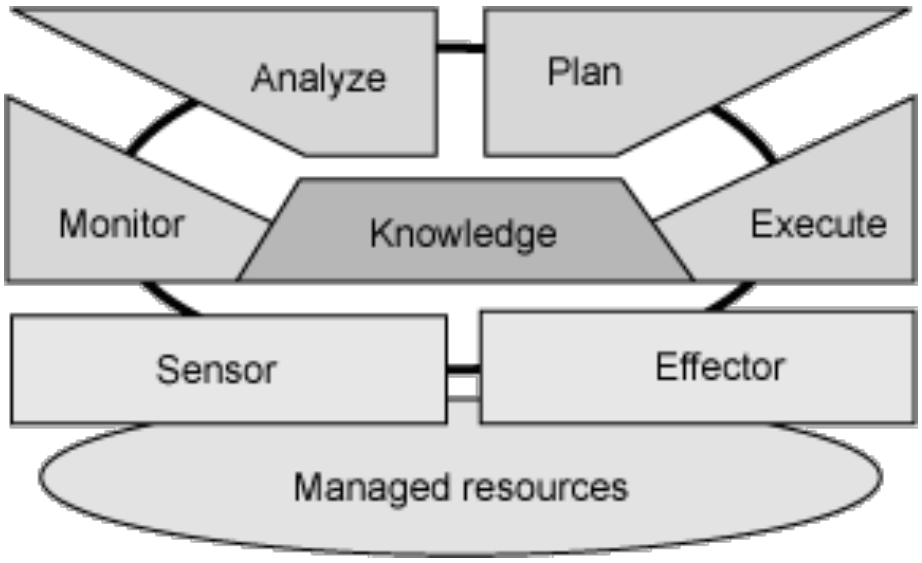


Figure 3. Closed feedback control-loop in autonomic systems (IBM, 2005)

Feedback control loops are considered a key issue in pursuing self-adaption for any system, because they support the MAPE-K activities. They play an integral role in adaptation decisions. Thus, key decisions about a self-adaptive system's control depend on the structure of the system and the complexity of the adaptation goals.

2.4 Reference Architectures

Reference architecture (RA) has emerged as an important area of research in software architecture. It is considered a blueprint of software development, since it guides the design of concrete architectures of systems for a given application domain (Angelov, Trienekens, & Grefen, 2008; G. Muller, 2008; E. Nakagawa, F. Oquendo, & M. Becker, 2012). Reference architectures can directly impact on the quality and design of a range of concrete architectures and software systems developed from them (Angelov, Grefen, & Greefhorst, 2009). Therefore, they must consider business rules, architectural styles, best practices of software development, and software elements that support the design of systems of the application domain.

According to (G. Muller, 2008), a RA can be used to facilitate the design of concrete architecture or as a standardization asset that supports interoperability among systems or components of systems. Figure 4 shows how the same RA can result in different concrete architectures, depending on the context and involved stakeholders.

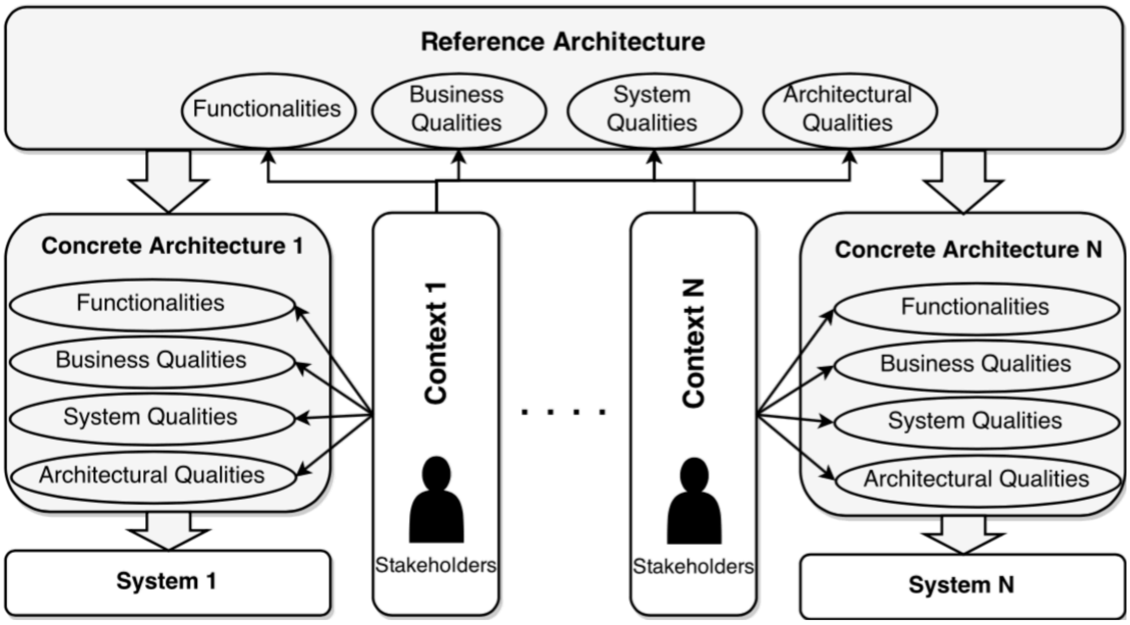


Figure 4. Role of stakeholder and contexts for RA and concrete architectures. Source: (Angelov et al., 2008)

2.4.1 Related Concepts

Sometimes the term reference architecture has been used interchanged with other terms, such as software architecture, reference model, product line architecture and domain specific software architecture. Moreover, other concepts as architectural patterns, architectural styles, concrete architecture, and ontologies are somehow related to reference architecture.

- **Software Architecture**: it is defined as the structure or structures of the system which comprise software elements (e.g., services, components, modules), the externally visible properties of those elements (i.e., behavior of each element), and the relationships among them (Bass, 2012). In this context, reference architectures are a special type of software architecture that captures the essence of the architectures of a set of software systems in a given domain (E. Nakagawa et al., 2012).
- **Reference model (RM)**: it is an abstract representation of the elements in a given domain of interest, the behavior of such elements, and the relationships among them (Bass, 2012). The RM mapped onto software elements (that cooperatively implement the functionality defined in the RM) and the data flow between them is considered as a reference architecture. The mapping does not need to be performed one to one.
- **Product Line Architecture (PLA)**: it is an abstract software architecture for describing elements of a family of similar products developed by the Software Product Line (SPL) approach (P. Clements & Northrop, 2002). RAs are generally on a higher level of abstraction compared to PLAs (Nakagawa et al., 2014).
- **Architectural patterns**: it expresses fundamental structural organization schema for software systems. An architectural pattern describes a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for the organization of relationships among them (Buschmann, Henney, & Schimdt, 2007). An architectural pattern not only provides the structure of a solution for a usual problem in software design, but also describes the context in which such a problem may occur and the consequences associated with the pattern adoption.
- **Architectural styles**: it is a type of architectural pattern that defines a family of

systems in terms of a pattern of structural organization, establishes vocabulary of components and connector types, and a set of constraints on how they can be combined (Eeles, 2008). A style reduces the set of possible forms and imposes a certain degree of uniformity on the architecture.

- **Concrete architecture**: it is the software architecture of a given software system tailored for a particular set of stakeholders and concerns. Stakeholders refer to any individual, team or organization that is interested in a software system and plays a relevant role during its development process (ISO/IEC/IEEE, 2011). Concern is an interest or need of one or more stakeholders on the software system under development (ISO/IEC/IEEE, 2011).
- **Ontologies**: specifically domain ontologies are an explicit formal representation of knowledge about a domain of application. This includes types of entities that exist in the domain, properties of those entities, relationships among entities, processes and events that happen with those entities.

Reference architectures (RAs) are created based on reference models (RMs), architectural patterns (Angelov, Grefen, & Greefhorst, April, 2012). Architectural patterns can be used in the design of reference architectures for achieving desired architectural qualities. Figure 5 represents the relationship between reference models, architectural patterns and reference architectures.

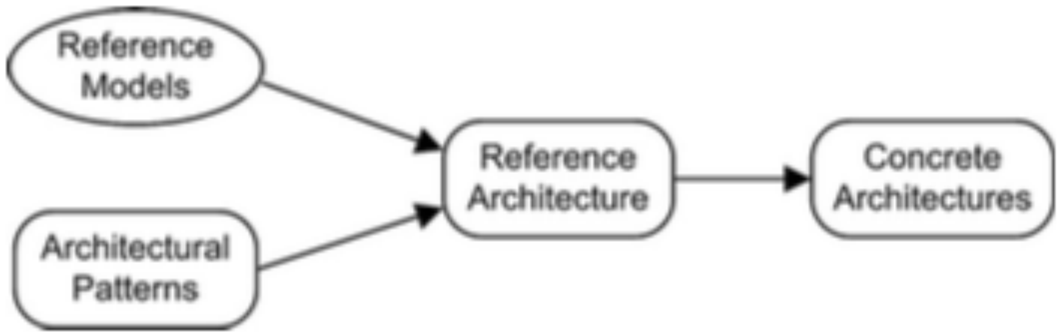


Figure 5. Relationship between reference models, architectural patterns, reference architectures, and concrete architectures. Source: (Bass, 2012)

Reference Architectures (RAs) are a special type of software architecture that has become an important element for the systematic reuse of architectural knowledge. The main purpose of RAs is to facilitate and guide (E. Nakagawa et al., 2012):

(i) The design of concrete architectures for new systems;

(ii) The systems extensions of neighboring domains of an RA;

(iii) The systems evolution derived from an RA; and

(iv) The improvement in the standardization and interoperability of different systems.

A reference architecture plays a dual role with regard to specific target software architectures: it generalizes and extracts common functions and configurations; and it provides a base for instantiating target systems. In other words, they can be seen as a knowledge repository of a given domain, contributing to the software development, since reuse of knowledge and improvement in the productivity are promoted.

2.4.2 Reference Architecture Classification

Due to the diversity of application domains and interests, RAs can be classified according to three dimensions, described as follows (Angelov et al., 2009, April, 2012).

- **Context Dimension**: RAs can be designed in the context of a single organization or multiple organizations that share a common characteristic, such as geographical location and market domain. Different type of organizations are usually involved in the establishment of these architectures. Besides, such architectures can be designed before any existing system (preliminary) or after accumulating the experience from the development of several systems (i.e., classical);
- **Goal Dimension**: as stated by (G. Muller, 2008), RAs can be designed with two main goals: standardization and facilitation. RAs for standardization aim at improving interoperability among systems by promoting unified understanding of the domain at the architectural level. RAs for facilitation aim at providing guidelines for development of concrete architectures;
- **Design Dimension**: RAs are represented by several types of elements, including components, interfaces, protocols, algorithms, policies, and guidelines. These elements can be described in different levels of detail, formalism, and abstraction.

2.4.3 Reference Architecture Evaluation

An architecture evaluation helps identifying the strong and weak aspects of the architecture and gives an indication for the success of the system development and implementation processes. A reference architecture serves as a guiding tool for many projects taking place in diverse contexts. Thus, its evaluation prior to its adoption by the stakeholders is of even greater importance. Furthermore, a strong positive evaluation of a reference architecture is an incentive for its wider adoption.

The cost to fix an error found during requirements on early design phases in a system development is orders of magnitudes smaller to correct than the same error found during testing. Software architecture is the product of the early design phase, and its effect on the system and the project is profound (P. C. Clements, Kazman, & Klein, 2002). There are a number of differences between reference architectures and concrete architectures, and reference architectures are considered by some authors as very distant from concrete architectures: "reference architectures are not architectures; they are useful concepts that capture elements of an architecture" (Angelov et al., 2008). The main difference between a concrete architecture and a reference architecture are: (i) Reference architectures are of a generic nature, (ii) There is not a clear group of stakeholders of a reference architecture (iii) Due to their generic nature, reference architectures are defined on a high level of abstraction, and (iv) A reference architecture has to address more architectural qualities than a concrete architecture (Angelov et al., 2008).

The concrete architecture evaluation occurs when the architecture has been specified but before implementation has begun. Users of iterative or incremental life-cycle models can evaluate the architectural decisions made during the most recent cycle. However, one of the appealing aspects of architecture evaluation is that it can be applied at any stage of an architecture's lifetime, there are two useful variations from the concrete architectures: early and late (P. C. Clements et al., 2002).

Early. Evaluation needs not to wait until an architecture is fully specified. It can be used at any stage in the architecture creation process to examine those architectural decisions already made and choose among architectural options that are pending. That is, it is equally proficient at evaluating architectural decisions that have already been made and those that are being considered.

Late. The second variation takes place when not only the architecture is nailed down but the implementation is complete as well. This case occurs when an organization inherits some sort of legacy system. The techniques for evaluating a legacy architecture are the same as those for one that is newborn.

An evaluation is a useful thing to do because it will help the new owners understand the legacy system, and let them know whether the system can be counted on to meet its quality and behavioral requirements. The evaluation will first identify what the areas of interest are and then highlight the strengths and weaknesses of each architecture in those areas.

Most of quality attributes lie squarely in the realm of architecture. For instance, the ATAM (Kazman, Klein, & Clements, 2000), a type of Early Evaluation, concentrates on evaluating an architecture for suitability in terms of imbuing a system with the quality attributes.

As mentioned before, concrete and reference architectures have certain differences. These differences lead to a number of problems that do not allow the direct application of methods for the evaluation of concrete architectures in the case of reference architectures. For Samuil Angelov at el (Angelov et al., 2008), existing methods fall short in providing techniques for the evaluation of the architecture qualities of Reference Architectures. They show that existing methods on the evaluation of concrete architectures are not directly applicable for an evaluation of reference architectures and present an adaptation and extension of ATAM approach to the Evaluation of Reference Architecture. These types of approaches are important initiatives in reference architecture evaluation once methods commonly used on concrete architecture evaluations cannot be directly applied in reference architectures; mainly they have differences in the level of abstraction.

2.4.4 Software Architecture Representation

Most software architects use informal box and arrow diagrams to describe the interrelationship between the various components abstracted from the analysis phase. But unfortunately these line and box diagrams are highly ambiguous. Also, software architectures can be described according to the structural and the behavioral viewpoints. The structural viewpoint specifies components, connectors and the configurations of both. The behavioral viewpoint specifies the dynamism of the architecture, in other words, how its components and connectors can change at runtime. This dynamism may be specified in terms of actions a system executes or participates in, relations among actions and behavior of components and connectors. Normally, WSNs are deployed in dynamic environments, with unexpected changes of the execution context, where a reconfiguration of nodes' behavior is needed, therefore, the choice of a modeling approach that supports this dynamism can aid to implement WSNs systems.

2.4.4.1 Software modeling approaches

Over the last decades, different proposals emerged for software modeling, from which we can highlight two well-known approaches: (i) Architecture Description Languages (ADLs); (ii) OMG (Object Management Group) standard modeling languages, such as UML (Unified Modeling Language) and SysML (Systems Modeling Language). ADL is a language for formally representing the architecture of a software system. ADLs provide notations for representing and analyzing architectural designs. According to (P. C. Clements, March, 1996) ADLs must support creation, refinement and validation of architectures, must represent most of the common architectural styles, must provide views of architectural information (such as Pipe and filters and Layered Systems), must express architectural information while being platform independent. Different ADLs have been developed in the academia and industry, and they can be Semi-formal ADLs and Formal ADLs. Semi-Formal ADLs aim to support communication among stakeholders and provide visual notations. ACME (Garlan, Monroe, & Wile, Nov, 1997), xADL (Dashofy, van der Hoek, & Taylor, August, 2001) and Aesop (Garlan, Allen, & Ockerbloom, December, 1994) are examples of Semi-formal ADLs. In practice, UML 2 is a Semi-Formal ADL. This language provides a graphical representation of architectures, it is a general purpose modeling language and supports the description of components. UML 2 can describe ports and interfaces, component behavior as state machines; by using OCL to support component constraints and can describe classifiers and instances. Most of Semi-Formal ADLs cannot describe architecture behavior and does not support automated analysis and validation. Formal ADLs aim to support automatized verification of properties and to provide textual notations based on mathematical principles.

Most formal ADLs provide limited support for dynamism. Pi-ADL (F. Oquendo, May, 2004), detailed in Section 2.4.4.2, supports full specification of dynamic architectures. Other examples of prominent formal ADLs are Dynamic Wright (Allen, Douence, & Garlan, 1998) and LEDA (Canal, Pimentel, & Troya, 1999).

Dynamic Wright is a formal ADL that allows describing the behavior and reconfiguration of a system by using a variant of the Communicating Sequential Processes (CSP). However, CSP is able to specify only static configurations, dynamic reconfigurations are not supported and have to be simulated. By adopting an exogenous approach, Dynamic Wright provides a special component in the architecture responsible for centralizing all reconfiguration operations since only it can modify the architecture.

LEDA is a formal ADL based on π-calculus. It is structured upon: (i) components, which represent system modules and can be either functional elements or connectors; (ii) roles, which describe the observable behavior of components; and (iii) attachments, which define connections among component instances. The approach adopted in LEDA for dynamism is endogenous, decentralized, so that the reconfiguration operations are described along with the behavior specification of components. However, the behavior of architectural elements is specified by directly using the operators defined in π-calculus. LEDA does not provide architectural abstractions over these constructs, thus making architecture descriptions more difficult.

2.4.4.2 Pi-ADL

Pi-ADL (π-ADL) was designed by the ArchWare European Project (F. Oquendo et al., May, 2004). This language enables the specification of static, dynamic and mobile architectures. Pi-ADL (F. Oquendo, May, 2004) is a well-founded formal language also based on π-calculus. Pi-ADL focuses on the formal description of architectures from the runtime viewpoint: the structure, the behavior and how they may evolve over time. The language follows the principle of executability, i.e., a virtual machine can run the specifications of the architecture. Pi-ADL allows description of software architecture in terms of components, architectures and their composition. Figure 6 shows the main elements of the pi-ADL architecture.

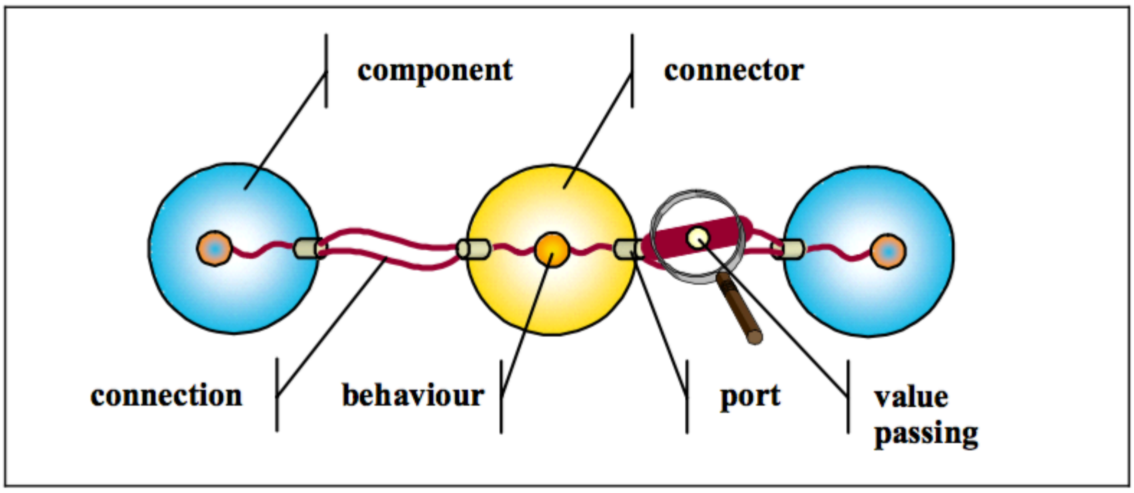


Figure 6. Main elements of Pi-ADL. Source: (F. Oquendo, May, 2004)

A description of the elements is shown below:

- Components are described in terms of external ports and internal behavior, and they specify the computational elements of a software;
- Ports are described in terms of connections between components and their environment. They put together connections providing an interface within the component and its environment;
- Connections are interaction points. They provide communication channels between the architectural elements. A component can send or receive values via connections. They can be declared as output connections (values can only be sent), input connections (values can only be received), or input-output connections (values can be sent or received).
- Connectors are special-purpose components. They are described as components in terms of external ports and an internal behavior. However, their architectural role is to connect together components. They specify interactions among components. Therefore, components provide the locus of computation, while connectors manage interaction among components. A component

cannot be directly connected to another component. In order to have actual communication between two components, there must be a connector between them.

Figure 7 illustrates an example of Pi-ADL description for a simple pipeline architecture composed of two components (filters) linked through one connector (pipe) (F. Oquendo et al., May, 2004). Filter components transform data received from their input and send the transformed data to their output, while pipe connectors transmit the output of one filter to the input of another filter.

The Filter component depicted in Figure 7-a is declared with two connections:

(i) inFilter, an input connection for receiving data to be processed, and;

(ii) outFilter, an output connection for sending processed data.

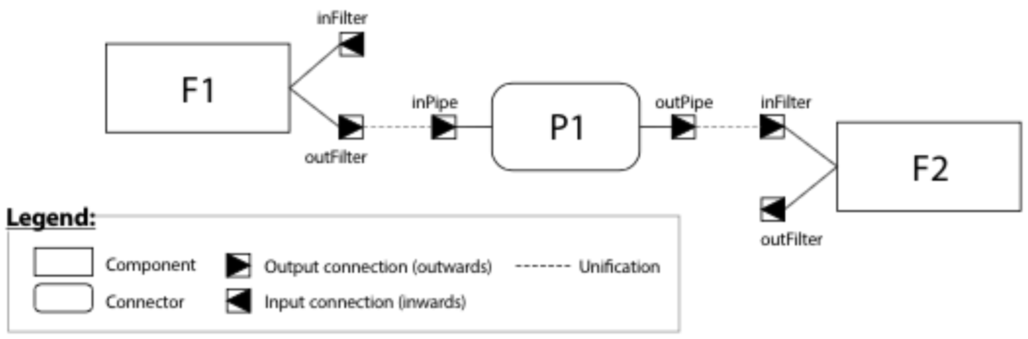
The behavior of the Filter component encompasses the transform function, which receives data from the inFilter connection and returns data to be sent through the outFilter connection. In such an architecture description, the transform function is unobservable, i.e., internal.

Similarly, the Pipe connector depicted in Figure 7-b is declared with two connections:

(i) an input connection (inPipe) that receives data as output of a filter, and;

(ii) an output connection (outPipe) that sends data to the input of another filter.

Finally, the PipeFilter architecture depicted in Figure 7-c is specified as a composition in which two filter components (F1 and F2) and one pipe connector (P1) are instantiated. The attachments of these architectural elements take place through the unification of the output connection of the filter F1 with the input connection of the pipe P1, and the unification of the output connection of the pipe P1 with the input connection of the filter F2. Therefore, data can be sent from filter F1 to filter F2 through the pipe P1 via the declared connections.



```
component Filter is abstraction() {
   connection inFilter is in(String)
   connection outFilter is out(String)
   protocol is {
      (via inFilter  receive String
       via outFilter send String)*
   }
   behaviour is {
      transform is function(d : String) : String {
         unobservable
      }
      via inFilter receive d : String
      via outFilter send transform(d)
      behavior()
   }
}
```

(a)

```
connector Pipe is abstraction() {
   connection inPipe is in(String)
   connection outPipe is out(String)
   protocol is {
      (via inPipe  receive String
       via outPipe send String)*
   }
   behaviour is {
      via inPipe receive d : String
      via outPipe send d
      behavior()
   }
}
```

(b)

```
architecture PipeFilter is abstraction() {
   behavior is {
      compose {
             F1 is Filter()
         and P1 is Pipe()
         and F2 is Filter()
      } where {
         F1::outFilter unifies P1::inPipe
         P1::outPipe   unifies F2::inFilter
      }
   }
}
```

(c)

Figure 7. Description of a simple pipeline architecture in pi-ADL, Source: (F. Oquendo, May, 2004)

The following general principles guided the design of Pi-ADL:

I. **Formality**: Pi-ADL is a formal language: it provides a formal system, in the mathematical sense, for describing dynamic architectures and reasoning about them;

II. **Runtime perspective**: Pi-ADL focuses on the formal description of architectures from the runtime viewpoint: the (runtime) structure, the (runtime) behavior, and how these may evolve over time;

III. **Executability**: Pi-ADL is an executable language (a virtual machine runs specifications of software architectures).

2.5 Final Considerations

In this chapter we introduced the basic concept of WSNs, Autonomic Computing, and Reference Architectures.

Regarded to WSNs, we noticed a gap of middleware system to satisfy the requirements of special features of wireless sensor networks such as the management of limited constraints and solutions for self-adaptation (Portocarrero et al., 2014). The adaptation functions should facilitate provision of quality of service to applications while using the limited resources of WSNs and extending their lifetime. Middleware is an approach to satisfy adaptation. Middleware for supporting such applications would comprise more functions, such as fusion, the ability to dynamically change the node behavior and operation mode, and effective adaptation between applications and sensor nodes. The quality of middleware for WSN not only depends on how well it has been designed and implemented in network nodes but also on how well it can deal with problems and events at runtime (F. C. Delicato et al., 2003).

Autonomic Computing is presented as an interesting option to meet basic requirements in WSNs design. Autonomic computing principles can be applied to WSNs in order to optimize network resources, facilitate their operations and achieve desired functionality in the wide field of sensing-based applications, by providing conditions for this type of network manage itself without involving human operators (Portocarrero et al., 2014).

References architectures can be used to facilitate the design of concrete architecture for self-adaptive WSNs or as a standardization asset that supports interoperability among components of WSNs applications. Hence, Pi-ADL provides a formal and theoretically well-founded language for describing dynamic software architectures under structural and behavioral perspectives. In this perspective, our RA, described in next Chapter, was specified using Pi-ADL since this formal ADL has the ability for representing the dynamism of behavior of components. The highly changeable environment of WSNs has Pi-ADL as a suitable candidate for representing the Reference Architecture of a self-adaptive WSNs.

CHAPTER 3: Research Proposal

3.1 Initial Considerations

In general, RAs have been built using an ad-hoc approach, without following a systematic process. However, systematizing their building will allow achieving more effective RAs that could more completely attempt their purpose. In this context, it is possible to find several initiatives (Angelov et al., April, 2012; Bayer et al., 2004; Cloutier et al., 2010) oriented to provide general high-level guidelines, principles and recommendations to build RAs. However, (Nakagawa et al., 2014) proposes ProSA-RA, a process for the building of RAs, focusing on how to design, represent, and evaluate such architectures.

In this perspective, Section 3.2 details the ProSA-RA process and the remainder of this Chapter aims to apply ProSA-RA for building the proposed RA.

3.2 ProSA-RA

The conception and evaluation of RAMSES were based on ProSA-RA. ProSA-RA is a process that systematizes the design, representation, and evaluation of RAs. The outline structure of ProSA-RA is illustrated in Figure 8.

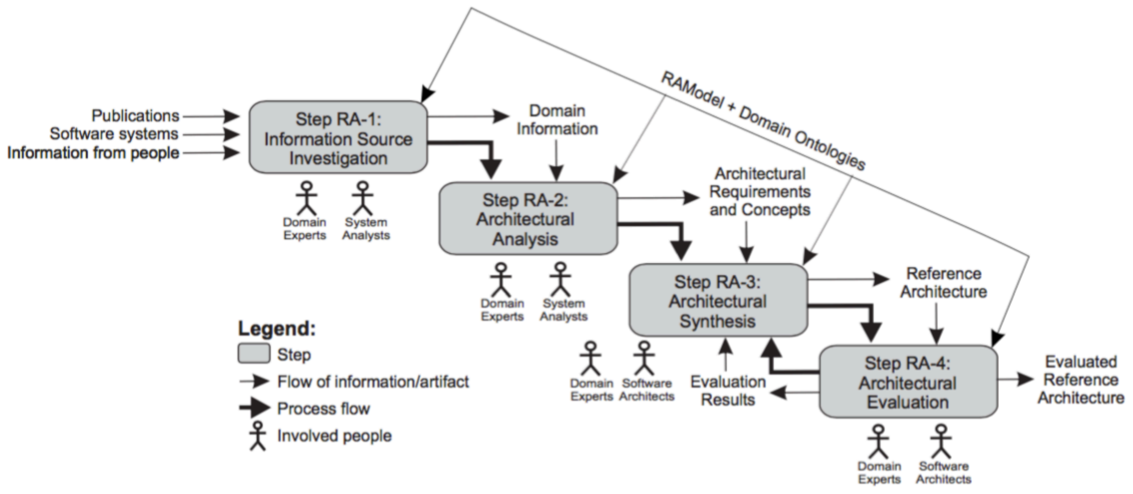

RAModel + Domain Ontologies
Publications
Software systems
Information from people
Step RA-1: Information Source Investigation
Domain Information
Domain Experts
System Analysts
Step RA-2: Architectural Analysis
Architectural Requirements and Concepts
Domain Experts
System Analysts
Step RA-3: Architectural Synthesis
Reference Architecture
Legend:
Step
Flow of information/artifact
Process flow
Involved people
Domain Experts
Software Architects
Evaluation Results
Step RA-4: Architectural Evaluation
Evaluated Reference Architecture
Domain Experts
Software Architects

Figure 8. Outline Structure of ProSA-RA. Source: (Nakagawa et al., 2014).

In short, four steps compose this process:

- **Step RA-1** - **Information Source Investigation**. In this step, the main information sources to be used in the design of the RA are selected. These sources must provide information about processes and activities that could be supported by software systems of the target domain. Since RAs should be the basis for various software systems of a given domain, these sources must involve a more comprehensive knowledge about the domain if compared with information sources when developing the architecture of a specific system. The output of this step is a set of information sources, which will be used to get requirements of the intended RA in the next step.

- **Step RA-2** - **Architectural Analysis**. Based on the selected sources, the set of requirements of software systems of the target domain is identified and, based on these requirements, the set of requirements of the RA is then identified. After that, the set of concepts that must be considered in the RA is established. Domain concepts identified in this task will be further used as an input to build the architectural description of the RA. In summary, outputs of this step are the requirements of the RA.

- **Step RA-3** - **Architectural Synthesis**. In this step, the architectural description of the RA is built using RAModel (Reference Architecture Model) as a framework (E. Nakagawa et al., 2012). RAModel provides information on possibly all elements (and their relationships) that could be contained in RAs, independently on application domains or purpose of such architectures. The outcome of this step is an architectural description composed of a set of architectural views and additional artifacts of the RA.

- **Step RA-4** - **Reference Architecture Evaluation**. RA evaluation refers to the task of checking the architectural description of such architecture along with diverse stakeholders intending to detect defects in this description. For this, ProSA-RA uses a checklist-based inspection approach, named FERA (Framework for

Evaluation of Reference Architectures) (J. Santos, Guessi, Galster, Feitosa, & Nakagawa, 2013).

Following, we describe the activities and methods conducted within each phase. In this perspective, Section 3.3 describes a set of activities of Step RA-1. The requirements of our RA and quality attributes are identified in Section 3.4 (Step RA-2). In Section 3.5, the activities of Step RA-3 were conducted and Section 3.6 details activities of Step RA-4.

3.3 Step RA-1: Information Source Investigation

In order to attend the Step RA-1 of ProSA-RA, a set of information sources was selected that allows identifying processes, activities, and tasks that must be supported by self-adaptive middleware systems for WSNs. Thus, the following activities were conducted:

A. Find out which interactions and behavior can be automated in WSN components, taking into account the hardware and software limitations of these networks.

B. Identify which development approaches of AC are used for designing WSN middleware system architectures that allow the self-management of the network.

C. Identity the most important RAs and reference models in WSN and autonomic computing domains that could serve as a starting point for our RA.

D. Identify architectural styles/patterns that are commonly found in the software architectures of autonomic computing systems and wireless sensor network applications.

In order to attend activities A and B of Step RA-1 of ProSA-RA, we have conducted a systematic literature review (SLR) (Kitchenham & Charters, 2007) to accomplish a methodological, fair analysis about the researched subject. Thus, Activity A is presented in Section 3.3.1 and Activity B is presented in Section 3.3.2. In order to attend activity C, in Section 3.3.3 we conduct two other SLRs that allow us to present related works that describe RAs for WSNs and for autonomic systems (AS). Finally, in

Section 3.3.4 architectural styles/patterns commonly used in self-adaptive systems and WSN applications are identified.

In recent years, SLRs have been used for presenting the state of the art regarding a subject topic in a comprehensive, non-biased way, and for identifying interesting and important research opportunities for further investigations. Activities A and B have been carefully planned as a Systematic Literature Review based on a rigorous methodological framework previously introduced by (Kitchenham & Charters, 2007), which provides a set of well-defined steps carried out in accordance with a predefined protocol. This rigorous methodology can be viewed as the main point that differentiates a systematic procedure from a simple, traditional literature review as it seeks to avoid the maximum of bias throughout the process, thus providing scientific value for the obtained findings.

SLRs are means of evaluating and interpreting available relevant research to particular research questions, topic area or phenomenon of interest, thus aiming to present a fair evaluation of a research topic. A SLR is typically divided in three basic steps (Kitchenham & Charters, 2007):

(i) **Planning**, which defines the research questions to be answered, the search strategy to be adopted, the selection criteria, and the data extraction and synthesis methods to be used, thus yielding a protocol that will guide the conduction of the whole process;

(2) **Conduction**, in which the primary studies are identified, selected, and evaluated according to the previously established protocol, and;

(3) **Reporting** (or Analysis), which aggregates extracted information from the relevant primary studies considering the research questions and outlines conclusions from them.

SLRs have been recently regarded as a useful way for dealing with research evidences, thus making it possible to systematically identify, select, analyze, and aggregate them for providing knowledge about a given research topic. Furthermore, they have been commonly used for synthesizing existing work from the literature in a comprehensive and non-biased way and for identifying research challenges and opportunities in the state of the art regarding the research subject. In our work, a SLR was conducted to attend activities A and B of Step RA-1 of ProSA-RA with the goals of (i) finding out which interactions and behavior can be automated in WSN components and (ii) identifying which development approaches of AC are used for designing WSN middleware system architectures. Therefore, the Planning and Conduction phases of the performed SLR are presented in (Portocarrero et al., 2014). We defined one search string to seek for studies related to activities A and B. The general form of the search string is shown below:

Query strings:

TITLE-ABS-KEY

(

(autonomic **OR** self-adaptive **OR** self-adapt **OR** self-adaptation **OR** self-adapted **OR** self-management **OR** autonomous**) AND**

("wireless sensor network" **OR** "sensor network" **OR** WSN **OR** WSAN **OR** "wireless sensor and actuator network" **OR** "wireless ad-hoc network" **OR** "wireless actuator network"**) AND**

("design project" **OR** "design model" **OR** "architecture" **OR** "architecture-based" **OR** framework **OR** middleware **OR** routing **OR** clustering **OR** "data aggregation" **OR** "data dissemination"**)**

)

As the first and most critical step of the tasks performed in our systematic literature review, we translate the goals of activities A and B into two research questions (RQ1 and RQ2). The results obtained from the performed SLR are presented in the reporting phase according to the defined research questions. They were used to find primary studies to understand and summarize evidences about the application of AC principles to WSN in order to optimize network resources. In this context, the following research questions (RQ) were defined:

RQ1 (Activity A): Which interactions and behavior should be automated in the components of WSNs?

RQ2 (Activity B): Which model/programming/design/developing approach can be applied to provide autonomic behavior to middleware systems for Wireless Sensor Networks?

3.3.1 Activity A – Step RA-1: Self-adaptive WSN system concerns

This activity is related to what can be automated in Wireless Sensor Networks (RQ1). We noticed that most of the selected studies are interested in a lightweight, autonomic behavior for WSNs in order to:

(i) Adapt the network to dynamic environments (for instance, node energy depletion and arrival of new applications) and unpredictable events (for instance, node and link failures);

(ii) Save energy, extending the network lifetime;

(iii) Provide network scalability;

(iv) Provide reliability of sensing data and;

(v) Provide independence between the management functions of applications and network configuration.

Selected studies, detailed in

Table 1, addresses the aforementioned concerns by leveraging the self-* properties of AC (self-configuration, self-healing, self-optimization and self-protection).

Table 1. Selected primary studies of self-adaptive WSN

ID	*Title*	*Year*	*Ref*
S1	Design of a generic management system for wireless sensor networks	2014	(Cao, Bellata, & Oliver, Sep, 2014)
S2	Smart Policy Generating Mechanism for Policy Driven Self-Management in Wireless Sensor Networks	2013	(S., Zeng, & Liu, 2013)
S3	A QoS-Driven Self-Adaptive Architecture For Wireless Sensor Networks	2013	(Jemal & Halima, June, 2013)
S4	Using Dynamic Software Variability to Manage Wireless Sensor and Actuator Networks	2013	(Mouronte, Ortiz, Garcia, & Capilla, May, 2013)
S5	DISON: A Self-organizing Network Management Framework for Wireless Sensor Networks	2013	(Minh, Bellalta, & Oliver, 2013)
S6	Autonomous Configuration of Spatially Aware Sensor Services in Service Oriented WSNs	2013	(Shah & Szymanski, Mar, 2013)
S7	A Novel Wireless Sensor and Actor Network Framework for Autonomous Monitoring and Maintenance of Lifeline Infrastructures	2012	(Imran, Alnuem, Alsalih, & Younis, Jun, 2012)
S8	Constraint-based Self-adaptation of Wireless Sensor Networks	2012	(Gamez, Romero, Fuentes, Ruovoy, & Duchien, 2012)

S9	Framework for a Self-managed Wireless Sensor Cloud for Critical Event Management	2012	(Nair, Morrow, & Parr, 2012)
S10	Autonomous Sensor Network Architecture Model	2012	(Tóth & Vajda, 2012)
S11	Developing Wireless Sensor Network Applications Based on a Function Block Programming Abstraction	2012	(Kerasiotis, Koulamas, & Papadopoulos, 2012)
S12	Autonomous Sensor Networks for Process Monitoring and Automation	2012	(Balakrishnan & Hiremath, Jan, 2012)
S13	An Autonomic Plane for Wireless Body Sensor Networks	2012	(Fortino, Galzarano, & Liotta, Jan, 2012)
S14	Framework for distributed policy-based management in wireless sensor networks to support autonomic behavior	2012	(Qwasmi & Liscano, Jan, 2012)
S15	Autonomic Role and Mission Allocation Framework for Wireless Sensor Networks	2011	(Bourdenas, Tei, Honiden, & Sloman, Oct, 2011)
S16	Towards Aware, Adaptive and Autonomic Sensor-Actuator Networks	2011	(ElGammal & Eltoweissy, Oct, 2011)
S17	Autonomic computing driven by feature models and architecture in FamiWare	2011	(Gamez, Fuentes, & Araguez, 2011)
S18	Autonomous Decentralized Mechanism of Structure Formation Adapting to Network Conditions	2011	(Takano, Aida, Murata, & Imase, Jul, 2011)
S19	Swarm behavior control of mobile multi-robots with wireless sensor networks	2011	(W. Li & Shen, Jul, 2011)
S20	Middleware Support for a Self-Configurable Wireless Sensor Network	2011	(Gotz, Rettberg, & Podolski, Mar, 2011)
S21	Starfish: Policy Driven Self-Management in Wireless Sensor Networks	2010	(Bourdenas & Sloman, 2010)
S22	Autonomic networking in wireless sensor networks	2009	(Garlan, Schmerl, & Cheng, 2009)
S23	Secure Self-Adaptive Framework for Distributed Smart Home Sensor Network	2009	(Muraleedharan & Osadciw, 2009)
S24	Agilla: A mobile agent middleware for self-adaptive wireless sensor networks	2009	(Fok, Roman, & Lu, 2009)
S25	A Survey on the Applicability of Trust Management Systems for Wireless Sensor Networks	2007	(Fernandez-Gago, 2007)

Selected studies S1, S2, S4, S5, S12, S13, S16 and S22, focus on proposing solutions to address the **self-configuration** property. Sensor nodes reconfigure and adapt their behaviors of communication and sensing by dynamically altering parameter values according to the changing conditions and states of the network. Some examples of network configuration adopted in S22 are decreasing the node sensing duty cycle if the monitored phenomenon has no significant changes during a certain period of time; and reducing radio transmission power to shorten the communication range if the residual node energy has dropped to a critical level. S1 adopted a context model where a context indicates which information can result in reconfiguration actions. For instance, a reconfiguration action can be changing the transmit radio power or refusing to perform a requested application task (for saving energy). One example of a context is "The remaining battery of the node is low".

S13, S21, S22, S23, S24 and S25 address the **self-healing** property. This property may be considered an essential characteristic that all sensor networks should incorporate for assuring their reliability and correctness. Fault tolerance is a common feature addressed in this context. For example, as noted by S22, to lessen the impact of faulty nodes, sensor nodes surrounding the phenomenon area generally regroup among themselves in order to maintain the reliability and consistence of network connectivity and sensing coverage. After the faulty neighbor is detected, a node will choose a new neighbor to route its sensing data to. In S13, a filtering task (Figure 9) is interposed between the sensing and the processing in order to guarantee the quality of raw data and to avoid that a data corruption affects the entire application correctness.

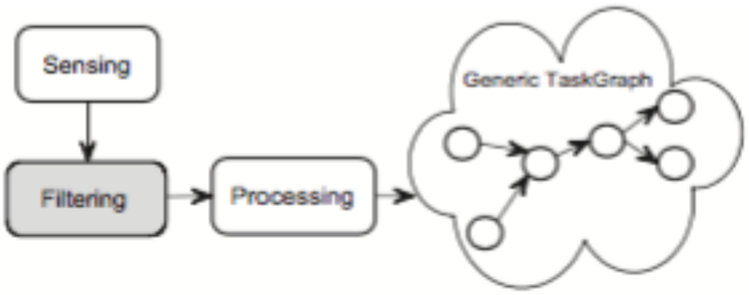


Figure 9. Self-healing example. Source: (Fortino et al., Jan, 2012)

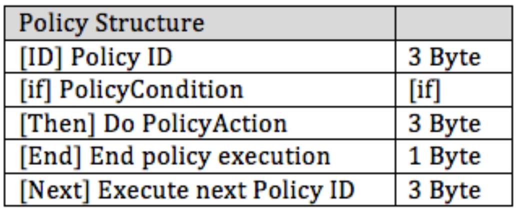
Regarding the **self-optimization** property, in order to extend the WSN operating lifetime, data transmission should be minimized and sensing data must be processed inside the nodes as much as possible aiming to minimize data transmission, as it is done in S2, S3, S4 S5, S10, S12, S13. For example, the authors in S10 propose a basic filtering process to recognize that there is unnecessary information in the raw sensor data able to attend application requirements. Thus, it is possible to reduce the number of transmitted data packets and consequently to reduce the total energy usage at sensor nodes.

Finally, regarding the **self-protection** property, an encryption process could be conceived in order to encrypt data coming from sensor nodes, if necessary. For S13, the privacy of information transmitted in a sensor network is one of the most priority goals regarding this AC property. In order to provide self-protection in WSN applications, the authors in S23 propose a secure communication method applied to Smart Homes. This method is a nature inspired framework based on mimicking ant's behavior. The protection mechanism depends on detecting the abnormal behavior in the network. This proposal adopts a clustered topology and the protection mechanism is located in a base station. All the cluster heads send their data directly to the base station. The operation of the mechanism is similar to the work of the human brain; the brain receives data from the whole body and detects abnormal behavior.

3.3.2 Activity B – Step RA-1: Approaches for self-adaptation in WSN

This activity is related to which development approaches can be applied to provide autonomic behavior to Wireless Sensor Networks applications (RQ2). From the analyzed studies, we have identified that selected studies use the following approaches: (i) policy-based reasoning approach; (ii) context-based reasoning approach; (iii) feedback control loops; (iv) mobile agents and; (v) model transformation and code generation. The analysis of each approach is shown below.

Studies S1, S2, S5, S6, S10, S14 and S21 rely on **policy-based reasoning (PBR)** approaches. As stated in S14, a general way of implementing autonomic behavior in distributed systems is through the use of policies. A policy is a constraint on the system behavior that can be expressed by using natural languages or mathematical notations. Policy-based systems use many existing expressive languages for specifying policies, but due to resource constraints, they are not appropriate for wireless sensor networks. Figure 10 depicts a policy structure specific for WSN (S14). Typically, a policy in WSN is specified in terms of tasks to monitor events, verify conditions and trigger actions whenever predefined events are detected by the monitoring task.



Policy Structure	
[ID] Policy ID	3 Byte
[if] PolicyCondition	[if]
[Then] Do PolicyAction	3 Byte
[End] End policy execution	1 Byte
[Next] Execute next Policy ID	3 Byte

Figure 10. Policy Structure. Source: (Qwasmi & Liscano, Jan, 2012)

In Figure 10, (i) ***[ID] Policy ID*** is used throughout the WSN to locate any particular policy. A policy identification consists of only two parts which are *Event ID* and Sequence Number of policy (*SeqNo*). *EventID* is 2 byte long. The first byte represents the event category such as (T,Temperature=1), and the second byte is an hexadecimal number representing the sequence number of possible events in the sensor. *SeqNo* is 1 byte long representing the policy sequence within the chain of applicable policies to the respective *EventID*. *EventID* and *SeqNo* are sensor dependent information and can be locally accessed from the sensor. Thus the sensor can identify the policy ID locally without the need to reach out to any other sensor; (ii) ***[If] Policy Condition*** is a Boolean expression based upon the data provided by the local sensor system and static or dynamic data provided by the triggered policy; whenever an event occurs, one or more conditions are checked as true, and if all of them pass, a corresponding action must be executed; (iii) ***[Then] Policy Action*** describes the desired action number (ID) to be executed whenever the IF condition is true; (iv) ***[End] End policy Execution*** indicates the end of the policy execution if the condition is False, otherwise the policy execution will move to the next policy in the chain; (v) ***[Next] Next Policy ID*** contains the key for the next policy in the chain of applicable polices.

Following the **context-based reasoning (CBR)** approach, studies S4, S5, S9, S13, S16, S18, S23 consider as meaningful context any information that affects node's operation. Context information constitutes an important source of data for systems that have to react dynamically to changes in the environment or to new context conditions. For instance, S5 predefines the following context formats and stores them in a context database:

[CONTEXT ID] [INFORMATION TYPE] [INFORMATION ID] [INFORMATION VALUE]

CONTEXT ID is the unique identifier of the context, INFORMATION TYPE describes the source of the sensing data; INFORMATION ID represents the identifier of each specific information such as the sensing capabilities, the residual energy, etc; and INFORMATION VALUE is the value of that specific information in bits. Once obtained the context, it is possible to apply a reasoning process in order to react to dynamic changes of network conditions. On the other hand, S16 uses fuzzy logic and machine learning techniques as a reasoning process for this stage. Finally, S1 applies a hybrid approach relying on policy-based and context-based approach.

The feedback control loop (FCL) approach (S1, S3, S5, S7, S15, S20) is commonly presented in AC systems and most of them use a four-step loop (monitoring, analysis, planning and execution). Generally, WSN middleware systems that apply this approach use hierarchical networks (Figure 11) typically defined with three (or more) levels, namely: sensor nodes level, cluster head level and a base station (or sink node) level. This makes the system able to provide quick adaptation to multiple context parameter changes.

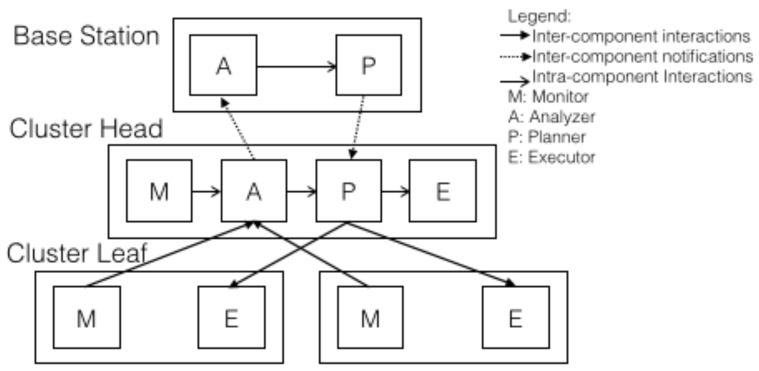


Figure 11. Self-adaptive control loop distributed in three levels of network architecture

Mobile agents (MA) approach (S11, S19, S24) provides a programming model in which applications consist of evolving communities of agents that share a WSN. Agents can dynamically enter and exit a network and can autonomously clone and migrate themselves in response to changes in the environment. Users inject mobile agents that spread across nodes performing application-specific tasks. Each agent is autonomous, allowing multiple applications to share a network. For instance, Agilla (S24) provides a programming model in which applications consist of evolving communities of agents that share a WSN, where coordination among the agents and access to physical resources are supported by a tuple space abstraction. Agents can dynamically enter and exit a network and can autonomously clone and migrate themselves in response to environmental changes.

Model transformation/Code Generation (MT/CG) approaches define an automatic process to derive different middleware configurations depending on the hardware and software of the deployed WSN. This process uses techniques such as model-driven development (MDD) (MDA) and Software Product Line (SPL) (Pohl, Bockle, & Linden, 2005). Model transformation and automatic code generation (S8, S12, S17 and S20) are used to create the concrete system from the model. This approach allows non-experts to develop WSN systems and to provide the needed mechanisms to adapt the network to dynamic environments and unpredictable events. In studies S8 and S17, authors achieve self-adaptation in WSN by proposing a family of middleware for Ambient Intelligence systems, deployed in sensors devices and smartphones. They use feature models and Dynamic Software Product Lines (DSPLs) to drive the self-adaptation process. The reconfiguration consists in replacing the current feature model configuration by a new configuration more suitable to the new contextual situation. Issues highlighted of each primary study are summarized in Table 2.

The first column (Development Approach) shows the approach followed for every study. The Management Approach column shows techniques used to implement the before-mentioned development approaches. These techniques can be centralized, distributed or hybrid. In a centralized approach, the control of the WSN management is centrally located in the sink node. Therefore, this approach allows controlling the network as a whole, in a high level of abstraction. However, in order to found out if an adaptation is required, all nodes need to communicate with the single sink node. A drawback of this approach appears when the WSN has a large scale, with a sheer number of spatially dispersed nodes; thus the number of communications towards the sink node increases and the network is more prone to connectivity failures (caused by bottlenecks in the nodes near to sink node). In a distributed approach, the control of management is fully distributed among the WSN nodes, thus, the before-mentioned centralized approach drawback is mitigated. The sensor nodes apply coordination functions to manage themselves.

Table 2. Summarization of primary studies relevant data

Ref	Develop. Approach	Management Approach	Type of Solution	Addressed Self-properties	Adaptation Time	Topology of WSN	Evaluation
S1	PBR and FCL	Hybrid	Framework	self-configuration	runtime	Flat/ Hierarchic	TinyOS/ TelosB
S2	PBR	Hybrid	Framework	self-configuration self-optimization	Design/ runtime	Hierarchic	Simulated in Contiki OS
S3	FCL	Hybrid	Middleware	self-optimization	runtime	Flat/ Hierarchic	Simulated in Avrora
S4	CBR	Hybrid	Middleware	self-configuration self-optimization	Design	Flat/ Hierarchic	No
S5	PBR and FCL	Hybrid	Framework	self-configuration self-optimization	runtime	Hierarchic	No
S6	PBR	Centralized, Distributed	Middleware Framework	self-configuration	runtime	Flat	Simulated in CORE and EMAN
S7	FCL	Hybrid	Framework	self-configuration self-optimization self-healing	runtime	Hierarchic	No
S8	MT/CG	Hybrid	Middleware	self-configuration	Design	Hierarchic	Simulated on RecosQos
S9	CBR	Distributed	Framework	self-configuration	runtime	Hierarchic	Libelium Wapmotes
S10	PBR	Distributed	Middleware	self-optimization	Design	Hierarchic	No
S11	MA	Distributed	Middleware	self-configuration	runtime	Flat/ Hierarchic	-
S12	MT/CG	Hybrid	Middleware	self-configuration self-optimization	Design runtime	Hierarchic	ContikiOS/ Atmel AVR
S13	CBR	Distributed	Framework	self-configuration self-healing self-optimization self-protection	Design/ runtime	Hierarchic	TinyOS
S14	PBR	Hybrid	Framework	self-configuration	Design/ runtime	Flat/ Hierarchic	Finger/Fing er2
S15	FCL	Distributed	Framework	self-configuration	runtime	Flat/ Hierarchic	TinyOS 2.x, Finger2
S16	CBR	Distributed	Framework	self-configuration	Design/ runtime	Hierarchic	Simulated
S17	MT/CG	Hybrid	Middleware	self-configuration	Design/ runtime	Flat/ Hierarchic	TinyOS 2.1.1, TOSSIM
S18	CBR	Distributed	Framework	self-configuration	Design	Flat/ Hierarchic	Simulations
S19	MA	Distributed	Framework	self-configuration	runtime	Not specific	Simulations
S20	MT/CG	Distributed	Middleware	self-configuration	Design/ runtime	Flat/ Hierarchic	No
S21	PBR	Distributed	Middleware , Framework	self-configuration self-healing	runtime	Flat/ Hierarchic	TinyOS
S22	-	Centralized, Distributed	-	self-configuration self-healing	runtime	Flat / Hierarchic	TinyOS
S23	CBR	Distributed	Framework	self-healing self-protection	runtime	Flat/ Hierarchic	Simulated in Matlab
S24	MA	Distributed	Middleware	self-configuration self-healing	Design/ runtime	Not specific	TinyOS Mica2, Telosb
S25	-	Centralized, Distributed	.	self-healing	runtime	Flat Hierarchic	-

Most of the selected studies use a hybrid approach, where parts of the management functions are performed in the sink nodes and parts are distributed among the sensor nodes. Therefore, the hybrid approach is able to accomplish management functions by both considering the network as a whole, and distributing adaptation responsibilities throughout the network in order to mitigate scalability issues. We also noticed that selected studies implement their proposals at the middleware level (see Type of Solution column). Middleware frameworks reduce the time and effort in developing WSN applications, by providing an easy way to integrate complex and distributed autonomic services, common programming abstractions, and hiding low-level programming details of different sensor platforms. Thus, developers can devote more time in developing the WSN application requirements, and to (re)use the autonomic management components of the middleware to configure/tailor the WSN self-adaptive behavior. In the column named Addressed Self-properties we noted that self-configuration, in autonomic WSN middleware systems, is a general-purpose property. However, implementations of self-healing and self-protection are more common in specific WSN applications.

The Adaptation Time column specifies when the adaptation process occurs. Most of the selected studies execute an adaptation plan at runtime. The MT/CG approach defines the adaptation plan at design time, before the creation of the concrete system. Regarding the WSN topology used in selected studies (Topology of WSN column), we noticed that hierarchic topologies are most used in hybrid/distributed approaches. Normally, hierarchical topologies favor scalability. They organize the nodes into clusters where some nodes work as cluster heads and collect the data from other (ordinary) nodes in the clusters. Then, the heads can consolidate the data and send it to the sink node/base station as a single packet, reducing the overhead from data packet headers and sometimes performing some data aggregation procedure (that can also improve data accuracy). According to Ding et al. (Ding, Holliday, & Celik, Jun/Jul, 2005), clustering reduces useful energy consumption by improving bandwidth utilization, reducing collisions caused by contention for the channel; and reduces wasteful energy consumption by reducing communication overhead.

Finally, the last column of Table 2 shows the Evaluation process used for validating the proposals of selected studies. Most of the selected studies implement their tests using TinyOS-based programs.

After performing the SLR, we noticed that most works are still in an initial stage of integration of AC principles into WSN systems. The goal of optimizing network resources by using these principles is a relatively new research topic in the WSN field. (Picco, 2010) considers that the lack of mature methodologies, software engineering techniques, and abstractions that improve the development process of software for WSNs may come from the fact that the Software Engineering community perceives WSNs as too “low-level”. Or, the fact that WSN software spans the entire network stack and reaches into the physical layer may constitute a steep learning curve for software engineering researchers. (Picco, 2010) also mentions that, unfortunately, the WSN and software engineering research communities have been mostly impermeable to each other. However, they believe that the contribution of SE concepts to the WSN field is inevitable (Picco, 2010), since WSNs are a key element of the grand vision of a physical world augmented by a myriad of computing devices (such as, Internet of Things, Cyber-Physical Systems and autonomic computing).

Dealing with AC in middleware systems for WSN is more complex when compared to their counterparts in traditional middleware systems mainly due to the need of handling limited computing resources of sensor nodes. However, we believe that the selected studies are useful means for addressing autonomic behavior in WSN applications. In the following, we present some challenges and research opportunities that we have identified from the analyzed studies. Figure 12 shows which development approaches are used to achieve each AC property.

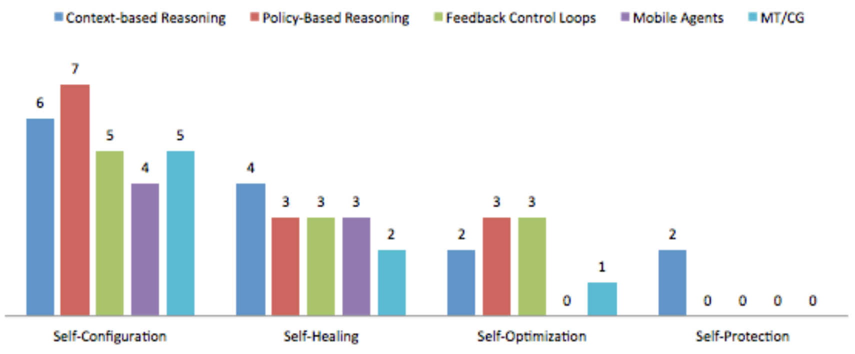


Figure 12. Correlation between the use of AC properties in development approaches

As we have mentioned, one of the investigated development approaches of AC used to provide autonomic behavior in WSN is Context-based reasoning. In this approach, the WSN adaptation is limited once it is a simple reaction to the current context of the network. The middleware system receives the network context as input and responds to it following a logic that selects the most appropriate action. The most important activity of this approach is to sense what is happening in the network and whenever the WSN receives a stimulus, it must react accordingly. The reactivity feature of context-based middleware systems derives from the fact that they just are able to perceive the environment. So, these systems can react to events that occur in the environment in order to satisfy their design objectives.

The Policy-based reasoning approach is widely used in goal-oriented WSN applications. In this approach, the choice of a specific action depends not only on the context of the network but also on how close to the goal each action will bring the system. Systems applying this approach do everything possible in order to achieve the goal of the WSN application. The conventional policy-based systems are generally too heavy to execute in a sensor node. Due to these limitations (memory and CPU constraints), devices in WSN can only store a limited number of policies in their memory and must recycle them when required. This process of loading/unloading policies might create a communication overhead that needs to be handled. Policy-based middleware systems are able to take initiatives towards the satisfaction of specific internal design objectives.

In the Mobile Agents approach, a WSN is seen as a platform that software agents can use in order to perform sensing and/or computing tasks. Agents are highly autonomous and can make adaptation decisions locally based on the changes of the environment. Such decisions take form when the agent decides to migrate or clone itself to neighbor nodes. This approach enables the creation of self-adaptive, self-organized and autonomous applications. As network nodes are directly exposed to the environment, agents can quickly detect changes and determine when adaptation is necessary. Therefore, autonomous and localized agents react faster and transmit less data than centralized adaptation approaches. This makes them suitable for applications in which local decisions significantly reduce the amount of data wirelessly transmitted. The paramount example is tracking of an object (person, fire, wildlife) as it passes through an area monitored by a WSN. The agents responsible for tracking can migrate along the WSN nodes as the object passes through them, thus avoiding wasting resources on nodes that are far from the object. On the other hand, the Mobile Agents approach is not meant for data collection applications that require deployment across the entire WSN. On such cases, this approach introduces an unnecessary overhead of agents switching, which requires wireless data transmission and dynamic-runtime memory allocation. One interesting and rather unexplored application of Mobile Agents in WSN is modeling the behaviors of swarm individuals. Such can be accomplished by rewarding or penalizing the agent's approaches and movements. The effect can then be observed as a whole across the WSN and can turn into a powerful way to study the emergence of swarm behavior.

Model Transformation and Code Generation approaches are traditionally used in Software Product Lines (SPL) in order to create static software systems from a set of software assets. However, these techniques can also be employed to create dynamic software able to perform self-configuration at runtime, such as described in (Gamez et al., 2011). These models can also be used at runtime to drive the reconfiguration of the middleware for failure recovery and/or self-configuration, this is known as a models@runtime approach.

Using models@runtime has the advantage of keeping the application within a known state described by a model, even when it mutates to cope with changes in the environment. This eases the burden of keeping track of architectural configurations in applications that are scattered across several nodes. Furthermore, by doing the self-configuration in a middleware layer common to all nodes, this approach allows a higher degree of adaptability than others like the Mobile Agents approach. For instance, it is possible to change the routing protocol in runtime. In S8, authors configure the network with some energy efficient protocols (TYMO, TinyHop and AODV) and, when the use of the road is drastically reduced and the information collected by its sensors is not as critical as before, they select the most energy efficient routing protocol (previously determined). Additionally, if they choose a protocol which is not preinstalled in several nodes, they produce additional reconfiguration costs in terms of energy expense by sending large size messages that contain the protocol functionality. However, architectural configurations that must take place in every node to ensure compatibility, like changing the routing protocol, require sinks or cluster heads to coordinate adaptation across the multiple nodes. This reduces the locality of the adaptation and incurs in high communication overheads. Also, changing the model on a node is a rare but computing intensive task. The node has to forge a plan detailing the configuration steps to take and then must use a domain modeler to check the correctness of the plan, i.e. check if it arrives at a valid state defined in the FM.

In the before-mentioned approaches, adaptation is performed with the explicit or implicit presence of feedback loops. WSN applications with explicit feedback loops define a part of the system that deals with feedback. This part of the system is able to interacting, communicating and coordinating among middleware components. The majority of the selected studies that applies a feedback control loop approach uses the MAPE-K model proposed by IBM (IBM, 2005).

Figure 13 depicts a summary of the main features of autonomic wireless sensor networks, gathered from selected studies retrieved in the performed SRL. These features are organized in terms of: (i) development approaches used to provide autonomic behavior in WSN, (ii) requirements of autonomic WSN, (iii) self-adaptable/self-manageable features of WSN, according to AC principles, and (iv) techniques used to manage and implement the development approaches.

Besides the development approaches described in this Chapter to tackle autonomy in WSN (Context-based reasoning, Policy-based reasoning, Feedback control loops, Mobiles Agents and Model transformation/code generation), we noticed that self-* properties were applied as follows. Regarding self-configuration, selected studies proposed solutions that are able for autonomously managing the network state, application requirements, node resources, communication protocols and quality of service; for self-healing, solutions for failure node detection and filtering sensing data are proposed; reduction of a number of data transmission, and processing of sensing data are some of methods used to address self-optimization. Self-protection was a self-* property poorly explored by the selected studies: only two (2) selected studies refer to this property, where cryptography was presented as a solution for this issue.

In order to extend and implement the aforementioned development approaches, system architectures, sitting between the sensing applications and the node operating system, are expected to provide a set of integrated functions for nodes to be self-manageable and self-configurable. Nevertheless, there is still no comprehensive system architecture design that supports these expectations.

Summarizing, most of the works that we have found in the literature that aim to provide autonomic behavior in WSNs are just in preliminary stages and they have still some open challenges. However, their proposals seem very adequate to tackle some aspects of the autonomic WSNs.

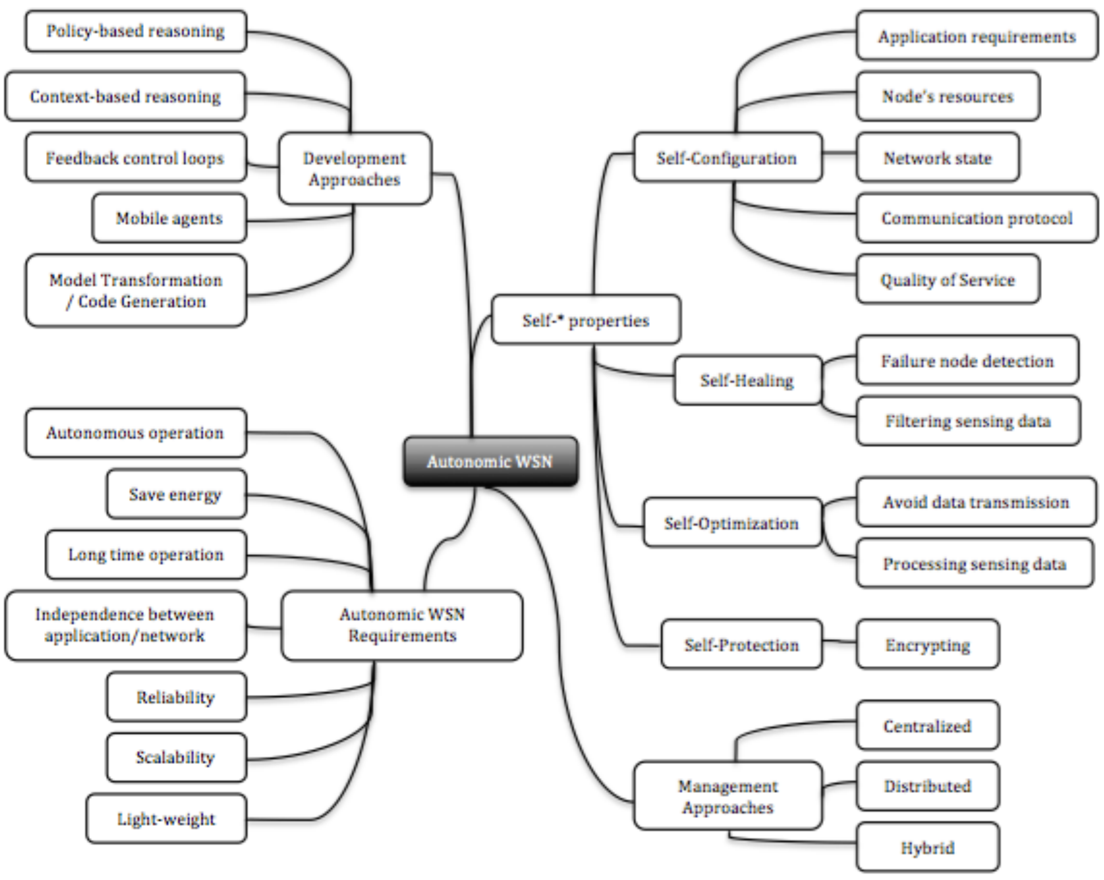


Figure 13. Autonomic Wireless Sensor Network characteristics

3.3.3 Activity C – Step RA-1: Reference Architectures for WSN and AS

In order to perform activity C of Step RA-1 of ProSA-RA, this section presents some RAs specific for WSNs and for Autonomic Systems (AS). In this activity we have conducted two SLRs focusing specifically in RAs for WSNs and RAs for autonomic systems. Section 3.3.3.1 presents a SLR conducted to find out the main RA for WSN. Section 3.3.3.2 presents a SLR conducted to find out the main RA for autonomic systems.

3.3.3.1 Reference Architectures for Wireless Sensor Networks

This activity aims to find out which are the main important RAs for WSN. In this perspective a SLR was conducted to retrieve all the literature relevant to answer the following research questions.

RQ1: Are there Reference Architectures (RA) or Reference Models (RM) specific for WSN?

RQ2: How such RA/RM for WSN have been represented?

These questions are motivated by the need for insights about the research trends in RA for WSN. The research strategy adopted in this SLR was the automatic search in research databases, as we performed in (Portocarrero et al., 2014). The general form of the search string is shown below:

Query strings:

TITLE-ABS-KEY
(("reference architecture" **OR** "reference architectures" OR "reference model" **OR** "reference models"**)** **AND (**"wireless sensor network" **OR** "sensor network" **OR** wsn OR wsan **OR** "wireless sensor and actuator network" **OR** "wireless ad-hoc network" **OR** "wireless actuator network"**))**

The primary studies were searched in all publication databases and 101 studies were obtained. After the reading of titles, abstracts, and, when necessary, the introduction of each study, 16 primary studies were selected to be read in full. After application of the inclusion and exclusion criteria, 9 primary studies were selected as included. The inclusion criteria used in this SLR was corroborating if the primary study provides evidence on RAs for WSN. The exclusion criteria were discarding primary studies with no evidence on RAs or RMs for WSN, primary study does not provide a reasonable amount of information, it is written in a language other than English and it is a previous work developed by the same author. Besides, one study suggested by a specialist was also evaluated (SNRA) (-. ISO/IEC, 2014). As a result, 10 primary studies were selected as the most relevant ones to our SLR. Table 3 shows the complete reference of the 10 primary studies of this SLR.

Table 3. Selected primary studies of RA for WSN

ID	*Title*	*Name of RA/RM*	*Year*	*Ref*
S1	A Reference Architecture for Sensor Networks Integration and Management	SeNsIM	2009	(Casola, Gaglione, & Mazzeo, July, 2009)
S2	Technical Document of ISO/IEC JTC 1 Study Group on Sensor Networks (SGSN) Study on Sensor Networks (Version 3)	SNRA	2009	(-. ISO/IEC, 2014)
S3	Reference Model for Sensor Networks in B3G Mobile Communication Systems	e-SENSE	2006	(Gluhak & al., June, 2006)
S4	A survey on developing publish/subscribe middleware over wireless sensor/actuator networks	Pub/Sub Mw	2015	(Sheltami, Al-Roubaiey, & Mahmoud, 2015)
S5	Knowledge-based environmental research infrastructure: moving beyond data	ENVRI-RM	2015	(Stocker, Rönkkö, & Kolehmainen, 2015)
S6	Integrated Technical Reference Model and Sensor Network Architecture	I-TRM	2008	(Joshi & Michel, 2008)
S7	Reference Architectures and Management Model for Ad hoc Sensor Networks	BSNF	2004	(Serri, 2004)
S8	On Complex Event Processing for Sensor Networks	EDA	2009	(Dunkel, 2009)
S9	Towards a Multiagent-based Software Architecture for Sensor Networks	iEPA	2011	(Dunkel, 2011)
S10	The ANGEL WSN Architecture	ANGEL	2007	(Willig, Hauer, Karowski, Baldus, & Huebner, 2007)

In order to answer how the proposed RAs were designed/represented, we associate each of them with four design techniques, adapted from (F. Affonso, Scannavino, Oliveira, & Nakagawa, 2014):

- **Block Diagram:** a system diagram whose main parts or functions are represented by blocks connected by lines that show the relationships among them;

- **Informal Notation:** a type of notation that follows no formal or semi-formal method or methodology;

- **Layer Diagrams:** a high-level representation for the organization of physical artifacts. These diagrams describe the major tasks that can be performed by artifacts or components.

- **UML Diagrams:** a graphical language for visualization, specification, construction, and documentation of object-oriented software-intensive system's artifacts.

The relationship between the design techniques and the studies are shown in Table 4.

Table 4. Decision Technique used in selected primary studies of RA/RM for WSN

Design Technique	***RA/RM***	***Primary Studies***
Block Diagram	Reference Architecture Reference Model	S1, S2, S7, S8, S9 S3, S4, S6
Informal Notation	Reference Architecture Reference Model	S10 S5
Layer Diagrams	Reference Architecture Reference Model	S1, S2, S7, S8, S9 S3, S6
UML Diagrams	Reference Architecture	S1, S8

These relations aim at guiding the choice of techniques used in the design of RAMSES. Most of RA/RMs reviewed in this Section has been designed using ad hoc approaches and represented using informal techniques. These RA/RMs for WSN are predominantly described using block definition diagrams and they have a shortage of well-defined interfaces. In addition, most of RA/RMs applies a combination of design techniques in order to facilitate the modeling activity. For instance, the combination of block diagrams and layer diagrams are used by the 70% of the primary studies, it must be highlighted because they enable the creation of modular relations among the components. However, we note a lack of formal methods of architectural representation. Moreover, there is a lack of an explicit definition of self-adaptive management of the network and an architectural representation of the dynamic behavior of components, a key issue in WSN applications.

Another finding of this SLR is related to the architectural design decisions (architectural patterns/styles) applied to build the proposed RA/RM. As showed in Table 5, the following architectural design decisions have been used in primary studies.

- Layered based architecture: reflect a division of the software into layers that represent a grouping of modules that offers a cohesive set of services.

- Event-based architecture: allows asynchronous communication among software components by generating and receiving notification of events occurred. Events are managed by an event service that implements a multicasting mechanism that decouples event generators from event receivers.

- Agent-based architecture: consist of three major components: (i) a platform manager, responsible for managing the agents; (ii) an advertisement registry, which contains descriptions of the agents in the system and facilitates discovery of those agents; (iii) and a set of agents that can communicate with each other, with the platform manager and with the advertisement registry.

- Publish/subscribe pattern: is a messaging pattern where senders of messages, called publishers, do not program the messages to be sent directly to specific receivers, called subscribers, but instead characterize published messages into classes without knowledge of which subscribers, if any, there may be. Similarly, subscribers express interest in one or more classes and only receive messages that are of interest, without knowledge of which publishers, if any, there are.

- Mediator Pattern: defines an object that encapsulates how a set of objects interacts. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

- Broker pattern: separates the communication functionality of a distributed system from its application functionality. It hides and mediates all communication between the objects or components of a system.

Table 5. Architectural design decision applied in primary studies of RA/RM for WSN

Architectural design decision	*Primary Studies*
Layered based architecture	S1, S2, S3, S4, S6, S7, S8, S9
Event-based architecture	S8, S9
Agent-based architecture	S9
Publish/Subscribe pattern	S4
Mediator Pattern	S1
Broker Pattern	S1, S3
No Reference	S5, S10

Based on results showed on Table 5, we noticed that 80% of studies (S1, S2, S3, S4, S6, S7, S8, S9) reported the applying of a layer-based architecture. The main benefits of this architectural decision are providing abstraction, isolation, manageability, performance, reusability and testability. Nevertheless, we noticed a lack of well-defined architecture design that supports the autonomy of sensor networking. We believe the study of RAs specific for autonomic systems may aid to support this gap. Thus, in next section we present a SLR to retrieve studies that investigate this subject.

3.3.3.2 Reference Architectures for Autonomic Systems

This activity aims to found out which are the most important RAs for autonomic systems (AS). In this perspective, a SLR was conducted to retrieve all the literature relevant to answer the following research questions.

RQ1: Are there Reference Architectures (RA) or Reference Models (RM) for AS?

RQ2: How such RA/RM for AS have been represented?

These questions were motivated by the need for insights about the research trends in RA for AS. The research strategy adopted in this SLR was the automatic search in research databases, as we describe in (Portocarrero et al., 2014). The general form of the defined search string is shown below:

Query strings:

TITLE-ABS-KEY

(("reference architecture" **OR** "reference architectures" **OR** "reference model" **OR** "reference models" **) AND (** autonomic **OR** autonomous **OR** "self-adaptive" **OR** "self-management" **OR**" self-managed" **OR** "self-adapt" **OR** "self-adapted" **OR** "self-adaptation"**))**

Before starting this SLR, one study suggested by an expert in reference architectures was evaluated (F. Affonso et al., 2014). Such study was entitled "Reference Architectures for Self-Managed Software Systems: a Systematic Literature Review" and it presents a detailed state-of-art on reference architectures for self-adaptive systems obtained from a systematic literature review. The authors retrieved 22 primary studies that represent the most relevant RA/RMs for AS until 2014. Thus, we complement this SLR by using our query string to search in all publication databases the most important RA/RMs for AS published since 2014. As a result, 33 studies were obtained. As a result of the conduction phase, 3 primary studies were selected as the most relevant ones to our SLR since 2014. Table 6 shows the complete reference of the 22 primary studies retrieved by (F. Affonso et al., 2014) and the three studies retrieved by us. In the ID column, this symbol (*) indicates that we added the study.

Table 6. Selected primary studies of RA/RM for AS, adapted of (F. Affonso et al., 2014)

ID	*Title*	*Year*	*Ref*
S1	Reflecting on self-adaptive software systems	2009	(Andersson, de Lemos, Malek, & Weyns, 2009)
S2	Mobile agent based elastic executor service: Reference architecture of an executor service using a mobile agent platform to control the elasticity of the system	2012	(Bhattacharya, 2012)
S3	A scalable approach to qos-aware self-adaption in service-oriented architectures	2009	(Cardellini, 2009)
S4	Towards a dynamic cloud-enabled service eco-system	2011	(Castejon, 2011)
S5	Domain-specific software architecture for adaptive intelligent systems	1995	(Hayes-Roth, 1995)
S6	An architectural blueprint for autonomic computing	2005	(IBM, 2005)
S7	Automated adaptations to dynamic software architectures by using autonomous agents	2004	(Jiao & Mei, 2004)
S8	Self-managed systems: an architectural challenge	2007	(Kramer & Magee, 2007)
S9	Soadapt: A process reference model for developing adaptable service-based applications	2012	(Lane, 2012)
S10	Dynamic software architectures: formal specification and verification with CSP	2012	(C. e. a. Li, 2012)
S11	A reference architecture for self-organizing service-oriented computing	2008	(L. Liu & et, 2008)
S12	Autonomic computing now you see it, now you don't: Design and evolution of autonomic software systems	2009	(H. Muller & et, 2009)
S13	A reference architecture for integrated development and run-time environment	2012	(Tajalii & Medvidovic, 2012)
S14	Observation and control of organic systems	2011	(Tomforde & al., 2011)
S15	A self-organizing architecture for pervasive ecosystems	2010	(Villalba, 2010)
S16	Nature-inspired spatial metaphors for pervasive service ecosystems	2008	(Villalba, 2008)
S17	DYNAMICO: A Reference Model for Governing Control Objectives and Context Relevance in Self-Adaptive Software Systems	2013	(Villegas, Tamura, Mu¨ller, Duchien, & Casallas, 2012)
S18	Architectural design of a distributed application with autonomic quality requirements	2005	(D. e. a. Weyns, 2005)
S19	Forms: a formal reference model for self-adaptation	2010	(D. Weyns, Malek, & Andersson, 2010a)
S20	On decentralized self-adaptation: lessons from the trenches and challenges for the future	2010	(D. Weyns, Malek, & Andersson, 2010b)
S21	Timing driven architectural adaptation	2006	(Wils, Berbers, Holvoet, & De Vlaminck, 2006)
S22	Adaptive component paradigm for highly configurable business components	2006	(Zewdie & Carlson, 2006)

S23*	A Reference Model as Automated Process for Software Adaptation at Runtime	2015	(F. J. Affonso, Vecchiato Saenz, Rodrigues, Luis, & Nakagawa, 2015)
S24*	Engineering Pervasive Service Ecosystems: The SAPERE Approach	2015	(Castelli, Mamei, Rosi, & Zambonelli, 2015)
S25*	ReMoSSA: Reference Model for Specification of Self-adaptive Service-Oriented-Architecture	2014	(Cherif, Djemaa, & Amous, 2014)

In order to answer how the proposed RA/RMs for AS were designed/represented, (F. Affonso et al., 2014) associate each RA with six existing design techniques: block diagram, informal notation, layer diagrams, UML diagrams, formal methods (such as Z notation) and formal notation for business process (such as BPMN- Business Process Modeling Notation, BPEL- Business Process Execution Language). In addition, in order to find out how the dynamic behavior of AS has been designed, (F. Affonso et al., 2014) identify ten knowledge types:

- **Action plane:** represents an information item that shows a systematic course of action for the achievement of a declared purpose.

- **Agents:** encompass the software agents, intelligent agents, and autonomous agents.

- **Autonomous subsystems:** implement the generic loop (Monitor - Analyze - Plan - Executor) defined by the architecture model of autonomic computing.

- **Computational Reflection:** represents the ability of a program to modify itself at runtime and is very similar to human reflection.

- **Nature-inspired pervasive service ecosystems:** should obtain inspiration from natural systems by enabling modeling and deployment of services as autonomous individuals spatially-situated in a system of other services, data sources, and pervasive devices.

- **Process flow:** represents a sequence of steps for performing of an activity. Decisions can be made in order to characterize the changes in the dynamic behavior of software.

- **Rule base:** represents a rule set that can be used to define the changes to be performed at runtime.

- **Service composition:** represents functionalities that can be coupled at runtime so that the new requirements are met.

- **Subsystems in layers:** represents a subsystem set organized in layers so that one or more activities are performed. They are autonomous units that enable the dynamic behavior of AS; and

- **Supervisor systems:** represents systems responsible for monitoring the operation of another system.

Based on their conducted SLR, (F. Affonso et al., 2014) point out that the surveyed RA/RM can be instantiated to operate in other domains of software systems, which can be considered a well-inclined aspect for the reuse, and most of the RA/RM are designed by one or more design techniques and one or more knowledge types, where the relationship between the knowledge type “autonomous subsystems” and the design technique “block diagram” must be highlighted for facilitating the modeling activity of RA/RM.

The relationship between design techniques and knowledge types, as well as the studies addressed to each relation are shown in Table 7 (this symbol (*) indicates that we added the study).

The RA/RM for autonomic system reviewed in this Section serve as a framework for our RA. These RA/RM were analyzed to acquire relevant knowledge in order to understand how autonomic computing principles are applied in order to manage a system. Therefore, these RA were used as an “inspiration” to introduce an autonomic module based on MAPE framework in our RA.

Table 7. Design technique vs. knowledge type for RA/RM for AS, adapted from (F. Affonso et al., 2014)

Design technique	***Knowledge Type***	***Primary Studies***
Block Diagrams	Agents	S2, S7
	Autonomous subsystems	S11, S12, S14, S17, S18, S25*
	Computational reflection	S1, S23*
	Nature-inspired pervasive service ecosystems	S16, S24*
	Rule base	S4, S7, S12, S13
	Service composition	S3, S17
	Subsystems in layers	S16
	Supervisor systems	S17, S22
Formal Methods	Agents	S7, S19
	Autonomous subsystems	S19
	Computational reflection	S19
	Rule base	S7, S10, S19
	Process Flow	S9
Formal Methods for Business Process	Process Flow	S6, S9
Informal Notation	Autonomous subsystems	S18
	Nature-inspired pervasive service ecosystems	S16, S24*
	Subsystems in layers	S16, S15
Layer Diagram	Action Plan	S5, S8
	Agents	S20
	Autonomous subsystems	S18, S20
	Computational reflection	S1, S4, S5, S8, S13
	Subsystems in layers	S1, S15
	Supervisor systems	S22
UML Diagram	Agents	S19, S20
	Autonomous subsystems	S11, S20, S25*
	Computational reflection	S19, S20, S23*
	Process Flow	S9
	Rule base	S21
	Supervisor systems	S22

3.3.4 Activity D – Step RA-1: Architectural Styles/Patterns

Patterns are increasingly popular techniques for addressing key aspects of software architecture design. Patterns codify reusable design expertise that provides time-proven solutions to commonly occurring software problems that arise in particular contexts and domains. A pattern describes a generic solution for a recurring design problem.

Design patterns provide a scheme for refining the elements of a software system and the relationships between them, and describe a common structure of communicating elements that solves a general design problem within a particular context. Architectural styles express the fundamental, overall structural organization of software systems and provide a set of predefined subsystems, specify their responsibilities, and include guidelines for organizing the relationships between them.

Patterns help enhancing reuse by capturing and reusing the static and dynamic structure and collaboration of key participants in software designs. They are particularly useful for documenting recurring micro-architectures, which are abstractions of software components that experienced developers apply to resolve common design and implementation problems. Patterns also raise the level of discourse in project design and programming activities, which helps improving team productivity and software quality.

Software architectures are almost never limited to a single architectural style; instead they are often a combination of architectural styles that make up the complete system. Aiming to build our RA we have conducted an informal review to complement the architectural patterns/styles found on the two SLR presented in Section 3.3.3. Therefore, we listed the following architectural patterns/styles commonly used in self-adaptive systems, such as Layered architectural style, broker pattern, service component for self-adaptive systems, aspect peer-to-peer style, aggregator-escalator-peer style, decorator pattern, decentralized patterns (hierarchical control pattern and master/slave pattern), and patterns commonly used in WSNs, such as mediator and data gathering:

3.3.4.1 Layered Architectural Style

This style focuses on grouping related functionality within an application into distinct layers that are stacked vertically on top of each other. Each layer provides a set of services to the layer above and uses the services of the layer below. Between two adjacent layers a clearly defined interface is provided. Commonly, Self-adaptive systems implement their architectures based on a three-layer architecture style. Examples of self-adaptive systems using this approach are (Menasce, Gomaa, Malek, & Sousa, Nov/Dec, 2011) and (Kramer & Magee, 2007).

Figure 14 summarizes the Kramer and Magee proposal (Kramer & Magee, 2007) of the three layer model for a self-managed system. The main criteria for placing function in different layers in this architecture style is one of time scale and this would seem to apply equally well to self-managed systems (Kramer & Magee, 2007). Immediate feedback actions are at the lowest level and the longest actions requiring deliberation are at the uppermost level. Kramer and Magee emphasize that they consider this a conceptual architecture or RA which identifies the necessary functionality for self-management. We will use it in the next section to organize the research challenges presented by self-management in WSN middleware systems, and to define the basic structure of our RA.

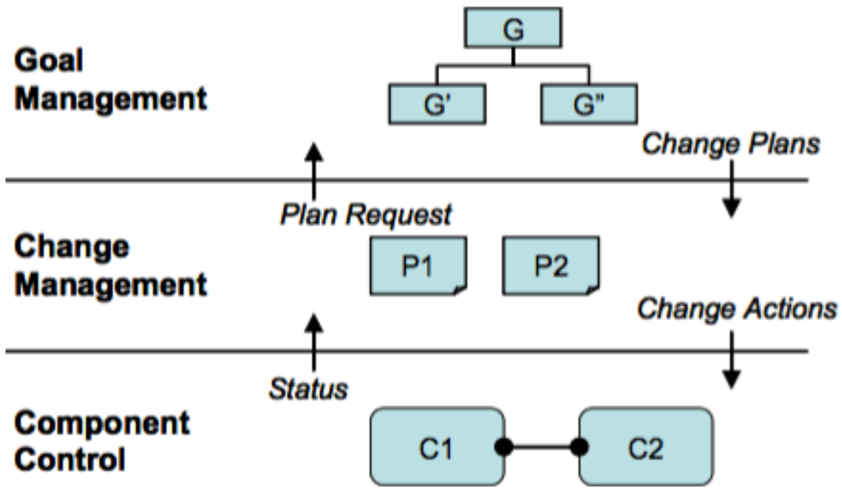


Figure 14. Three Layer Architecture Model for Self-Management (Kramer & Magee, 2007)

3.3.4.2 Broker Pattern

One major challenge in distributed software systems is communication and integration of heterogeneous components into coherent applications, as well as the efficient use of networking resources. A Broker (see Figure 15) separates the communication functionality of a distributed system from its application functionality. The Broker hides and mediates all communication between the objects or components of a system.

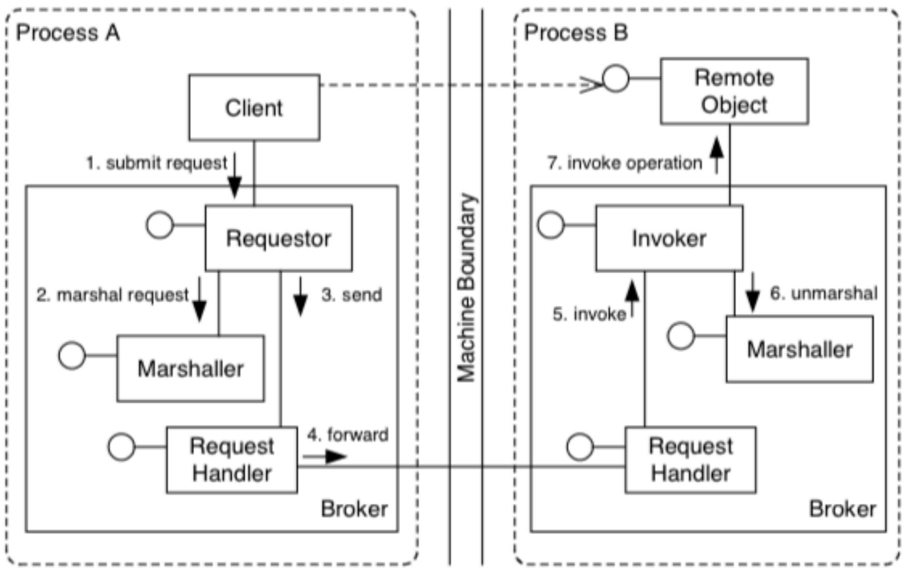


Figure 15. Broker Pattern (Avgeriou & Zdun, 2005)

3.3.4.3 Service Components for self-adaptive systems

Component-based systems and Service-oriented systems are architectural styles commonly applied together by self-adaptive systems. Component architectural style focuses on the decomposition of the design into individual functional or logical components that expose well-defined communication interfaces containing methods, events, and properties. The main benefits of this approach are: ease of deployment, reduced cost, component reusability and mitigation of technical complexity. Service-oriented architectural style, in turn, enables application functionality to be provided as a set of services. Thus, in (Puviani, Cabri, & Zambonelli, 2013) a pattern is proposed that describes the structure of service components for using in self-adaptive systems. This pattern proposes the use of a component with well-defined interfaces specific for self-adaptive systems.

The interfaces are (see Figure 16): Input, used to receive information; Output, used to send information; Sensor, that makes the component able for achieving information from the external; Effector that makes the component be able for managing the external; Emitter, used to emit status information to an external manager. Thus, the structure of service components provides a base for instantiating a concrete system from our RA.

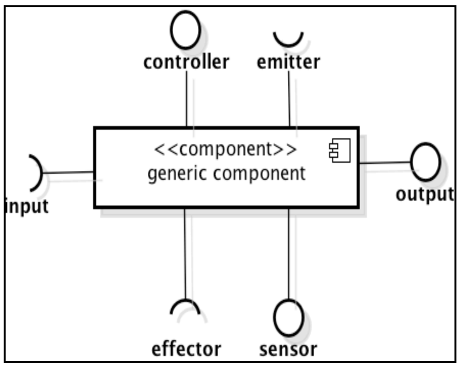


Figure 16. Structure of service components (Puviani et al., 2013)

3.3.4.4 Aspect peer-to-peer Architectural Style:

In this architectural style, a monitor component observes each aspect of the system or its environment and a peer configurator component to reconfigure the system. This style (Figure 17) (Neti & Muller, May, 2007) can also be viewed as a set of autonomic elements with one autonomic element for each component of the system. Each autonomic element in the set is independent and complete on its own.

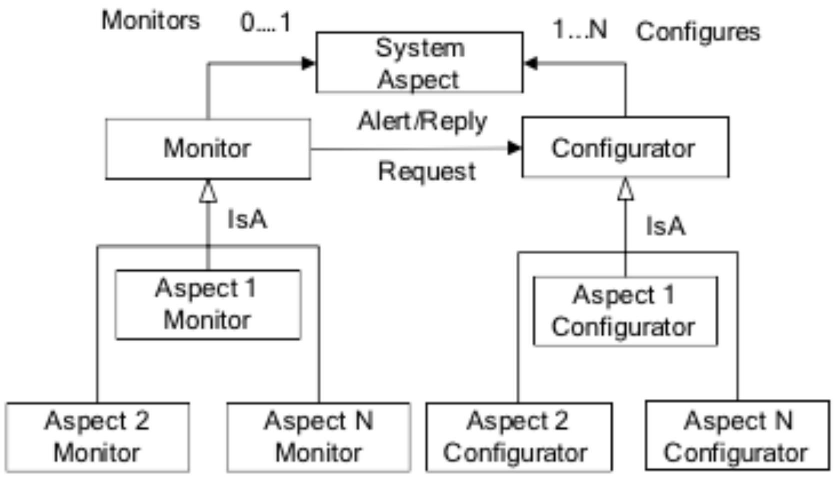


Figure 17. Aspect peer-to-peer Architectural Style (Neti & Muller, May, 2007)

3.3.4.5 Aggregator-escalator-peer Architectural Style:

This architectural style (Neti & Muller, May, 2007) allows to monitors to pass their outputs to higher-level aggregator monitors. A higher-level configurator can then make better configuration decisions. This style (Figure 18) can also be viewed as a set of autonomic elements where one autonomic element exists for each component of the system. In this architectural style, higher-level elements need information from lower level elements to function properly.

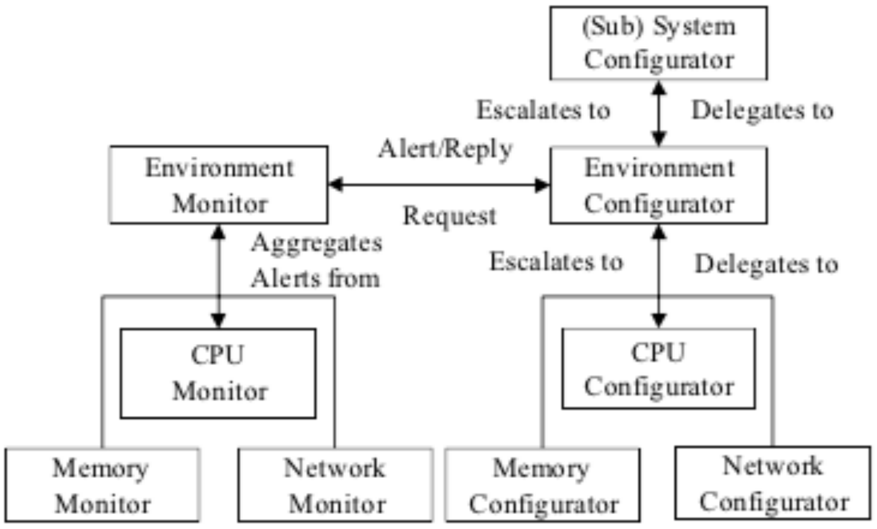


Figure 18. Aggregator-escalator-peer Architectural Style (Neti & Muller, May, 2007)

3.3.4.6 Decorator Pattern.

The design pattern Decorator allows behavior to be added to an individual object, either statically or dynamically, without affecting the behavior of other objects from the same class. This pattern, depicted in Figure 19, is based on three types of entities: (i) Interfaces that define services provided by components, (ii) Abstract classes with the definition of basic methods, services and references to other components, and (iii) Implementation classes that define the specific required behavior.

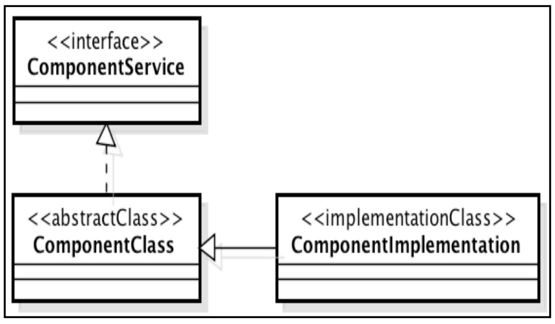


Figure 19. General implementation of components

3.3.4.7 Decentralized Patterns for Self-Adaptation in WSN

In many cases, centralizing control for self-adaptation is simply not feasible. Among the possible reasons for this we can mention (Lemos, 2013): (i) an inherent distribution of information in the system makes it too costly or even infeasible to collect all the data required for adaptation actions; (ii) due to the scale of the system the cost to process all the information at one place may be too high; and (iii) the system spans multiple ownership domains with no trustworthy authority to control adaptations. Thus, when systems are large, complex, and heterogeneous as it is typical in WSN applications, a simple MAPE loop may not be sufficient for managing all adaptations in a system, so multiple MAPE loops may be introduced. However, there is a dearth of practical and effective techniques to build systems using decentralized control of self-adaptive software (Lemos, 2013).

Decentralization of control may be the only option in cases where no single entity has the knowledge or authority to coordinate adaptation across a set of managed subsystems. Therefore, we now present two MAPE patterns that model different types of interacting MAPE loops with different degrees of decentralization:

a) Hierarchical control pattern. This pattern provides a layered separation of concerns to manage the complexity of self-adaptation. It organizes the adaptation logic as a hierarchy of MAPE loops. Different layers typically focus on different concerns at different levels of abstraction, and may operate at different time scales. Loops at lower layers operate at a short time scale, guaranteeing timely adaptation concerning the part of the system under their direct control. Higher level operates at a longer time scale with a more global vision.

The architecture of the managed system will likely influence which patterns are applicable. A hierarchical pattern will be unlikely to work if there is no obvious hierarchy of authority in the managed system (Lemos, 2013). This pattern, depicted in Figure 20, is applied when control loops need to interact and coordinate actions to avoid conflicts and provide certain guarantees about adaptation.

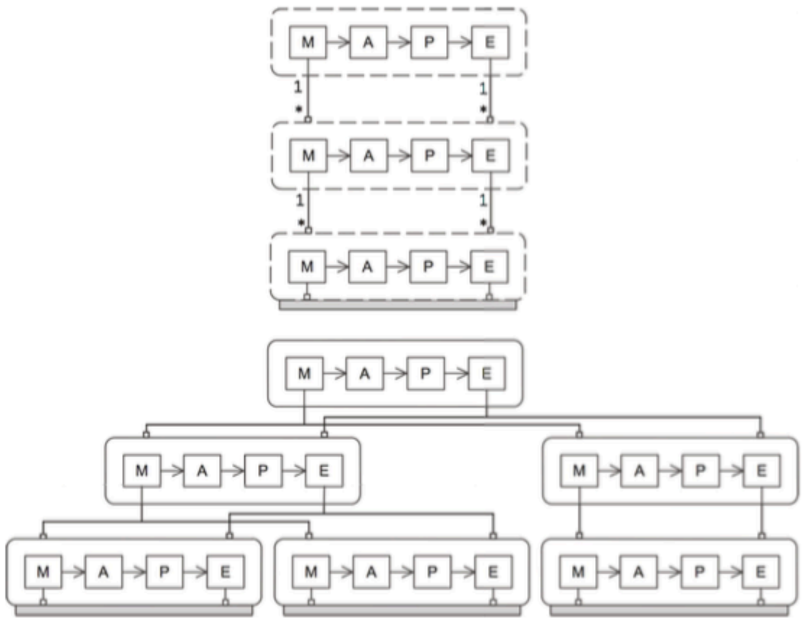


Figure 20. Top: Hierarchical control pattern. Bottom: concrete instance of the pattern. Source: (Lemos, 2013)

The hierarchical control pattern enables adaptation logic to be structured so that the complexity of self-adaptation can be managed. In the context of WSN, the adoption of a hierarchical structure allows bottom layer control loops, deployed inside the network, focus on concrete adaptation objectives (sensor nodes and clusters), while higher level control loops, deployed outside the network, take increasingly broader perspectives by considering an adaptation plan for the entire network. This corresponds to the layered organization of self-adaptation as proposed in Section 3.3.4.1.

b) Master/Slave pattern. This pattern organizes the adaptation logic by creating a hierarchical relationship between one (centralized) master component, that is responsible for the analysis and planning of adaptations, and multiple slave components which are responsible for monitoring and execution. This pattern is applied when there is a need to adapt a distributed software system. This pattern consists of two abstract groups of MAPE components. There is a single instance of the group with a Planer (P) and an Analyzer (A) component, and there can be an arbitrary number of instances of the group with a Monitor (M) and an Executor (E) component. As such, the pattern supports the typical flow of component interactions of a MAPE loop, but with multiple instances of M and E component as depicted in Figure 21.

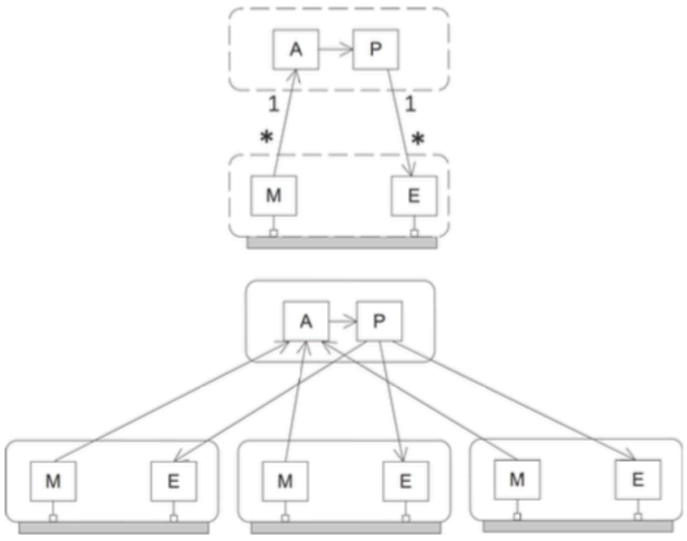


Figure 21. Top: master/slave pattern. Bottom: concrete instance of the pattern. Source: (Lemos, 2013)

The master/slave pattern is a suitable solution for WSN scenarios hierarchically organized in clusters in which slave control components, implemented by ordinary nodes, need to process monitored information to derive the required data allowing centralized decision making on cluster heads, and execute local adaptation on each cluster. Centralizing the A and P components facilitates the implementation of efficient algorithms for analysis and planning aimed at achieving global objectives of each cluster.

3.3.4.8 Mediator Pattern

This pattern defines that an object shall encapsulate the interaction mechanism of other objects. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently (Amin & Hong, 2005). This pattern designs an intercessor to decouple many peers.

An architecture with numerous inter-connections make it more difficult for a component to work without the support of others. In addition to that, making significant changes to the overall behavior of the system becomes unnecessarily difficult, since behavior is distributed among many modules. These problems can be avoided by encapsulating collective behavior in a separate mediator module.

Mediator pattern, depicted in Figure 20, contains two participants: (i) Mediator, an intermediary node/component or module which provides an interface for communicating with other nodes/components or modules; and (ii) Colleagues, that communicates with its mediator whenever it would have otherwise communicated with another colleague.

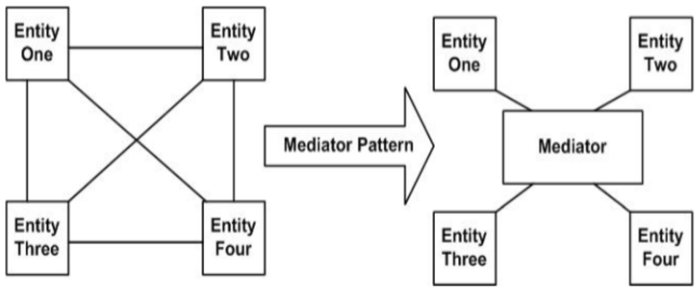


Figure 22. Mediator pattern. Source (Amin & Hong, 2005)

This pattern may be applicable in the following sensor network scenario:

- When a set of modules communicate in well-defined but complex ways.

- When it is difficult to reuse a module because it refers to and communicates with many other modules.

Mediator pattern allows system managers to vary and reuse Colleague and Mediator independently. It also simplifies the maintenance of the system and any new functionality can be added at mediator without affecting colleagues. Moreover, mediator pattern can simplify the communication protocol by replacing many-to-many relationship to one-to-many relationship as it is easy to inspect one-to-many relationship. A drawback of the mediator pattern is that without proper design the mediator itself can become overly complex.

3.3.4.9 Data Gathering Pattern

This pattern is applicable when sensor nodes store routing tables and execute routing protocols in order to route data from one node to another. In (Cardei, Fernandez, Sahu, & Cardei, 2011) this pattern is detailed considering a hierarchical WSN composed by sensor nodes, cluster head, super nodes and sink.

In this topology, all sensor nodes in a cluster are within direct communication range with the cluster head. Sensor nodes collect data and forward the data to the nearest cluster head. Cluster heads aggregate the data received from sensor nodes and forward them to the nearest super node and super nodes forward the data to the sink. Finally, sink nodes forwards the data to the user or the user requests data from the sink. Data gathering is depicted in Figure 21.

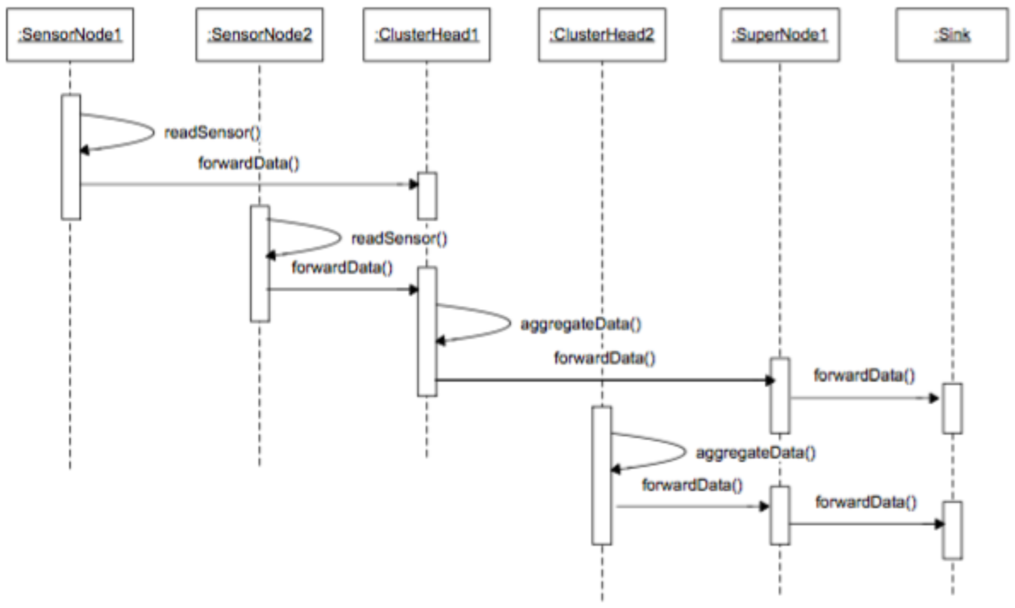


Figure 23. Data Gathering Pattern (Cardei, Fernandez, Sahu, & Cardei, 2011)

3.4 Step RA-2: Architectural Analysis

Based on the resulting information of the previous phase, this step of ProSA-RA aims to establish the requirements of the RA. For this, the following activities are undertaken:

A. Establishment of architectural requirements for the RA

B. Establishment of quality attributes for the RA.

To conduct activities A and B, we consider the outcome of the performed systematic literature reviews, presented in Sections 3.3.1 and 3.3.2. Regarding activity A, Section 3.4.1 presents the requirements for self-adaptation in WSN identified in the SLR. Regarding activity B, Section 3.4.2 presents the quality attributes identified for the RA.

3.4.1 Activity A – Step RA-2: Requirements for self-adaptation in WSN

Considering the information detailed in the Activity A of previous step of ProSA-RA, the following self-adaptive WSN system concerns were identified (see section 3.3.1):

I. Adapt the network to dynamic environments (for instance, node energy depletion and arrival of new applications) and unpredictable events (for instance, node and link failures);

II. Save energy, extending the network lifetime;

III. Provide network scalability;

IV. Provide reliability of sensing data and;

V. Provide independence between the management functions of applications and network configuration.

Knowing that the addressing of these concerns allows self-adaptation in WSN systems and based on the taxonomy presented in Figure 3, where the main characteristics of autonomic WSN were identified and their most important requirements were highlighted, we established a set of requirements for self-adaptation in WSN addressed by our RA. These requirements are described below.

- **Req-A**: A WSN should contain one or more sink nodes or base stations, which must be endowed with a wireless communication interface and one interface with a Gateway, in order to integrate the WSN with external systems and other networks. In this perspective, the sensor network consists of the managed resource according to the autonomic computing vision. The composition of these elements (Sink nodes and Gateway) in the RA contributes to deal with balancing the network traffic;
- **Req-B**: The RA must enable the definition of a set of sensing-based applications, responsible for defining sensing tasks and quality attributes to be attended by the system. The specification of concrete and independent components to handle WSN applications and the MAPE process contribute to guarantee independence between the management functions of applications and network configuration;
- **Req-C**: The RA must enable the definition of a set of high-level goals. For instance, a goal can specify that the response time of a simple environmental monitoring WSN application should be under 20 seconds, while that of a real-time WSN application needs to be less than 5 second. These goals are composed of execution/adaptation policies that will guide decisions and

strategies to be used for configuring and adapting action plans in the network. These high-level goals include both point of views: the applications' and the network, and they aim to optimize the use of resources by keeping the system performance and the separation of system concerns;

- **Req-D**: The RA must enable the self-management of the network by defining a set of software components (middleware platform) responsible for managing the WSN. Such components must be responsible for implementing the feedback loops (MAPE-K) and, therefore, they must include functionalities for context management (for enabling the MAPE process), goal management (to guarantee WSN application requirements) and adaptation management (to adapt the network behavior whenever it is needed, mainly in order to extending the network lifetime);
- **Req-E**: The RA must consider a hierarchical topology for the WSN organization. Hierarchical WSNs organize the nodes into clusters. In a clustered network, the communication is divided into intra and inter cluster. The intra-cluster communication is from cluster nodes to their respective cluster head. The inter-cluster communication is from the cluster heads to the sink nodes. The composition of clusters in the RA contributes to deal with scalability. Scalability in this context implies the need for load balancing, efficient resource utilization, and data aggregation.

Table 8 shows the mapping between the aforementioned self-adaptive system concerns and the RAMSES requirements.

Table 8. Mapping between self-adaptive system concerns for WSN and RA requirements

Concern	**Req-A**	**Req-B**	**Req-C**	**Req-D**	**Req-E**
I				X	
II				X	
III	X				X
IV				X	
V		X	X		

3.4.2 Activity B – Step RA-2: Self-Adaptive WSN System Quality Attributes

This section aims to attend the goals of activity B of Step RA-2 recommended by ProSA-RA. This activity intends to establish a set of quality attributes for the RA. Quality Attributes are non-functional requirements used to evaluate the performance of a system. Since quality attributes are system-wide, their implementation must also be system-wide; satisfaction of a quality attribute requirement cannot be constrained to a single module or subsystem. Thus, a system-level vision of the system is required in order to ensure that the system can properly satisfy its quality attributes. One of the primary purposes of the system architecture is to create a system design to satisfy the quality attributes. Wrong or bad architectural choices can cost significant time and effort in later development, or may cause the system to fail to meet its quality attribute goals. The most important quality models used in software development is the ISO/IEC 25010:2011 standard (ISO/IEC, 2011), which replaced its predecessor ISO/IEC 9126-1:2001. The ISO/IEC 25010:2011 standard defines quality characteristics relevant to all software systems. These characteristics are further subdivided into subcharacteristics, which can be measured by one or more quality properties. The ISO/IEC 25010:2011 presents five characteristics associated with quality in use and eight about product quality.

Quality in use characteristics focus on the interactions of users and the product in a particular context of use. These characteristics are: Effectiveness, efficiency, satisfaction, freedom from risk, and context coverage.

Product quality characteristics are related to static properties of software and dynamic properties of a computer system. These characteristics are: Functional suitability, performance efficiency, compatibility, usability, reliability, security, maintainability and portability.

Besides ISO/IEC 25010:2011, it is also possible to identify studies in the literature focusing on proposing quality models and sets of quality characteristics for particular system domains, such as self-adaptive systems (Yang, Li, Jin, & Chen, 2014) and wireless sensor networks (Ravula, Kim, Petrus, & Stoermer, 2005) These studies add quality characteristics relevant to a particular domain and adapt definitions of existing characteristics to better describe quality in this domain. For instance, (Yang et al., 2014) adds adaptability as a relevant quality characteristic of self-adaptive systems. Adaptability is the ability of the system to identify the root of a failure within the software and to change to new specification or operating environments. Other studies, (Ravula et al., 2005) include fault-tolerance, scalability and security as relevant quality characteristics of wireless sensor network.

Fault-tolerance is the ability of software to withstand (and recover) from component or environmental, failure. WSNs require high availability guaranteeing reliable services. The consistency of the data states and the redundancy of the persistent data in the distributed wireless networks is one of the critical software qualities necessary to achieve reliable services even when faults are found in the system. For instance, how does the software system meet the deadline of high priority messages when some of the system’s sub-networks are disconnected from the entire network? How does the system provide correct and consistent states when faults occur over the network? Middleware technologies also need to enable the modeling and analysis of fault states and modes, and design the strategies to support a fault tolerant system.

Scalability is another important quality attribute for WSN, as WSNs grow from small residential applications to commercial/industrial applications with thousands of nodes, their processing capacity must be able to grow, to both expand the complexity of the states and the amount of sensing data the system manages, and maintain acceptable performance levels for the real time services. The middleware system needs to resolve these scalability design issues by decoupling all the networking, messaging and operating system related development from the application level development.

Finally, security relates to unauthorized access to the software functions. Sensor networks must ensure network security and user privacy. Security and privacy are of extreme importance for many WSN applications. Standardization of security should provide different security levels according to each application need. Moreover, the privacy of users and information should be protected. In (Yang et al., 2014), authors noted a gap in research on security requirements engineering related to self-adaptive systems.

Quality characteristics for self-adaptive systems and WSN have a profound effect on the proposed RA by providing a sound basic for making objective decisions about design trade-offs. They represent areas of concern for self-adaptive WSN middleware systems that directly impacted design decisions for the conception of our RA.

3.5 Step RA-3: Architectural Synthesis

In this section RAMSES is described using RAModel (Reference Architecture Model) as a framework (E. Nakagawa et al., 2012). The outcome of this step of ProSa-RA is an architectural description composed of a set of architectural views and additional artifacts of the RA. RAModel, depicted in Figure 22, provides information on all elements (and their relationships) that could be contained in RAs, independently on application domains or purpose of such architectures. RAModel is composed of four groups of elements (detailed in Appendix A).

- **Domain:** It contains elements related to self-contained, specific information of the space of human action in the real world, such as domain legislations, standards, and certification processes, which impact systems and related RAs of that domain.
- **Application:** It contains elements that provide a good understanding about the RA, its capabilities and limitations (such as goals and needs of the RA, scope and functional requirements). It also contains elements related to the business rules (or functionalities) that could be present in software systems built from the RA.
- **Infrastructure:** it refers to elements that could be used to build the software systems based on the RA. These elements are responsible to enable these systems to automate, for instance, processes, activities, and tasks of a given domain; and
- **Crosscutting Elements:** this group aggregates a set of elements that are usually spread across and/or tangled with elements of the other three groups (domain, application and infrastructure). Examples of crosscutting elements are communication (internal and external) in the software systems built from the RA, as well as the domain terminology (set of terms of the domain that are used in the description of the RA) and decisions (such as alternatives, rationale and tradeoffs reported during the development of the RA).

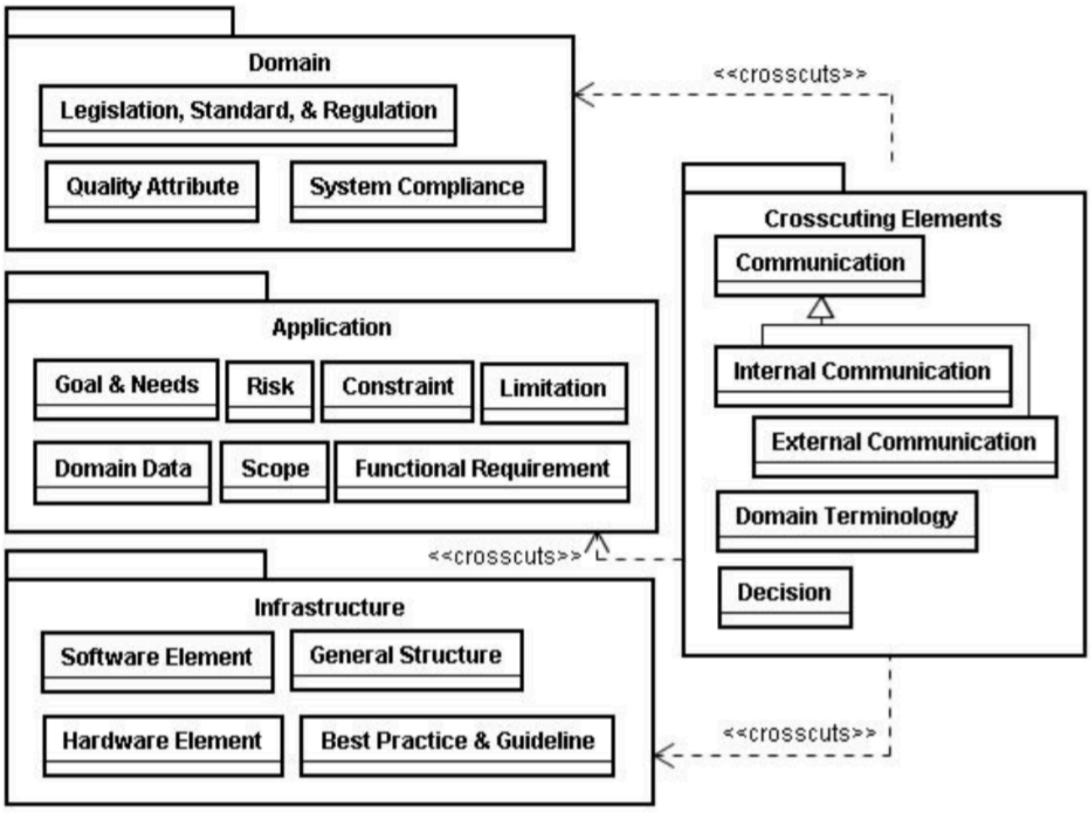


Figure 24. RAModel: Reference model for reference architectures

Considering the effectiveness of using architectural views to represent software architectures (see Section 2.4.4), they can be also adopted to describe reference architectures (Guessi, de Oliveira, & Nakagawa, 2011). Thereby, (Nakagawa et al., 2014) have observed that different sets of views can be used to represent most of RAModel elements. (Nakagawa et al., 2014) organize architectural views in four groups of architectural viewpoints. Each group has one or more related views. Views in the same group usually describe similar types of information of the RA or information in different abstraction levels of such architecture. In RAMSES, these viewpoints, listed below, are described by combining UML and Pi-ADL.

(i) Crosscutting viewpoint,

(ii) Source Code viewpoint,

(iii) Runtime viewpoint,

(iv) Deployment viewpoint.

(Nakagawa et al., 2014) consider that these architectural viewpoints are sufficient to represent the most of reference architectures, independently of the application domain.

During the development of these architectural views, we checked RAModel in order to verify if all elements present in RAModel and required in the reference architecture were considered.

3.5.1 Crosscutting viewpoint:

Crosscutting viewpoint shows for all stakeholders the general information about RAMSES. Three views compose this group:

(i) Conceptual view, that presents the structure of information managed within the RA, where a glossary can be used to describe each term used, optionally replaced by a domain ontology, such the Semantic Sensor Network Ontology (SSN) (Neuhaus & Compton, 2009);

(ii) Functionality view, that identifies, at a higher-level abstraction, the main functionalities of the RA components, their relationships, and possible consumers;

(iii) System Services view, that identifies at a higher abstraction level the minimal set of services (functions) that need to be supported by all implementations of the RA.

3.5.1.1 Conceptual View

A conceptual view of RAMSES is represented through a UML Class Diagram notation (Figure 25). The elements contained in this conceptual model are the elements that compose a Self-Adaptive WSN application and they can be stored as meta-data in the knowledge base of RAMSES.

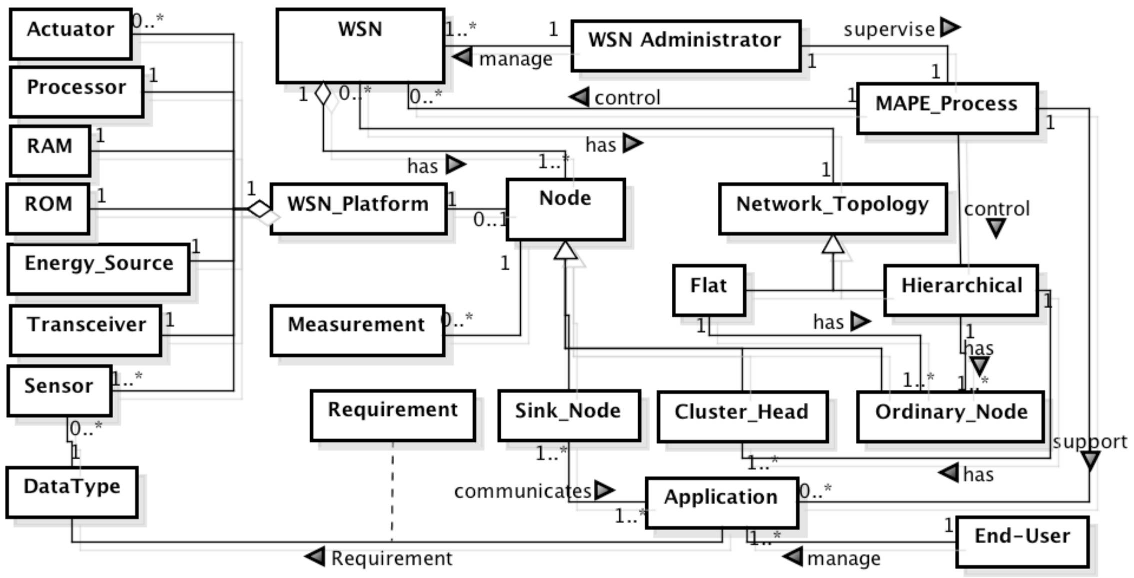


Figure 25. Conceptual view

Table 9 details the conceptual domain of RAMSES.

Table 9. Conceptual domain

Terminology	*Definition*
WSN	Wireless Sensor Network, which comprises a set of **Nodes** that cooperatively monitor environmental conditions, such as temperature, humidity, motion, and so forth
Node	A device responsable for collecting information from its surrounding environment and transmit it to one or more points of centralized control. A node may
Network Topology	It is the arrangement of the **Nodes** of the **WSN**. RAMSES considers that a WSN may have a **Flat** or **Hierarchical** topology
Flat Topology	This is actually the case of no topology or the absence of any defined topology. In flat topology, each sensor plays equal role in network formation
Hierarchical Topology	RAMSES considers hierarchical topologies as a cluster-based topology
Sink_Node	A type of **Node** that connects the WSN with external networks and applications
Cluster Head	A type of **Node** that acts as autonomic managers of nodes belonging to the cluster
Ordinary Node	A type of Node that receives adaptation messages from cluster heads
WSN Platform	It is an underlying computer system on which application program for WSN can run.
Measurement	It refers to sensing data collected by **Sensors**
Sensor	It is an object whose purpose is to detect events or changes in its environment, and then provide a corresponding output
Actuator	It is a type of motor that is responsible for moving or controlling a mechanism or system
Processor	It is the logic circuitry that responds to and processes the basic instructions that drive a WSN platform
RAM	Random Access Memory
ROM	Read Only Memory
Energy Source	It is a system that provides power to **Node**
Transceiver	It is a transmitter/receiver of a single WSN package
WSN Administrator	It is a user that supervises and manages the WSN operation
End-User	It is a user that defines and interacts with WSN application
Application	It is a set of application requirements defined by an End-User
Requirement	An application requirement can be defined as: application lifetime, data collection rate, maximum desired delay, physical phenomena to be monitored, geographical area when the data collection will be performed, among others.
DataType	It is a classification that identifies the type of data, such as: Real, Integer or Boolean.
MAPE-Process	It refers to autonomic computing process (for Monitoring, Analyzing, Planning and Executing) which is performed for managing the WSN

3.5.1.2 Functionality view

This view defines the main functionalities of RAMSES. It introduces for all stakeholders, in a higher abstraction level, the main elements that deliver the RA functionality, the key functional elements, their responsibilities, the interfaces they expose, and the interactions between them. This view demonstrates how RAMSES performs the functions required of it.

RAMSES aims to meet the efficiency and flexibility requirements for self-adaptive WSN middleware systems and it builds on autonomic computing principles by mapping the MAPE-K model, proposed by IBM (IBM, 2005), to an implementable architecture for managing and control autonomic WSNs.

RAMSES assumes a hierarchical topology for the WSN organization by grouping nodes into clusters, where some nodes work as cluster heads and collect the data from other (ordinary) nodes in the clusters. Then, the cluster heads can process the data (by performing operations such as data aggregation, data fusion and simple data analysis) and send it to the sink node as a single packet, decreasing the overhead from data packet headers and reducing the transmission of redundant data. Therefore, the hierarchical network is defined with three types of nodes: sink nodes, cluster heads and sensor nodes. Sink nodes connect the WSN and the gateway, which in turn connects the WSN to external networks, and applications. Cluster heads act as autonomic managers of nodes belonging to their respective clusters. Finally, a sensor node plays the role of a managed node that receives adaptation messages from its cluster head. The overall system is controlled by a hierarchical control structure where complete MAPE-K loops are present at all architecture layers of the hierarchy. MAPE-K loops at different levels interact with each other by exchanging sensing data and control information. The MAPE-K loop at a given level may pass to the level above information it has collected, possibly filtered or aggregated, along with information about locally planned actions, and may issue to the level below directives about adaptation plans that should be refined into corresponding actions.

It is important to note that most of the feedback control loop (FCL) solutions use context-based reasoning for the monitoring process and a policy-based reasoning to execute action plans. Therefore, since we are proposing a MAPE-based solution, our

RA fits in this type of approach (FCL approach), also following the context-based and policy-based reasoning principles to monitor WSN network context and execute action plans, respectively.

Among the patterns and architectural styles described in section 3.3.4, the following were adopted in the design of our RA: (i) Layered architectural style, (ii) broker pattern, (iii) decorator pattern, (iv) decentralized patterns, (v) mediator pattern and, (vi) data gathered pattern.

RAMSES adopts the Layered architectural style (see Section 3.3.4.1), by encompassing three architectural layers (Figure 26): Sensor MAPE-K Layer (SML), Network MAPE-K Layer (NML), and Goal Management Layer (GML). SML concerns the autonomic management inside sensor devices; NML concerns the autonomic management in the whole network and GML aims to manage WSN applications and to set adaptation policies used by the underlying layers to perform network adaptations. In order to manage the autonomic behavior of the whole network, the NML requires the collection of information monitored by underlying levels (from sensor nodes). Thereby, NML acts as an autonomic manager and SML acts as a managed resource. On the other hand, SML concerns the autonomic management inside each cluster. Each cluster, besides forwarding monitoring information to the NML, behaves as an independent autonomic system, where cluster heads are autonomic managers and ordinary nodes are managed resources.

The SML allows adapting a node configuration according to context information of nodes and following sensor adaptation policies. RAMSES considers as node context the following information: battery level, data delivery model (event-based, periodic and request-reply models), data sending rate, operational state (active/inactive/idle), type of node (node manager/cluster heads and managed node/ordinary nodes), tasks that the nodes are currently performing (routing/sensing/storing), signal strength, and localization. Node managers may create new configurations for managed nodes localized inside their respective cluster. Each node manager is responsible for managing its own cluster. In order to determine all adaptation actions needed to reconfigure managed nodes of a cluster, a node manager continuously receives context information from its managed nodes and a MAPE-K process, at the cluster level, is performed to verify the need of an adaptation. Whenever an adaptation action is required, a node configuration message is created containing:

(i) Information about changes in the topology of the cluster (activating and deactivating managed nodes);

(ii) Adjustments of the data delivery model in use by cluster members; and

(iii) Tasks of nodes (routing, sensing, storing).

After the adaptation plan is performed, the node manager disseminates the configuration message to all cluster members. In addition, a node manager is able to receive a cluster configuration message sent by the gateway (via the sink node) containing a set of tasks to be executed by sensor nodes (for instance, to configure cluster nodes for performing a sensing task aiming at delivering to cluster heads temperature measures at a predefined interval) and the adaptation policies used by the nodes to support adaptation decisions.

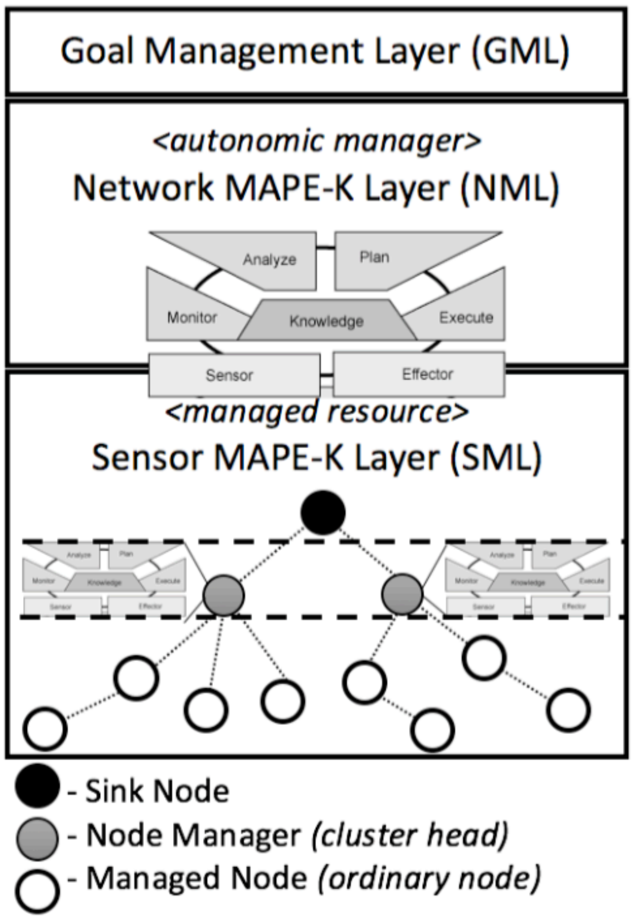

Goal Management Layer (GML)
<autonomic manager>
Network MAPE-K Layer (NML)
Analyze
Plan
Monitor
Knowledge
Execute
Sensor
Effector
<managed resource>
Sensor MAPE-K Layer (SML)
- Sink Node
- Node Manager (cluster head)
- Managed Node (ordinary node)

Figure 26. Layered architecture of RAMSES

RAMSES organizes MAPE loops in a hierarchical fashion, where each level of the hierarchy contains instances of all four MAPE components. In this setting, higher-level MAPE loops (at the NML layer) determine the configuration values for the subordinate MAPE loops (SML layer). Monitoring and execution are delegated to the different nodes of the network, whereas analyzing and planning are centralized in cluster heads and gateways. It is worth to highlight that the hierarchical control pattern implementation is used to allow the deployment of two MAPE processes, where NML is in a higher level and SML is in a lower level of operation. Moreover, the decentralized deployment of MAPE process inside clusters (SML) is enabled by the Master/Slave pattern implementation, where ordinary nodes monitor their context and a cluster head manages them.

3.5.1.3 System Services View

The System Services View identifies at a higher abstraction level the minimal set of services that need to be supported by all implementations of RAMSES. This information is useful to ensure interoperability among different instances of the RA. Architectural services were defined based on architectural requirements specific for each architectural layer of RAMSES: GML, NML and SML, presented in Table 10, Table 11 and Table 12 respectively. These specific architectural requirements are defined based on the general requirement for self-adaptation identified in Section 3.4.1.

Table 10. GML Architectural Requirements

Req. ID	**GML Requirements**	**Ref.**
Req-GML-[1]	The RA must enable to end-users to configure WSN applications in the system	Req-B
Req-GML-[2]	The RA must enable to end-users to monitor sensing data regarded to WSN applications created by them	Req-B
Req-GML-[3]	The RA must enable to WSN administrators to configure adaptation policies for managing the whole network	Req-C
Req-GML-[4]	The RA must enable to WSN administrators to configure adaptation policies for managing sensor node clusters	Req-C
Req-GML-[5]	The RA must enable multiple WSN applications running at same time	Req-B
Req-GML-[6]	The RA must enable to end-users to perform searches and queries about WSN applications	Req-B
Req-GML-[7]	The RA must enable to WSN administrators to perform searches and queries about active adaptation policies	Req-C
Req-GML-[8]	The RA must enable to software architects to know the current state of RAMSES architecture.	Req-B Req-C
Req-GML-[9]	The RA must enable to WSN administrators to know about autonomic computing decisions made by the system.	Req-B Req-C

Table 11. NML Architectural Requirements

Req. ID	**NML Requirements**	**Ref.**
Req-NML-[1]	The RA must allow to store sensing data	Req-B
Req-NML-[2]	The RA must allow to store application data	Req-B
Req-NML-[3]	The RA must allow to store adaptation policies	Req-B
Req-NML-[4]	The RA must allow to store context information data	Req-D
Req-NML-[5]	The RA must allow to store users data	Req-B
Req-NML-[6]	The RA must allow to store control data	Req-C
Req-NML-[7]	The RA must allow store network services	Req-B
Req-NML-[8]	The RA must provide mechanisms to monitor context information data	Req-D
Req-NML-[9]	The RA must provide mechanisms to monitor application data	Req-D
Req-NML-[10]	The RA must provide mechanisms to monitor sensing data	Req-D
Req-NML-[11]	The RA must provide mechanisms to monitor network services	Req-D
Req-NML-[12]	The RA must provide mechanisms to analyze coverage of network	Req-D
Req-NML-[13]	The RA must provide mechanisms to analyze connectivity of network	Req-D
Req-NML-[14]	The RA must provide mechanisms to analyze QoS requirements	Req-D
Req-NML-[15]	The RA must provide mechanisms to analyze Application requirements	Req-D
Req-NML-[16]	The RA must provide mechanisms to analyze the residual energy of network	Req-D
Req-NML-[17]	The RA must provide mechanisms to plan the best network topology	Req-D
Req-NML-[18]	The RA must provide mechanisms to plan the best routing protocol	Req-D
Req-NML-[19]	The RA must provide mechanisms to plan the task to be configured on nodes	Req-D
Req-NML-[20]	The RA must enable to adapt the network behavior	Req-D
Req-NML-[21]	The RA must provide mechanisms to recover due a network failure	Req-D
Req-NML-[22]	The RA must provide mechanisms to configure a WSN platform	Req-D
Req-NML-[23]	The RA must provide mechanisms to communicate with WSNs	Req-A
Req-NML-[24]	The RA must provide mechanisms to protect the communication with WSNs	Req-A
Req-NML-[25]	The RA must publish sensing data to user applications	Req-B
Req-NML-[26]	The RA must publish autonomic decisions to network administrators	Req-C

Table 12. SML Architectural Requirements

Req. ID	**SML Requirements**	**Ref.**
Req-SML-[1]	The RA must enable to perform physical measurements	Req-B
Req-SML-[2]	The RA must enable a standard sensor nodes communications	Req-B
Req-SML-[3]	The RA must provide mechanisms to communicate WSN with a gateway	Req-A
Req-SML-[4]	The RA must provide mechanisms to protect the communications between nodes	Req-A
Req-SML-[5]	The RA must provide mechanisms to nodes publish sensing data to sink node	Req-A
Req-SML-[6]	The RA must publish context information to cluster heads	Req-E
Req-SML-[7]	The RA must analyze the residual energy of nodes	Req-D
Req-SML-[8]	The RA must analyze sensing data collected within a cluster	Req-D
Req-SML-[9]	The RA must provide mechanisms to analyze the HW health of nodes	Req-D
Req-SML-[10]	The RA must provide mechanisms to analyze the task configuration of nodes	Req-D
Req-SML-[11]	The RA must provide mechanisms to dynamically plan a cluster topology	Req-D
Req-SML-[12]	The RA must provide mechanisms to dynamically configure a node	Req-D
Req-SML-[13]	The RA must provide mechanisms to recover due a node failure	Req-D
Req-SML-[14]	The RA must allow to store sensor adaptation policies	Req-C
Req-SML-[15]	The RA must allow to store sensor node tasks	Req-B
Req-SML-[16]	The RA must allow to store preconfigured alert messages	Req-D
Req-SML-[17]	The RA must allow to store contextual data of clusters	Req-D
Req-SML-[18]	The RA must provide mechanisms to include new nodes on the network	Req-E

Table 13 shows the relationship between services and architectural requirements. In this table are also identified the RAMSES component that provide these service interfaces.

Table 13. System Services View

ID Service	Service	Description	Component	Related Req.
RA-S [1]	*publishWSNData*	To collect measurement data gathered by the WSN	Application Manager	Req-GML-[1], Req-NML-[9]
RA-S [2]	*setAppReq*	To receive application requirements to be monitored by the MAPE process	Network Monitor	Req-GML-[1], Req-NML-[2] Req-GML-[5], Req-NML-[8], Req-NML-[9]
RA-S [3]	*publishContext*	To publish all collected context data from WSN to be monitored by the MAPE process	Network Monitor	Req-GML-[2], Req-NML-[8], Req-NML-[10], Req-NML-[11], Req-NML-[25]
RA-S [4]	*getServices*	To get current network services from the Knowledge Base	Network Knowledge Base	Req-GML-[6]
RA-S [5]	*setAdaptationPolicies*	To publish adaptation policies	Network Knowledge Base	Req-GML-[3], Req-GML-[4] Req-GML-[7], Req-NML-[3]
RA-S [6]	*sendCurrState*	To update Knowledge Base with the current state of network (context and sensing data)	Network Knowledge Base	Req-NML-[1], Req-NML-[2], Req-NML-[4], Req-NML-[5], Req-NML-[6], Req-NML-[7]
RA-S [7]	*getCurrState*	To get the current state of network (current configuration, context and sensing data)	Network Knowledge Base	Req-NML-[12], Req-NML-[13], Req-NML-[14], Req-NML-[15], Req-NML-[16], Req-NML-[17], Req-NML-[18], Req-NML-[19]
RA-S [8]	*analysingRequest*	To request an analysis of network context	Network Analyzer	Req-NML-[12], Req-NML-[13], Req-NML-[14], Req-NML-[15], Req-NML-[16]
RA-S [9]	*getAdaptationInfo*	To publish adaptation network information	Inspection Manager	Req-GML-[8], Req-GML-[9], Req-NML-[26]
RA-S [10]	*adaptationRequest*	To request a network adaptation	Network Planner	Req-NML-[17], Req-NML-[18], Req-NML-[19]
RA-S [11]	*setAdaptationPlan*	To publish a network adaptation plan	Network Configuration Manager	Req-GML-[5], Req-NML-[20], Req-NML-[21], Req-SML-[18]

ID Service	Service	Description	Component	Related Req.
RA-S [12]	*WSNConfiguration*	To send a configuration message to WSN	Gateway Communication	Req-NML-[22]
RA-S [13]	*marshallRequest*	To marshall/unmarshall transmitted information by NML components	Security Marshaller	Req-NML-[24]
RA-S [14]	*receiveMsg*	To receive a transmitted information from WSN	Gateway Communication	Req-GML-[5], Req-NML-[23]
RA-S [15]	*sendGWMsg*	To send a configuration message to WSN	Request Handler	Req-GML-[5], Req-NML-[23]
RA-S [16]	*sensorMarshallRequest*	To marshall/unmarshall transmitted information by sensor nodes	Sensor Sercurity Marshaller	Req-SML-[4]
RA-S [17]	*receiveGWMsg*	To receive transmitted information from Gateway	Sensor Communication	Req-SML-[3]
RA-S [18]	*sendMsg*	To send a message towards a Gateway	Sensor RequestHandler	Req-SML-[3], Req-SML-[5]
RA-S [19]	*receiveMessage*	To receive a message from a sensor node	Sensor Manager	Req-SML-[2], Req-SML-[5], Req-SML-[6], Req-SML-[13]
RA-S [20]	*sendMessage*	To send a message towards a sensor node	Sensor Communication	Req-SML-[2], Req-SML-[5], Req-SML-[6], Req-SML-[13]
RA-S [21]	*publishData*	To publish sensing data	Sensor Manager	Req-SML-[1]
RA-S [22]	*nodeAdaptationPlan*	To publish a cluster adaptation plan	Sensor Manager	Req-SML-[12], Req-SML-[18]
RA-S [23]	*setConfiguration*	To configure sensor tasks	Acquisition Manager	Req-SML-[12], Req-SML-[13]
RA-S [24]	*analyseData*	To request an analysis of a cluster context	Sensor Analyzer	Req-SML-[7], Req-SML-[8], Req-SML-[9], Req-SML-[10]
RA-S [25]	*update*	To update Knowledge Base with the current state of cluster (context and sensing data)	Sensor Knowledge Base	Req-SML-[14], Req-SML-[15], Req-SML-[16], Req-SML-[17]
RA-S [26]	*select*	To get the current state of cluster (current configuration, context and sensing data)	Sensor Knowledge Base	Req-SML-[9], Req-SML-[10], Req-SML-[12]
RA-S [27]	*sensorAdaptationRequest*	To request a cluster adaptation	Sensor Planner	Req-SML-[11], Req-SML-[12]

3.5.2 Source Code Viewpoint

This viewpoint shows specific details, such as software structures and modules, about the implementation of the systems resulting from the reference architecture. A module view was used for presenting this viewpoint to architects and developers stakeholders. In Figure 27 is depicted a module view of RAMSES represented with a UML component diagram.

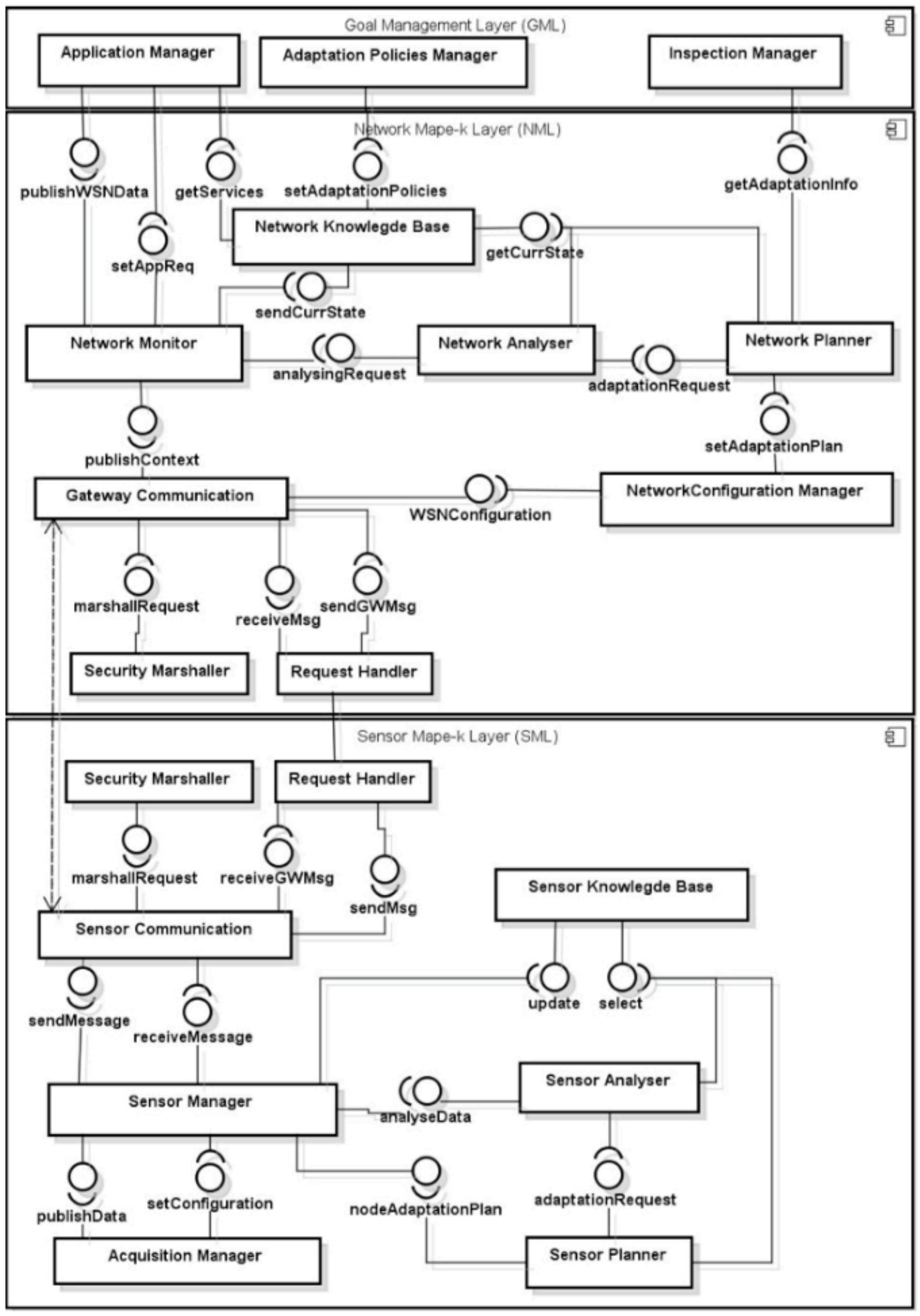


Figure 27. Module View of RAMSES

GML layer encompasses the Application Manager, Adaptation Policies Manager and Inspection Manager. NML components consist of Gateway Communication, Network Monitor, Network Analyzer, Network Planner, Network Knowledge Base and Network Configuration. Finally, SML is composed by the Sensor Manager, Sensor Analyzer, Sensor Planner, Sensor Knowledge Base, Sensor Communication, and Acquisition Manager. GML and NML components are deployed in a Gateway node, while the SML components are deployed in sensor nodes.

A. Goal Management Layer (GML):

RAMSES is based on self-adaptation principles and to perform this autonomic behavior, a minimal human intervention is required. Components that allow human interaction to define the policies and configurations of network adaptation mechanisms are in the GML and its components are Application Manager, Adaptation Policies Manager, and Inspection Manager.

a. Application Manager:

This component is used by end-users to create applications by using services provided by network and to enable the monitoring of sensing data. The Application Manager component is declared with connections for setting the application requirements; for requesting services provided by the network and; for receiving sensing data. A WSN Application contains a set of requirements that can be defined as: application lifetime, data collection rate, maximum desired delay, physical phenomena to be monitored, geographical area when the data collection will be performed, among others. Network services are a set of attributes describing capabilities of sensor nodes (for instance, temperature, luminosity, and humidity). These capabilities are advertised to cluster heads that in turn forwards this information to the NML to be managed and offered to applications. Sensing data are measurements of physical variables performed by sensor nodes.

b. Adaptation Policies Manager:

This component is used to define adaptation policies and is declared with an output connection for sending adaptation policies to underlying levels of the architecture (NML and SML). A policy is defined by humans, according to business goals, and specifies a set of actions to be performed whether the Network Monitor Component detects predefined events.

In RAMSES, a policy consists in a set of variables (Id, ContextInfo, Operators, Value, Actions, Priority) that allows:

(i) Identification of adaptation policy (Id);
(ii) Type of adaptation policy (specific for NML or specific for SML), the current context information (ContextInfo) of network;
(iii) Operators for indicating which logic operator Is used (>, <, >=, <=, ==);
(iv) Value representing the threshold to fire a management decision;
(v) Actions to be applied; and
(vi) Priority of the policy to prevent conflicting policies.

c. Inspection Manager:

This component is used to inspect adaptation information handled by the middleware that is accessible to network administrators. Inspection manager component is declared with an input connection for receiving adaptation information of the network.

B. Network MAPE-K Layer (NML):

NML is responsible for performing adaptation actions to (re)configure the network. The contextual information used for this activity is provided by the whole network. For the general implementation of MAPE-K components of the NML, we adopted the design pattern Decorator (see Section 3.3.4.4). This approach allows clarifying the component relationships of the RA and defining the behavior of components. Thus, the adoption of this pattern promotes a clear and complete separation among mechanisms that handle adaptation issues of the RA and the specificity of the instantiated middleware. Figure 28 represents the applying of Decorator pattern considering the classes' implementation of MAPE-K component used in RAMSES.

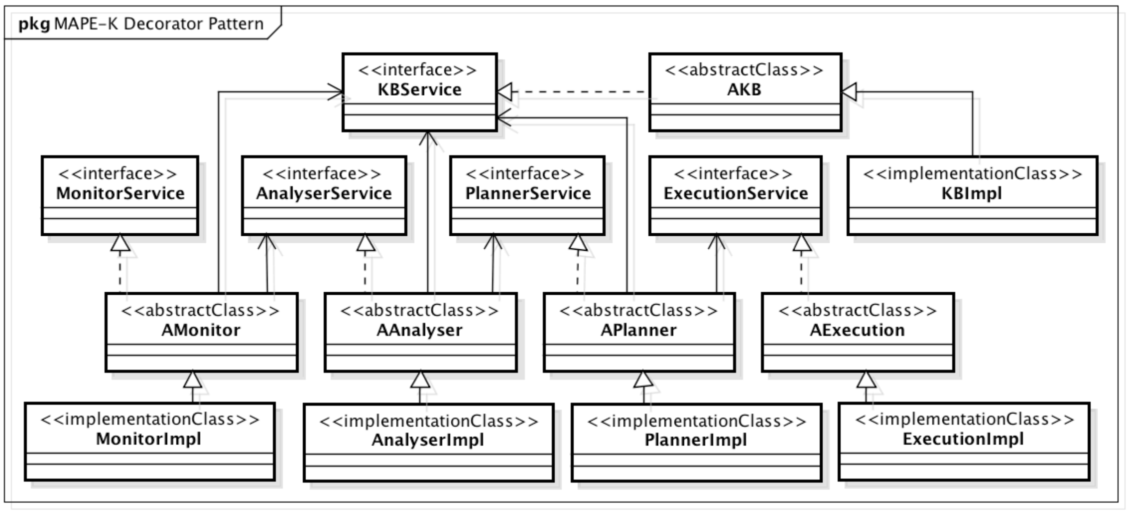

pkg MAPE-K Decorator Pattern
<<interface>> KBService
<<abstractClass>> AKB
<<interface>> MonitorService
<<interface>> AnalyserService
<<interface>> PlannerService
<<interface>> ExecutionService
<<implementationClass>> KBImpl
<<abstractClass>> AMonitor
<<abstractClass>> AAnalyser
<<abstractClass>> APlanner
<<abstractClass>> AExecution
<<implementationClass>> MonitorImpl
<<implementationClass>> AnalyserImpl
<<implementationClass>> PlannerImpl
<<implementationClass>> ExecutionImpl

Figure 28. MAPE-K Decorator Pattern

a. **Gateway Communication:**

This component provides to the Network Monitor component, contextual information collected through the SML. This component was established based on a broker pattern and it has a connection to Security Marshaller component for marshaling the transmitted information in a safety way.

Our RA adopts the broker pattern aiming to enable the communication between Network Mape-k Layer (NML) and Sensor Mape-k Layer (SML). In RAMSES, the communication technology used by SML depends on hardware/software platform of sensor nodes. There are currently many hardware and software platforms available for WSN. These platforms may work together in the same network infrastructure since they often use the same wireless communication protocol. Actually, these platforms execute specific operating systems and applications for WSN, and they are implemented in specific programming languages, then a method for enabling the communication and integration between these platforms and a gateway system is needed. A well-known and widely used example of WSN platform is the sensor nodes manufactured by Crossbow, currently MEMSIC (MEMSIC) (such as Mica and MicaZ). Such platform is supported by TinyOS operating system (TinyOS). Other example are the sensor nodes manufactured by Sun (Oracle) named Sun Spot that use Java programming; and the sensor nodes of Arduino platform (Arduino) (such as Arduino Mega, Arduino Uno, among others). Therefore, the broker pattern hides and mediates all communication between NML and the heterogeneity of SML components.

b. Network Monitor:

This component receives sensing data from the Gateway and publishes these data to the Application Manager component. It also publishes the sensing data to the Network Analyzer component to analyze the network context. The Network Monitor monitors context information provided by the whole network and uses the hierarchical control pattern to summarize the underlying monitoring information. It monitors sensing data (measures of physical phenomena), context information (such as battery status of sensor nodes), services provided by nodes (such as temperature, humidity) and application requirement (such as data delivery model, desired services, and QoS requirements) as depicted in Figure 27.

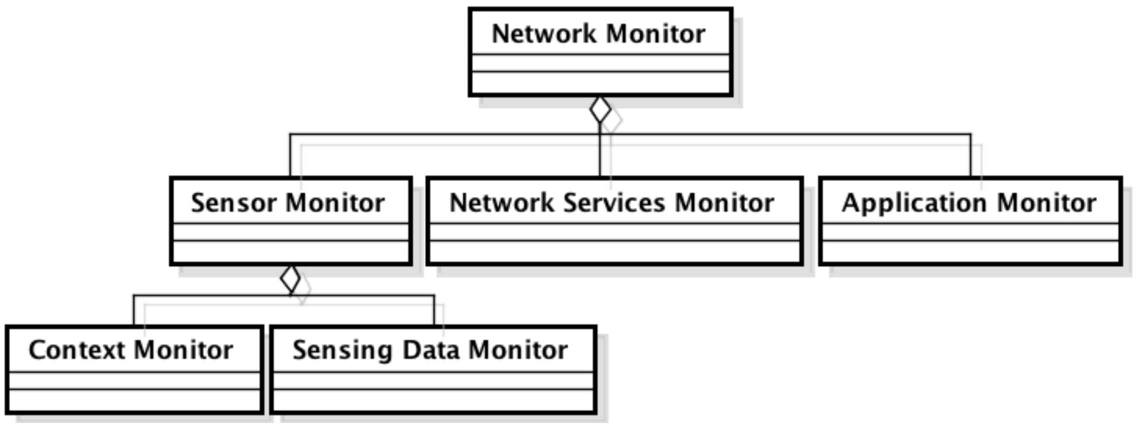


Figure 29. Network Monitor concerns

c. **Network Analyzer:**

This component detects symptoms to determine the need of network adaptation. With the monitored information and adaptation policies, the Network Analyzer is able to support the implementation of methods for verifying if application requirements, coverage, and connectivity are guaranteed, and for verifying the energy state of the network. If an adaptation need is detected, then an adaptation request must be performed. Figure 28 depicts the main concerns of this component.

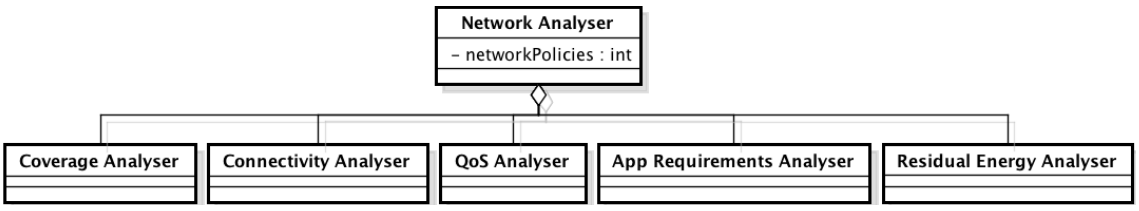


Figure 30. Network Analyzer concerns

d. Network Planner:

This component plans a network configuration once an adaptation request is sent by the Network Analyzer. The Network Planner component considers adaptation policies to generate a network configuration. A policy specifies a set of actions that should be taken by the middleware upon the occurrence of adaptation requests. These plans concern, as depicted in Figure 29, to Topology Control, a Sensor Tasks and Routing Protocols.

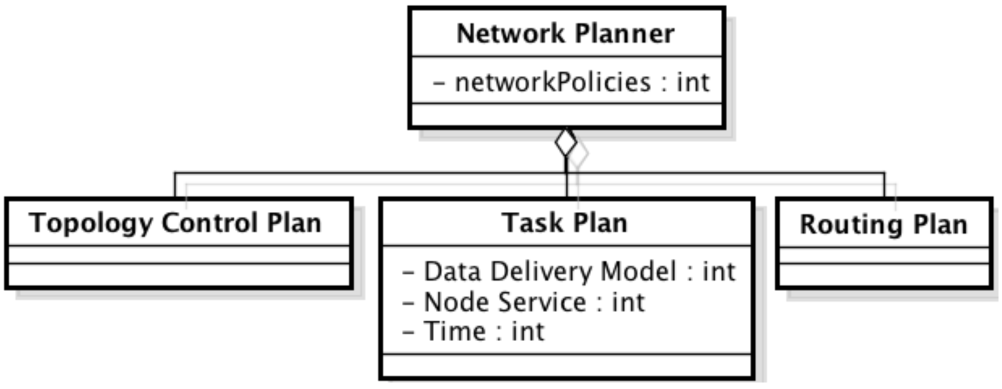


Figure 31. Network Planner concerns

e. Network Configuration Manager:

This component receives configuration parameters from Network Planner, translates these parameters in a configuration message, and disseminates this message to the nodes, through the Gateway Communication component. Since our RA adopts a hierarchical control pattern, the underlying monitoring information, provided by the Sensor Manager component, is able for configuring an adaptation plan for whole network.

f. Network Knowledge Base:

This component stores all context information and supports the MAPE-K process. This component stores information (Figure 30) regarding Sensor Context, Sensing Data, Network Services, Application requirements, and Network Adaptation Policies.

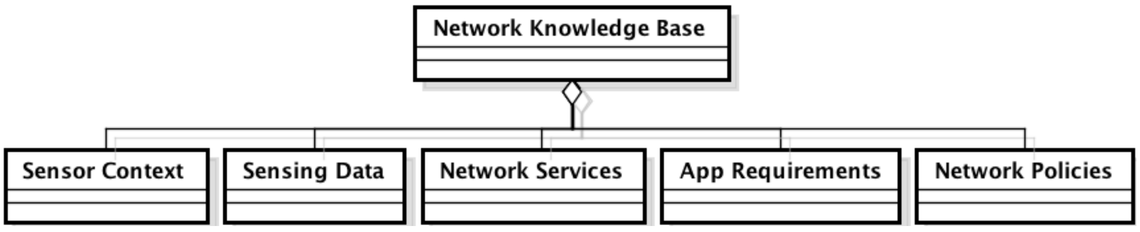


Figure 32. Network Knowledge Base concerns

C. Sensor MAPE-K Layer (SML):

SML consists in: Sensor Manager, Sensor Analyzer, Sensor Planner, Sensor Knowledge Base, Sensor Communication, and Acquisition Manager. In order to promote loose coupling and simplify the communication between SML components, the Mediator pattern (see 3.3.4.6) was applied. Sensor Manager encapsulate the collective behavior in a separate mediator module.

a. Sensor Manager:

Manages the nodes behavior and determines all adaptation actions needed to reconfigure: (i) a cluster, if the node is configured as a manager; and (ii) itself, if the node is configured as a managed node. This component is responsible for executing the MAPE-K process.

The Sensor Manager component is declared with connections for receiving the new configuration from NML, for setting the new configuration in its own sensors, and for forwarding the new configuration for the other nodes of cluster, if the component belongs to a node that is a cluster head (node manager).

Moreover, in the case of a node manager, the component can use connections for receiving an adaptation plan. In this case, this component encompasses a function aiming to convert the adaptation plan in a sensor node configuration. If the node is a manager, this data will be analyzed by the Sensor Analyzer component.

b. Sensor Analyzer:

This component uses the *analyseData* connector to collect contextual information of sensor and detects symptoms to determine an adaptation need. With the information monitored by Sensor Manager (via Acquisition Manager Component) and adaptation policies, the Sensor Analyzer component is able to support implementation of simple methods for verifying if QoS of nodes are guaranteed and verifies hardware resources. If an adaptation need is detected an adaptation request must be performed. Figure 31 depicts the main concerns of this component.

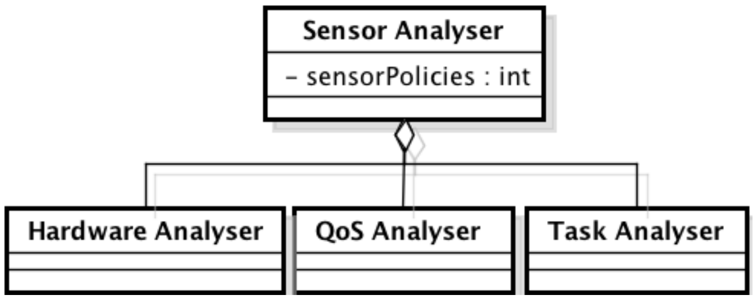


Figure 33. Sensor Analyzer concerns

c. **Sensor Planner:**

Plans a sensor configuration once an adaptation request is sent by Sensor Analyzer component. The Sensor Planner component considers adaptation policies to generate a new sensor configuration (such as a new topology cluster plan, an alert plan or a task plan). Since we adopt the hierarchical control pattern for distributing the management of the WSN we consider each cluster as an autonomic element. Each cluster in the set is independent and complete. Figure 32 depicts the main concerns of this component.

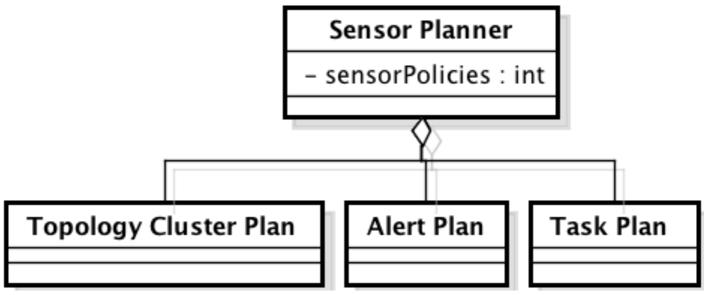


Figure 34. Sensor Planner concerns

d. **Sensor Knowledge Base:**

Stores sensor policies, tasks of nodes, alert messages, and information to support the MAPE-K process. Figure 33 depicts the main concerns of this component.

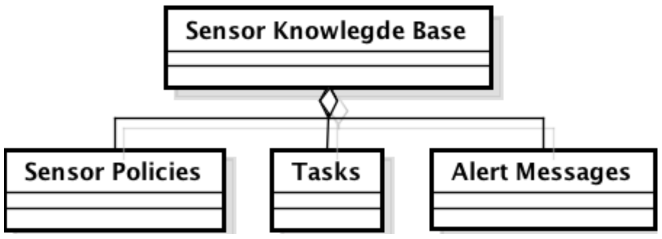


Figure 35. Sensor Knowledge Base concerns

e. Acquisition Manager:

Collects measures of physical phenomena monitored by sensors and executes the data delivery. If an adaptation request defines changes in the data delivery model, this component will be notified. Acquisition Manager component is declared with an output connection for sending sensing data to cluster head to it executes a local MAPE-K (Master Slave pattern) and an input connection for receiving the new configuration (tasks) of the sensor node.

f. **Sensor Communication:**

This component allows the communication among nodes. On one hand, when this component is instantiated into a node manager, the Broker pattern is applied for supporting the communication between sensor network and Gateway. On the other hand, when this component is instantiated into a managed node, the Data Gathering pattern is applied.

3.5.3 Runtime Viewpoint

This viewpoint shows the dynamic behavior of middleware systems that will be built based on RAMSES. Since this viewpoint must represent a RA, it shows the behavior of RAMSES in a high level of abstraction. This viewpoint presents the structure of a system at the moment when it is executed, through the representation of components, interfaces, packages, provided and required interfaces, ports and connectors. In this viewpoint, two architectural views are presented for architects and developer stakeholders:

(i) Collaborative components view, that describes the RA as a set of components interacting and using a set of shared data repositories (data flow) to perform the required functionality at runtime;

(ii) Process view, that describes the RA as a set of concurrently executing units and their interactions.

Dynamic behavior is particularly relevant in the domain of wireless sensor networks (WSNs), where numerous and unexpected changes of the execution context prevail. In this perspective, we noted a lack of well-defined architecture design that supports the autonomy of sensor networking. Hence, RAMSES follows the autonomic computing model MAPE-K, for making decisions aiming to attend self-adaptive WSN requirements.

The choice of a modeling approach that supports this dynamism is a challenge. WSN applications have to face environments in which operation conditions change very often, requiring that the system be able to adapt and reconfigure according to these dynamic scenarios. Managing the dynamic and context-aware adaptation is not an easy task, and whereas nodes in a WSN can be heterogeneous, having different sensor devices, and communication constraints (Sohraby et al., 2007). In terms of architecture design, most of current WSN applications use the same fixed software architecture for each network node, and these are designed in tightly coupled closed architectures. By architecture we mean a set of system components, the externally properties of those components and the relationship between them. For simple applications, the choice of a static architecture might not be a problem. However, when we consider more complex applications, with dynamic requirements that may change at runtime, and assume the WSN infrastructure must be shared by multiple applications, the need for a more advanced, dynamically reconfigurable architecture, arises. The idea behind node reconfigurability is that the network can adapt its functionality to the current situation, in order to lessen the use of the scarce energy and memory resources of nodes, while maintaining the integrity of its operation.

These peculiarities demand WSN to have a dynamic behavior to deal with both the environmental nature where they are deployed and the HW/SW constraints of sensor nodes. WSN must be able to monitor their context and to take adaptation decisions in runtime to, for instance, change the current routing protocol with a new one that requires less energy consumption; reconfigure the network topology to attend coverage and connectivity requirements that are no longer guaranteed due to a failure or node malfunction; add new node tasks to attend requirements of new started applications; load a module of cryptography to attend a security policy against a threat; etc. However, current techniques of architectural modeling for WSN systems mainly focus on the static and structural representation of WSN architectures and, as we argued in (Portocarrero et al., 2014), there is a lack of proposals to represent behavioral and dynamic aspects of WSN architectures and with a high level of formality that could make their verification and implementation easier.

In literature there are several architecture description languages (ADLs) that enable runtime representation of a software architecture. However, most existing ADLs: (i) are focused on structural, topological aspects of the architecture; (ii) do not provide an adequate support for representing behavioral concerns of the architecture; (iii) do not support an expressive description of dynamic aspects of the architecture; (iv) have limitations in terms of automated verification of architectural properties and constraints; and (v) are disconnected from the implementation level, thus entailing architectural mismatches and inconsistencies between architecture and implementation.

In order to tackle these problems, we are proposing the use of Pi-ADL to support WSN modeling. This ADL is a formal language for describing software architectures under both structural and behavioral viewpoints while fostering a rigorous, automated analysis of such architectures.

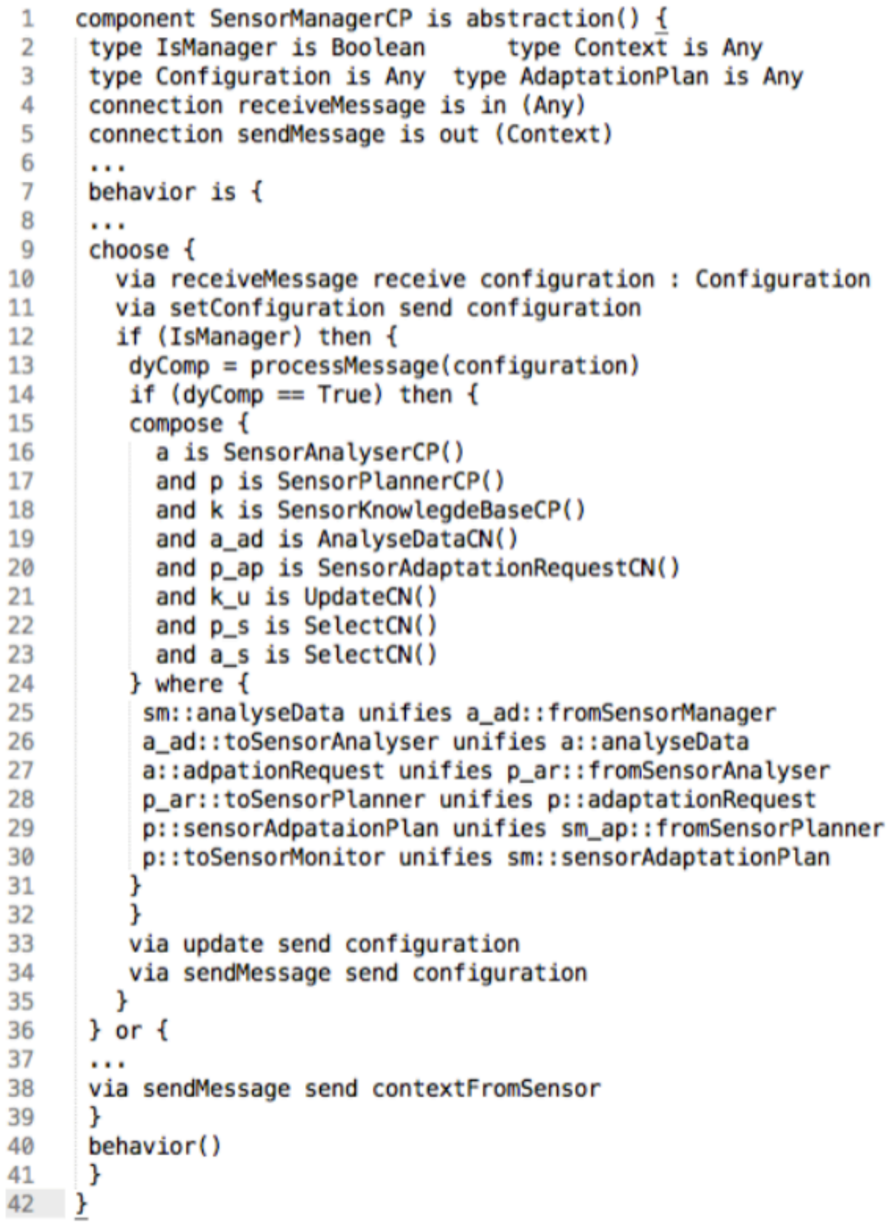
3.5.3.1 Collaborative Components view

This viewpoint describes RAMSES as a set of components interacting to perform the required functionality at runtime. This view is depicted in Figure 36 through a Pi-ADL language.

In Pi-ADL language, an architecture is described in terms of components, connectors and their composition. Components are described in terms of external connections and an internal behavior. Components can send or receive values via connections. Connections provide communication channels between two architectural elements. A connection can be declared as an output connection (for sending values), input connection (for receiving values), or input-output connection. Connectors are special-purpose components responsible to specify interactions among components. Components provide the locus of computation, while connectors manage interaction among components. A component cannot be directly connected to other component; there must be a connector between them.

Figure 36 depicts the specification of a fragment of code of the Sensor Manager component (*SensorManagerCP*). In line 2, the Boolean type *IsManager* is used to manage the behavior of nodes. When *IsManager* is True, a sensor node must act as a node manager (cluster head) and, if False, a sensor node must act as a managed node (ordinary node). Types Context, Configuration and *AdaptationPlan* are used to manage data messages exchanged by this component. In lines 4 and 5, an input connection (*receiveMessage*) and an output connection (*sendMessage*) are respectively specified. From line 7, the behavior of this component is specified. It is important to note that this component performs different behaviors depending on the type of messages received by it. The possible types of message are: Context messages received from nodes, Configuration messages received from NML or a node manager, and Adaptation Plan message received from the Sensor Planner component. The choose structure (line

9) is used to select the current behavior of the node. If the node needs to act as a manager, then it needs to dynamically upload (lines 12 to 30) the components responsible for executing the MAPE-K process (Sensor Analyzer, Sensor Planner and Sensor Knowledge Base).



```
1  component SensorManagerCP is abstraction() {
2   type IsManager is Boolean      type Context is Any
3   type Configuration is Any  type AdaptationPlan is Any
4   connection receiveMessage is in (Any)
5   connection sendMessage is out (Context)
6   ...
7   behavior is {
8   ...
9   choose {
10     via receiveMessage receive configuration : Configuration
11     via setConfiguration send configuration
12     if (IsManager) then {
13      dyComp = processMessage(configuration)
14      if (dyComp == True) then {
15      compose {
16        a is SensorAnalyserCP()
17        and p is SensorPlannerCP()
18        and k is SensorKnowlegdeBaseCP()
19        and a_ad is AnalyseDataCN()
20        and p_ap is SensorAdaptationRequestCN()
21        and k_u is UpdateCN()
22        and p_s is SelectCN()
23        and a_s is SelectCN()
24      } where {
25       sm::analyseData unifies a_ad::fromSensorManager
26       a_ad::toSensorAnalyser unifies a::analyseData
27       a::adpationRequest unifies p_ar::fromSensorAnalyser
28       p_ar::toSensorPlanner unifies p::adaptationRequest
29       p::sensorAdpataionPlan unifies sm_ap::fromSensorPlanner
30       p::toSensorMonitor unifies sm::sensorAdaptationPlan
31      }
32      }
33      via update send configuration
34      via sendMessage send configuration
35     }
36  } or {
37  ...
38  via sendMessage send contextFromSensor
39  }
40  behavior()
41  }
42 }
```

Figure 36. Pi-ADL Specification of Sensor Manager

Architecture-based approaches to self-adaptation, as RAMSES, exploits architecture models to reason about and control adaptation at runtime. An architecture model represents the adaptable system as a composition of components and their interconnections, with mappings to their implementations. Pi-ADL allows this representation where changes to the architectural model are mapped to the application implementation (as described in lines 12 to 30 of Figure 27), and can be executed at runtime. RAMSES introduces an architectural runtime view in in order to evaluate and adapt the running system.

Appendix B details the complete Pi-ADL specification of the runtime viewpoint of RAMSES. It is also available on is available at http://146.164.247.214/wordpress/rawsn/.

3.5.3.2 Process View

This view describes RAMSES as a set of concurrently executing units and their interactions. RAMSES Process View is represented as the activity diagram depicted in Figure 37

The process starts with activity 1, where users define application requirements, and activity 2, where network administrators define policies to be applied by the middleware in order to guide the adaptive network behavior whenever a context change occurs. Both, application requirements and adaptation policies are stored in a database managed by the Network Knowledge Base component. Following, the Network Monitor component starts monitoring the context (for instance, new application requirements, new sensing data and current battery level of nodes) and performs activity 3 to update the database with the current state of the network.

Activity 4 is performed by the Network Analyzer component to analyze the context changes in search of symptoms that indicate deviations between the current and desired behaviors of the network. A desired behavior of the network is one that attends all application requirements and complies with the current policies. Activity 5 is performed to plan a new network configuration according to the stored application requirements and policies. In activity 6, a message is created with the definition of the new configuration parameters of nodes in order to be sent to the network in a format compatible with the target platform. Activity 7 is performed by the Gateway Communication component to send the configuration message to the network using a dissemination protocol specific to the target platform.

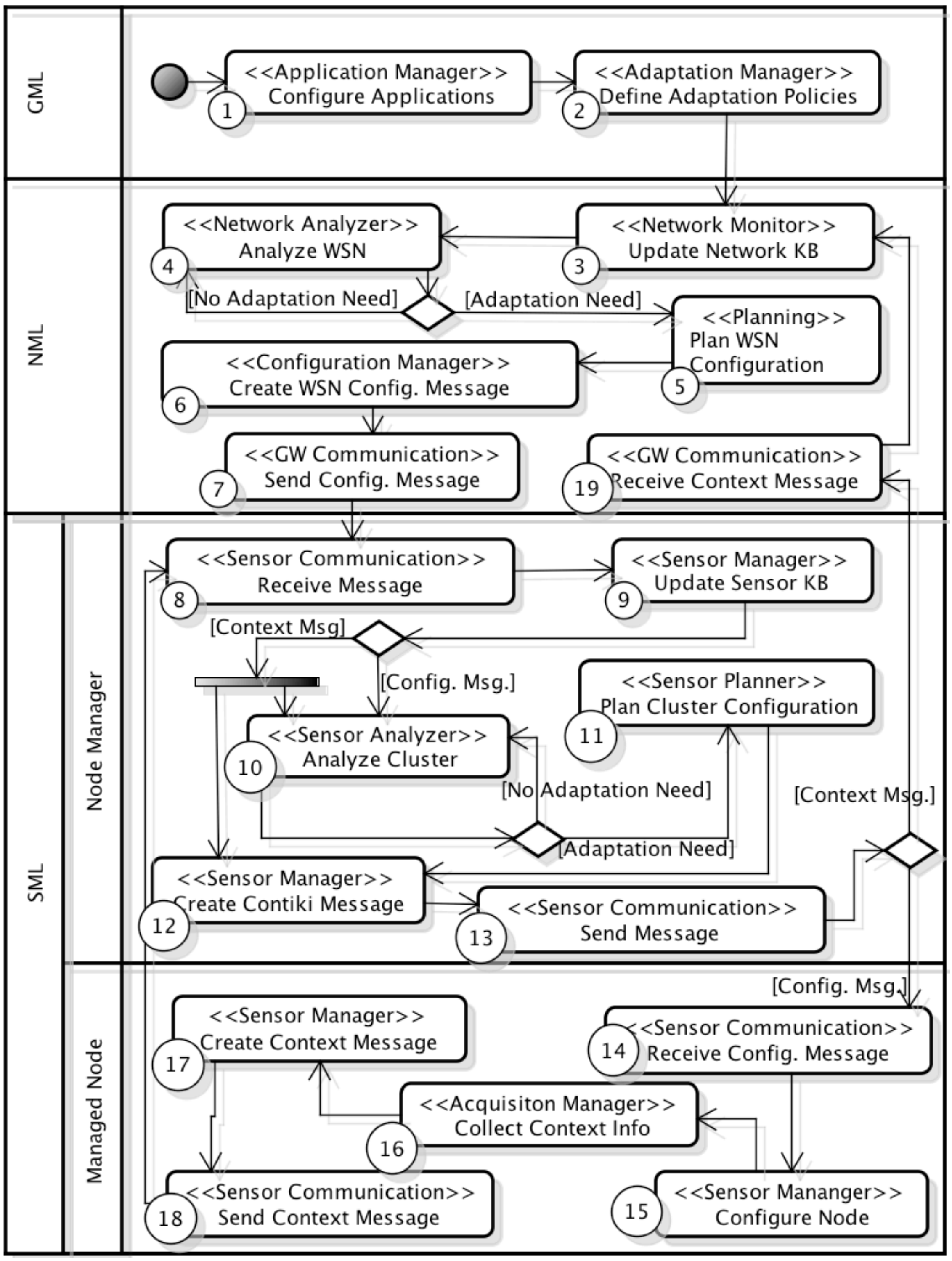

GML
NML
SML
Node Manager
Managed Node
<<Application Manager>> Configure Applications
1
<<Adaptation Manager>> Define Adaptation Policies
2
<<Network Analyzer>> Analyze WSN
4
<<Network Monitor>> Update Network KB
3
[No Adaptation Need]
[Adaptation Need]
<<Planning>> Plan WSN Configuration
5
<<Configuration Manager>> Create WSN Config. Message
6
<<GW Communication>> Send Config. Message
7
<<GW Communication>> Receive Context Message
19
<<Sensor Communication>> Receive Message
8
<<Sensor Manager>> Update Sensor KB
9
[Context Msg]
[Config. Msg.]
<<Sensor Planner>> Plan Cluster Configuration
11
<<Sensor Analyzer>> Analyze Cluster
10
[No Adaptation Need]
[Adaptation Need]
[Context Msg.]
<<Sensor Manager>> Create Contiki Message
12
<<Sensor Communication>> Send Message
13
[Config. Msg.]
<<Sensor Manager>> Create Context Message
17
<<Sensor Communication>> Receive Config. Message
14
<<Acquisiton Manager>> Collect Context Info
16
<<Sensor Communication>> Send Context Message
18
<<Sensor Mananger>> Configure Node
15

Figure 37. RAMSES Process View

Activity 8 is performed whenever a node manager (cluster head) receives a message (configuration message or context message). If a Configuration Message is received, the Sensor Manager component updates the local database (activity 9) and requests to the Sensor Analyzer component to analyze the new configuration (activity 10) in order to verify if the current configuration of nodes is able to guarantee application requirements and if the cluster nodes are working in conformance with policies. If the cluster needs to be adapted, activity 11 is performed to plan a new configuration for the cluster and a configuration message, containing the current node tasks, is created (activity 12) and sent to all nodes of the cluster (activity 13). When a managed node (ordinary node) receives a configuration message from the cluster head (activity 14), the Sensor Manager component updates configuration parameters of nodes (activity 15) and starts collecting the context information (activity 16) such as battery level and physical measures. Next, a context message is created and sent to the cluster head (activities 17 and 18 respectively). After a context message is sent to the cluster head, activity 8 is performed again in order to receive and process such message. Context information of nodes is updated in the database (activity 9) and, after that, two activities are performed in parallel. On one hand, activities 10 to 18 are executed until the network stabilization. On the other hand, the Sensor Manager component creates a context message containing aggregated data of all the cluster nodes and sends it towards the sink. When the context message is received by the Gateway Communication component (activity 19), the Network Monitor component updates the knowledge base and resumes all the process (activity 3).

An adaptation can be triggered mainly by a node (or link) fault, an application requirement not fulfilled by the network, a node configuration error, and/or a chance of optimization. The main causes of failures in sensor nodes are (Paradis, 2007): (i) nodes may fail due to depletion of batteries or destruction by an external event, (ii) links are failure-prone, causing network partitions and dynamic changes in network topology, and (iii) congestion may lead to packed loss.

RAMSES provides two strategies for network adaptation.

The first is supported by the reconfiguration of node parameters (parametrical adaptation). Five message structures are defined:

(i) Configuration message, used to configure sensor tasks;

(ii) Topology message, used to configure clusters and cluster topology by selecting active/inactive nodes;

(iii) Data message; used to deliver to cluster head the sensing data and context information of nodes;

(iv) CH message; used to deliver to NML the aggregated cluster data; and

(v) Adaptation policies messages, used to update the network with the current adaptation policies that support the autonomic behavior.

The second strategy is supported by reprogramming the sensor nodes. Many different mechanisms for reprogramming sensor nodes have been developed ranging from full image replacement to virtual machines. RAMSES suggests to use method based on dynamic loading of components in order to load specific components required for a managed node (ordinary node) becomes a node manager (cluster head). Dynamic loading method is effective for reprogramming even resource constrained WSN.

3.5.4 Deployment Viewpoint

Deployment viewpoint describes the hardware and software installed on each piece of hardware, the technology constraints and network connections. For this architectural viewpoint, we describe the deployment view in Section 3.5.4.1 and its Pi-ADL specification in Section 3.5.4.2. System administrators, developers and testers are the stakeholders interested in this architectural view.

3.5.4.1 Deployment View

This architectural view, depicted in Figure 38, describes RAMSES through the hardware structure, on which middleware components are allocated, and their network connections symbolizing the interaction between these devices.

In this view, each node represents a physical piece from the equipment where a RAMSES instantiation will be deployed. In RAMSES Deployment view are considered the following elements:

(i) Global Server, it is a node that contains the services provided by GML layer components (Application Manager, Adaptation Policies Manager and Inspection Manager), the following NML layer components (Network Monitor, Network Analyzer, Network Planner and Network Configuration Manager) and the data server to support the Network Knowledge Base component.

(ii) Gateway Server, it is a node deployed belong a WSN and it is responsible for enabling the communication between the network and the Global Server (via serial interface). This node contains the services provided by the Gateway Communication component, Security Marshaller component and the Request

(iii) Sensor Node, representing the sensor devices that establish the WSN. These devices contain SML components. RAMSES considers three types of Sensor Nodes:

   a. Base Station: This type of node contains two communication interfaces. The first is used to communicate a WSN with the Gateway Server via serial interface, and the second interface is used to communicate this node with other type of nodes (Cluster Head) via Zigbee interface. Zigbee technology implements the standard for wireless communication IEEE 802.15.4, specifically projected for this kind of communication.

b. Cluster Head: In this type of node are deployed all SML components and it is responsible for managing a group of Ordinary Nodes via Zigbee technology.

c. Ordinary Node: In this type of node are deployed all SML components. However, if a target platform is capable for dynamically loading components, then this node can dynamically unload the Sensor Analyzer component and the Sensor Planner component (these components are specifically used in MAPE process, that is an exclusive Cluster Head responsibility.

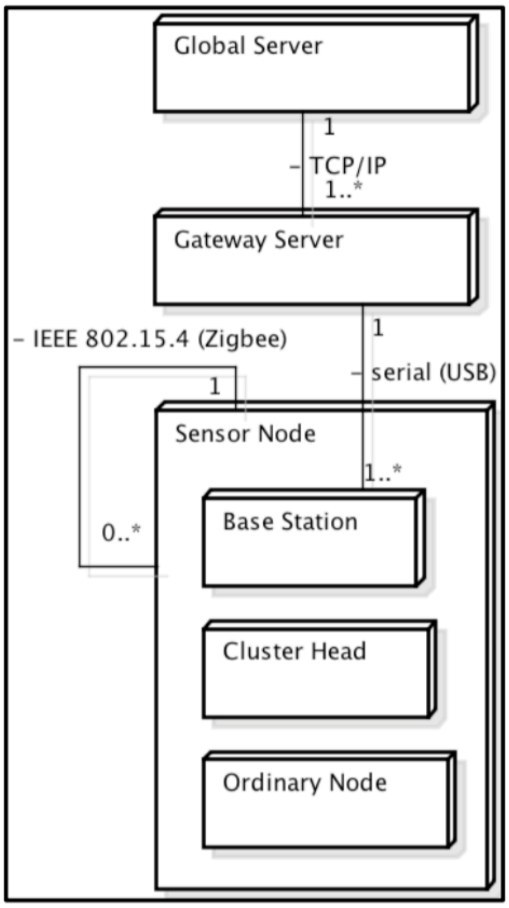
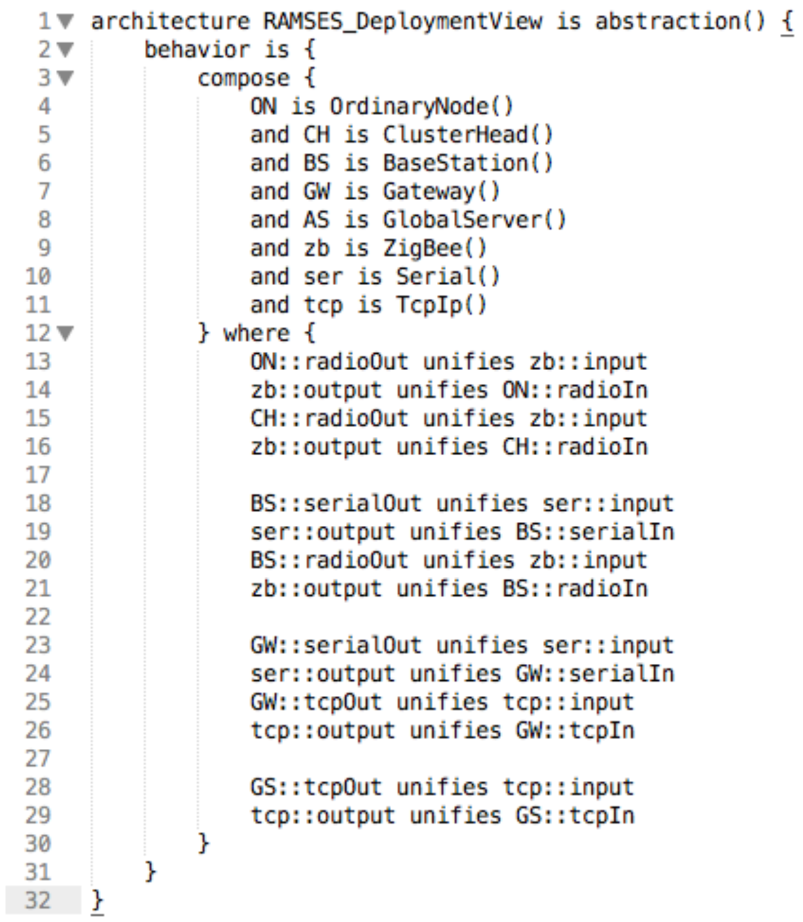


Figure 38. RAMSES Deployment View

3.5.4.2 Deployment Pi-ADL Specification

Figure 39 presents a fragment of Pi-ADL specification of the Deployment View detailed in Section 7.2.2.1. The complete Pi-ADL specification of this architectural view is presented in Appendix C. It is available at http://146.164.247.214/wordpress/rawsn/.



```
1  architecture RAMSES_DeploymentView is abstraction() {
2      behavior is {
3          compose {
4              ON is OrdinaryNode()
5              and CH is ClusterHead()
6              and BS is BaseStation()
7              and GW is Gateway()
8              and AS is GlobalServer()
9              and zb is ZigBee()
10             and ser is Serial()
11             and tcp is TcpIp()
12         } where {
13             ON::radioOut unifies zb::input
14             zb::output unifies ON::radioIn
15             CH::radioOut unifies zb::input
16             zb::output unifies CH::radioIn
17
18             BS::serialOut unifies ser::input
19             ser::output unifies BS::serialIn
20             BS::radioOut unifies zb::input
21             zb::output unifies BS::radioIn
22
23             GW::serialOut unifies ser::input
24             ser::output unifies GW::serialIn
25             GW::tcpOut unifies tcp::input
26             tcp::output unifies GW::tcpIn
27
28             GS::tcpOut unifies tcp::input
29             tcp::output unifies GS::tcpIn
30         }
31     }
32 }
```

Figure 39. Pi-ADL Deployment View Specification

In this deployment view representation, lines 4 to 8 represent the deployed node elements detailed in previous Section (Ordinary Node, Cluster Head, Base Station, Gateway Server and Global Server), and lines 9 to 11 represent the used communication technology between them. We note that Base Station (BS) and Gateway Server (GW) have two interfaces of communication (an input and an output connector for each interface), Zigbee and Serial for BS (lines 17 to 20) and, Serial and TCP/IP for GW (lines 21 to 24).

Although RAMSES does not depend on any specific implementation technologies, in Deployment view were presented some communication protocols (Zigbee, Serial, TCP/IP) aiming support an understanding of it. However, the proposed components interaction does not depend of such technology, and they can be implemented by equivalents technologies, since these technologies support the target platform.

RAMSES can be deployed on the main target platform for WSN such as the sensor motes manufactured by MEMSIC, Sun Spots, Contiki and Arduino platform.

3.5.4.3 Technology view

This view presents requirements for standards and tools commonly used within instances of a reference architecture to guide selection of appropriate or compatible technologies for the purposes of the reference architecture.

Hence, we present the RAMSES instantiation process (Figure 40) in order to determine the involved technologies, actors and their respective activities and roles. In this process, the required steps to develop a self-adaptive middleware for WSN based on RAMSES are described.

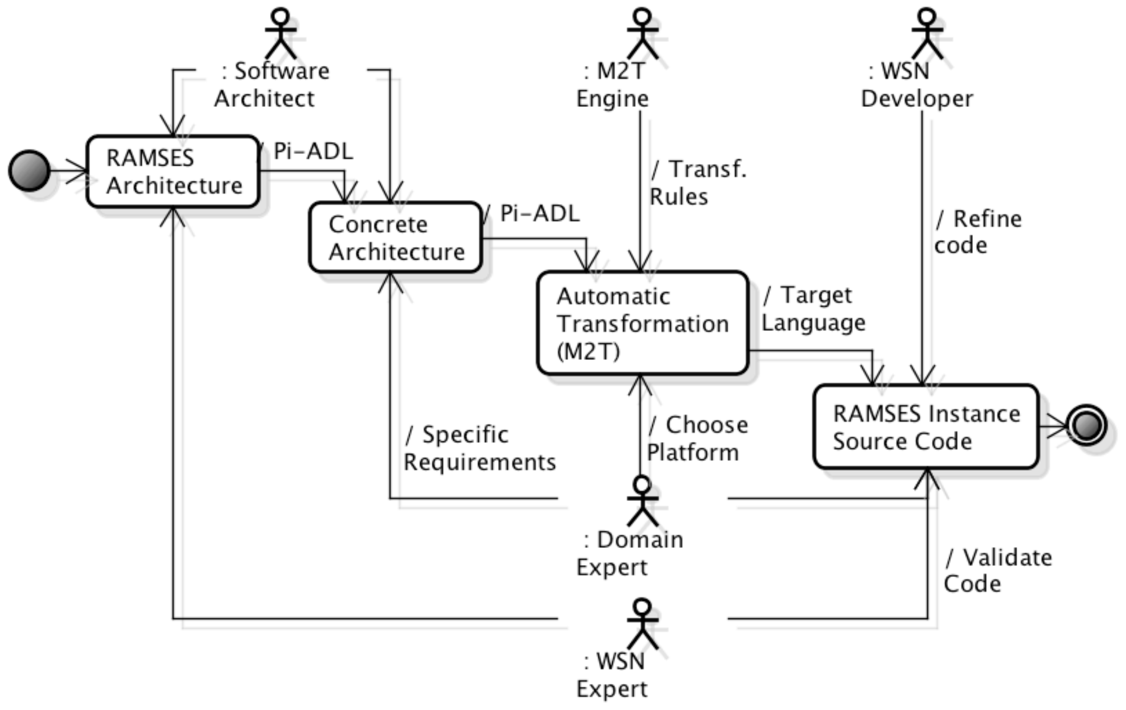


Figure 40. RAMSES Instantiation process

All the activities take place previously to the WSN nodes deployment. The software artifacts produced as outcome of the first activity represent the RAMSES architecture runtime view specification, written in pi-ADL. Aiming to instantiate RAMSES into a concrete software architecture, a domain expert, and a software architect should refine the RA, in order to extend it with architectural decisions (specified with PI-ADL) that address the specific requirements of the concrete middleware, such as target platform constraints and extension of RAMSES components that address specific requirements of the instantiated middleware.

Following, “Automatic Transformation (M2T)” is accomplished; this activity takes as input the Pi-ADL specification and the transformation code concerning the chosen target platform (for the Global Server, Gateway Server and Sensor Node) and generates as output the source code to be deployed in these RAMSES deployment nodes. Pi-ADL specifications are able to be automatically transformed into a source code of the target platform, chosen by a domain expert, by applying a model to text transformation engine (M2T). The output source code is refined by a WSN developer and validated by a domain and network experts in order to fulfill the target programming language constraints and specific middleware requirements.

A Model to Text (M2T) transformation was used to generate the source code for both gateway and sensor nodes. For gateway, the M2T transformation solution, proposed in (Cavalcante, 2014), was used to generate the RAMSES source code for gateway components in Go programming language. For sensor nodes, we implemented a M2T transformation (see Chapter 4) to the target sensor node platform Contiki (A. Dunkels, Grönvall, & Voigt, November 2004), a lightweight and flexible operating system for tiny-networked sensors, with support for dynamic loading.

3.6 Step RA-4: Reference Architecture Evaluation

In this Section we conduct the fourth step of the ProSA-RA process. In this step, ProSA-RA proposes the use of a checklist-based inspection approach named FERA (Framework for Evaluation of Reference Architectures) (J. Santos, Guesse, Gaister, Feitosa, & Nakagawa, 2013). This checklist corresponds to a list of questions that guide reviewers in detecting defects in documents related to reference architectures.

The main intention of using FERA is to validate if:

(i) The RA is adequately represented; i.e. if it provides general information (such as the potential risk, constraints, and scope).

(ii) The RA contains an adequate set of architectural viewpoints, view and models;

(iii) The documentation related to the RA contain important information; such as architectural decision, best practices and guidelines, polices and rules, international standards, and interfaces among modules;

(iv) The RA considers quality attributes important to its domain

(v) The RA can be easily instantiated

(vi) The RA could be changed, if necessary, in order to improve the reference architecture documentation.

Therefore, we followed the recommendation and used FERA to conduct an evaluation. Four evaluators (two specialists in the RA domain and two specialists in WSN domain) answered 93 questions (FERA checklist is available at http://146.164.247.214/wordpress/rawsn/ra- evaluation/).

As main results, 66.93% of the questions were answered as satisfactory or partially satisfactory, and 33.07% as not satisfactory. Summarizing, the results showed that the provided abstraction level is adequate for the RA purposes, the concepts underlying the RA are clearly explained and its detail level favors the RA understanding. Moreover, all modules of the RA are clearly identified, the relationships between these modules can be determined, and the runtime dependencies of these modules can be identified. Finally, the required hardware elements can be identified, the RA is in conformance and complete regarding domain requirements, and it addresses the key issues of the WSN domain. On the other hand, the main problems reported in the conducted inspection are the lack of explicit of the variability mechanisms of our RA and documentation regarding the threats for introducing the RA.

Based on these results, we complete our architectural description by using the RAModel in order to overcome the pointed drawbacks of the RA adoption. Considering all documentation of RAMSES presented in Section 3.5 of this Chapter we could analyze this architecture considering the elements proposed in RAModel. These elements are organized in four groups (detailed in Appendix A).

i) Domain

RAMSES addresses non-functional requirements related to quality attributes, such as fault-tolerance, adaptability and scalability. Thus, we can say that RAMSES presents information related to quality attributes or non-functional requirements. Besides that, RAMSES provides information about their WSN modules, it also uses IEEE 802.15.4.1 as a communication standard and, considering that RAMSES is supported by most of WSN platforms, we consider that the system compliance element and, the legislation, standard and regulation elements are partially addressed.

ii) Application

Throughout the documentation of RAMSES, *goals and needs* are addressed, since the RA proposes an autonomic computing method to manage WSN through the implementation of a hierarchical MAPE process. The *scope* of RAMSES also is presented, by supporting a framework for the implementation of self-adaptive WSN. RAMSES also defines a set of *functional requirements* as detailed in Section 3.4.1. Regarding *domain data*, i.e., data used by systems resulting from RAMSES and related directly to the domain is present (as detailed Table 9). In addition, data used in the application interfaces are generally established. Moreover. RAMSES restricts its implementation in hierarchical WSN networks, then we consider *constraint and limitations* elements addressed. However, the *risks* of its implementation could be deeeper explored.

iii) Infrastructure

RAMSES presents a fully detailed and layered architecture. Hence, the RA addresses information regarding the *general structure* of RAMSES, such as architectural styles and patterns. It therefore aggregates the *best practices* for developing an instance of the RA by proposing an automatic instantiation process provided by the formal specification of the RA. Throughout the documentation of RAMSES, a set of software elements can be identified. Thus, we can say that information about *software elements* has been addressed by RAMSES. It is also observed that information about *hardware elements* that compose a system. We can find elements, such as each type of sensor node device, where RAMSES explicitly presents how to manage these elements, in a general way. Hence, information about hardware elements is also presented in RAMSES.

iv) Crosscutting Elements

RAMSES explicitly establishes a set of standardized interfaces, including interfaces between its architectural layers that support hardware independence and the separation of concerns between the network management and the WSN application management. These interfaces are defined on a lower level of abstraction, on the source code level written in Pi-ADL. In addition, RAMSES provides the well-defined interfaces for each component of the RA. These interfaces refer to communication among different software components that are present in these applications. Thus, we can say that information related *internal communication* is present in RAMSES.

Regarding *external communication* and considering a WSN application as a unique system, this communication is established by RAMSES through Gateway Communication component (involving, for instance, communication with other WSN, internet, or other systems). Information related to external communication is therefore addressed by RAMSES. This architecture also presents a glossary containing the definitions of all mayor terms used, thereby establishing the domain terminology adopted by the architecture (Section 3.5.1.1). Thus, information about *domain terminology* used in the RA is contained in RAMSES. Throughout RAMSES documentation, mainly in Section 3.5.1, we addressed the main architectural decision when establishing the RA. Thus, we can say that *decision element* is addressed by RAMSES.

Through this analysis, we identified which elements of RAModel are present in RAMSES and which are not. We consider RAMSES a complete architecture, according to RAModel. However, some elements, such as risk, laws and current regulations could be added to this RA.

3.7 Final Considerations

In this Chapter, we presented RAMSES, a reference architecture of a self-adaptive middleware for WSN. To build RAMSES, we conduct each step of ProSA-RA. The first step was performed an investigation of useful information sources to design the RA. In this step, a set of requirements for self-adaptation in WSN and the different approaches used to build an autonomic WSN application were identified. In the second step, the architectural requirements for self-adaptive WSN applications and their main quality attributes were obtained. They were defined from a primary studies analysis, selected in the first step, where the main related works of reference architectures for autonomous systems and WSN applications were analyzed. In third step, we design RAMSES using RAModel as a framework and modeling the architecture through the following architectural viewpoints: crosscutting viewpoint, runtime viewpoint and deployment viewpoint. Pi-ADL specification of runtime view and deployment view were applied in order to explicitly define the dynamic behavior of WSN networks, a key issue in this type of systems.

In fourth and final step for ProSA-RA was performed an evaluation of RAMSES through FERA, a framework for evaluation RAs. It is important to note that RAMSES contains characteristics of a research-driven RA, in which details of development of some modules needs to be investigated. In the context of this work, in addition to FERA evaluation, in order to identify related failure to omission, ambiguity, inconsistency and incorrect information, the evaluation of RAMSES was also carried out through a case study, detailed in Section 4.

CHAPTER 4: Proof of Concept – SAMSON

4.1 Initial Considerations

In this Chapter, we present the evaluation of the proposed approach through a proof of concept. We assess through a series of adaptation scenarios the execution of an instance of RAMSES, named SAMSON (Self-Adaptive Middleware for wireless SensOr Networks), to verify if it achieves the requirements of an adaptive system by performing the MAPE-K model. Thus, in this evaluation we executed four different scenarios to verify the following self-* properties:

- Self-configuration (Scenario 1) to check the mechanisms that enable the WSN to adapt itself to the environment, changing its behavior as specified through high-level policies, by dynamically altering values of parameters according to the changing
- Self-healing (Scenarios 2 and 3) to verify the ability of SAMSON adaptive mechanism to react to node failures (both ordinary nodes and cluster heads) at runtime.
- Self- optimization (Scenario 4) to verify the capabilities of the autonomic management process of extending the network operational lifetime by applying actions defined in adaptation policies.

Furthermore, we evaluate the benefits of RAMSES instantiation in terms of implementation effort (quantified as lines of code) when using the proposed model-driven transformations to generate a middleware instance (SAMSON) from the reference architecture.

4.2 Instantiation Process

Software architects, who are responsible for the RA instantiation, often do not have the knowledge about programming, especially on coding for WSN platforms. In this context, model transformations can be used to enable a mapping between models from different abstraction levels, namely architecture instance, into platform-specific code. Thus, to fulfill the gap between the architecture specification and platform implementation, in this Section we describe how an architectural description (architecture instance) created using RAMSES can be mapped into an executable source code for a target platform.

A model transformation is the process to generate text or documentation from a source model. To run a model-to-text transformation and generate code for a target platform, some software artifacts are mandatory: (i) the source model, (ii) its respective meta-model, (iii) the mappings between meta-model elements of the source model and code elements of the target platform respecting the syntax rules of the native programming language.

Following the process presented in Section 3.5.4.3, the specified RA (in Pi-ADL language) is considered the source meta-model and the software architect job is to specify the concrete architecture, or the source model (also in Pi-ADL language). Thus, it is only necessary to select the target platform and its respective transformation rules that were already developed. In this context, potentially any platform that uses a programming language defined under a well-defined set of syntax rules can be used as target platform.

In this proof of concept, we considered Contiki as the target WSN platform for code generation. In the remainder of this section we specify the necessary steps to develop the model transformation rules for Contiki platform. This proof of concept can be used as reference to specify M2T transformations for other platforms.

The first step in the process of generating the middleware source code is to perform the logical mapping between components of SML layer of RAMSES (which conforms to pi-ADL meta-model) and the coding elements of the target sensor platform (see Table 14 for a summary of the mapping for Contiki).

Table 14. Correspondences between RAMSES and Contiki

RAMSES (Pi-ADL)	***Contiki (C)***
Component	Process
Connector	Process
Behavior	Main process implementation
Connection	Send/Receive data process
Declaration of Connections	Send/Receive library includes
Architecture	Main process declaration
Basic types (except Any)	Primitive C types
Unobservable elements	Empty Interface
Any type	Empty Body

- The Component, Connector, and Behavior elements of RAMSES represent, respectively, a unit of computation of a system, an interconnection among components to support their interactions, and the internal behavior of the component. Components and connectors are mapped to Contiki functions and processes. All Contiki programs are processes. A process is a piece of code that is executed regularly by the Contiki system and run until an event is triggered, such as a timer firing or the occurrence of an external event. The behavior element is represented in Contiki as the sequence of actions that are specified in the main process, including timers, event triggers, and others.

- The Connection and the Declaration of Connections elements of RAMSES provide communications channels between two components. A component can send/receive values via connections. A connection is represented in Contiki as the invocation of a function to send/receive data to/from a remote component of the network. As there are different protocols to send data through the network, the declaration of connections establishes the mapping to the chosen protocol.

- The Architecture element of RAMSES encompasses the composition of component and connector instances. The architecture is represented in Contiki by the specification of the main process and of the list of processes that will auto-start when the application starts.

- The Basic types, Unobservable elements, and Any type of RAMSES express, respectively, atomic values, the capability to enact an action invisibility and a generic type. As Contiki is implemented in C language, all C's basic types are present, thus there is a direct map between RAMSES Basic types to Contiki types. Moreover, the unobservable elements and any type can be mapped to the declaration of empty functions or the use of the void type in Contiki, respectively.

Finally, once the transformation rules are specified, the M2T transformation can be executed as many times as necessary to generate the code of the source model produced by the software architect. To perform the transformation from the architectural description to source code, an M2T transformation engine is necessary. We used Xtext to validate the pi-ADL architectural descriptions developed using pi-ADL. Therefore, we used the Xtend tool to map an architectural description into Contiki code following the mapping presented in Table 14. SAMSON source code is available at http://146.164.247.214/wordpress/rawsn/.

4.3 Evaluation Methodology

We performed this proof of concept to evaluate SAMSON operation in four different scenarios. We simulated a WSN with two clusters, each one containing a set of thirty (30) nodes with different sensing capabilities (temperature and luminosity). Moreover, for purposes of comparison, in addition to the regular implementation of SAMSON (S1) we instantiated another version of the middleware (S2) without the dynamic adaptation features, where the MAPE-K process initialized by the Sensor Manager component was deactivated.

Both instances (S1 and S2) were executed with the same network configuration. This proof of concept was executed using Cooja Simulator on virtual SKY nodes. All scenarios defined for this proof of concept specify a WSN application to collect temperature and luminosity measures every 15 seconds and the application must remain active for one week.

4.3.1 Scenario 1: Self-configuration:

For this scenario, in a pre-deployment phase each node was statically configured to associate itself to a predefined cluster head. All clusters were previously grouped based on the distance between nodes and statically defined by the network administrator in a pre-deployment phase.

When the middleware starts running, the selection of cluster heads is performed by SAMSON (when needed) as part of the autonomic management of the network and based on the residual energy of nodes. So, at runtime all nodes of a cluster are dynamically reconfigured to associate themselves to the new selected cluster head.

4.3.2 Scenario 2: Self-healing (node failure)

In this scenario, after one day of simulation, node failures were manually introduced (by turning off nodes) until the network coverage or connectivity were no longer guaranteed. The goal was to verify the self-healing property through the adaptation by parameter. The coverage is guaranteed when a given area is monitored by at least k different active sensors (D. L. Li, H, 2009), and connectivity is guaranteed when the messages sent by all active sensor nodes can reach the sink (D. L. Li, H, 2009).

When any of these events is detected, the MAPE-K process of the node manager (CH) reads the activated adaptation policy in order to plan (activation of redundant nodes that are inactive) and execute an action (sending of a configuration message to nodes of the cluster) to reconfigure the cluster and keep a minimal number of active nodes. There are sophisticated heuristic-based algorithms for selecting active nodes, as in (F. Delicato, Protti, Pirmez, & Rezende, 2006). In this experiment, we applied a simple approach, based on the energy level of nodes, for activating them until achieving, when possible, 80% of active nodes per cluster. The minimal percentage of active nodes required by the defined policy is 60% (18 nodes per cluster).

4.3.3 Scenario 3: Self-healing (cluster head failure).

In this scenario, we adopted a proactive policy for replacing a low energy cluster head. The new manager must dynamically load specific components of cluster heads, in order to perform the MAPE-K process inside the cluster.

The goal of this scenario is to verify the self-healing property thought the adaptation by dynamic load of components. The adaptation policy is activated whenever the residual energy of the current cluster head is below 30% (value reached after about 60 hours of simulation). When this policy is activated, the CH broadcasts a configuration message to the cluster nodes advertising the ID of the new CH. When a node with the same ID receives that message, then this node dynamically loads the specific components to become the new CH.

4.3.4 Scenario 4: Self-optimization:

In this scenario, we defined an application to periodically monitor the temperature of a given area and notify users whenever the sensed data reaches a value higher than 15oC (this condition is simulated to happen every 12 hours per day).

To optimize the energy consumption of nodes, an adaptation policy was defined to decrease the data delivery rate of a node (from 15 sec to 60 sec) if the collected data is frequently under a given threshold. Thus, reducing the number of messages and increasing the system lifetime, since in WSNs the communication consumes most of the energy.

4.4 Analysis

Table 15 shows the collected data regarding energy consumption of nodes (EC) and the average number of transmitted messages (TM) sent by ordinary nodes (ON) and received by each cluster head (CH) for each scenario. These data indicate, in average, that the energy consumption of nodes is more efficiently managed when the adaptation strategy employed by SAMSON is used (S1).

Table 15. SAMSON Evaluation

	Scenario 2		Scenario 3		Scenario 4	
	S1	S2	S1	S2	S1	S2
EC	646.5 J	92.36 J	646.5 J	646.5 J	404 J	646.5 J
TM	1209600_{CH}	172800_{CH}	432000_{CH1} 777600_{CH2}	1209600_{CH1}	25200_{ON}	40320_{ON}

In scenario 2, the adaptive mechanism allowed the recovery of the network coverage and assured that the network was kept alive, extending the network lifetime for all the simulation time (one week). Such mechanism was autonomously executed by the CH without interrupting the network operation or human interference. On the other hand, after the coverage fault in S2, the network was not able to recover its operation, and the monitored area was uncovered, thus incurring in a failure in meeting application requirements. Therefore, the network stopped to provide its functionality and was declared inoperative after one day of operation in simulation time.

In scenario 3, the residual energy of the cluster head (CH1) is depleting, thus compromising its lifetime and the cluster functionality. If a cluster head fails, all area covered by the cluster will be unmonitored and disconnected to the rest of the network. In this case, a reconfiguration of the cluster topology is required and a selection of a new cluster is a mandatory operation. Thus, in S1, when the problematic cluster achieves a predetermined residual energy, an adaptation policy is activated to select a new leader (CH2), based on the amount of residual energy of nodes.

When the chosen node receives a configuration messages with instruction for turning itself in a CH, the exclusive components for a node manager (responsible for performing the MAPE-K) are dynamically loaded. Such result is very important for the WSN domain, as memory is a valuable resource that is often very restrict in the current hardware.

Therefore, Table 16 shows that by applying dynamic loading of components, ordinary nodes (managed nodes) consume less memory than cluster heads. In addition, RAM consumption was 31% below the memory capacity of SKY nodes, since

SKY platform contains 10 Kb of RAM and 48 KB of flash memory. Thus, SAMSON memory consumption is aligned with current WSN hardware platforms. Regarding SAMSON image size, in ordinary nodes it is smaller (- 10%) than in cluster heads, demonstrating that only the necessary components are loaded according to the node responsibility (manager/managed), thus saving the valuable resources of nodes.

Table 16. SAMSON Overhead

	Node Manager	**Managed Node**
RAM	6.9 Kb	6.6 Kb
Image Size	29.3 Kb	26.3 Kb

In scenario 4, the average of energy consumption of nodes supported by S1 (404 J) was approximately 37% lower than a network running without the adaptation support (646.5 J). Such result can be explained by the policy applied by S1 to reduce the number of transmitted messages. Thus, ordinary nodes that measure values below the predefined threshold were configured to send messages every 60 seconds (instead of every 15 seconds), thereby reflecting into less energy usage by reducing the number of message transmissions from 40320 to 25200, for each ordinary node.

Finally, this evaluation process also included an analysis of the middleware instantiation process with the purpose of evaluating its implementation effort from the RA. We analyzed the number of lines of code as a metric to evaluate: (i) how simple it is to instantiate RAMSES and generate SAMSON using the M2T engine and, (ii) directly with Contiki programming. RAMSES instantiation method (supported by a formal ADL specification – pi-ADL– and a M2T transformation engine) makes it simple to implement SAMSON, by requiring 128 lines of codes against 453 for Contiki programming. Thus, the developer is only concerned in the implementation of predefined functions and the architectural decisions are inherited from the RA.

4.5 Related self-adaptive middleware instances for WSN

In this section, SAMSON is compared with self-adaptive middleware systems found in literature.

DISON (Minh et al., 2013) adopts a Policy-based reasoning (PBR) approach to provide a generic management system for WSN. Its main goal is to allow sensor nodes to adapt autonomously to changes in application requirements and network resources. A multilevel management mechanism is used where every sensor nodes is empowered to participate in the management process at different levels according to their resources. A management function in DISON aims to monitor network resources, detecting faults, and reconfiguring nodes operation.

Our middleware also allows individual sensor nodes to perform management functions locally based on adaptation policies. In SAMSON, these functions are responsibility of the SML layer. However, the main difference concerns SAMSON NML layer, located outside the network. Such layer can work as a central manager to maintain operations from a global perspective. Thus, the self-adaptation process is performed in two levels of abstraction. One level is performed in the SML layer and aims to provide more localized management decisions. The second level is performed in the NML layer and allows global decisions involving the entire WSN. Such distribution of responsibilities favors energy efficiency while at the same time allows potentially optimal decisions (not possible with a partial view of the network). Moreover, the management functions applied in our SAMSON are based on the MAPE-K model, which provides well-grounded guidelines for specifying architectural aspects of autonomic systems.

(Jemal & Halima, June, 2013) describe a QoS-Driven Self-Adaptive Architecture for WSN. This architecture, based on feedback control loop (FCL), adopts MAPE-K. Generally, WSN middleware systems that apply this approach use hierarchical networks typically defined with three levels: sensor nodes, cluster head and base station levels. Accordingly, in (Jemal & Halima, June, 2013), three different levels of adaptation are defined. Cluster heads perform actions like message filtering, adjustment of frequency, and optimization of the transmission signal frequency. Base station provides the actions of global network reorganization, including components deployment, redeployment, undeployment, activation and deactivation. SAMSON is also based on MAPE-K model with different levels of adaptation, including a global perspective operating outside the network (NML). Its adaptation process operates via configuration messages and, when needed, we applied a dynamic run-time linking and loading process.

Starfish (Bourdenas & Sloman, 2010; Bourdenas et al., Oct, 2011) is a PBR system proposed for specifying and dynamically managing policies in sensor nodes. They include a policy system to specify dynamic adaptation. Such system is named Finger 2, an extension of Finger. Another PBL middleware based on Finger is SMART507 (S. et al., 2013). Its main goal is to automatically generate policies which introduce swarm intelligence into the WSN management. In SAMSON, the structure of adaptation policies is defined in Pi-ADL, at the architectural level. Hence, it is not necessary to learn another language for specifying policies. In SAMSON, users are able for managing policies via the Adaptation Policies Manager component.

Table 17 summarizes the highlighted characteristics of these self-adaptive middleware systems.

Table 17. Characteristics of self-adaptive middleware for WSN

Reference	Develop. Approach	Deploy. Approach	RM-based/ RA-based	ADL spec.	Self-adaptation process	Evaluation
DISON (Cao et al., Sep, 2014; Minh et al., 2013)	PBR	Distributed	-	Blocks	Structured messages	TinyOS and SENSE sim.
SMART507 (S. et al., 2013)	PBR	Hybrid	Finger/1,2 extension	Blocks	Structured messages	Contiki OS/Cooja
Ahmed (Jemal & Halima, June, 2013)	FCL, PBR	Distributed	MAPE	Blocks	Deploy, Structured messages	AZEM sim.
Starfish (Bourdenas & Sloman, 2010; Bourdenas et al., Oct, 2011)	PBR	Distributed	Finger2	Blocks	Virtual machine	TinyOS 2.x
SAMSON	FCL, PBR	Hybrid	MAPE/ RAMSES	Pi-ADL, UML	Dynamic Loading, Structured Messages	Contiki OS/Cooja sim.

The first column “Develop. Approach” shows the type of development approach followed for each middleware. We noticed that all FCL-based (Feedback control loop) systems are supported by some kind of policy management reasoning (PBR). Starfish depends on virtual machines, these systems can somewhat simplify the WSN management tasks, but they still require a lot of management effort to specify the policies or writing management code. Scripts are significantly smaller to transmit than binary images and do not require rebooting nodes, but operational overheads of a full virtual machine can be significant. SMART507 improve these drawbacks by implementing a management server equipped with a smart policy generation subsystem. After the generation process, management server assigns the policies to node clients through the sensor network. However, the node client needs to run a policy interpreter. Finally, both DISON and SAMSON introduce a policy data model aiming to attend the limited resources of WSN.

The column “Deploy. Approach” shows the deployment approaches applied for each middleware. A distributed approach enables to apply a robust autonomic process, the management overhead is reduced and the complexity and coordination issues can be addressed by applying an hybrid approach deploying a central manager with a complete view of network. SAMSON and SMART507 follow hybrid approaches.

The third column shows which middleware systems are based on reference architectures (RA-based) or reference models (RM-based). Some PBR systems are based on Finger and FCL are based on MAPE model. SAMSON is also a FCL system based on a RA. RAMSES supports SAMSON in a systematic and principled way. This support relies on design pattern and well-defined architectural styles specific for autonomic WSN. SAMSON architecture is also more comprehensive due to it contains a formal runtime specification (pi-ADL). As showed in the fourth column (ADL spec.), our middleware is the unique among the related systems that uses an ADL for specifying its architecture. It contributes for minimizing the development and maintenance cost.

The fifth column (Adaptation Process) shows the mechanisms that middleware systems use for adaptation process. We noticed that all systems define a structured scheme of messages. These messages are used for collecting and disseminating policy-based messages. Furthermore, we found four types of mechanisms used for applying self-adaptation in WSN. These mechanisms are based on: (i) interpreter of policies, (ii) redeploying of applications, (iii) virtual machines and, (iv) dynamic loading. Dynamic loading process is more efficient in terms of WSN resource management (A. F. Dunkels, N; Eriksson, J; Voigt, T, November, 2006). Finally, the last column shows that TinyOS-based and Contiki OS were the sensor platforms most used for evaluating the afore-mentioned WSN middleware systems.

4.6 Final Considerations

In this chapter, we aimed at leveraging the adoption of RAMSES, by automatizing the mapping process between a RA and its instantiation in a concrete middleware implementation. Following the presented approach, software architects can design self- adaptive WSNs at a high level of abstraction, independent of a specific platform, and generate, through an MDD approach, the target middleware code that meets the requirements raised and the chosen platform. This approach reduces the gap between the RA specification and its implementation.

An important benefit of RAMSES is to reduce the effort in developing and maintaining WSN applications. The literature points out that complexity in the development of WSN application is a critical issue, even with the latest advances in the area. With RAMSES, will be possible to generate the middleware code for different WSN platforms. Thus, the proposed approach contributes to deal with heterogeneity issues, and promotes the integration of several WSN platforms.

CHAPTER 5: Comparative Analysis

5.1 Initial Considerations

In this Chapter, we present a comparative analysis between RAMSES and related reference architectures, and we also evaluate the completeness of those RAs.

In order to obtain the RAs proposed in the self-adaptive WSN domain, in Section 3.3.3 we conducted two systematic literature reviews to identify the most important RAs in WSN and autonomic computing domains. As result we discovered 10 reference architectures proposed for WSN domain and 25 for autonomic computing. Based on RAMSES requirements (see Section 3.4.1), we classified 6 of these reference architectures as related to self-adaptive WSN domain, and that are presented in Table 18.

Table 18. Reference Architectures for self-adaptive WSNs

ID	Reference	Title	Related to
RAMSES	Our proposal	RAMSES	WSN, AC
RA1	(Casola et al., July, 2009)	SeNsIM	WSN
RA2	(ISO/IEC, 2014)	SNRA	WSN
RA3	(Gluhak & al., June, 2006)	e-SENSE	WSN
RA4	(Cherif et al., 2014)	ReMoSSa	AC
RA5	(Villegas et al., 2012)	DYNAMICO	AC
RA6	(IBM, 2005)	ACRA	AC

Moreover, in order to analyse the completeness of the related reference architectures for self-adaptive middleware for WSN, we used the RAModel, as proposed by (Garces, Ampatzoglou, Avgeriou, & Nakagawa, 2015), to present (i) an evaluation of the completeness of reference architetures with a focus on its elements; and (ii) a more focused study, in the sense that it is stricktly related to self-adaptive middleware for WSN and not in the entire WSN or autonomic computing domains.

In this perspective, Section 5.2 analyses if the identified requirements for self-adaptation in WSN, presented in section 3.4.1, are addressed by the selected reference architectures. Section 5.3 describes the elements defined by each RA, and we analyse their level of completeness.

5.2 Requirements for self-adaptation in WSN

Requirements for self-adaptation in WSN have been established in section 3.4.1. Table 19 identifies which ones are addressed by the selected RAs.

Req.	RAMSES	RA1	RA2	RA3	RA4	RA5	RA6
Req-A	X	X	X	X	X		
Req-B	X	X	X	X	X		
Req-C	X		X	X	X	X	X
Req-D	X				X	X	X
Req-E	X	X	X				

- **Req-A**: *A WSN should contain one or more sink nodes or base stations, endowed with a wireless communication interface and one interface with a Gateway.* This requirement is addressed in reference architectures RAMSES, RA1, RA2, RA3 and RA4. RA1, RA2 and RA3 are designed for connecting a WSN with external networks; RA-4 partially attends this requirements, however, it may be adapted to design a WSN once it was projected for attending distributed systems.
- **Req-B**: *The RA must enable the definition of a set of sensing-based applications.* This requirement is also addressed in reference architectures RAMSES, RA1, RA2, RA3, RA4. These RAs offer a set of services for defining specific applications for WSN.
- **Req-C**: *The RA must enable the definition of a set of high-level goals.* RAMSES, RA2, RA3, RA4, RA5 and RA6. This requirement is addressed with support of the Adaptation Manager component in RAMSES where a set of adaptation policies are defined, in RA2, this requirement is supported by the Rule engine, a Policy Management component is defined in RA3. A definition of system's objectives is proposed in RA4. RA5 specifies the Control objective manager and RA6 guides the uses of high-level goal to support adaptation planning.
- **Req-D**: *The RA must enable the self-management of the network by defining a set of software components (middleware platform) responsible for managing the WSN.* RA4, RA5 and RA6 define a set of components based on MAPE-K

control loops. RAMSES defines these components at the middleware level. A future work of RA4 aims to develop a set of generic self-adaptable middleware.

- **Req-E**: *The RA must consider a hierarchical topology for the WSN organization.* This requirement is addressed in reference architectures RAMSES, RA1, RA2. RA3 does not provide enough information to determine whether the architecture supports this requirement.

5.3 Analysis of Completeness

Table 20 presents the RAModel elements (as described in Appendix A), in order to undertand which information is contained, and which is missing, in the definitions of RAs for self-adaptive middleware for WSN. Following, we present a brief discussion of the elements defined by each RA. It is worth to highlight that a complete evaluation of RAModel elements of RAMSES was detailed in Section 3.6.

Table 19. RAModel elements defined by each RA

Group	**Element**	**RAMSES**	**RA1**	**RA2**	**RA3**	**RA4**	**RA5**	**RA6**
Domain	Legislation, standards, and regulations			X				X
	Quality attributes	X		X			X	X
	System compliance			X				
Applications	Constraints	X						
	Domain data	X		X		X	X	X
	Functional requirements	X		X	X		X	X
	Goals & Needs	X	X	X	X	X	X	X
	Limitations	X		X				
	Risks							
	Scope	X	X	X				X
Infrastructure	Best practices and guidelines	X	X		X	X	X	X
	General structure	X	X	X	X	X	X	X
	Hardware elements	X	X	X	X			X
	Software elements	X	X	X	X	X	X	X
Crosscuting elements	Decisions	X	X			X		X
	Domain terminology	X		X		X	X	X
	External communication	X	X	X	X		X	X
	Internal communication	X	X	X	X	X	X	X

- **RA1 – SeNsIM:** This reference architecture for Sensor Networks Integration and Management, proposed by (Casola et al., July, 2009), allows a single unified view of sensor systems in order to satisfy user or application queries, and enables a flexible deployment and interconnection between them, even if located in different places. The architecture is based on the Mediator/Wrapper paradigm in order to provide a layered and scalable architecture. For designing phase, authors have used a box diagram in order to model the main components of the architecture and for the implementing phase, they have used Java programming and TinyDB.

SeNsIM details documentation about the rationale obtained during the reference architecture development, mainly regarded to design patterns and architectural styles. Hardware and software elements are also defined. The general structure of the RA is made of four logical layers (Figure 41): (i) an application or user layer to submit queries and elaborate the retrieved data; (ii) a mediator layer to format and forward queries to specific wrappers; (iii) a wrapper layer to extract and manage network information and data; (iv) the sensor system layer with or without a specific middleware or operating system.

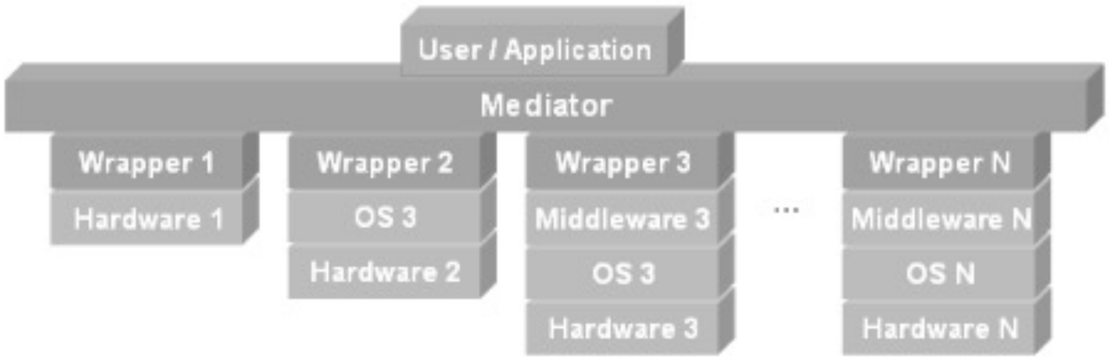


Figure 41. Abstraction layers of SeNsIM (Casola et al., July, 2009)

SeNsIM partially details crosscutting element, they do not report all details of domain terminology. External communication is detailed through network interfaces, and internal communication centralized by the mediator layer.

SeNsIM reference architecture does not describe domain elements hindering the understanding of laws, standards, regulations that SeNsIM systems must address. There is a lack of quality attributes and it makes difficult the architectural validation. Additionally, it does not offer a clear understanding about the limitation of its use, the set of systems that could be developed using it, the risk of using it, and the functional requirements that are important when software systems are evaluated by stakeholders. It lacks guidelines defining well-experimented practices to develop SeNsIM systems.

- **RA2 – SNRA:** The Sensor Network Reference Architecture (SNRA), proposed by (-. ISO/IEC, 2014), describes generic and generalized sensor network services. SNRA is an architectural representation of sensor network entities' (e.g., sensor nodes, gateway nodes, and other hardware in the node) functions, activities, and roles through operation layer and interoperable interfaces to provide the sensor network developers and implementers with reusable sensor network architecture for their target applications. SNRA depicts (in Figure 42) the general operational, functional and technical characteristics of WSNs.

  SNRA applies a multi-layered style, defining four layers: (i) sensor node hardware layer, that contains sensors, actuators, and power supply components; (ii) basic function layer responsible to process the data acquired by sensors, storage such data, and communicate sensor nodes; (iii) service layer that consists of services specific to domain and common services; (iv) application layer in where are located user applications; and (v) Cross-layer that comprises components to manage and monitoring different characteristics of the sensor network system.

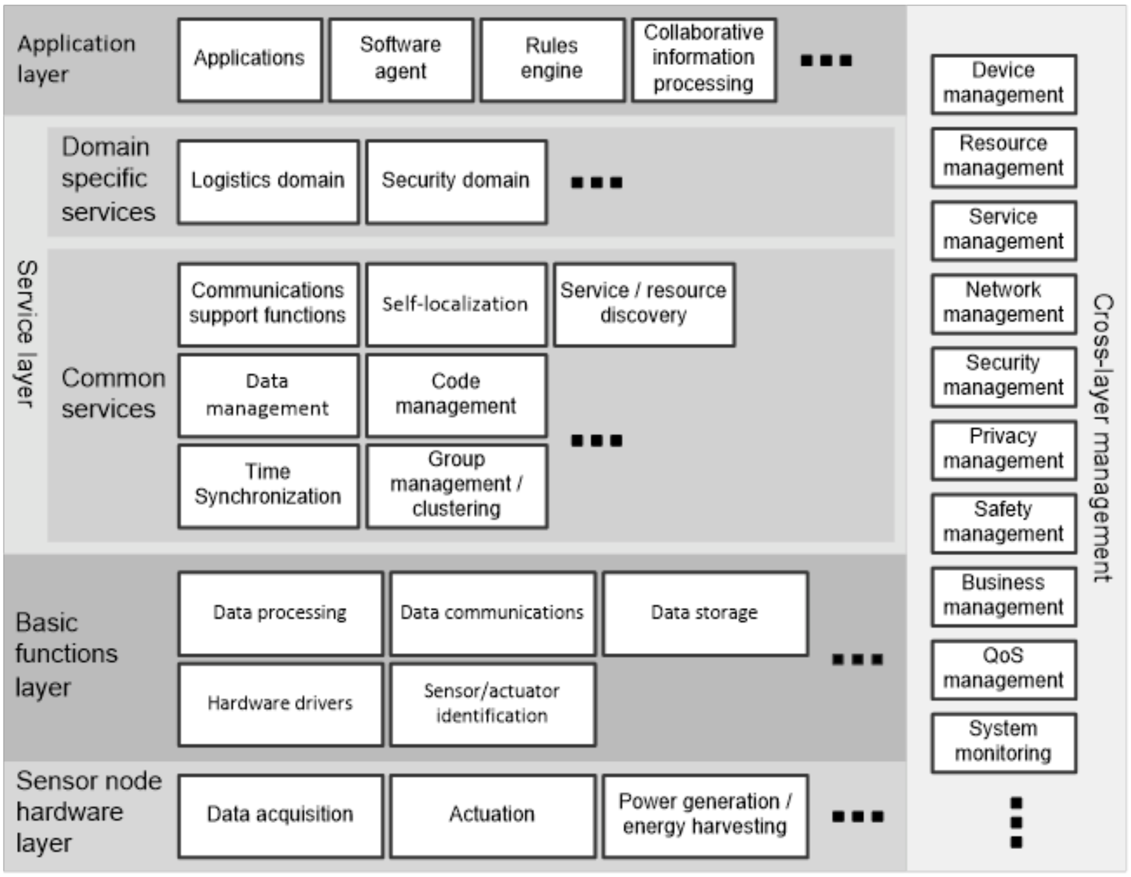


Figure 42. Sensor Network Reference Architecture (-. ISO/IEC, 2014)

SNRA details documentation related legislation, standards, and regulations. SNRA describes some laws related to privacy of sensor networks and generic security services where the accountability services are highlighted. Accountability means that users can rely on the information provided, it implies that all actions are traceable. However, this requires ethical standards and legal regulations. SNRA also considers the following quality attributes: security and privacy, robustness, scalability, quality of service, heterogeneity, mobility and power management.

Application and crosscutting elements are widely described. Regarding infrastructure elements. SNRA details the structure of the architecture. However, there is a lack of means (architectural viewpoints) that allow verifying that SNRA systems are created following best practices and guidelines definition for its use and instantiation.

- **RA3 - e-SENSE:** The e-SENSE Project (Reference Model for Sensor Networks in B3G Mobile Communication Systems) (Gluhak & al., June, 2006) aims to provide the enabling technology required to capture the desired ambient intelligence surrounding the users of services and service related objects through a WSN environment for B3G mobile communication systems. e-SENSE defines the elementary building blocks of the architecture and identifies the communication interfaces between those blocks.

  The ambient intelligence is captured with sensor nodes, which are the elementary building blocks of the WSN. Based on the application spaces envisioned in e-SENSE, the reference model considers various types of WSN architectural views. The e-SENSE reference model aims to provide an infrastructure where information sensed by nodes is processed in a totally distributed fashion and, if necessary, the result is transmitted to actuating nodes and/or to the fixed infrastructure by the means of a relaying gateway. Figure 43 depicts an overview of the distributed processing middleware architecture.

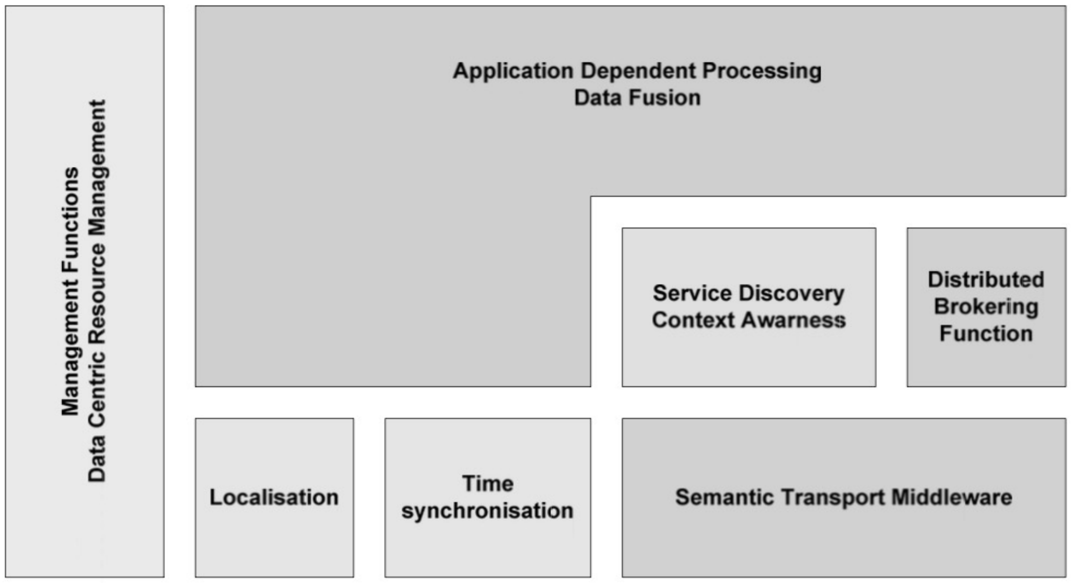


Figure 43. Reference model for the distributed processing middleware of e-SENSE (Gluhak & al., June, 2006)

Infrastructure and communication elements are detailed in e-SENSE reference architecture. The general structure is described in high-level and low-level of abstraction, the gateway of e-SENSE ensures required connectivity to facilitate the exchange of middleware messages.

e-SENSE documentation lacks information about its risks, limitations and quality attributes at using either as a whole or parts of it. Quality attributes, domain data, risk, and limitations are not defined.

- **RA4 – ReMoSSa:** This is a formal reference model for specifying self-adaptive Service-Based Applications. ReMoSSA integrates self-adaptation mechanisms and strategies to provide autonomic and adaptable services. It provides a dynamic monitoring and dynamic adaptation in the design phase. ReMoSSA reduces the cost and the effort of maintenance. ReMoSSA was inspired by FORMS model (Cherif et al., 2014) and the automated element proposed by IBM researchers (IBM, 2005). ReMoSSA can be used to check if the dynamic monitoring and the dynamic adaptation are being considered in the designs.

  ReMoSSA is divided into four main functionalities of the MAPE-K loop. ReMoSSA aims to incorporate various points of view into a unifying reference model. The strength of ReMoSSA model includes three mechanisms; Reflection, MAPE-K, and self-adaptive strategies pattern. ReMoSSA emphasizes the visibility of formal representation (based in Z language). ReMoSSA contains a set of relationships between the entities. It constitutes a guide to design self-adaptive SOA applications.

  Figure 44 shows an overview of ReMoSSA model which extending the primitives of FORMS (D. Weyns et al., 2010a) and adding the new elements required to support dynamic adaptation. With this model we can dynamically add or move conditions, planning strategies, or adaptation actions to modify the way the autonomic behavior is implemented in the application.

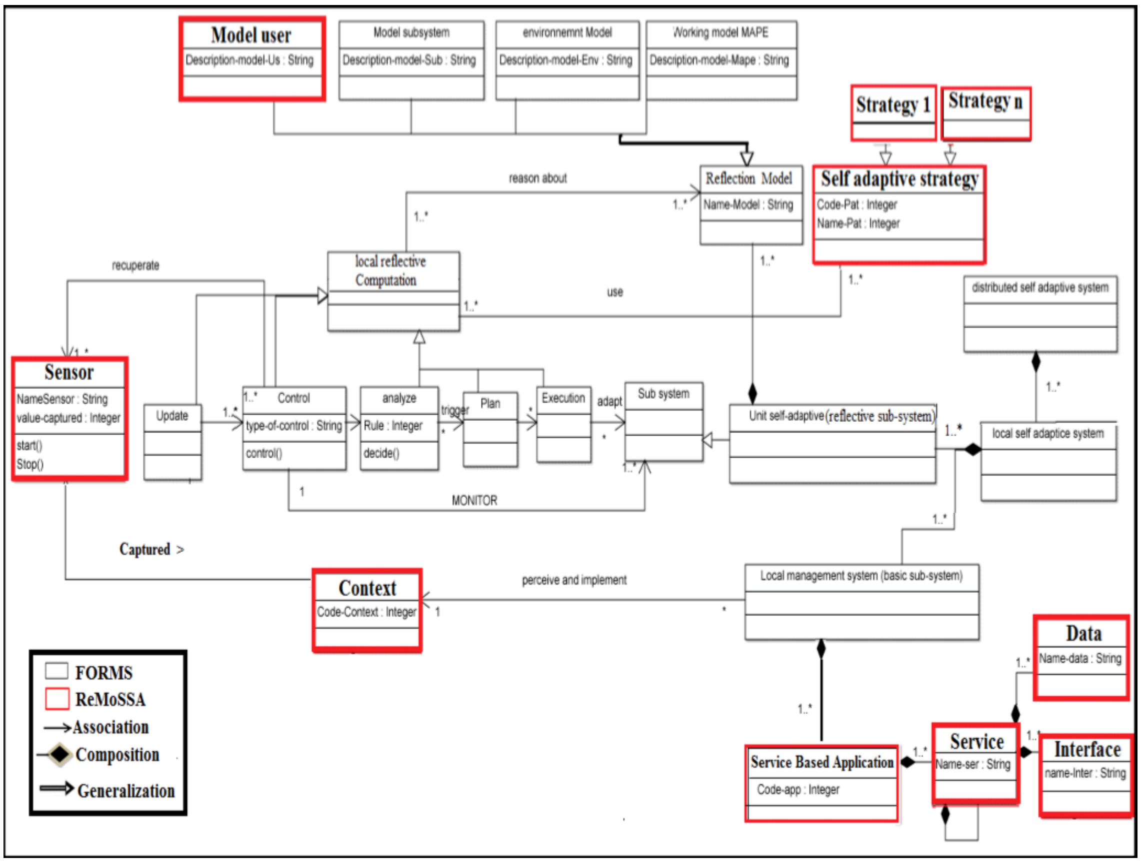


Figure 44. ReMoSSA reference model. Source: (Cherif et al., 2014)

ReMoSSA uses self-adaptation mechanisms like reflection mechanisms that can be used to adapt the behavior of applications dynamically. A reflection mechanism provides the ability for the application to observe and to modify its computation.

ReMoSSA establishes a set of interfaces that support internal communication between components. However, there is a lack of communication interfaces to communicate ReMoSSA with external networks. ReMoSSA documentation also lacks information about its risks, limitations and quality attributes.

- **RA5 – DYNAMICO:** A Reference Model for Context-Based Self-Adaptation (DYNAMO) (Cherif et al., 2014) is based on feedback control with explicit functional elements and corresponding interactions to control dynamic adaptation. These are the MAPE-K loop elements.
The separation of concerns is a key aspect of this model, which defines three

subsystems to achieve self-adaptation (i) Control objective manager: Manages the target system’s purpose in terms of its control objectives, according to the policies given by administrators. (ii) Context manager: Responsible for maintaining the pertinence and relevance of the context monitoring infrastructure with respect to the target system under changing conditions of execution; and (iii) Adaptation mechanism: Responsible for the adaptive actions over the target system according to the evaluation of its behavior. Figure 45 shows the general representation of DYNAMO.

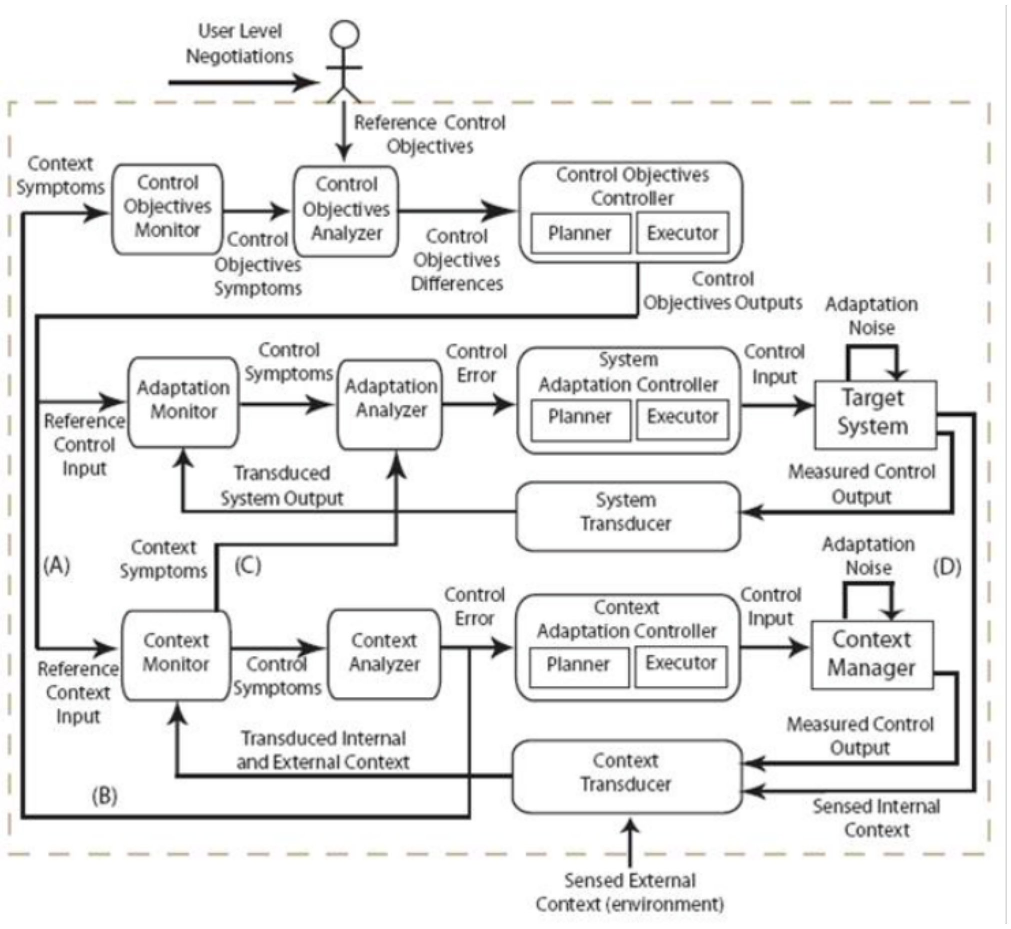


Figure 45. Reference model for DYNAMO. Source: (Cherif et al., 2014)

DYNAMO lacks techniques to evaluate system compliance to legislation, standards, and regulations. Quality attributes are addressed with the support of Control objective manager and domain terminology element is supported by the Smarter Context ontology.

DYNAMO does not specify its constraints and risks at using this RA. At the infrastructure level, there is a lack of hardware elements.

DYNAMO establishes interfaces that support internal communication between MAPE-K components. External communication is also supported, where a Semantic Web inference rules are defined as part of the Smarter-Context ontology.

- **RA6 – ACRA:** The Autonomic Computing Reference Architecture (ACRA) (IBM, 2005) consists of three parts: a set of architectural elements for constructing autonomic systems, patterns for using these elements in a system context, and interface and data interchange specifications that facilitate integration.

  As shown in Figure 46, ACRA provides a basic systems management topology that includes a hierarchical set of managers which manage a set of resources. The orchestrating managers control the management operations of the resource managers, and the resource managers provide the management support for a set of resources. Both types of managers may implement autonomic manager capabilities and typically support user interaction through one or more manual manager elements. The managers might also access management data from one or more knowledge sources.

  The central component in the ACRA is the autonomic manager (shown in Figure 3). It automates certain management functions and externalizes these functions according to the behavior defined by management standards.

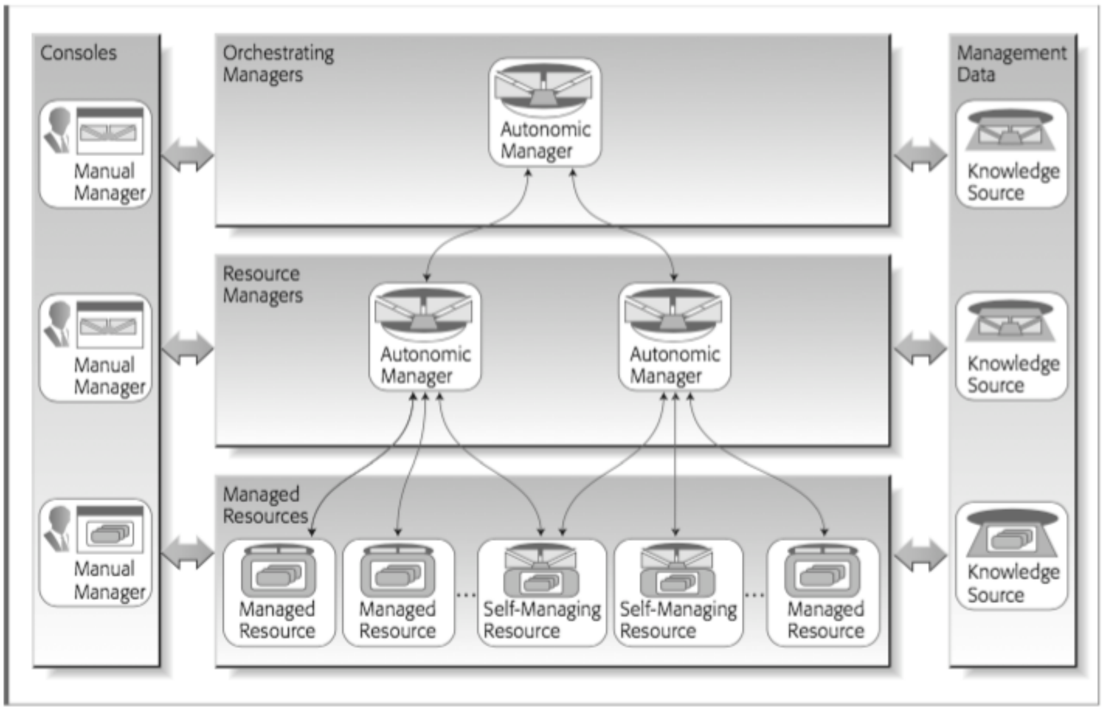


Figure 46. Autonomic computing referece architecture (ACRA). Source: (IBM, 2005)

ACRA is based on the IBM blueprint. Thus it details documentation related legislation, standards, and regulations. ACRA identifies relevant existing computing industry standards related to autonomic computing.

This architecture does not prescribe a particular management protocol or instrumentation technology because the architecture needs to work with the various computing technologies and standards that exist in the industry today.

Infrastructure and crosscutting elements are detailed in ACRA reference architecture. However, it does not offer documentation about the risk of use it and its limitations.

5.4 Final Considerations

In this chapter, a comparative analysis between RAMSES and related works was presented. Moreover, we evaluated the completeness of those related RAs. Through this analysis, we identified which elements of RAModel are present in the reference architectures and which are not. We consider that most of the related reference architectures are not according to RAModel because there were built in an ad hoc way since there is a lack of a well-defined methodology used to built the reference architecture. RAMSES was built by following the ProSA-RA process and based on RAModel.

We notice that most of the previously studied works were still insufficient since they do not reveal the coherence and traceability between specifications and implementation of internal parts of systems resulted from the reference architecture. We also notice a lack of usage of architectural viewpoints for representing different perspectives of RAs for supporting a better understanding of the proposed RAs.

It is important to note that most of the related reference architectures contain characteristics of a research-driven RA, in which details of the development of some modules, laws, standards and regulations needs to be more investigated.

CHAPTER 6: CONCLUSION

Autonomic Computing is a promising option to meet basic requirements in WSN design. Despite this fact, we noticed a lack of well-defined reference architecture designs that support the autonomy of WSN and middleware systems able to support network management without involving human operators. This thesis contributed in this sense, supporting a better understanding and systematization of the architecture design of self-adaptive middleware for WSN. The proposed reference architecture, named RAMSES, provides a common structure and guidelines for dealing with core aspects of developing and using self-adaptive middleware for WSNs.

The set of contributions described in the thesis is revisited in Section 5.1. Section 5.2 summarizes directions for further research. Finally, the list of publications related to this research is detailed in Section 5.3.

6.1 Revisiting the thesis contribution

This section summarizes the main contributions of this thesis.

- **Definition of a reference architecture that meets requirements for self-adaptive middleware systems for WSNs.**

  We proposed a reference architecture, named RAMSES, to facilitate the design of self-adaptive middleware for WSN (see Chapter 3). The conception of RAMSES was motivated by the lack of guidelines and well-defined middleware architectures for WSNs that provides an explicit way for defining the underlying autonomic behavior of WSNs.

  Our architecture follows ProSA-RA, a process that systematizes the design, representation, and evaluation of reference architectures. Our RA was inspired on autonomic computing systems by applying the MAPE-K for providing autonomy to networks. We designed RAMSES using RAModel as a framework, that defines a set of architectural views needed for documenting a reference architecture. Moreover, the patterns applied in RAMSES are derived from common knowledge in the field of self-adaptation and experiences acquired by the authors with building self-adaptive systems and WSN middleware systems.

Finally, we applied both an early and a late evaluation in our RA. The early evaluation was based on FERA, which mainly indicates that (i) the provided abstraction level and the use of autonomic computing principles, based on MAPE-K, are adequate for the RA purposes, (ii) RAMSES is clearly explained, and its detail level favors the RA understanding. For late evaluation, we conducted a proof of concept, which indicates that: (i) adaptation capabilities of the WSN were provided by the middleware architectural design and, (ii) the instantiation of RAMSES reduces the effort for developing and maintaining WSN applications.

- **Specification of the reference architecture using pi-ADL Architecture Description Language.**

  We specified the runtime view and deployment view of RAMSES using pi-ADL. This ADL enables the representation of dynamic software architectures as required by WSNs, which demands continuously adapting the network to dynamic environments and unpredictable events. Therefore, we used Pi-ADL specification to explicitly define the dynamic behavior of WSN networks, a key issue in this type of systems.

  In this perspective, it is worth highlighting that it was the first time Pi-ADL was used in the representation context of middleware systems specific to these networks, where the main contributions of Pi-ADL for WSNs area are: (i) behavioral representation of WSN architectures; (ii) representation of dynamic aspects of a WSN; and (iii) facilitating the source-code generation due to the formal modeling of the architecture.

- **A middleware instantiation of the reference architecture.**

  RAMSES was instantiated by building a self-adaptive middleware for WSN, named SAMSON. This middleware was built for evaluating RAMSES in deployment time. For properly assessing the middleware functionality, four adaptation scenarios were established to verify if adaptation capabilities managed by MAPE components were correctly executed.

SAMSON was deployed in a hierarchical network of virtual SKY nodes that run Contiki OS. The Cooja Simulator supported this evaluation. The results show that SAMSON correctly monitors and analyzes the network context, and a diagnosis based on adaptation policies is properly used for planning a network adaptation, whenever it is needed. Furthermore, SAMSON was compared with other self-adaptive middleware systems found in the literature. We noticed that SAMSON stood out for offering a well-defined RA-based design for supporting dynamic behavior in WSNs.

- **A model-driven solution composed of a model to text transformation engine (M2T) for mapping from RAMSES components (specified with pi-ADL) into Contiki code.**

  We established the RAMSES instantiation process that describes the steps required to develop a self-adaptive middleware for WSN based on RAMSES. In this process was determined the involved stakeholders and their respective activities and roles in order to translate the RAMSES elements, specified in pi-ADL into a concrete middleware instance.

  This solution takes as input the Pi-ADL specification and the transformation code concerning the chosen target platform and generates as output the source code to be deployed in the sensor nodes. The M2T transformation can be executed as many times as necessary to generate the code of the source model produced by the software architect in the design phase. This approach reduces the gap between the RA specification and its implementation. Moreover, an important benefit of RAMSES is to reduce the effort in developing WSN applications.

The achievements of this multidisciplinary thesis contribute to the areas of Software Architecture and WSNs, as they advance the current state-of-art on the architectural design of self-adaptive middleware for WSN based on autonomic computing principles.

6.2 Limitations and Future Work

This section describes limitations of this thesis and how they can be tackled. We conclude by pointing out future directions of research in the areas associated with the design of software architectures for self-adaptive WSNs.

- **Evolution of the domain terminology into an ontology:**

  The evolution of the domain terminology and conceptual view, detailed in Section 3.5.1.1, into an ontology can be an important contribution to this reference architecture. Therefore, investigating the relationship among the domain terminology proposed in RAMSES and available ontologies for WSN or related areas, such as the SNN-Ontology (Neuhaus & Compton, 2009), can be considered a promising extension of this work.

- **Modeling of Variability Viewpoint:**

  RAMSES was designed for various WSN domains and purposes. However, a deeper analysis must be conducted in order to establish a more reasoned definition of variability aspects in RAMSES. Variability is the ability of the software artifacts built from reference architectures to easily be adapted for a specific context. In this perspective, research on variability views on reference architectures can contribute to clear understanding about how to adapt and evolve our RA to a concrete architecture for a given WSN domain.

- **Automatized verification of architectural properties:**

  The critical nature of many complex software systems calls for rigorous architectural models (such as architecture descriptions) as means of supporting the automated verification and enforcement of architectural properties and constraints. Pi-ADL is a formal ADL that supports automatized verification of properties and provides textual notations based on mathematical principles. In this perspective, research for developing tools for automatically verifying RAMSES properties would contribute to precisely determine if a WSN architecture can satisfy properties related to user requirements. Additionally, automated verification provides an efficient method to check the correctness of architectural design.

- **Limitation of the Instantiation process of RAMSES:**

  The instantiation process described in Section 3.5.4.3 provides guidelines that facilitate the development of self-adaptive middleware for WSN based on the proposed reference architecture, RAMSES. However, additional information and support could be provided to facilitate the application of its phases. For instance, we could improve the proposed process by providing document templates and further directions on how stakeholders must conduct their activities.

- **Limitation of the M2T transformations**

  In RAMSES, automatic transformations (M2T) takes as input the Pi-ADL specification and the transformation code concerning the chosen target platform and generates as output the source code to be deployed in RAMSES deployment nodes. In order to generate the source code for gateway components, the M2T transformation solution maps the source code in Go programming language, and for sensor nodes, it maps into C code for Contiki OS. However, the proof of concept described in section 4.2 can be used as the reference to specify M2T transformations for other platforms, such as TinyOS.

- **Limitation in the Evaluation Methodology of the Proof of Concept:**

  We conducted a proof of concept to evaluate RAMSES by instantiating it into a concrete architecture used to build SAMSON. The preliminary results were promising. However, the benefits of adopting reference architectures are inherently difficult to estimate and generalize. Therefore, conducting larger and longer scenarios to observe qualitative aspects and conceiving other adaptation scenarios (such as a self-protection scenario to evaluate security issues) may contribute to validate the RA. Moreover, the proof of concept used to evaluate SAMSON was performed on virtual nodes supported by Cooja simulator. The obtained results provide encouraging evidence, but the scope was limited and opportunities for evaluations could include real sensor nodes. This future work can provide additional evidence that increase the confidence on the adoption of RAMSES.

6.3 List of Publications

The publication results directly provided from this PhD thesis are presented bellow:

- **PORTOCARRERO, JESÚS M. T.**; Delicato, Flávia C.; Pires, Paulo F.; Rodrigues, Taniro C.; Batista, Thais (***in press***). "SAMSON: Self-Adaptive Middleware for Wireless Sensor Networks".
  Event: 31st ACM Symposium on Applied Computing SAC'16 (SA-TTA), 2016, Pisa, Italy (***in press***)

- **PORTOCARRERO, J. M. T.;** DELICATO, F. C.; PIRES, P. F.; NAKAGAWA, ELISA Y.; OQUENDO, FLAVIO. "Self-Adaptive Middleware for Wireless Sensor Networks: A Reference Architecture". In:
  Event: European Conference on Software Architecture, 2015, Dubrovnik/Cavtat, Croatia. ECSAW '15.

- **PORTOCARRERO, JESÚS M. T.;** Delicato, Flávia C.; Pires, Paulo F.; GÁMEZ, NADIA; FUENTES, LIDIA; LUDOVINO, DAVID; FERREIRA, PAULO. "Autonomic Wireless Sensor Networks: A Systematic Literature Review".
  Journal: Journal of Sensors, v. 2014, p. 1-13, 2014.

- **PORTOCARRERO, J. M. T.;** DELICATO, F. C.; PIRES, P. F.; BATISTA, T.V. "Reference Architecture for Self-Adaptive Management in Wireless Sensor Networks".
  Event: 2014 International Conference on Adaptive and Intelligent Systems - ICAIS'14, 2014, Bournemouth - UK.

Other related publications:

- SILVA, J. R.; DELICATO, F. C.; PIRMEZ, L.; PIRES, P. F.; **PORTOCARRERO, J. M. T.;** RODRIGUES, T. C.; BATISTA, T.V. "PRISMA: Publish / Subscribe and Resource Oriented Middleware for Wireless Sensor Networks".

  Event: The Tenth Advanced International Conference on Telecommunications AICT 2014, Paris, France.

- Delicato, F. C.; **PORTOCARRERO, J. M. T.;** SILVA, J. R. ; Pires, Paulo F.; ARAUJO, R.; BATISTA, T. V. "MARINE: MiddlewAre for resource and mIssion-oriented sensor Networks".

  Journal: ACM SIGMOBILE Mobile Computing and Communications Review, v. 17, p. 40-54, 2013.

- Pires, Paulo F.; Delicato, Flávia C.; **PORTOCARRERO, J. M. T.** "Enfoque basado en MDA para apoyar evoluciones seguras en sistemas".

  Journal: Novática, v. 221, p. 25-33, 2013.

- ARAUJO, R. P. M.; **PORTOCARRERO, J. M. T.;** DELICATO, F. C.; PIRES, P. F.; PIRMEZ, L.; BATISTA, T.; ROSSETTO, S.; SOUTO, A. L. "Middleware Baseado em Componentes e Orientado a Recursos para Redes de Sensores sem Fio".

  Event: XXX Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC), 2012, Ouro Preto, MG, Brazil.

- ARAUJO, R. P. M.; **PORTOCARRERO, J. M. T.;** SILVA, J. R.; DELICATO, F. C.; PIRES, P. F. "MARINE: MiddlewAre for Resource and mIssion oriented sensor Networks".

  Event: The First ACM Annual International Workshop on Mission-Oriented Wireless Sensor Networking (ACM MiSeNet 2012), 2012, Istanbul, Turkey.

REFERENCES

Affonso, F., Scannavino, K., Oliveira, L., & Nakagawa, E. (2014). *Reference Architectures for Self-Managed Software Systems: a Systematic Literature Review.* Paper presented at the Software Components, Architectures and Reuse (SBCARS), Eighth Brazilian Symposium on.

Affonso, F. J., Vecchiato Saenz, M., Rodrigues, L., Luis, E., & Nakagawa, E. Y. (2015). A Reference Model as Automated Process for Software Adaptation at Runtime. *Latin America Transactions, IEEE (Revista IEEE America Latina), 13(1)*, 214-221.

Allen, R., Douence, R., & Garlan, D. (1998). *Specifying and analyzing dynamic software architectures.* Paper presented at the Proceedings of the First International Conference on Fundamental Approaches to Software Engineering (FASE'98), Lisbon, Portugal.

Amin, S. O., & Hong, C. S. (2005). On Design Patterns for Sensor Networks.

Andersson, J., de Lemos, R., Malek, S., & Weyns, D. (2009). *Reflecting on self-adaptive software systems.* Paper presented at the Software Engineering for Adaptive and Self-Managing Systems, 2009. SEAMS '09. ICSE Workshop on, Vancouver, Canada.

Angelov, S., Grefen, P., & Greefhorst, D. (2009). *A classification of software reference architectures: Analyzing their success and effectiveness.* Paper presented at the Software Architecture, 2009 \& European Conference on Software Architecture. WICSA/ECSA 2009.

Angelov, S., Grefen, P., & Greefhorst, D. (April, 2012). A framework for analysis and design of softwre reference architectures. *Information and Software Technology, 54*(4), 417-431.

Angelov, S., Trienekens, J. J., & Grefen, P. (2008). *Towards a Method for the Evaluation of Reference Architectures: Experiences from a Case.* Paper presented at the ECSA '08 Proceedings of the 2nd European conference on Software Architecture, Springer-Verlag Berlin, Heidelberg.

Arduino. Arduino. Retrieved from http://arduino.cc/.

Aslam, M. S. a. O. R., Eoin and Rea, Susan and Pesch, Dirk. (2009). *Open framework middleware: an experimental middleware design concept for wireless sensor networks.* Paper presented at the Proceedings of the 6th international workshop on Managing ubiquitous communications and services.

Avgeriou, P., & Zdun, U. (2005). *Architectural patterns revisited – a pattern language.* Paper presented at the In 10th European Conference on Pattern Languages of Programs (EuroPlop 2005), Irsee, Germany.

Balakrishnan, G., & Hiremath, S. S. (Jan, 2012). Autonomous sensor networks for process monitoring and automation. *IEEE 10th Int. Symp. Appl. Mach. Intell. Informatics*, 47–52.

Bass, L., Clements, P., Kazman, R. (2012). *Software Architecture in Practice. Addison-Wesley*: Longman Publishing Co., Inc., Boston, MA, USA.

Bayer, J., Forster, T., Ganesan, D., Girard, J., John, I., Knodel, J., . . . Muthig, D. (2004). *Definition of reference architectures based on existing systems*. Retrieved from

Bhattacharya, A. (2012). *Mobile agent based elastic executor service: Reference architecture of an executor service using a mobile agent platform to control the elasticity of the system.* Paper presented at the JCSSE'2012.

Blair, G. (2004). Middleware Technologies for Future Communication Networks. *IEEE Netw, 18(1)*.

Bourdenas, T., & Sloman, M. (2010). Starfish: policy driven self-management in wireless sensor networks. *Proc. 2010 ICSE Work. Softw. Eng. Adapt. Self-Managing Syst*, 75–83.

Bourdenas, T., Tei, K., Honiden, S., & Sloman, M. (Oct, 2011). Autonomic Role and Mission Allocation Framework for Wireless Sensor Networks. *2011 IEEE Fifth Int. Conf. Self-Adaptive Self-Organizing Syst.*, 61-70.

Buschmann, F., Henney, K., & Schimdt, D. (2007). Pattern-oriented Software Architecture: On Patterns and Pattern Language. *John wiley and sons, 5*.

Canal, C., Pimentel, E., & Troya, J. (1999). *Specification and refinement of dynamic software architectures.* Paper presented at the Proceedings of the TC2 First Working IFIP Conference on Software Architecture (WICSA1), San Antonio, TX, USA. .

Cao, T. M., Bellata, B., & Oliver, M. (Sep, 2014). Design of a generic management system for wireless sensor networks. *Ad Hoc Networks, 20*, 16-35.

Cardei, M., Fernandez, E., Sahu, A., & Cardei, I. (2011). *A pattern for sensor network architectures.* Paper presented at the Proceedings of the 2nd Asian Conference on Pattern Languages of Programs.

Cardellini, V. e. a. (2009). *A scalable approach to qos-aware self-adaption in service-oriented architectures.* Paper presented at the QSHINE'2012.

Casola, V., Gaglione, A., & Mazzeo, A. (July, 2009). *A reference architecture for sensor networks integration and management.* Paper presented at the IEEE Proceedings of GSN 2009, Oxford, UK.

Castañeda, L. (2012). *A Reference Architecture for Component-Based Self-Adaptive Software Systems.* (Magister in Informatics and Telecommunications Management Magister Graduation Project), ICESI University.

Castejon, H., et al. (2011). *Towards a dynamic cloud-enabled service eco-system.* Paper presented at the ICIN'2011.

Castelli, G., Mamei, M., Rosi, A., & Zambonelli, F. (2015). *Engineering pervasive service ecosystems: the SAPERE approach.* Paper presented at the ACM Transactions on Autonomous and Adaptive Systems (TAAS).

Cavalcante, E. O., F; Batista, T. (2014). *Architecture- Based Code Generation: From π-ADL Architecture Descriptions to Implementations in the Go Language.* Paper presented at the European Conference on Software Architecture (ECSA), Vienna, Austria.

Cherif, S., Djemaa, R. B., & Amous, I. (2014). *ReMoSSA: Reference Model for Specification of Self-adaptive Service-Oriented-Architecture.* Paper presented at the New Trends in Databases and Information Systems.

Chong, C., & Kumar, S. (Aug, 2003). Sensor Networks: Evolution, Opportunities, and Challenges. *Proceedings of the IEEE, 91(8)*, 1247.

Clements, P., & Northrop, L. (2002). *Software product lines: practices and patterns*: Addison-Wesley.

Clements, P. C. (March, 1996). *A Survey of Architectural Description Languages.* Paper presented at the Eighth International Workshop on Software Specification and Design, Germany.

Clements, P. C., Kazman, R., & Klein, M. (2002). *Evaluating Software Architectures*. Boston.

Cloutier, R., Muller, G., Verma, D., Nilchiani, R., Hole, E., & Bone, M. (2010). The concept of reference architectures. *Systems Engineering, 13 (1)*, 14--27.

Costa, P., Coulson, G., Gold, R., Lad, M., Mascolo, C., Mottola, L., . . . Zachariadis, S. (2007). *The RUNES Middleware for Networked Embedded Systems and its Application in a Disaster Management Scenario.* Paper presented at the Fifth Annual IEEE International Conference on Pervasive Computing and Communications (PerCom'07).

Dashofy, E., van der Hoek, A., & Taylor, R. (August, 2001, 28–31 August 2001). *A highly-extensible, XML-based architecture description language.* Paper presented at the Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA 2001), Amsterdam, The Netherlands.

Delicato, F., Protti, F., Pirmez, L., & Rezende, J. (2006). An efficient heuristic for selecting active nodes in wireless sensor networks. *Computer Networks, 50(18)*, 3701-3720.

Delicato, F. C., Pirmez, L., Rust, L., & Pires, P. C. (2003). A Flexible Middleware System for Wireless Sensor Networks. *ACM/IFIP/USENIX International Middleware Conference. Lecture Notes on Computer Science, 2672*, 474-492.

Ding, P., Holliday, J., & Celik, A. (Jun/Jul, 2005). *Distributed energy-efficient hierarchical clustering for wireless sensor networks.* Paper presented at the Proceedings of the First IEEE international conference on Distributed Computing in Sensor Systems, Marina del Rey, CA.

Dunkel, J. (2009). *On complex event processing for sensor networks.* Paper presented at the Autonomous Decentralized Systems, 2009. ISADS'09. International Symposium on.

Dunkel, J. (2011). *Towards a multiagent-based software architecture for sensor networks.* Paper presented at the Autonomous Decentralized Systems (ISADS), 2011 10th International Symposium on.

Dunkels, A., Grönvall, B., & Voigt, T. (November 2004). *Contiki - a lightweight and flexible operating system for tiny networked sensors.* Paper presented at the First IEEE Workshop on Embedded Networked Sensors, ampa, Florida, USA.

Dunkels, A. F., N; Eriksson, J; Voigt, T. (November, 2006). *Run-Time Dynamic Linking for Reprogramming Wireless Sensor Networks.* Paper presented at the 4th International conference on Embedded networked sensor systems. SenSys'06, Boulder, Colorado, USA.

Eeles, P. (2008). *Understanding architectural assets.* Paper presented at the Software Architecture, 2008, WICSA 2008. Seventh Working IEEE/IFIP.

ElGammal, M., & Eltoweissy, M. (Oct, 2011). Towards Aware, Adaptive and Autonomic Sensor-Actuator Networks. *2011 IEEE Fifth Int. Conf. Self-Adaptive Self-Organizing Syst.*, 210-211.

Fernandez-Gago, M. (2007). A survey on the applicability of trust management systems for wireless sensor networks. *Secur. Priv. Trust Pervasive Ubiquitous Comput*.

Fok, C.-L., Roman, G.-C., & Lu, C. (2009). Agilla: A mobile agent middleware for self-adaptive wireless sensor networks. *ACM Transactions on Autonomous and Adaptive Systems (TAAS), 4(3)*.

Fortino, G., Galzarano, S., & Liotta, A. (Jan, 2012). An autonomic plane for Wireless Body Sensor Networks. *Int. Conf. Comput. Netw. Commun*, 94-98.

Gamez, N., Fuentes, L., & Araguez, M. (2011, September 13–16). *Autonomic computing driven by feature models and architecture in FamiWare.* Paper presented at the Software Architecture: Proceedings of 5th European Conference, ECSA 2011,, Essen, Germany.

Gamez, N., Romero, D., Fuentes, L., Ruovoy, R., & Duchien, L. (2012). Constraint-based self-adaptation of wireless sensor networks. *Proc. 2nd Int. Work. Adapt. Serv. Futur. Internet*, 20–27.

Garces, L. M., Ampatzoglou, A., Avgeriou, P., & Nakagawa, E. (2015). *A Comparative Analysis of Reference Architectures for Healthcare in the Ambient Assisted Living Domain.* Paper presented at the Computer-Based Medical Systems (CBMS), 2015 IEEE 28th International Symposium on.

Garlan, D., Allen, R., & Ockerbloom, J. (December, 1994). *Exploiting style in architectural design environments.* Paper presented at the Proc. of SIGSOFT'94: The second ACM SIGSOFT Symposium on the Foundations of Software Engineering.

Garlan, D., Monroe, R., & Wile, D. (Nov, 1997). *ACME. An architecture description language.* Paper presented at the Proceedings of CASCON'97.

Garlan, D., Schmerl, B., & Cheng, S. (2009). Software Architecture-Based Self-Adaptation. *Autonomic Computing and Networking*, 31-55.

Gluhak, A., & al., e. (June, 2006). *e-SENSE Reference Model for Sensor Network in B3G Mobile Communications Systems.* Paper presented at the 15th IST Summit 2006, Myconos, Greece.

Gotz, M., Rettberg, A., & Podolski, I. (Mar, 2011). Middleware Support for a Self-Configurable Wireless Sensor Network. *2011 14th IEEE Int. Symp. Object/Component/Service-Oriented Real-Time Distrib. Comput. Work*, 143–151.

Guessi, M., de Oliveira, L., & Nakagawa, E. Y. (2011). *Representation of Reference Architectures: A Systematic Review.* Paper presented at the Software Engineering and Knowledge Engineering (SEKE'11).

Hadim, S., & Mohamed, N. (Mar, 2006). Middleware: Middleware Challenges and Approaches for Wireless Sensor Networks. *IEEE Distrib. Syst, Online 7(3)*, 1-1.

Hayes-Roth, B. e. a. (1995). Domain-specific software architecture for adaptive intelligent systems. *Transactions on Software Engineering, 21(4)*, 288-301.

IBM. (2005). An Architectural blueprint for autonomic computing.

Imran, M., Alnuem, M. a., Alsalih, W., & Younis, M. (Jun, 2012). A novel wireless sensor and actor network framework for autonomous monitoring and maintenance of lifeline infrastructures. *2012 IEEE Int. Conf. Commun*, 6484–6488.

ISO/IEC. (2011). Systems and software engineering - Systems and software Quality Require- ments and Evaluation (SQuaRE) – System and software quality models *Tech Report 25010/2011*.

ISO/IEC, -. (2014). Sensor networks: Sensor Network Reference Architecture (SNRA) *Information technology*.

ISO/IEC/IEEE. (2011). ISO/IEC/IEEE 42010.

Jemal, A., & Halima, R. B. (June, 2013). *A QoS-driven self-adaptive architecture for wireless sensor networks.* Paper presented at the Proceedings of the 22nd

IEEE International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE ’13), Hammamet, Tunisia.

Jiao, W., & Mei, H. (2004). *Automated adaptations to dynamic software architectures by using autonomous agents.* Paper presented at the JEAAI.

Joshi, H., & Michel, H. (2008). *Integrated Technical Reference Model and Sensor Network Architecture*, ICWN, International Conference on Wireless Networks.

Kay, I., Kaste, O., & Mattern, F. (2008). Middleware Challenges for Wireless Sensor Networks. *ACM SIGMOBILE Mob. Comput. Commun, 6(4)*, 59-61.

Kazman, R., Klein, M., & Clements, P. C. (2000). *ATAM: Method for Architecture Evaluation* Paper presented at the Software Engineering Institute, Pittsburgh, PA.

Kerasiotis, F., Koulamas, C., & Papadopoulos, G. (2012). Developing wireless sensor network applications based on a function block programming abstraction. *IEEE Int. Conf. Ind. Technol*, 372–377.

Khedo, K., & Subramanian, R. (2009). A Service-Oriented Component-Based Middleware Architecture for Wireless Sensor Networks. *Int. J. Comput. Sci. Netw. Secur, 9(3)*, 174-182.

Kitchenham, B., & Charters, S. (2007). Guidelines for performing Systematic Literature reviews. *Software Engineering. EBSE Technical Report. EBSE-2007-01, 2(3)*

Kramer, J., & Magee, J. (2007). Self-managed Systems: an Architectural Challenge. *EEE Computer*, 259–268.

Lane, S. e. a. (2012). Soadapt: A process reference model for developing adaptable service-based applications. *Information and Software Technology, 54(3)*, 299-316.

Lemos, R. a. G., Holger and Müller, Hausi A. and Shaw, Mary. (2013). On Patterns for Decentralized Control in Self-Adaptive Systems *Software Engineering for Self-Adaptive Systems II: International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers* (pp. 76-107): Springer Berlin Heidelberg.

Li, C. e. a. (2012). *Dynamic software architectures: formal specification and verification with CSP.* Paper presented at the Internetware'2012, New York, NY, USA.

Li, D. L., H. (2009). Sensor Coverage in Wireless Sensor Networks. *Wireless Networks: Research, Technology and Applications*, 3-31.

Li, W., & Shen, W. (Jul, 2011). Swarm behavior control of mobile multi-robots with wireless sensor networks. *J. Netw. Comput. Appl, 34(4)*, 1398–1407.

Liu, L., & et, a. (2008). *A reference architecture for self-organizing service-oriented computing.* Paper presented at the ARCS'2008.

Liu, T., & Martonosi, M. (2003). *Impala : A Middleware System for Managing Autonomic , Parallel Sensor Systems.* Paper presented at the Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming.

Loureiro, A., Gonzàlez, R., & Mini, R. (2010). QoS: Requirements, Design Features, and Challenges on Wireless Sensor Networks. *Handbook of Research on Developments and Trends in Wireless Sensor Networks: From Principle to Practice. IGI Global*, 56-78.

MacKenzie, C. M. a. L., Ken and McCabe, Francis and Brown, Peter F and Metz, Rebekah and Hamilton, Booz Allen. (2006). Reference model for service oriented architecture 1.0. *OASIS standard, 12*.

Macolo, I., Capra, L., & Emmerich, W. (2002). Mobile Computing Middleware. *Advanced lectures on networking*, 20-58.

MDA. Model Driven Architecture. Retrieved from http://www.omg.org/mda/

MEMSIC. MEMSIC. Retrieved from http://www.memsic.com/.

Menasce, D., Gomaa, H., Malek, S., & Sousa, J. (Nov/Dec, 2011). SASSY: A Framework for Self-Architecting Service-Oriented Systems. *IEEE Software, 28(6)*.

Minh, T., Bellalta, B., & Oliver, M. (2013). DISON: A Self-organizing Network Management Framework for Wireless Sensor Networks. *Ad Hoc Networks*.

Molla, I., & Ahamed, I. (2006). A Survey of Middleware for Sensor Network and Challenges. *Int. Conf. Parallel Process*, 223-228.

Mouronte, M. L., Ortiz, O., Garcia, A. B., & Capilla, R. (May, 2013). *Using dynamic software variability to manage wireless sensor and actuator networks.* Paper presented at the Proceedings of the IFIP/IEEE International Symposium on Integrated Network Management (IM'13).

Muller, G. (2008). A reference architecture primer. *Eindhoven University of Technology, Eindhoven, White paper*.

Muller, H., & et, a. (2009). *Autonomic computing now you see it, now you don't: Design and evolution of 2009 autonomic software systems.* Paper presented at the LNCS.

Muraleedharan, R., & Osadciw, L. A. (2009). Secure self-adaptive framework for distributed smart home sensor network. *2009 Conf. Rec. Forty-Third Asilomar Conf. Signals, Syst. Comput.*, 284–287.

Nair, N., Morrow, P., & Parr, G. (2012). Framework for a Self-managed Wireless Sensor Cloud for Critical Event Management. *Sens. Syst. Softw*, 15-29.

Nakagawa, E., Oquendo, F., & Becker, M. (2012). *RAModel: A reference model for reference architectures.* Paper presented at the Software Architecture (WICSA) and European Conference on Software Architecture (ECSA).

Nakagawa, E. Y., Guessi, M., Maldonado, J., Feitosa, D., & Oquendo, F. (2014). *Consolidating a Process for the Design, Representation, and Evaluation of Reference Architectures.* Paper presented at the Proc. Working IEEE/IFIP Conf. of Software Architecture (WICSA 2014), Sydney, Australia.

Nakagawa, E. Y., Oquendo, F., & Becker, M. (2012). *RAModel: A reference model of reference architectures.* Paper presented at the ECSA/WICSA 2012, Helsinki, Finland.

Neti, S., & Muller, H. A. (May, 2007). *Quality Criteria and an Analysis Framework for Self-Healing Systems.* Paper presented at the International Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'07).

Neuhaus, H., & Compton, M. (2009). *The semantic sensor network ontology.* Paper presented at the AGILE workshop on challenges in geospatial data harmonisation, Hannover, Germany.

Oquendo, F. (2004). π-ADL: An architecture description language based on the higher- order typed-calculus for specifying dynamic and mobile software architectures. *ACM SIGSOFT Software Engineering Notes, 29(3)*, 1-4.

Oquendo, F. (May, 2004). *π-ADL: an Architecture Description Language based on the higher-order typed π-calculus for specifying dynamic and mobile software* *architectures.* Paper presented at the ACM SIGSOFT Software Engineering Notes, New York, NY, USA.

Oquendo, F., Warboys, B., Morrison, R., Dindeleux, R., Gallo, F., Garavel, H., & Occhipinti, C. (May, 2004). *ArchWare: Architecting Evolvable Software.* Paper presented at the Proceedings of the 1st European Workshop on Software Architecture, LNCS 3047, St Andrews, UK.

Oracle. oracle.

Paradis, L. H., Q. (2007). A Survey of Fault Management in Wireless Sensor Networks. *Journal of Network and Systems Management*. doi:DOI: 10.1007/s10922-007-9062-0

Picco, G. P. (2010). *Software engineering and wireless sensor networks: happy marriage or consensual divorce?* Paper presented at the Proceedings of the FSE/SDP workshop on Future of software engineering research, Santa Fe, New Mexico, USA.

Pohl, K., Bockle, G., & Linden, F. (2005). Software product line engineering: foundations, principles, and technique. *Springer, Berlin*.

Portocarrero, J. M. T., Delicato, F. C., Pires, P. C., Gamez, N., Fuentes, L., Ludovino, D., & Ferreira, P. (2014). Autonomic Wireless Sensor Networks: A Systematic Literature Review. *Journal of Sensors, 2014*, 13. doi:http://dx.doi.org/10.1155/2014/782789

Potdar, V., Sharif, A., & Chang, E. (2009). *Wireless sensor networks: A survey.* Paper presented at the Internation Conference Advanced Information Networking and Applications Workshops, WAINA'09.

Puccinelli, D., & Haenggi, H. (2005). Wireless sensor networks: applications and challenges of ubiquitous sensing. *IEEE Circuits and Systems Magazine, 3*, 19-29.

Puviani, M., Cabri, G., & Zambonelli, F. (2013). *A taxonomy of architectural patterns for self-adaptive systems.* Paper presented at the International Conference on Computer Science and Software Engineering (C3S2E'13), ACM, New York, NY, USA.

Qwasmi, N., & Liscano, R. (Jan, 2012). Framework for Distributed Policy-Based Management in Wireless Sensor Networks to Support Autonomic Behavior. *Procedia Comput. Sci., 10*, 232-239.

Rahman, A. (April, 2009). *Middleware for wireless sensor networks: Challenges and Ap- proaches*. Paper presented at the Seminar on Internetworking, Helsinki University of Technology, Finland.

Ravula, S., Kim, J. E., Petrus, B., & Stoermer, C. (2005). *Quality attributes in wireless sensor networks.* Paper presented at the Third IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems. SEUS 2005.

S., S., Zeng, B., & Liu, L. (2013). Smart policy generating mechanism for policy driven self-management in wireless sensor networks. *Sensor & Transducers, 154(7)*, 9-14.

Salehie, M., & Tahvildari, L. (2009). Self-Adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.,, 4*(2).

Santos, J., Guesse, M., Gaister, M., Feitosa, D., & Nakagawa, E. (2013). *A checklist for evaluation of reference architectures for embedded systems,.* Paper presented at the SEKE'13, Boston, USA.

Santos, J., Guessi, M., Galster, M., Feitosa, D., & Nakagawa, E. Y. (2013). *A checklist for evaluation of reference architectures for embedded systems.* Paper presented at the SEKE'13, Boston, USA.

Serri, J. A. (2004). *Reference architectures and management model for ad hoc sensor networks.* Paper presented at the Sensor and Ad Hoc Communications and Networks, First Annual IEEE Communications Society Conference.

Shah, S., & Szymanski, B. (Mar, 2013). Autonomous configuration of spatially aware sensor services in service oriented WSNs. *Pervasive Comput. Commun. Work. (PERCOM)*, 312–314.

Sheltami, T., Al-Roubaiey, A., & Mahmoud, A. (2015). A survey on developing publish/subscribe middleware over wireless sensor/actuator networks. . *Wireless Networks*, 1-22.

Sohraby, K., Minoli, D., & Znati, T. (2007). *Wireless Sensor Networks: Technology, Protocols and Applications*. Paper presented at the John Wiley & Sons, Inc., Hoboken, NJ, USA.

Stocker, M., Rönkkö, M., & Kolehmainen, M. (2015). Knowledge-based environmental research infrastructure: moving beyond data. *Earth Science Informatics*, 1-19.

Tajalii, H., & Medvidovic, N. (2012). *A reference architecture for integrated development and run-time environments.* Paper presented at the TOPI'2012.

Takano, C., Aida, M., Murata, M., & Imase, M. (Jul, 2011). Autonomous Decentralized Mechanism of Structure Formation Adapting to Network Conditions. *2011 IEEE/IPSJ Int. Symp. Appl. Internet*, 524–531.

TinyOS. TinyOS.

Tomforde, S., & al., e. (2011). *Observation and control of organic systems.* Paper presented at the Organic Computing - A Paradigm 2011 Shift for Complex Systems.

Tóth, A., & Vajda, F. (2012). Autonomous Sensor Network Architecture Model. *Inf. Commun. Technol*, 298–308.

Villalba, C. e. a. (2008). *Nature-inspired spatial metaphors for pervasive service ecosystems.* Paper presented at the SASOW'2008.

Villalba, C. e. a. (2010). *A self-organizing architecture for pervasive ecosystems.* Paper presented at the Self-Organizing Architectures.

Villegas, N., Tamura, G., Mu¨ller, H., Duchien, L., & Casallas, R. (2012). DYNAMICO: A Reference Model for Governing Control Objectives and Context Relevance in Self-Adaptive Software Systems. *Software Engineering for Self-Adaptive Systems. Springer, 7475 of LNCS*, 282 – 310.

Wang, M., Cao, J., Li, J., & al., e. (May, 2008). Middleware for wireless sensor networks: A survey. *Journal of Computer Science and Technology, 23*(3), 305-326.

Weyns, D., Malek, S., & Andersson, J. (2010a). *FORMS: a formal reference model for self-adaptation.* Paper presented at the Proceedings of the 7th international conference on Autonomic computing.

Weyns, D., Malek, S., & Andersson, J. (2010b). *On decentralized self-adaptation: lessons from the trenches and challenges for the future.* Paper presented at the Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems.

Weyns, D. e. a. (2005). *Architectural design of a distributed application with autonomic quality requirements.* Paper presented at the DEAS'2005, New York, NY, USA.

Willig, A., Hauer, J. H., Karowski, N., Baldus, H., & Huebner, A. (2007). *The ANGEL WSN Architecture.* Paper presented at the 14th IEEE International Conference on Electronics, Circuits and Systems.

Wils, A., Berbers, Y., Holvoet, T., & De Vlaminck, K. (2006). *Timing driven architectural adaptation.* Paper presented at the Distributed Applications and Interoperable Systems.

Yang, Z., Li, Z., Jin, Z., & Chen, Y. (2014). A Systematic Literature Review of Requirements Modeling and Analysis for Self-adaptive Systems. *Salinesi, C., van de Weerd, I. (eds.) REFSQ 2014. LNCS, Springer, Heidelberg, 8396*, 55-71.

Yu, I., Krishnamachari, B., & Prasanna, V. (2004). Issues in Designing Middleware for Wireless Sensor Networks. *IEEE Netw, 18(1)*, 15-21.

Zewdie, B., & Carlson, C. (2006). *Adaptive component paradigm for highly configurable business components*. Paper presented at the Electro/information Technology, IEEE International Conference on.

APPENDIX A: RAModel – Reference model for Reference Architectures

Table 20. RAModel group of elements (E. Nakagawa et al., 2012)

Group	Element	Description
Domain	Legislations, standards, and regulations	Laws, standards, and regulations existing in the domain that should be present in systems resulted from the reference architecture.
	Quality attributes	Quality attributes, for instance, maintainability, portability, and scalability, that are desired in systems resulted from the reference architecture.
	System compliance	Means to verify if systems developed from the reference architecture follow existing legislations, standards, and regulations.
Application	Constraints	Constraints presented by the reference architecture and/or constraints in specific part of a reference architecture.
	Domain data	Common data found in systems of the domain. These data are presented in a higher level of abstraction, considering the higher level of abstraction of the reference architecture.
	Functional requirements	Set of functional requirements that are common in systems developed using this architecture.
	Goal and needs	Intention of the reference architecture and needs that could be covered by the reference architecture.
	Limitations	Limitations presented by the reference architecture and/or limitations in specific part of a reference architecture.
	Risks	Risks in using the reference architecture and/or risks in using some part of such architecture.
	Scope	Scope that is covered by the reference architecture, i.e., the set of systems developed based on the reference architecture.
Infrastructure	Best practices and guidelines	Well-experimented practices to develop systems of the do- main, These practices could be accompanied by guidelines describing how to apply these practices.
	General structure	General structure of the reference architecture, represented sometimes by using existing architectural styles.
	Hardware elements	Elements of hardware, such as server and devices, which host systems, resulted from the reference architecture.
	Software elements	Elements of software present in the reference architecture, e.g., subsystems and classes, which could be used to develop software systems.

Crosscutting Elements	Decisions	Decisions, including description of the decision, options (alternatives), rationale, and tradeoffs, must be reported during the development of the reference architecture.
	Domain Terminology	Set of terms of the domain that are widely accepted by the community related to that domain and are, therefore, used in the description of the reference architecture.
	External communication	Means by which occurs exchange of information between the systems resulted from the reference architecture and the external environment.
	Internal communication	Means by which occurs exchange of information among internal parts of systems resulted from the reference architecture.

APPENDIX B: Pi-ADL specification of runtime view

```
/**************************************************
 * GOAL MANAGEMENT LAYER
 **************************************************/

// COMPONENT APPLICATION MANAGER
component ApplicationManagerCP is abstraction() {
        type WSNData is Any
        type AppReq is Any
        type WSNServices is Any
        connection publishWSNData is in(WSNData)
        connection setAppReq is out(AppReq)
        connection getServicesRequest is out(Boolean)
        connection getServicesReply is in(WSNServices)
        protocol is {
                (via publishWSNData receive WSNData
                via setAppReq send AppReq
                via getServicesRequest send Boolean
                via getServicesReply receive WSNServices)*
        }
        behavior is {
                via publishWSNData receive wsnData : WSNData
                via setAppReq send AppReq
                via getServicesRequest send True
                via getServicesReply receive wsnServices: WSNServices
                behavior()
        }

}

// CONECTOR PublishWSNData
connector PublishWSNDataCN is abstraction() {
        type WSNData is Any
        connection fromNetworkMonitor is in (WSNData)
        connection toApplicationManager is out (WSNData)
        protocol is {
                (via fromNetworkMonitor receive WSNData
                via toApplicationManager send WSNData)*
        }
        behavior is {
                via fromNetworkMonitor receive wsnData : WSNData
                via toApplicationManager send wsnData
                behavior()
        }
 }


//COMPONENT ADAPTATION POLICIES MANAGER
component AdaptationPoliciesManagerCP is abstraction(){
        type Id is Integer
        type Type is Integer
        type ContextInfo is Any
```

```
            type Operator is String
            type Value is Real
            type Action is Any
            type Priority is Integer

            type AdaptationPolicy is tuple[Id, Type, ContextInfo, Operator, Value, Action, Priority]

            connection setAdaptationPolicy is out(AdaptationPolicy)
            protocol is {
                        (via setAdaptationPolicy send AdaptationPolicy)*
            }
            behavior is {
                        via setAdaptationPolicy send AdaptationPolicy
                        behavior()
            }
}


// COMPONENT INSPECTION MANAGER
component InspectionManagerCP is abstraction(){
            type AdaptationInfo is Any
            connection getAdaptationInfo is in(AdaptationInfo)
            protocol is {
                        (via getAdaptationInfo receive AdaptationInfo)*
            }
            behavior is {
                        via getAdaptationInfo receive adaptationInfo : AdaptationInfo
                        behavior()
            }
}

// CONECTOR AdaptationInfo
connector AdaptationInfoCN is abstraction() {
            type AdaptationInfo is Any
            connection fromNetworkPlanner is in (AdaptationInfo)
            connection toInspectionManager is out (AdaptationInfo)
            protocol is {
                        (via fromNetworkPlanner receive AdaptationInfo
                        via toInspectionManager send AdaptationInfo)*
            }
            behavior is {
                        via fromNetworkPlanner receive adaptationInfo : AdaptationInfo
                        via toInspectionManager send adaptationInfo
                        behavior()
            }
 }

 /**************************************************
 * NETWORK MANAGEMENT LAYER
 **************************************************/

// COMPONENT NETWORK MONITOR
 component NetworkMonitorCP is abstraction() {
            type Context is Any
            connection publishContext is in (Context)
            connection setAppReq is in (Context)
            connection sendCurrState is out(Context)
```

```
        connection analysingRequest is out(Boolean)
        connection publishWSNData is out(Context)
        protocol is {
                (
                        via publishContext receive Context
                        via setAppReq receive Context
                        via sendCurrState send Context
                        via analysingRequest send Boolean
                        via publishWSNData send Context
                )*
        }
        behavior is {
                via publishContext receive wsnContext : Context
                via setAppReq receive appContext : Context
                via sendCurrState send wsnContext
                via sendCurrState send appContext
                if (wsnContext) then {
                        via analysingRequest send True
                        via sendCurrState send wsnContext
                        via publishWSNData send wsnContext
                }
                if (appContext) then {
                        via analysingRequest send True
                        via sendCurrState send appContext
                }
                behavior()
        }
}

// CONECTOR AppReq
connector AppReqCN is abstraction() {
        type Context is Any
        connection fromApplicationManager is in (Context)
        connection toNetworkMonitor is out (Context)
        protocol is {
                (via fromApplicationManager receive Context
                via toNetworkMonitor send Context)*
        }
        behavior is {
                via fromApplicationManager receive context : Context
                via toNetworkMonitor send context
                behavior()
        }
}

// CONECTOR publishContext
connector PublishContextCN is abstraction() {
        type Context is Any
        connection fromGatewayCommunication is in (Context)
        connection toNetworkMonitor is out (Context)
        protocol is {
                (via fromGatewayCommunication receive Context
                via toNetworkMonitor send Context)*
        }
        behavior is {
                via fromGatewayCommunication receive context : Context
                via toNetworkMonitor send context
```

```
			behavior()
		}
	}

	// COMPONENT NETWORKANALYSER
	component NetworkAnalyserCP is abstraction() {
		type Id is Integer
		type Type is Integer
		type ContextInf is Any
		type Value is Real
		type Context is tuple[Id, Type, ContextInfo, Value]
		connection analysingRequest is in(Boolean)
		connection getCurrStateRequest is out(Boolean)
		connection getCurrStateReply is in(Any)
		connection adaptationRequest is out(Boolean)
		protocol is {
			(via analysingRequest receive Boolean
			via getCurrStateRequest send Boolean
			via getCurrStateReply receive Any
			via adaptationRequest send Boolean
			)*
		}
		behavior is {
			analyseContext is function (c : Context) : Boolean {
				unobservable
				return True
			}
			via analysingRequest receive b_analysingRequest : Boolean
			if (b_analysingRequest) then {
				via getCurrStateRequest send true
				via getCurrStateReply receive context : Any
				if analyseContext(context) then {
					via adaptationRequest send (true)
				}
			}
			behavior()
		}
	}

	// CONECTOR analysingRequest
	connector AnalysingRequestCN is abstraction() {
		connection fromNetworkMonitor is in (Boolean)
		connection toNetworkAnalyser is out (Boolean)
		protocol is {
			(via fromNetworkMonitor receive Boolean
			via toNetworkAnalyser send Boolean)*
		}
		behavior is {
			via fromNetworkMonitor receive adaptationIsNeeded : Boolean
			via toNetworkAnalyser send adaptationIsNeeded
			behavior()
		}
	}

	// COMPONENT NETWORKPLANNER
	component NetworkPlannerCP is abstraction() {
		type AdaptationPlan is Any
```

```
        type Id is Integer
        type Type is Integer
        type ContextInf is Any
        type Operator is relational_operator
        type Value is Real
        type Action is Any
        type Priority is Integer
        type Context is tuple[Id, Type, ContextInfo, Value]
        type AdaptationPolicy is tuple[Id, Type, ContextInfo, Operator, Value, Action, Priority]
        connection adaptationRequest is in(Boolean)
        connection adaptationPlan is out(AdaptationPlan)
        connection adaptationInfo is out(Any)
        protocol is {
                (via adaptationRequest receive Boolean
                via adaptationPlan send AdaptationPlan
                via adaptationInfo send Any)*
        }
        behavior is {
                adaptationPlanFunction is function (c : Context) : AdaptationPlan {
                        unobservable
                }
                via adaptationRequest receive b_adaptationRequest : Boolean
                if (b_adaptationRequest) then {
                        via adaptationPlan send adaptationPlanFunction(context)
                }
                via adaptationInfo send context
                behavior()
        }
}

//CONECTOR adaptationRequestCN
connector AdaptationRequestCN is abstraction() {
        connection fromNetworkAnalyser is in (Boolean)
        connection toNetworkPlanner is out (Boolean)
        protocol is {
                (via fromNetworkAnalyser receive Boolean
                via toNetworkPlanner send Boolean)*
        }
        behavior is {
                via fromNetworkAnalyser receive adaptationRequest : Boolean
                via toNetworkPlanner send adaptationRequest
                behavior()
        }
}

// COMPONENT NETWORK KNOWLEDGE BASE
component NetworkKnowlegdeBaseCP is abstraction() {
        connection setObject is in(Any)
        connection getObject is out(Any)
        protocol is {
                (via setObject receive Any |
                via getObject send Any
                )*
        }
        behavior is {
                o is location[Any]
                storeValue is function(data : Any) {
                        o = data
                }
                returnValue is function(data2 : Any) : Any {
                        return o
                }
                via setObject receive c : Any
                storeValue(c)
                via getObject send returnValue(d)
                behavior()
        }
}

//CONECTOR getServices
connector GetServicesCN is abstraction() {
        connection fromNetworkKnowlegdeBase is in (Any)
        connection toApplicationManager is out (Any)
        protocol is {
                (via fromNetworkKnowlegdeBase receive Any
                via toApplicationManager send Any)*
        }
        behavior is {
                via fromNetworkKnowlegdeBase receive services : Any
                via toApplicationManager send services
                behavior()
        }
}

//CONECTOR setAdaptationPolicies
connector SetAdaptationPoliciesCN is abstraction() {
        connection fromAdaptationPoliciesManager is in (Any)
        connection toNetworkKnowlegdeBase is out (Any)
        protocol is {
                (via fromAdaptationPoliciesManager receive Any
                via toNetworkKnowlegdeBase send Any)*
        }
        behavior is {
                via fromAdaptationPoliciesManager receive adaptationPolicies : Any
                via toNetworkKnowlegdeBase send adaptationPolicies
                behavior()
        }
}

//CONECTOR sendCurrState
connector SendCurrStateCN is abstraction() {
        connection fromNetworkMonitor is in (Any)
        connection toNetworkKnowlegdeBase is out (Any)
        protocol is {
                (via fromNetworkMonitor receive Any
                via toNetworkKnowlegdeBase send Any)*
        }
        behavior is {
                via fromNetworkMonitor receive currentContext : Any
                via toNetworkKnowlegdeBase send currentContext
                behavior()
        }
}
```

```
 //CONECTOR getCurrState
connector GetCurrStateCN is abstraction() {
        connection fromNetworkKnowlegdeBase is in (Any)
        connection toNetworkAnalyser is out (Any)
        connection toNetworkPlanner is out (Any)
        protocol is {
                ((via fromNetworkKnowlegdeBase receive Any
                via toNetworkAnalyser send Any) |
                (via fromNetworkKnowlegdeBase receive Any
                via toNetworkPlanner send Any))*
        }
        behavior is {
                via fromNetworkKnowlegdeBase receive data : Any
                via toNetworkAnalyser send data
                //via fromNetworkKnowlegdeBase receive data : Any
                via toNetworkPlanner send data
                behavior()
        }
}

// COMPONENT NETWORK CONFIGURATION MANAGER
component NetworkConfigurationManagerCP is abstraction() {
        type AdaptationPlan is Any
        type Configuration is Any
        connection adaptationPlan is in (AdaptationPlan)
        connection wsnConfiguration is out (Configuration)
        protocol is {
                (via adaptationPlan receive AdaptationPlan
                via wsnConfiguration send Configuration)*
        }
        behavior is {
                generateConfiguration is function (a : AdaptationPlan) : Configuration {
                        unobservable
                }
                via adaptationPlan receive adaptationPlan : AdaptationPlan
                via wsnConfiguration send generateConfiguration(adaptationPlan)
                behavior()
        }
}
  //CONECTOR adaptationPlan
connector AdaptationPlanCN is abstraction() {
        type AdaptationPlan is Any
        connection fromNetworkPlanner is in (AdaptationPlan)
        connection toNetworkConfigurationManager is out (AdaptationPlan)
        protocol is {
                (via fromNetworkPlanner receive AdaptationPlan
                via toNetworkConfigurationManager send AdaptationPlan)*
        }
        behavior is {
                via toNetworkPlanner receive adaptationPlan : AdaptationPlan
                via toNetworkConfigurationManager send AdaptationPlan
                behavior()
        }
}

 //CONECTOR WSNConfiguration
connector WSNConfigurationCN is abstraction() {
        type Configuration is Any
        connection fromNetworkConfigurationManager is in (Configuration)
        connection toGatewayCommunication is out (Configuration)
        protocol is {
                (via fromNetworkConfigurationManager receive Configuration
                via toGatewayCommunication send Configuration)*
        }
        behavior is {
                via fromNetworkConfigurationManager receive configuration : Configuration
                via toGatewayCommunication send configuration
                behavior()
        }
}

component SecurityMarshallerCP is abstraction() {
        connection marshallRequest is in (Any)
        connection marshallReplay is out (Any)
        protocol is {
                (via marshallRequest receive Any
                via marshallReplay send Any)*
        }
        behavior is {
                encriptionFunction is function (e : Any) : Any {
                        unobservable
                }
                marshallerFunction is function (m : Any) : Any {
                        unobservable
                }
                via marshallRequest receive data : Any
                data2 = encriptionFunction(data)
                via marshallReply send marshallerFunction(data2)
                behavior()
        }
}

connector MarshallRequestCN is abstraction(){
        connection fromGatewayCommunication is in (Any)
        connection toSecurityMarshaller is out (Any)
        protocol is {
                (via fromGatewayCommunication receive Any
                via toSecurityMarshaller send Any)*
        }
        behavior is {
                via fromGatewayCommunication receive data : Any
                via toSecurityMarshaller send data
                behavior()
        }
}

connector MarshallReplayCN is abstraction(){
        connection fromSecurityMarshaller is in (Any)
        connection toGatewayCommunication is out (Any)
        protocol is {
                (via fromSecurityMarshaller receive Any
                via toGatewayCommunication send Any)*
        }
        behavior is {
                via fromSecurityMarshaller receive data : Any
                via toGatewayCommunication send data
                behavior()
        }
}

component NetworkRequestHandlerCP is abstraction() {
        connection sendMsgHandler is in (Any)
        connection receiveMsgHandler is out (Any)
        protocol is {
                (via receiveMsgHandler send Any |
                 via sendMsgHandler receive Any
                )*
        }
        behavior is {
                datafromNetworkFunction is function() : Any {
                        unobservable
                }
                datatoNetworkFunction is function(d : Any) {
                        unobservable
                }
                choose {
                        via receiveMsgHandler send datafromNetworkFunction()

                } or {
                        via sendMsgHandler receive data : Any
                        datatoNetworkFunction(data)
                }
                behavior()
        }
}

connector ReceiveMsgHandler_networkCN is abstraction(){
        connection fromRequestHandler is in (Any)
        connection toGatewayCommunication is out (Any)
        protocol is {
                (via fromRequestHandler receive Any
                via toGatewayCommunication send Any)*
        }
        behavior is {
                via fromRequestHandler receive data : Any
                via toGatewayCommunication send data
                behavior()
        }
}

 connector SendMsgHandler_networkCN is abstraction(){
        connection fromGatewayCommunication is in (Any)
        connection toRequestHandler is out (Any)
        protocol is {
                (via fromGatewayCommunication receive Any
                via toRequestHandler send Any)*
        }
        behavior is {
                via fromGatewayCommunication receive data : Any
                via toRequestHandler send data
                behavior()
        }
}

/*************************************************
* SENSOR MANAGEMENT LAYER
*************************************************/
 // COMPONENT SENSOR MANAGER
component SensorManagerCP is abstraction() {
        type IsManager is Boolean           type Context is Any
        type Configuration is Any type AdaptationPlan is Any
        connection receiveMessage is in (Any)
        connection sendMessage is out (Context)
        connection publishData is in (Context)
        connection setConfiguration is out(Configuration)
        connection analyzeData is out (Context)
        connection nodeAdaptationPlan is in (AdaptationPlan)
        connection update is out (Any)
        protocol is {
                ((via receiveMessage receive Configuration
                via setConfiguration send Configuration
                via update send Any) |      (via receiveMessage receive Context
                via publishData receive Configuration via analyzeData send Context
                via sendMessage send Context via update send Any ))*
        }
        behavior is {
                processConfiguration is function (a : AdaptationPlan) : Configuration {
                        unobservable
                        return c
                }
                processMessage is function (c : Configuration) : Boolean {
                        unobservable
                        return
                }

                choose {
                        via receiveMessage receive configuration : Configuration
                        via setConfiguration send configuration
                        if (IsManager) then {
                                dyComp = processMessage(configuration)
                                if (dyComp == True) then {
                                        compose{
                                                a is SensorAnalyserCP()
                                                and p is SensorPlannerCP()
                                                and a_ad is AnalyseDataCN()
                                                and p_ap is SensorAdaptationRequestCN()
                                                and k_u is UpdateCN()
                                                and p_s is SelectCN()
                                                and a_s is SelectCN()
                                        } where {
                                        sm::analyseData unifies a_ad::fromSensorManager
                                        a_ad::toSensorAnalyser unifies a::analyseData
                                        a::adaptationRequest unifies p_ar::fromSensorAnalyser
                                        p::sensorAdaptationPlan unifies sm_ap::fromSensorPlanner
                                        p::toSensorMonitor unifies sm::sensorAdaptationPlan
                                        }
                                }
                                via update send configuration
                via sendMessage send configuration
            }
        } or{
            if (IsManager) then {
                via nodeAdaptationPlan receive adaptationPlan : AdaptationPlan
                via update send adaptationPlan
                via setConfiguration send processConfiguration(adaptationPlan)
                via sendMessage send adaptationPlan
            }
        }or {
            via receiveMessage receive contextFromWSN : Context
            if (IsManager) then {
                via udpate send contextFromWSN
                via analyzeData send contextFromWSN
            }
            via sendMessage send contextFromWSN
        }or{
            via publishData receive contextFromSensor : Context
            if (IsManager) then {
                via udpate send contextFromSensor
                via analyzeData send contextFromSensor
            }
            via sendMessage send contextFromSensor
        }
        behavior()
    }
}

// COMPONENT SENSOR ANALYSER
component SensorAnalyserCP is abstraction() {
    type Context is Any
    connection analyseData is in(Boolean)
    connection selectRequest is out(Boolean)
    connection selectReply is in(Any)
    connection adaptationRequest is out(Boolean)
    protocol is {
        (via analyseData receive Boolean
        via selectRequest send Boolean
        via selectReply receive Any
        via adaptationRequest send Boolean
        )*
    }
    behavior is {
        analyseContext is function (c : Context) : Boolean {
            unobservable
            return True
        }
        via analyseData receive b_analyseData : Boolean
        if (b_analyseData) then {
            via selectRequest send true
            via selectReply receive context : Any
            if analyseContext(context) then {
                via adaptationRequest send (true)
            }
        }
        behavior()
    }
```

```
}

 // COMPONENT sensorPLANNER
component SensorPlannerCP is abstraction() {
        type AdaptationPlan is Any
        type Id is Integer
        type Type is Integer
        type ContextInf is Any
        type Operator is relational_operator
        type Value is Real
        type Action is Any
        type Priority is Integer
        type Context is tuple[Id, Type, ContextInfo, Value]
        type AdaptationPolicy is tuple[Id, Type, ContextInfo, Operator, Value, Action, Priority]
        connection adaptationRequest is in(Boolean)
        connection nodeAdaptationPlan is out(AdaptationPlan)
        protocol is {
                (via adaptationRequest receive Boolean
                via nodeAdaptationPlan send AdaptationPlan
                )*
        }
        behavior is {
                adaptationPlanFunction is function (c : Context) : AdaptationPlan {
                        unobservable
                }
                via adaptationRequest receive b_adaptationRequest : Boolean
                if (b_adaptationRequest) then {
                        via selectRequest send true
                        via nodeAdaptationPlan send adaptationPlanFunction(context)
                }
                behavior()
        }
}

component AcquisitonManagerCP is abstraction() {
        type Context is Any
        type Configuration is Any
        connection publishData is out(Context)
        connection setConfiguration is in(Configuration)
        protocol is {
                (via publishData send Context |
                via setConfiguration receive Configuration)*
        }
        behavior is {
                choose {
                        readContext is function() : Context {
                                unobservable
                        }
                        via publishData send readContext()
                } or {
                        configureSensors is function(c : Configuration) {
                                unobservable
                        }
                        via setConfiguration receive configuration  : Configuration
                        configureSensors(configuration)
                }
                behavior()
```

```
          }
}

 component SensorKnowlegdeBaseCP is abstraction() {
          connection setObject is in(Any)
          connection getObject is out(Any)
          protocol is {
                    (via setObject receive Any |
                    via getObject send Any
                    )*
          }
          behavior is {
                    o is location[Any]
                    storeValue is function(data : Any) {
                              o = data
                    }
                    returnValue is function(data2 : Any) : Any {
                              return o
                    }
                    via setObject receive c : Any
                    storeValue(c)
                    via getObject send returnValue(d)
                    behavior()
          }
}

component SensorCommunicationCP is abstraction() {
          type Context is Any
          type Configuration is Any
          connection sendMessage is in(Context)
          connection sendMsgHandler is out(Any)
          connection receiveMsgHandler is in(Any)
          connection receiveMessage is out(Configuration)
          connection securityMarshallRequest is out(Any)
          connection securityMarshallReplay is in(Any)
          protocol is {
                    (
                              (via receiveMsgHandler receive Any
                              via securityMarshallRequest send Any
                              via securityMarshallReplay receive Any
                              via receiveMessage send Configuration |
                              via sendMsgHandler send Any
                              ) |
                              (
                              via sendMessage receive Context
                              via securityMarshallRequest send Any
                              via securityMarshallReplay receive Any
                              via sendMsgHandler send Any
                              )
                    )*
          }
          behavior is {
                    choose {
                              via receiveMsgHandler receive configuration : Configuration
                              via securityMarshallRequest send configuration
                              via securityMarshallReplay receive marshalledConfiguration : Configuration
                              via receiveMessage send marshalledConfiguration
                              behavior()
                    } or {
                              via receiveMsgHandler receive contextHandler : Context
                              via securityMarshallRequest send context
                              via securityMarshallReplay receive marshalledContextHandler : Context
                              if (IsManager) then {
                                      via receiveMessage send marshalledContext
                              }
                              via sendMsgHandler send marshalledContext
                              behavior()
                    }or {
                              via sendMessage receive context : Context
                              via securityMarshallRequest send context
                              via securityMarshallReplay receive marshalledContext : Context
                              via sendMsgHandler send marshalledContext
                              behavior()
                    }
          }
}

component SensorSecurityMarshallerCP is abstraction() {
          connection marshallRequest is in (Any)
          connection marshallReplay is out (Any)
          protocol is {
                    (via marshallRequest receive Any
                    via marshallReplay send Any)*
          }
          behavior is {
                    encriptionFunction is function (e : Any) : Any {
                              unobservable
                    }
                    marshallerFunction is function (m : Any) : Any {
                              unobservable
                    }
                    via marshallRequest receive data : Any
                    data2 = encriptionFunction(data)
                    via marshallReply send marshallerFunction(data2)
                    behavior()
          }
}

connector MarshallRequest_sensorCN is abstraction(){
          connection fromSensorCommunication is in (Any)
          connection toSensorSecurityMarshaller is out (Any)
          protocol is {
                    (via fromSensorCommunication receive Any
                    via toSensorSecurityMarshaller send Any)*
          }
          behavior is {
                    via fromSensorCommunication receive data : Any
                    via toSensorSecurityMarshaller send data
                    behavior()
          }
}

connector MarshallReplay_sensorCN is abstraction(){
          connection fromSensorSecurityMarshaller is in (Any)
```

```
        connection toSensorCommunication is out (Any)
        protocol is {
                (via fromSensorSecurityMarshaller receive Any
                via toSensorCommunication send Any)*
        }
        behavior is {
                via fromSensorSecurityMarshaller receive data : Any
                via toSensorCommunication send data
                behavior()
        }
}

component SensorRequestHandlerCP is abstraction() {
        connection receiveMsgHandler is out (Any)
        connection sendMsgHandler is in (Any)
        protocol is {
                (via receiveMsgHandler send Any |
                 via sendMsgHandler receive Any
                )*
        }
        behavior is {
                datafromGatewayFunction is function() : Any {
                        unobservable
                }
                datatoGatewayFunction is function(d : Any) {
                        unobservable
                }
                choose {
                        via receiveMsgHandler send datafromGatewayFunction()
                } or {
                        via sendMsgHandler receive data : Any
                        datatoGatewayFunction(data)
                }
                behavior()
        }
}

connector ReceiveMsgHandler_sensorCN is abstraction(){
        connection fromRequestHandler is in (Any)
        connection toSensorCommunication is out (Any)
        protocol is {
                (via fromRequestHandler receive Any
                via toSensorCommunication send Any)*
        }
        behavior is {
                via fromRequestHandler receive data : Any
                via toSensorCommunication send data
                behavior()
        }
}

 connector SendMsgHandler_sensorCN is abstraction(){
        connection fromSensorCommunication is in (Any)
        connection toRequestHandler is out (Any)
        protocol is {
                (via  fromSensorCommunication receive Any
                via toRequestHandler send Any)*
        }
        behavior is {
                via  fromSensorCommunication receive data : Any
                via toRequestHandler send data
                behavior()
        }
}

connector SendMessageCN is abstraction() {
        connection fromSensorManager is in (Any)
        connection toSensorCommunication is out (Any)
        protocol is {
                (via fromSensorManager receive Any
                        via toSensorCommunication send Any)*
        }
        behavior is {
                via fromSensorManager receive data : Any
                via toSensorCommunication send data
                behavior()
        }
}

connector ReceiveMessageCN is abstraction() {
        connection fromSensorCommunication is in (Any)
        connection toSensorManager is out (Any)
        protocol is {
                (via fromSensorCommunication receive Any
                via toSensorManager send Any)*
        }
        behavior is {
                via fromSensorCommunication receive data : Any
                via toSensorManger send data
                behavior()
        }
}

connector PublishDataCN is abstraction() {
        type Context is Any
        connection fromAcquisitionManager is in (Context)
        connection toSensorManager is out (Context)
        protocol is {
                (via fromAcquisitionManager receive Context
                via toSensorManager send Context)*
        }
        behavior is {
                via fromAcquisitonManager receive context : Context
                via toSensorManager send context
                behavior()
        }
}

connector SetConfigurationCN is abstraction() {
        type Configuration is Any
        connection fromSensorManager is in (Configuration)
        connection toAcquisitionManager is out (Configuration)
        protocol is {
                (via fromSensorManager receive Configuration
                via toAcquisitionManager send Configuration)*
        }
        behavior is {
                via fromSensorManager receive configuration : Configuration
                via toAcquisitionManager send configuration
                behavior()
        }
}

connector AnalyseDataCN is abstraction() {
        connection fromSensorManager is in (Boolean)
        connection toSensorAnalyser is out (Boolean)
        protocol is {
                (via fromSensorManager receive Boolean
                via toSensorAnalyser send Boolean)*
        }
        behavior is {
                via fromNetworkMonitor receive adaptationIsNeeded : Boolean
                via toNetworkAnalyser send adaptationIsNeeded
                behavior()
        }
}

 connector SensorAdaptationPlanCN is abstraction() {
        type Configuration is Any
        connection fromSensorPlanner is in (Configuration)
        connection toSensorManager is out (Configuration)
        protocol is {
                (via fromSensorPlanner receive Configuration
                via toSensorManager send Configuration)*
        }
        behavior is {
                via fromSensorPlanner receive configuration : Configuration
                via toSensorManager send configuration
                behavior()
        }
}

 connector SensorAdaptationRequestCN is abstraction() {
        connection fromSensorkAnalyser is in (Boolean)
        connection toSensorPlanner is out (Boolean)
        protocol is {
                (via fromSensorkAnalyser receive Boolean
                via toSensorPlanner send Boolean)*
        }
        behavior is {
                via fromSensorkAnalyser receive adaptationRequest : Boolean
                via toSensorPlanner send adaptationRequest
                behavior()
        }
}

 connector UpdateCN is abstraction() {
        connection fromSensorManager is in (Any)
        connection toSensorKnowlegdeBase is out (Any)
        protocol is {
                (via fromSensorManager receive Any
            via toSensorKnowlegdeBase send Any)*
      }
      behavior is {
            via fromSensorManager receive data : Any
            via toSensorKnowlegdeBase send data
            behavior()
      }
}

connector SelectCN is abstraction() {
      connection fromSensorKnowlegdeBase is in (Any)
      connection toSensorAnalyser is out (Any)
      connection toSensorPlanner is out (Any)

      protocol is {
            (via fromSensorKnowlegdeBase receive Any
            via toSensorAnalyser send Any |
            via toSensorPlanner send Any)*
      }
      behavior is {
            choose {
                  via fromSensorKnowlegdeBase receive ap_a : Any
                  via toSensorAnalyser send ap_a
            } or {
                  via fromSensorKnowlegdeBase receive ap_p : Any
                  via toSensorPlanner send ap_p
            }
            behavior()
      }
}

/**************************************************
* REFERENCE ARCHITECTURE
**************************************************/
architecture rawsn is abstraction() {
      behavior is {
            compose {
                  //COMPONENTS GML
                  gml_am is ApplicationManagerCP()
                  and gml_apm is AdaptationPoliciesManagerCP()
                  and gml_im is InspectionManagerCP()

                        //CONNECTORS GML
                        and gml_am_pd is PublishWSNDataCN()
                        and gml_im_ai is AdaptationInfoCN()

                  //COMPONENTS NML
                  and nml_m is NetworkMonitorCP()
                  and nml_a is NetworkAnalyserCP()
                  and nml_p is NetworkPlannerCP()
                  and nml_e is NetworkConfigurationManagerCP()
                  and nml_k is NetworkKnowlegdeBaseCP()
                  and nml_g is GatewayCommunicationCP()
                  and nml_sm is SecurityMarshallerCP()
                  and nml_rh is NetworkRequestHandlerCP()

                        //CONNECTORS NML
```

```
                and nml_m_ar is AppReqCN()
                and nml_m_pc is PublishContextCN()
                and nml_a_ar is AnalysingRequestCN()
                and nml_p_ar is AdaptationRequestCN()
                and nml_e_ap is AdaptationPlanCN()
                and nml_g_wc is WSNConfigurationCN()
                and nml_g_rh is ReceiveMsgHandler_networkCN()
                and nml_k_gs is GetServicesCN()
                and nml_k_scs is SendCurrStateCN()
                and nml_k_ap is SetAdaptationPoliciesCN()
                and nml_k_gcs is GetCurrStateCN()
                and nml_sm_mrq is MarshallRequestCN()
                and nml_sm_mrp is MarshallReplayCN()
                and nml_rh_sm is SendMsgHandler_networkCN()

        //COMPONENTS SML
        and sml_sm is SensorManagerCP()
        and sml_am is AcquisitonManagerCP()
        and sml_a is SensorAnalyserCP()
        and sml_p is SensorPlannerCP()
        and sml_k is SensorKnowlegdeBaseCP()
        and sml_sc is SensorCommunicationCP()
        and sml_m is SensorSecurityMarshallerCP()
        and sml_rh is SensorRequestHandlerCP()

                //CONNECTORS SML
                and sml_m_mrq is MarshallRequest_sensorCN()
                and sml_m_mrp is MarshallReplay_sensorCN()
                and sml_rh_sm is SendMsgHandler_sensorCN()
                and sml_sc_rh is ReceiveMsgHandler_sensorCN()
                and sml_sc_sm is SendMessageCN()
                and sml_sm_rm is ReceiveMessageCN()
                and sml_sm_pd is  PublishDataCN()
                and sml_sm_ap is SensorAdaptationPlanCN()
                and sml_am_sc is SetConfigurationCN()
                and sml_a_ad is AnalyseDataCN()
                and sml_p_ap is SensorAdaptationRequestCN()
          and sml_k_u is UpdateCN()
          and sml_p_s is SelectCN()
          and sml_a_s is SelectCN()

} where {
        // GML
        gml_am::setAppReq unifies nml_m_ar::fromApplicationManager
        nml_m_ar::toNetworkMonitor unifies nml_m::setAppReq
        nml_p::adaptationInfo unifies gml_im_ai::fromNetworkPlanner
        gml_im_ai::toInspectionManager unifies gml_im::getAdaptationInfo

        // NML
        nml_m::publishWSNData unifies gml_am_pd::fromNetworkMonitor
        gml_am_pd::toApplicationManager unifies gml_am::publishWSNData
        nml_m::analysingRequest unifies nml_a_ar::fromNetworkMonitor
        nml_a_ar::toNetworkAnalyser unifies nml_a::analysingRequest
        nml_m::sendCurrState unifies nml_k_scs::fromNetworkMonitor
        nml_k_scs::toNetworkKnowlegdeBase unifies nml_k::setObject
        nml_a::adaptationRequest unifies nml_p_ar::fromNetworkAnalyser
        nml_p_ar::toNetworkPlanner unifies nml_p::adaptationRequest
```

```
                        nml_p::adaptationPlan **unifies** nml_e_ap::fromNetworkPlanner
                        nml_e_ap::toNetworkConfigurationManager **unifies** nml_e::adaptationPlan
                        nml_e::wsnConfiguration **unifies**
nml_g_wc::fromNetworkConfigurationManager
                        nml_g_wc::toGatewayCommunication **unifies** nml_g::wsnConfiguration

                        // SML
                        sml_sc::receiveMessage **unifies** sml_sc_rm::fromSensorCommunication
                        sml_sc_rm::toSensorManager **unifies** sml_sm::receiveMessage
                        sml_sm::sendMessage **unifies** sml_sc_sm::fromSensorManager
                        sml_sc_sm::toSensorCommunication **unifies** sml_sc::sendMessage
                        sml_sm::setConfiguration **unifies** sml_am_sc::fromSensorManager
                        sml_am_sc::toAcquisitonManager **unifies** sml_am::setConfiguration
                        sml_sm::analyseData **unifies** sml_a_ad::fromSensorManager
                        sml_a_ad::toSensorAnalyser **unifies** sml_a::analyseData
                        sml_am::publishData **unifies** sml_sm_pd::fromAcquisitionManager
                        sml_sm_pd::toSensorManager **unifies** sml_sm::publishData
                        sml_a::adpationRequest **unifies** sml_p_ar::fromSensorAnalyser
                        sml_p_ar::toSensorPlanner **unifies** sml_p::adaptationRequest

                        sml_p::sensorAdpataionPlan **unifies** sml_sm_ap::fromSensorPlanner
                        sml_sm_ap::toSensorMonitor **unifies** sml_sm::sensorAdaptationPlan
                }
        }
}
```

APPENDIX C: Pi-ADL specification of deployment view

```
component OrdinaryNode is abstraction(){
        type SMLMessage is Any
        connection sensor is in (String)
        connection radioOut is out (SMLMessage)
        connection radioIn is in (SMLMessage)
        protocol is {
                ((via sensor receive String | via radioIn receive SMLMessage))
                via radioOut send SMLMessage
                )*
        }
        behavior is {
                convertRawData is function(m: String) : SMLMessage {
                        unobservable
                }
                verifyTypeMessage is function(msg: SMLMessage) : Boolean {
                        unobservable
                }
                choose {
                        via sensor receive measure : String
                        via radioOut send convertRawData(measure)
                        behavior()
                } or {
                        via radioIn receive message : SMLMessage
                        if verifyTypeMessage(message) then {
                                via radioOut send message
                        }
                        behavior()
                }
        }
}

component ClusterHead is abstraction(){
        type SMLMessage is Any
        connection sensor is in (String)
        connection radioOut is out (SMLMessage)
        connection radioIn is in (SMLMessage)
        protocol is {
                ((via sensor receive String | via radioIn receive SMLMessage))
                via radioOut send SMLMessage
                )*
        }
        behavior is {
                convertRawData is function(m: String) : SMLMessage {
                        unobservable
                }
                verifyTypeMessage is function(msg: SMLMessage) : Boolean {
                        unobservable
                }
                choose {
                        via sensor receive measure : String
                        via radioOut send convertRawData(measure)
                        behavior()
                } or {
                        via radioIn receive message : SMLMessage
                        if verifyTypeMessage(message) then {
                                via radioOut send context
                        }
                        behavior()
                }
        }
}

component BaseStation is abstraction(){
        type SMLMessage is Any
        connection radioOut is out (SMLMessage)
        connection radioIn is in (SMLMessage)
        connection serialOut is out (SMLMessage)
        connection serialIn is in (SMLMessage)
        protocol is {
                ((via radioIn receive SMLMessage)
                via serialOut send NMLMessage) |
                (via serialIn receive NMLMessage)
                via radioOut send SMLMessage)
                ) *
        }
        behavior is {
                choose {
                        via radioIn receive smlMessage : SMLMessage
                        via serialOut send smlMessage
                        behavior()
                } or {
                        via serialIn receive nmlMessage : NMLMessage
                        via radioOut send nmlMessage
                        behavior()
                }
        }
}

component Gateway is abstraction(){
        type NMLMessage is Any
        type SMLMessage is Any
        connection tcpOut is out (NMLMessage)
        connection tcpIn is in (NMLMessage)
        connection serialOut is out (NMLMessage)
        connection serialIn is in (NMLMessage)
        protocol is {
                ((via tcpIn receive NMLMessage)
                via serialOut send NMLMessage) |
                (via serialIn receive NMLMessage)
                via tcpOut send NMLMessage)
                ) *
        }
        behavior is {
                convertMessage is function(smlM: SMLMessage) : NMLMessage {
                        unobservable
                }
                convertMessage is function(nmlM: NMLMessage) : SMLMessage {
                        unobservable
```

```
		}
		choose {
			via tcpIn receive nmlMessage : NMLMessage
			via serialOut send convertMessage(nmlMessage)
			behavior()
		} or {
			via serialIn receive smlMessage : SMLMessage
			via tcpOut send convertMessage(smlMessage)
			behavior()
		}
	}
}

component GlobalServer is abstraction(){
	type NMLMessage is Any
	connection tcpOut is out (SMLMessage)
	connection tcpIn is in (SMLMessage)
	protocol is {
		(via tcpIn receive SMLMessage |
		via tcpOut send SMLMessage)*
	}
	behavior is {
		processMessage is function(m: NMLMessage) {
			unobservable
		}
		choose {
			via tcpIn receive message : NMLMessage
			processMessage(message)
			behavior()
		} or {
			via tcpOut send message
			behavior()
		}
	}
}

connector ZigBee is abstraction() {
	type SMLMessage is Any
	connection input is in (SMLMessage)
	connection output is out (SMLMessage)
	protocol is {
		(via input receive SMLMessage
		via output receive SMLMessage)*
	}
	behavior is {
		via input receive rm : SMLMessage
		via output send rm
		behavior()
	}
}

connector Serial is abstraction() {
	type SMLMessage is Any
	connection input is in (SMLMessage)
	connection output is out (SMLMessage)
	protocol is {
		(via input receive SMLMessage
                via output receive SMLMessage)*
        }
        behavior is {
                via input receive rm : SMLMessage
                via output send rm
                behavior()
        }
}

connector TcpIp is abstraction() {
        type NMLMessage is Any
        connection input is in (NMLMessage)
        connection output is out (NMLMessage)
        protocol is {
                (via input receive NMLMessage
                via output receive NMLMessage)*
        }
        behavior is {
                via input receive rm : NMLMessage
                via output send rm
                behavior()
        }
}

architecture RAMSES_DeploymentView is abstraction() {
        behavior is {
                compose {
                        ON is OrdinaryNode()
                        and CH is ClusterHead()
                        and BS is BaseStation()
                        and GW is Gateway()
                        and AS is GlobalServer()
                        and zb is ZigBee()
                        and ser is Serial()
                        and tcp is TcpIp()
                } where {
                        ON::radioOut unifies zb::input
                        zb::output unifies ON::radioIn
                        CH::radioOut unifies zb::input
                        zb::output unifies CH::radioIn

                        BS::serialOut unifies ser::input
                        ser::output unifies BS::serialIn
                        BS::radioOut unifies zb::input
                        zb::output unifies BS::radioIn

                        GW::serialOut unifies ser::input
                        ser::output unifies GW::serialIn
                        GW::tcpOut unifies tcp::input
                        tcp::output unifies GW::tcpIn

                        GS::tcpOut unifies tcp::input
                        tcp::output unifies GS::tcpIn
                }
        }
}
```