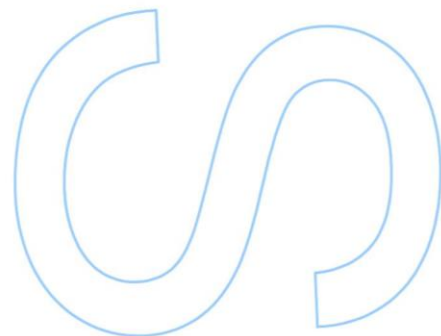
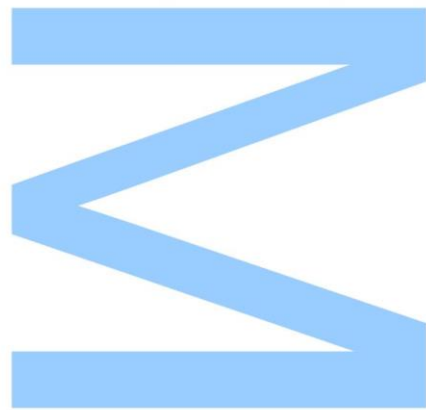


# Towards Out-of-the-Box Programming of Wireless Sensor Networks

Edgard Quirino dos Santos Neto  
Mestrado em Ciência de Computadores  
Departamento de Ciência de Computadores  
2014

## **Orientador**

Luís Miguel Barros Lopes, Professor Associado, Faculdade de Ciências da Universidade do Porto

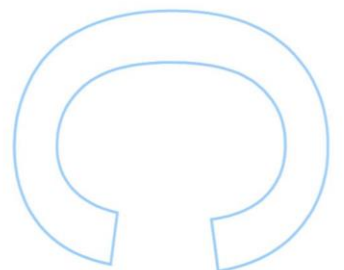
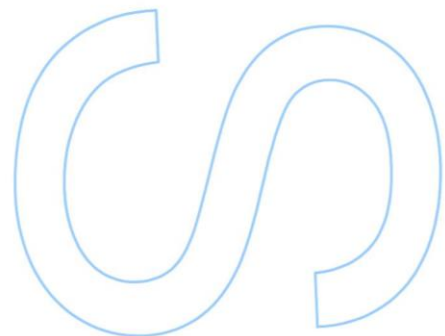
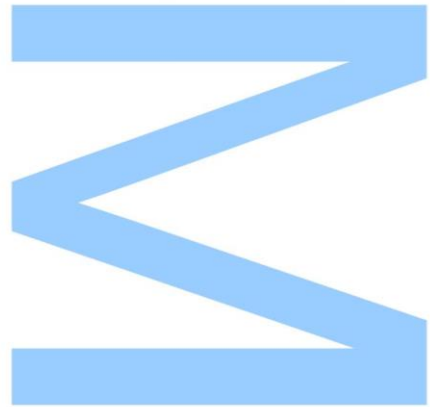




Todas as correções determinadas pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, \_\_\_\_ / \_\_\_\_ / \_\_\_\_



To my grandfather that once told me:

*"For those who are free, the mind shall be key."*

# Acknowledgements

I would like to thank my friend and supervisor Prof. Luís Lopes for in 2012 offering me a research scholarship, and the possibility to work with him. Further, I would like to express my gratitude to him, for all the support given in the work during this thesis, and the time dispended helping me with the writing of this thesis. I also would like to thank Roberto Silva for his contribution with the new version of the data layer, Carlos Machado for all the help with the hardware configuration and assembly, and Francisco Martins (Faculty of Science, University of Lisbon) for advice in the programming of the Arduino devices.

I would like to express my thanks to my dear friends Inês Castro Dutra, Tiago Travassos Vieira Vinhoza, Pedro Miguel Ferreira and José Serra, who have supported me all the time, helped me to achieve all my purposes and also provided me with the funniest moments. I am very grateful to all of you.

I am very grateful for all the support given by my family, namely to my mother, my stepfather and my little sister Sofia. In particular, I would like to say that is thanks to Sofia's smiles (which I can only see through a Skype video call), that I could find the necessary strength to continue this journey.

This work has been sponsored by projects MACAW (FCT contract: PTDC/EIA-EIA/115730/2009) and RTS (contract: NORTE-07-0124- FEDER-000062).

# Resumo

Uma Rede de Sensores-Actuadores Sem Fio (RSSF) é um conjunto de nós destinados a gravar, interagir e monitorizar as condições em diversas localizações. Os nós são dispositivos portáteis e leves que são usados para monitorizar parâmetros como por exemplo temperatura, humidade, pressão arterial e intensidade de luminosa.

A versatilidade desta tecnologia permite que ela seja usada em diferentes aplicações de diferentes áreas, tais como saúde, vigilância ambiental e casas inteligentes. No entanto, mesmo sendo uma tecnologia tão versátil, o seu uso ainda é restrito a utilizadores com alguma experiência na área. Esta experiência é necessária uma vez que a maioria das aplicações requerem do utilizador conhecimentos em programação de baixo nível e habilidades de configuração de *hardware*. No entanto, estas habilidades não são encontradas entre os utilizadores não especialistas, e isso certamente torna mais difícil a massificação desta tecnologia.

Com base nos factos mencionados acima, o nosso objectivo é produzir um sistema de fácil utilização, que permita que até utilizadores não especializados possam monitorizar e interagir com suas RSSF. Para alcançar esse objectivo nós propomos como solução a arquitetura SONAR.

SONAR é uma arquitetura de *software* de três camadas: cliente, processamento e dados. A camada cliente é usada para monitorizar, programar e injetar tarefas, sobre a RSSF. As tarefas são programadas utilizando uma linguagem de domínio específico, e posteriormente são executadas nos nós. A camada processamento é responsável pelo gerenciamento da RSSF e do armazém de dados. A camada dados, abstrai as especificidades das RSSF, pois os nós já vem com um sistema operativo e uma máquina virtual previamente instalados. Esta arquitetura faz do SONAR, uma solução completa e atraente, inclusive para utilizadores finais não especialistas, uma vez que não é preciso recorrer a programação de baixo nível, nem configurações de *hardware*, para implementar a sua RSSF.

# Abstract

A Wireless Sensor-Actuator Network (WSN) is a set of sensor nodes intended to record, to interact, and to monitor conditions at diverse locations. The sensor nodes are portable, lightweight devices that are used to monitor parameters such as temperature, humidity, blood pressure and light intensity.

The versatility of this technology allows it to be used in different applications of different areas, such as: health monitoring, environment surveillance, and smart houses. However, even being such a versatile technology, its usage still restricted to users with expertise on the field. This expertise is needed, since most applications require low level programming and hardware configuration skills. These skills, are certainly not found among non-specialist users, and this makes the massification of this technology more difficult.

Based on the aforementioned facts, our goal is to produce a user-friendly software system, that allows for even non-specialist users, to monitor and interact with WSN. In order to achieve such a goal, we propose the SONAR architecture.

SONAR is a 3-layer software architecture: client, processing and data. The client layer is used to monitor, program and inject tasks, over the WSN. The tasks are programmed using a simple domain specific language, and are executed in the nodes. The processing layer is responsible for managing WSN and data-stores. The data layer, abstracts WSN deployments with factory installed software, composed of an operating system and a virtual machine. This architecture makes SONAR, a complete and attractive solution, even for non-specialist end-users, since one does not need to resort to low level programming nor hardware configuration, to configure and deploy its WSN.

# Contents

<b>Acknowledgements</b>	<b>4</b>
<b>Resumo</b>	<b>5</b>
<b>Abstract</b>	<b>6</b>
<b>List of Tables</b>	<b>10</b>
<b>List of Figures</b>	<b>13</b>
<b>1 Introduction</b>	<b>14</b>
1.1 Motivation . . . . .	14
1.2 Problem Statement and Proposed Solution . . . . .	15
1.3 Outline . . . . .	16
<b>2 Related Work</b>	<b>17</b>
2.1 Wireless Sensor Networks . . . . .	17
2.2 Middleware Infrastructure . . . . .	19
2.3 Virtual Machines and Operating Systems . . . . .	22
2.4 Web Services and Data Store Models . . . . .	23
2.5 Summary . . . . .	25

<b>3</b>	<b>The SONAR Architecture</b>	<b>26</b>
3.1	Overview . . . . .	26
3.2	The Data Layer . . . . .	27
3.3	The Processing Layer . . . . .	28
3.4	The Client Layer . . . . .	30
3.5	Summary . . . . .	31
<b>4</b>	<b>Prototype</b>	<b>32</b>
4.1	Overview . . . . .	32
4.2	The Data Layer . . . . .	33
4.2.1	Building the Data layer . . . . .	33
4.2.2	Data Layer Components . . . . .	36
4.3	The Processing Layer . . . . .	40
4.4	The Client Layer . . . . .	44
4.4.1	The Network Tab . . . . .	44
4.4.2	The Data Tab . . . . .	44
4.4.3	The Task Tab . . . . .	46
4.5	Summary . . . . .	47
<b>5</b>	<b>Discussion</b>	<b>48</b>
5.1	Weaknesses . . . . .	48
5.2	Solutions . . . . .	49
5.3	Summary . . . . .	50
<b>6</b>	<b>The SONAR Task Language</b>	<b>51</b>
6.1	Syntax . . . . .	51
6.2	Operational Semantics . . . . .	52



6.3	Static Semantics . . . . .	55
<b>7</b>	<b>The SVM and SOS</b>	<b>59</b>
7.1	The SONAR Virtual Machine . . . . .	59
7.1.1	Byte-code . . . . .	59
7.1.2	Translation . . . . .	62
7.1.3	Semantics . . . . .	64
7.2	The SONAR Operating System . . . . .	64
7.2.1	Gateway . . . . .	65
7.2.2	Nodes . . . . .	66
7.3	Implementation . . . . .	68
<b>8</b>	<b>The Processing and Client Layers</b>	<b>71</b>
8.1	Processing Layer . . . . .	71
8.1.1	The Data Store . . . . .	71
8.1.2	The Generic Interface for communication with data store . . . . .	73
8.2	The Client Layer . . . . .	75
8.2.1	Deployment List View . . . . .	77
8.2.2	Task View . . . . .	78
8.2.3	Admin Mode . . . . .	79
<b>9</b>	<b>Conclusions and Future Work</b>	<b>83</b>

# List of Tables

7.1	Memory consumption for the gateway and nodes. . . . .	70
-----	---	----

# List of Figures

2.1	Sensor node. . . . .	18
2.2	Smart city Santander Project WSN Infrastructure. . . . .	19
2.3	Web Service architecture. . . . .	24
3.1	The SONAR architecture. . . . .	27
3.2	Data layer control and data flow . . . . .	28
3.3	Processing layer control and data flow . . . . .	29
3.4	Client layer control and data flow . . . . .	30
3.5	The SONAR architecture's data flow. . . . .	31
4.1	Builder tool application process . . . . .	34
4.2	Builder Tool Interface for the SunSPOT. . . . .	34
4.3	The startApp method for the MIDlet running on the nodes. . . . .	35
4.4	The code for a generic reader. . . . .	36
4.5	The code for a generic interpreter. . . . .	37
4.6	The code for the data forwarder class in the gateway. . . . .	39
4.7	The code for the command forwarder class in the gateway. . . . .	39
4.8	The interface of the adapter. . . . .	40
4.9	EER diagram for the SONAR data-store. . . . .	41
4.10	The interface of the SONAR web-service. . . . .	42

4.11	The network tab. . . . .	45
4.12	The data visualization tab. . . . .	45
4.13	The task management tab. . . . .	46
6.1	The syntax of STL. . . . .	52
6.2	STL program that turns on and off a sprinkler according to ambient temperature and humidity. . . . .	53
6.3	Reduction rules for STL instructions. . . . .	54
6.4	Reduction rules for STL expressions. . . . .	56
6.5	Type system for STL (part I). . . . .	57
6.6	Type system for STL (part II). . . . .	58
7.1	Byte-code syntax. . . . .	60
7.2	Translation to bytecode (part I). . . . .	61
7.3	Translation to bytecode (part II). . . . .	62
7.4	Transition rules for SVM. . . . .	63
8.1	EER diagram for the new SONAR data-store . . . . .	72
8.2	DAO for Data mapped object . . . . .	74
8.3	DAO for Task mapped object . . . . .	75
8.4	DAO for Deployment mapped object . . . . .	76
8.5	Factory Method for retrieve the appropriate implementation . . . . .	76
8.6	Login and SONAR-service connect window. . . . .	77
8.7	Deployment view. . . . .	77
8.8	Task view: Context Menu. . . . .	78
8.9	Data view. . . . .	79
8.10	The Management menu. . . . .	80

8.11	The register data store window. . . . .	80
8.12	The register user window. . . . .	81
8.13	The register deployment window. . . . .	81
8.14	The Authorize window. . . . .	81
8.15	EER diagram of the admin's database. . . . .	82

# Chapter 1

## Introduction

### 1.1 Motivation

Consider the following case scenario: A greenhouse owner buys a kit composed of a few nodes, a gateway node, and a software package to be installed, say, in his home computer. The nodes have temperature and humidity sensors and an actuator that is used to turn on and off a water sprinkler. After placing the nodes strategically in the greenhouse, the user installs the software in his home computer. Afterwards, the user connects the gateway node to a USB port. The software first starts a web-service, used to manage the deployment. Afterwards, the user can manage the deployment by running a web client, from anywhere in the Internet with access to his home computer. The client connects to the web-service and allows the user to visualize incoming data from the sensor nodes, and to manage their WSN with dynamically programmed tasks.

The above paragraph exemplifies a success scenario, in which a non-specialist end-user, is able to configure, to deploy and to monitor its WSN. However, that perfect scenario, does not correspond to the reality, in which a degree of expertise is required from the end-user. In particular, part of this required expertise, results from the variety of hardware platforms available on the market e.g., SunSPOT, Arduino, Mica, Firefly, WASP-motes. These platforms are based on nodes with the following features:

- different combinations of sensors and actuators;
- distinct communication and routing protocols, e.g., ZigBee, XBee, Bluetooth, WiFi;

- programmed using distinct programming languages, e.g., Java, nesC, C.

This heterogeneity has a positive impact, from the point of view of specialists and/or end-users with expertise in the field, since it allows them to build and deploy different WSN, optimized for a given application. On the other hand, it has a negative impact on the effort required to port, configure, and deploy a given application onto distinct platforms. In addition, a typical end-user without expertise in the field wishing to deploy such an infra-structure, for personal or business use, would see such heterogeneity as daunting. This makes WSN technology less appealing to the consumer market and certainly precludes its wider dissemination.

Despite this heterogeneity, we argue that most end-user applications running on WSN have similar *modus operandi*, namely: (a) periodically reading values from sensors on the nodes and sending them to a gateway, each node generating a data-stream for each set of sensed environment variables; (b) executing commands on the on-board actuators of the nodes, usually triggered by off-line processing of the aforementioned data-streams.

Based on this observation, we argue that it is possible to design and implement an architecture that would allow even non-specialist end-users to buy hardware/software kits and seamlessly configure, deploy and manage a WSN, without performing subtle hardware or software configurations. This philosophy, we believe, would go a long way in making the technology more appealing to end-users.

## 1.2 Problem Statement and Proposed Solution

Currently the configuration, deployment and management of a WSN is still restricted to specialists on the field. This restriction exists, mainly because of the diversity in hardware platforms, which requires different knowledge in low level programming and hardware configuration from the end-user. However, the aforementioned knowledge is not commonly found among non-specialist end-users, and it certainly prevents the massification of the technology. The problem we address in this thesis is therefore:

*How to massify the use of WSN, by providing a software framework that is easily ported to different platforms and at the same time, is user-friendly and accessible to non-specialist end-users ?*

Our answer to the above question is the Sensor Observation aNd Actuation aRchitecture (SONAR). SONAR is a 3-layer middleware architecture, composed of: (a) client layer; (b) processing layer and; (c) data layer. The client layer provides a graphical and user-friendly application to the end-user. This application allows for the end-user to inject dynamic programmed tasks into their WSN and monitor its produced data. The processing layer manages all the WSN, by storing relevant information concerning its tasks and produced data, into an appropriate data-store. The data layer, abstracts away the low level complexity of the hardware platform. This abstraction, is achieved with the usage of two pre-installed software components: operating system and virtual machine. The operating system is responsible for managing the resources and scheduling the tasks. The virtual machine is responsible for the execution of the tasks and the abstraction of the hardware platform. This architecture makes SONAR a complete solution for WSN, since it provides the end-user with not only a user-friendly application for the management of their WSN, but also with the necessary hardware, with pre-installed software components, ready to be deployed.

### 1.3 Outline

The remainder of this thesis is structured as follows. Chapter 2 presents relevant work for this thesis concerning WSN applications, middleware infrastructures, virtual machines, operating systems, web services and data stores. Chapter 3 presents SONAR's architecture, followed by a detailed description of each layer and its components. Chapter 4 details the design and implementation of a prototype, based on the SONAR's architecture. Chapter 5 follows by detailing the significant improvements made on the prototype, of the previous chapter. Finally, in the concluding Chapter 6, we provide conclusions, and present some ideas and opportunities for future work.



# Chapter 2

## Related Work

In this chapter we give an overview of work which is relevant for this thesis.

### 2.1 Wireless Sensor Networks

Akyildiz et al. describe a WSN as "[...] composed of numerous sensor nodes, which are densely deployed either inside the phenomenon or very close to it. The position of sensor nodes need not be engineered or pre-determined. This allows random deployment in inaccessible terrains or disaster relief operations. On the other hand, this also means that sensor network protocols and algorithms must possess self-organizing capabilities. Another unique feature of sensor networks is the cooperative effort of sensor nodes. Sensor nodes are fitted with an on-board processor. Instead of sending the raw data to the nodes responsible for the fusion, sensor nodes use their processing abilities to locally carry out simple computations and transmit only the required and partially processed data." [1]. An illustrative example of a typical sensor node, is depicted in the Figure 2.1. In particular, this sensor node, is an Arduino mega 2560 microcontroller, fitted with temperature and humidity sensors, and also a radio module, for transmitting and receiving data. These are the nodes, we use to test SONAR, later in this thesis. However, a sensor node can be assembled with any type of sensors, depending on the application. This makes the technology, more versatile and not restricted to just only a few case scenarios. An example of the usage of WSN, in a large scale, is the project Smart Santander [2]. This project uses the sensor nodes to monitor different parameters of the city Santander, such as noise, temperature, luminosity and CO<sub>2</sub> levels. Each node sends its data to the gateway. That forwards it

to the system platform, which is responsible for processing and storing the data. This WSN infrastructure is depicted in the Figure 2.2 ( from [2]).

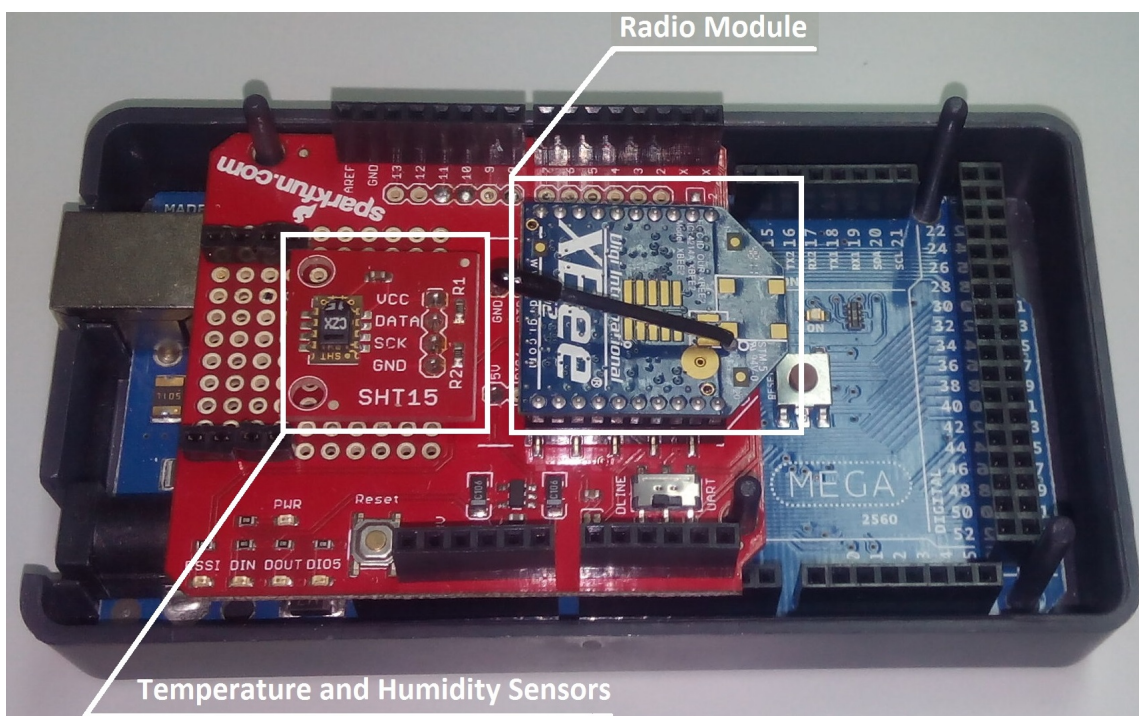


Figure 2.1: Sensor node.

Thus, WSNs can be made of a variety of sensor nodes, e.g., temperature, humidity, pressure and accelerometer. As a consequence, their usage in different applications and environments have been steadily growing, namely those related to:

- E-Health: body monitoring, medical drug preservation and security for wireless medical sensors [3, 4, 5].
- Environmental Surveillance: Monitoring fires in plantations, soil moisture, wind speed and rainfall [6].
- Intelligent City: monitoring different parameters such as noise, traffic light and CO levels [2].

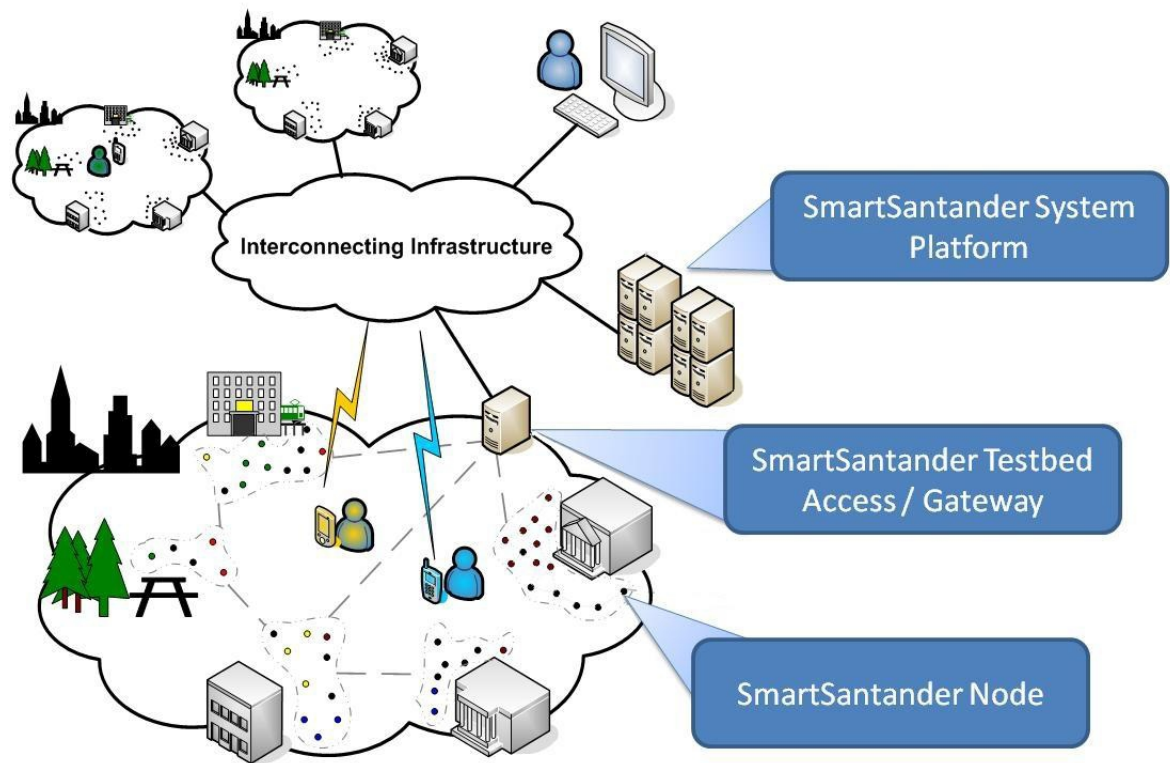


Figure 2.2: Smart city Santander Project WSN Infrastructure.

## 2.2 Middleware Infrastructure

Wang et al. describe a middleware as a layer of "Software and tools that can help hide the complexity and heterogeneity of the underlying hardware and network platforms, ease the management of system resources, and increase the predictability of application executions. WSN middleware is a kind of middleware providing the desired services for sensing-based pervasive computing applications that make use of a wireless sensor network and the related embedded operating system or firmware of the sensor nodes." [7].

TinySOA is a multi-platform service-oriented architecture for WSN that can be used to monitor data from different deployments [8]. The architecture has four main components: *node*, *gateway*, *registry* and *server*. The node component encapsulates all the functionality of a sensing node and resides in all sensing nodes in the network. The gateway component is normally located in a specialized node or computer and acts as a bridge between a WSN and the Internet. It is possible to have multiple WSNs with

different platforms, but each one must have its own gateway. The registry component is basically a database where all the information about the infrastructure is stored (sensor readings, available sensor networks). Finally, the server component acts as a provider of a web service, containing an interface used to consult the services offered by each of the networks. This general architecture trait is similar to SONAR's. Although, when compared with SONAR, it does not provide the possibility to directly interact with the WSN, by sending actuation commands and/or dynamically reprogram the deployment.

Global Sensor Networks (GSN) introduces the concept of virtual sensor, to allow users to focus on XML-based high-level descriptions of deployments, to describe the applications running on a WSN platform [9]. It provides built-in distributed querying, filtering and combination of sensor data algorithms. The lower level building blocks that compose the virtual sensor are platform specific and are provided in the platform or, for new unsupported platforms, a port must be available. To the user, however, this is completely transparent. This philosophy is similar to that adopted in SONAR, in the sense that all the specific and technical details, concerning the deployment, are completely transparent to the user. However, SONAR offers the possibility to directly interact with the WSN, through dynamic programmed tasks, using a graphical and intuitive application, which is a feature that GSN does not provide.

Sens-ation is a service-oriented architecture that facilitates the development of context-aware sensor-based infrastructures [10]. It is aimed not only to WSN infrastructures but also to ubiquitous computing platforms. The architecture is composed of multiple layers: *discovery and request*, *processing*, *persistence*, *handling and registry*, *adapter* and *sensor and actor*. In a typical deployment, there will be various sensors (in the sensor and actor layer) that capture and send data to the Sens-ation platform (layers: *discovery and request*, *processing*, *persistence*, and *handling and registry*, from top to bottom, respectively) via adapters (adapter layer). The handling layer manages registered sensors and the persistence layer stores the data. In the processing layer it is possible to process sensor data using inference engines. Finally, the clients can retrieve data from the server via various gateways provided by the discovery and request layer. The Sens-ation project, presents us with a complex and complete architecture, which in some points resembles SONAR, namely adapter and processing layers, that requires a considerable expertise to configure and use with a deployment. However, it does not offer the possibility to direct interact with the WSN, through dynamically programmed tasks.

IrisNet envisions a world-wide sensor web in which users, via standard web-services,

can transparently make queries on data from thousands to millions of widely distributed, heterogeneous nodes [11]. The manifold sensor nodes in the network collect data and store observations in local databases. The data is then transmitted across the Internet as it is requested by user queries. The user sees these multiple databases as a single unit supported by a high-level query language, and the queries can be made from anywhere in the Internet. One particular characteristic of IrisNet is that a single sensor node may run several tasks, providing data for different services at different rates. These tasks are managed by an IrisNet run-time installed at the node. While apparently not supporting the use of actuators on the sensors, IrisNet does allow users to reconfigure data-collection and filtering processes in reaction to sensed data (e.g., changing sampling rates, invoking a special-purpose processing routine). The system allows programmers to develop sensing services by providing high-level abstractions and interfaces for the sensing infrastructure that hide the complexities of the underlying distributed-data-collection and query-processing mechanisms. Although the IrisNet is similar to SONAR regarding the usage of web services for providing access to stored information, the goal is totally different. IrisNet aims to manage large deployments geographically distributed. SONAR targets small to medium deployments, with simple applications.

HERA is an agent-based architecture that allows the creation of a wireless sensor network using devices with different technologies [12]. It is based on SYLPH, a platform that provides a service-oriented approach to managing heterogeneous WSN [13]. SYLPH has its own language to represent and define services. In order to support a new WSN platform, developers need to provide the implementation of the relevant services for the target platform. Nodes in a WSN use a SYLPH Gateway to communicate with the other software components of the platform. A gateway is a device with several hardware network interfaces, each of which connected to a distinct WSN, allowing nodes with distinct hardware specifications to transfer data. The HERA platform adds reactive agents to SYLPH. These agents are pre-programmed in each node and provide the means to develop intelligent context-aware applications for heterogeneous WSNs. The agents can make use of planning mechanisms and adapt dynamically to new situations. However, to program the agents, the user needs a good knowledge of the programming language used by the devices. All agents, once initialized, register in an agent directory where the information about their functionality is stored. Although the HERA project provides a direct interaction with the WSN, it requires from the end-user a good knowledge of the programming language used by the devices. This knowledge, is certainly not found in a typical end-user, which can see it as daunting. This project also differs from SONAR, since we allow for dynamic programming and

uploading of tasks to nodes.

Corona is a WSN distributed query processor implemented for SunSPOT devices [14, 15]. Each node in the platform has a run-time query engine implemented on top of the Squawk virtual machine. The query language is a simplified form of SQL and supports distributed queries and in-network data aggregation in order to reduce transmission size and conserve battery. The platform supports time-triggered queries since the nodes in the SunSPOT network initially synchronize. This allows time-scheduled queries to be performed on the network. The data transmitted by the sensors is compressed to reduce transmission size. Another feature of Corona is a RMI interface for application programmers allowing Java applications to interact with Corona. This platform is extendable to new custom sensors or different data types, but for that, the user needs to be comfortable with the Java programming language. Although Corona, offers to the end-user a way to direct interact with the WSN, it is more restrictive than SONAR, since it only supports the SunSPOT platform. Also, even this platform being extendable to new custom sensors, it requires from the end-user a good knowledge of Java programming language. However, it fundamentally differs from SONAR, since we can offer support for any platform, and at the same time we do not require any technical knowledge from the end-user.

### 2.3 Virtual Machines and Operating Systems

A virtual machine (VM) is a software implementation of a machine (e.g., a computer) that executes programs like a physical machine. A VM was originally defined by Popek and Goldberg as "[...] an efficient, isolated duplicate of a real machine.". However, current use includes virtual machines that have no direct correspondence to any real hardware [16].

Mate [17] is a compact virtual machine implemented on top of TinyOS. Programs, called capsules, may be injected in the network at any time to perform specific tasks. They are written in a very simple assembly language and have the capability to move between sensor nodes.

The Regiment macro-programming language implements the Distributed Token Machine, based on an event-based programming model [18]. Each token is a typed message with some data or code that triggers a specific handler upon reception.

Sun Microsystems (now Oracle) introduced the Squawk virtual machine to support

applications for their SunSPOT devices [19]. Squawk is a very compact Java virtual machine, with a simplified byte-code layout that runs without an underlying operating system.

Operating systems, on the other hand, have their focus on resource management, since the usage of the low memory capacity, cpu speed and battery on embedded systems have to be optimized to their maximum.

In what concerns operating systems, TinyOS is perhaps the most widespread [20]. It provides a simple event-based execution-model with non-preemptive tasks. The system is loaded onto the sensor nodes as a set of modules linked with the user application.

Contiki is also based on an event-driven execution-model but supports multi-threaded applications, using very lightweight threads, and the dynamic loading of program modules [21].

SOS, also event-driven, is built from very small modules these are dynamically loaded, using a clever memory management scheme [22].

MANTIS and Nano-RK diverge from the above systems in that they support preemptive multi-threading, required for real-time and critical systems [23, 24]. A more comprehensive survey can be found in [25].

In SONAR the operating system (OS), and the virtual machine (VM) are integrated with the hardware. The OS is responsible for scheduling non-preemptive tasks, according to their periods. These tasks are run in an instance of the VM, and only one instance of the VM is running at a time. Thus, the OS does not provide support for preemptive tasks nor multiple threads, which makes it different from MANTIS, Nano-RK and Contiki. The VM is stack based, with a simple Instruction Set Architecture (ISA). It is compact in the same way as Mate, and can also receive tasks at any time. However, unlike Mate capsules, SONAR tasks do not have the capability to move between sensor nodes.

## 2.4 Web Services and Data Store Models

The World Wide Web Consortium (W3C) defines a Web service as [26]:

*"A software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-*

*processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.”*

As stated in the above definition, web services traditionally uses the following technologies:

- eXtensible Markup Language (XML) [27];
- Simple Object Access Protocol (SOAP) [28];
- Web Services Description Language (WSDL) [29];
- Universal Description, Discovery and Integration (UDDI) [30].

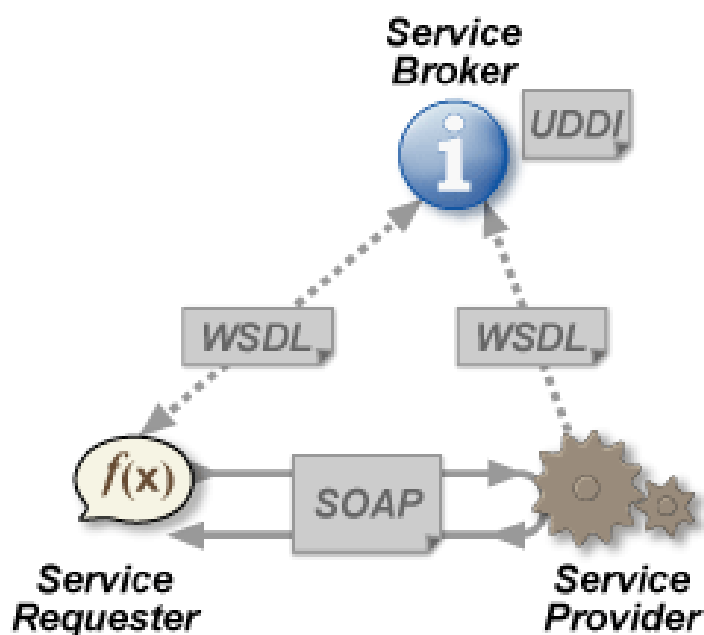


Figure 2.3: Web Service architecture.

The interaction among these components is depicted in the Figure 2.3, and are as follows: the service provider sends a WSDL file to the service broker (UDDI). The service requester contacts the broker to find out who is the provider for the data it needs, and then it contacts the service provider using the SOAP protocol. The service provider validates the service request and sends structured data in an XML file, using



the SOAP protocol. This XML file would be validated again by the service requester using an XSD file [31].

Web services are used for providing a programmatic interface across the Internet, without concerning administrative domains. This is possible, since web services are passing messages over HTTP, which is different from other methods such as Remote Procedure Call (RPC), which uses local ports to establish a connection with the remote machine, and then perform the routine.

A Data Store is a data repository of a set of integrated objects. These objects are modeled using classes defined in database schemes. Data store includes not only data repositories like databases, it is a more general concept that includes also flat files that can store data. Examples of different types of data stores are: Relational Database Management System (RDBMS); Wide Column Stores (WCS) and; Document Store (DS). RDBMS data stores may implement SQL databases, such as MySQL and PostgreSQL [32, 33]. However, WCS and DS, may use NoSQL databases such as Cassandra and MongoDB [34, 35].

A complete example of usage of both web service and data store, is the Amazon Simple Store Service (Amazon S3). Amazon S3 provides a web service interface that can be used to store and retrieve any amount of data, at any time, from anywhere on the web. It gives any developer access to the same highly scalable, reliable, secure, fast, inexpensive infrastructure that Amazon uses to run its own global network of web sites. The service aims to maximize benefits of scale and to pass those benefits on to developers [36].

## 2.5 Summary

In this chapter we presented an overview of the relevant work, for this thesis, such as WSN applications, middleware infrastructures, web services and data stores.

In the next chapter, we are going to introduce the SONAR's architecture, and detail all of its layers and components.

# Chapter 3

## The SONAR Architecture

In this chapter we present the architecture for SONAR, describing its layers and components. The chapter is organized into five sections: Overview, which will introduce the architecture and its basic features and functionalities; The Data Layer, The Processing Layer and The Client Layer, all of them corresponding to the architecture's layers and finally, a Summary.

### 3.1 Overview

SONAR is a typical 3-layer architecture, and it is depicted in the Figure 3.1. The data layer abstracts the WSN deployments managed by the architecture. These deployments generate data-streams, that are stored in a data-store in the processing layer. This data can be queried by the SONAR-service and processed in the task pool or in the client layer, in order to extract information on the status of nodes.

Clients are allowed to manage deployments in a disconnected way, through management tasks, and to forward tasks to the task pool. Every task, in the task pool, periodically queries the data store, processes the obtained results, and eventually issues actuation commands. Also notice that all layers are composed of multiple components, as required to abstract away the details of the WSN, in order to make data management and processing fully generic and modular.

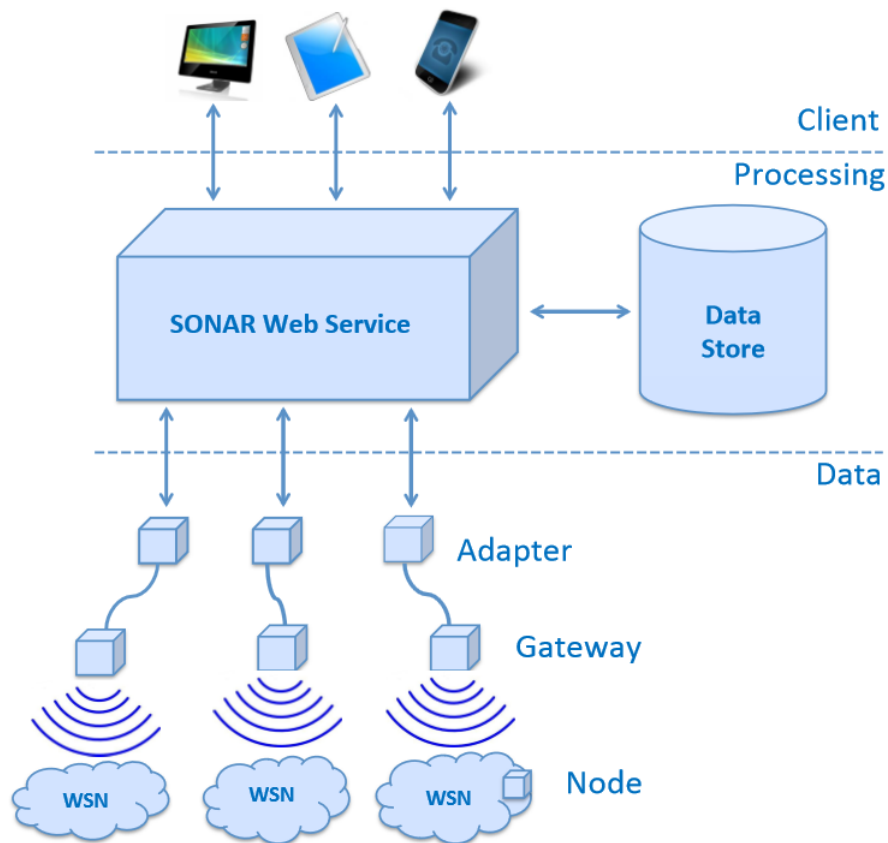


Figure 3.1: The SONAR architecture.

## 3.2 The Data Layer

The data layer abstracts each wireless sensor-actuator platform as three components:

- adapter;
- gateway;
- nodes.

The adapter is a dedicated component, and its only function is to act as a forwarder, since it behaves as an intermediate component between the processing layer and a gateway of a WSN.

The gateway may receive data messages from nodes, or control messages from the adapter. In the case of receiving a control message, it radios it to the appropriate set of nodes in the deployment. When receiving a data message from a node it forwards it to the adapter.

The nodes are composed of sensors and actuators. They send data produced by the sensors to the gateway, or receive and execute actuation commands from the gateway. The control flow and the data flow are both depicted in the Figure 3.2.

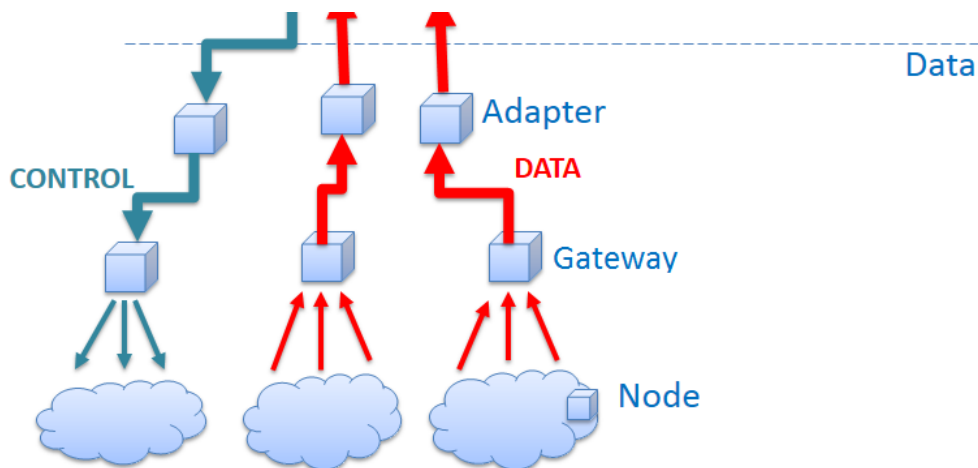


Figure 3.2: Data layer control and data flow

### 3.3 The Processing Layer

The processing layer is the intermediate layer between the client layer and the data layer. It manages all the control and data flow, and is made up of three components:

- SONAR service;
- Task Pool;
- Data Store.

The data store, must be able to provide a generic abstraction for a deployment, storing all the information produced by the deployment and handling all the incoming data requests from the SONAR service.

The task pool is responsible for scheduling and executing all the incoming tasks from the client layer. When being executed, a task typically uses the SONAR service to query the data store. The query returns a result that is processed by the task, and used to decide whether a control message is issued or not, to a region in the deployment.

The SONAR service, manages all the data flow by receiving data messages from the data layer and control messages from the client layer. In case the message comes from the client layer, the SONAR service processes it, in order to decide the type of control message received. This message can be of two types: data-request or task-submission. In a data-request, the SONAR service redirects the query to the data-store and forwards the results to the client. In a task-submission, it adds the task to the pool and registers the task in the data-store. In case of a message from the data layer, the SONAR service must process it, in order to decide whether it is a registration message or a data message. The registration message, contain all the information about a deployment (gateway address and available actuators and sensors), and indicates to the web service, that this is its first connection. This also means that it must be registered, in the data store. Afterwards, the SONAR service, can start to receive the deployment's data stream messages and store their values, in the data store.

The control flow and the data flow are both depicted in the Figure 3.3.

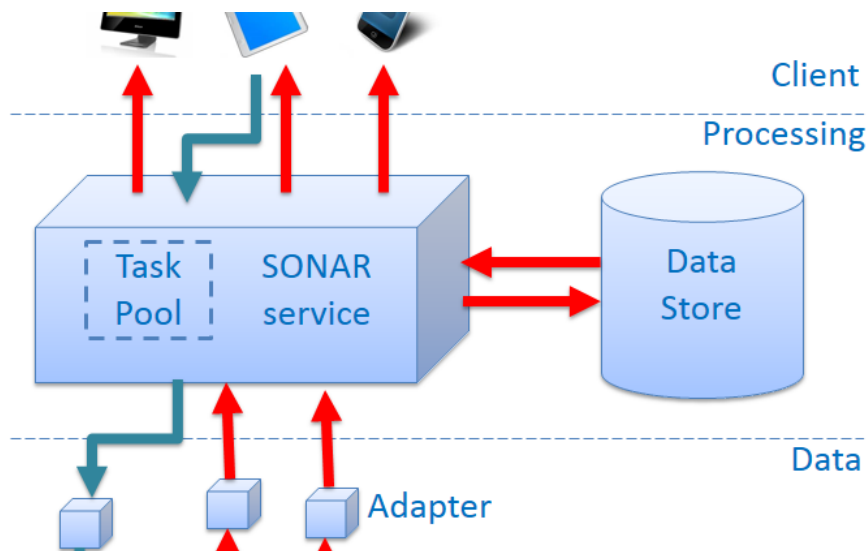


Figure 3.3: Processing layer control and data flow

### 3.4 The Client Layer

Finally the client layer provides web-based applications, for the end-user, which are focused on the SONAR-service's implementation. It allows for the user to send their tasks to the task pool and/or issues data requests to the web service, concerning a specific deployment. It returns its results to the client, which will process the received data, and display it to the user.

The clients need not continuous access to the Internet. On the contrary, the architecture allows for a disconnected management of the deployment, e.g., from a client installed in a smartphone, tablet or notebook with only occasional network connectivity.

The control flow and the data flow are both depicted in the Figure 3.4.

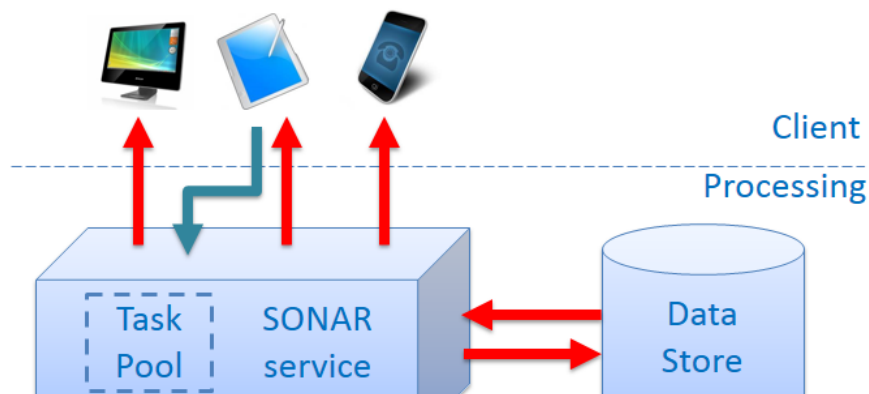


Figure 3.4: Client layer control and data flow

A full view of the control and data flow detailed along this chapter is depicted in the Figure 3.5.

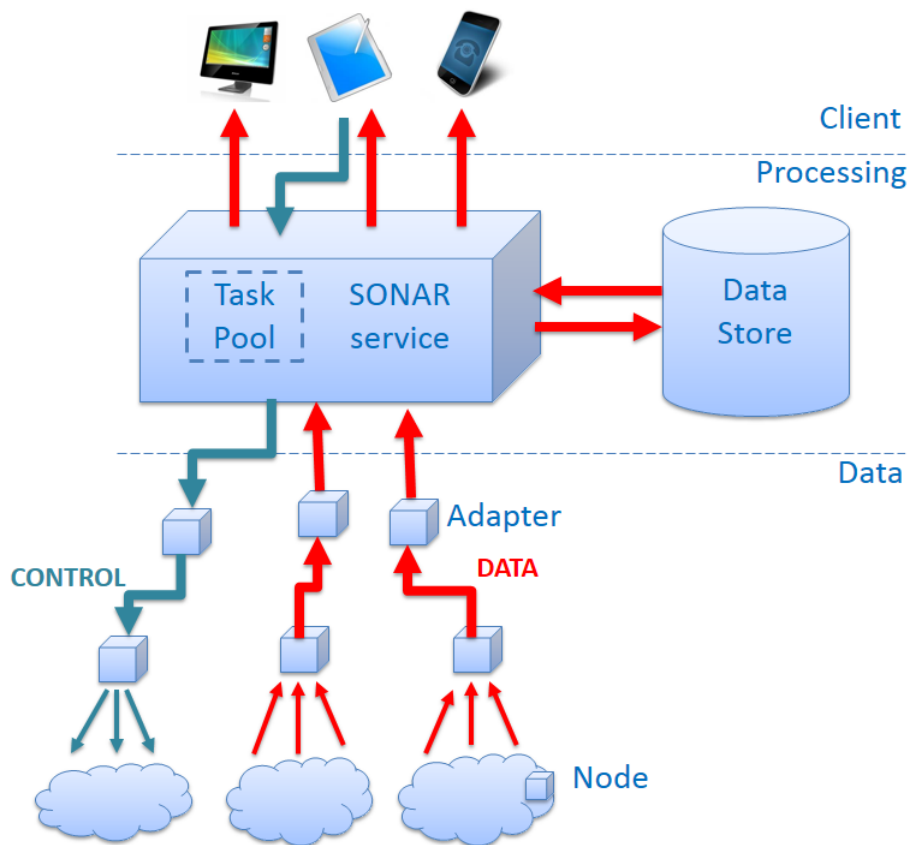


Figure 3.5: The SONAR architecture's data flow.

## 3.5 Summary

This chapter introduced the SONAR's architecture, by detailing its layers and components. In the next chapter, we are going to use the presented architecture, to detail the SONAR's first prototype implementation.

# Chapter 4

## Prototype

In this chapter addresses the implementation of the first prototype, based on the architecture proposed in Chapter 3. We start with an overview of the implementation, and in the next three sections we describe each layer in detail. Finally, we conclude the chapter with a summary.

### 4.1 Overview

SONAR is an out-of-the-box solution for the configuration, deployment and management of WSN, and requires the following software packages for being completely installed:

- Java Standard Edition (SE) 7
- Apache TomCat Server 6
- MySQL Community Edition

The Java SE 7 is used to execute the client and the builder tool applications, the adapter and the SONAR service components; the Apache TomCat Server 6 is used to publish the adapter and the SONAR service components, since both of them are implemented as web servers, and; the MySQL Community Edition is used to provide a concrete implementation of the data store. In order to keep the simplicity of this overview, we assume all the aforementioned components are installed in the same machine. However, they can be installed in separated machines, and have only their



individual software packages requirements installed. For instance, suppose the user has a dedicated machine for the data store. This machine must have installed the MySQL Community Edition but, since the others components are not present, it is not necessary to have installed the Java SE 7 and the Apache TomCat Server 6.

When using SONAR, the deployment and management of an application onto a WSN is performed in three steps. The first step consists on the automatic generation of the data layer, using a builder tool to configure, build and deploy it onto every node of a WSN. Afterwards, once the user has the data layer deployed, the second step consists on starting the SONAR service and the adapter components. The two aforementioned components can be started by just copying their respective files to the folder *webapps*, in the installation directory of the Apache TomCat Server 6. The third and final step consists on the registration of the deployment's gateway, with the processing layer. This process is also done automatically, when the gateway turns on for the first time, by sending a registration message. The adapter receives the registration message and forwards it to the processing layer. In the processing layer, the SONAR service component, receives the registration message and populates the data store with its content. Once the registration process is done, the user can start using the client application to manage and to interact with WSN. This interaction with WSN is done through periodic tasks, which are also managed by the client application. In addition, the user can also select a specific set of nodes of the deployment, and send the task to them. This set of nodes forms a region which, from an implementation point of view, is just a set of MAC addresses.

## 4.2 The Data Layer

### 4.2.1 Building the Data layer

An essential characteristic of SONAR is that the data layer for a deployment, is automatically generated for the user by a Java-based builder application, as depicted in the Figure 4.1. However, this generated data layer is platform dependent, which means that the user must provide some initial information such as the hardware platform, and the available sensors and actuators. This information is stored in an XML file that is then parsed with the help of the JAXB tool, and used to adjust the builder's GUI to reflect the available platform functionality, as it is depicted in the Figure 4.2. This XML file, also includes the locations of the pre-compiled modules and the scripts

required for the build. In particular, for this prototype, the available platforms are: SunSPOT and Arduino. However, all the implementation addressed in this chapter, is relative to the SunSPOT platform.

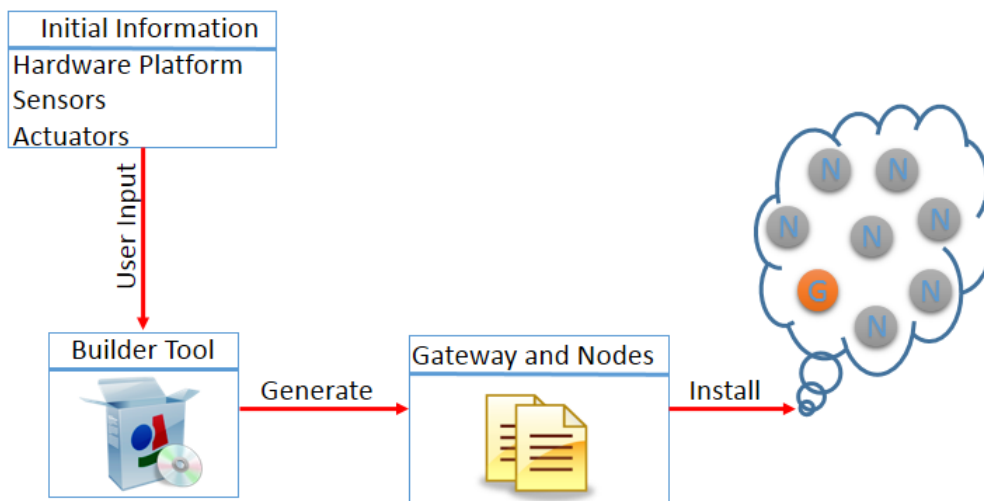


Figure 4.1: Builder tool application process

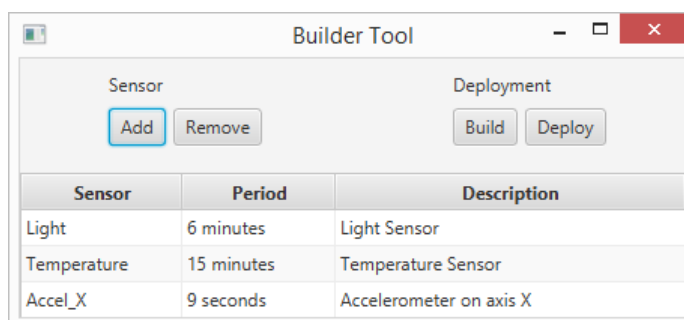


Figure 4.2: Builder Tool Interface for the SunSPOT.

Once the builder's interface is properly adjusted, the user can press the Add button, to choose which sensors are going to be used in the deployment and their sampling frequencies. Thereafter, the Build button is used to generate the code that will be deployed to the gateway and nodes.

The code generated for the SunSPOT nodes, uses Java Micro Edition (Java ME) for the Squawk virtual machine [19]. The node component is made up of multiple readers, one per selected sensor, as indicated by the user to the builder, and a command interpreter. The readers and the interpreter are generic pre-compiled classes. The builder simply

generates a class `Node` that creates an instance of each object attached to a thread (Figure 4.3).

```

public class Node extends MIDLet {
    . . .
    protected void startApp() throws . . . {
        new Reader(LIGHT,5000).start();
        new Reader(TEMP,3000).start();
        new Interpreter().start();
    }
}

```

Figure 4.3: The `startApp` method for the MIDLet running on the nodes.

The implementation of the readers (Figure 4.4) is rather simple: a thread periodically wakes up, reads the value of the appropriate sensor, and sends it to the gateway. The variable `sensorID` is set in the constructor (c.f. Figure 4.3) whereas `connection` and `board` denote, respectively, the radiogram connection used to radio the data, and an object providing access to the hardware sensors. More energy efficient data layers can be implemented using techniques such as data aggregation and/or sensor fusion, but we shall not address this point here.

The code for the interpreter (Figure 4.5) is also quite simple: a thread waits for a message from the gateway. When the message arrives it is unpacked and parsed in order to determine which actuator is to be activated and the corresponding parameters (e.g., which LED and which RGB combination, or which PIN and turn ON or OFF).

Once the code of the gateway and nodes are built, the Deploy button is used to deploy the generated code to the gateway and to every node onto the WSN. This deployment process is done using whatever scripts are required by the platform, indicated in the aforementioned XML file, preferably using Over-The-Air (OTA) programming, a feature that, although not essential, greatly facilitates the deployment of applications. Finally, the gateway application is installed and executed. Algorithm 1 summarizes this build procedure.

```
class Reader extends Thread {  
    ...  
    public void run() {  
        double value;  
        while (true) {  
            switch (sensorID) {  
                case TEMP:  
                    value = board.getTemp().getVal();  
                    break;  
                case LIGHT:  
                    value = board.getLight().getVal();  
                    break;  
                case ACX:  
                    value = board.getAccelX().getVal();  
                    break;  
                case ACY:  
                    value = board.getAccelY().getVal();  
                    break;  
                case ACZ:  
                    value = board.getAccelZ().getVal();  
                    break;  
            }  
            conn.writeData(sensorID);  
            conn.writeData(value);  
            conn.sendPackage();  
            Utils.sleep(period);  
        }  
    }  
}
```

Figure 4.4: The code for a generic reader.

## 4.2.2 Data Layer Components

Once the gateway application starts to run it registers itself with the SONAR web server and forwards information describing the deployment, used to create an appro-

```
class Interpreter extends Thread {  
    ...  
    public void run() {  
        while (true) {  
            conn.receivePackage();  
            int commandCode = conn.readInt();  
            switch (commandCode) {  
            case LED:  
                board.setLED(conn.readInt(), ...);  
                break;  
            case PIN:  
                board.setPIN(conn.readInt(), ...);  
                break;  
            }  
        }  
    }  
}
```

Figure 4.5: The code for a generic interpreter.

appropriate table in the data-store, and a reference for the adapter. From this point on, the gateway can forward data received from the nodes to the processing layer, via the adapter, and the deployment is visible to clients that connect to the SONAR web server. The web server will also be able to send commands to nodes in the WSN, via the adapter and the gateway. When a client connects to the SONAR web server and selects one of the registered deployments it receives the description of it kept in the data-store. The interface of the client is automatically adjusted to reflect the kind of information stored for that deployment, e.g., the available sensors in the node and the actuation commands it can execute, including the range of the input parameters for the commands.

The gateway receives data from the nodes and forwards it to the processing layer. It also receives actuation commands from the processing layer and forwards them to the nodes. This interaction between data and processing layers is mediated by the adapter. In our prototype, the gateway is implemented as two threads. The first thread is an instance of a class, `DataForwarder`, that continuously listens for incoming messages from

---

**Algorithm 1** Building a platform specific data layer and deploying it.

---

**function** BUILDATALAYER(*platform*, *port*, *config*, *macs*)

▷ *platform*: Platform identifier

▷ *port*: Hardware port for gateway

▷ *config*: Set of pairs (sensor,frequency)

▷ *macs*: Set of mac addresses

$\{libs, scripts\} \leftarrow \text{GETCODE}(platform)$

$\{gateway, node\} \leftarrow \text{BUILDATALAYER}(libs, config)$

RUN(*adapter*, *port*)

**for each** *mac* in *macs* **do**

*node\_script* ← GETSCRIPT(*scripts*, *mac*)

    RUN(*node\_script*, *mac*, *node*)

**end for**

*gateway\_script* ← GETSCRIPT(*scripts*, *gateway\_mac*)

RUN(*gateway\_script*, *gateway\_mac*, *gateway*)

**end function**

---

the nodes in the WSN, unpacks the data, and forwards it to the Adapter (Figure 4.6). The other thread is an instance of a class `CommandForwarder`, that receives pairs of the form (command,region) and radios the commands to the nodes whose MAC addresses belong to the region using unicast communication (Figure 4.7).

The adapter is a web service implemented in Java. It is a fully generic module and is activated in the usual way by placing the appropriate class bundle in the Apache TomCat folder installation of the computer. The implementation is quite straightforward as can be inferred from the interface (Figure 4.8). The three methods are used to:

- register new deployments with the middleware - `registerDeployment`;
- forward data to the processing layer - `forwardData`, and;
- receive commands from the processing layer to be forwarded to a region of the deployment - `forwardCommand`.

The communication between the gateway and the adapter is implemented through a Java serial library, `RXTXComm` [37].

```
class DataForwarder extends Thread {  
    ...  
    public void run() {  
        int sensorID;  
        double value;  
        String mac;  
        while (true) {  
            conn.receivePackage();  
            mac      = conn.readMAC();  
            sensorID = conn.readCode();  
            value    = conn.readValue();  
            adapter.forwardData(sensorID , value , mac);  
        }  
    }  
}
```

Figure 4.6: The code for the data forwarder class in the gateway.

```
class CommandForwarder extends Thread {  
    ...  
    public void run() {  
        int [] command = adapter.forwardCommand().getCommand();  
        String [] region = adapter.forwardCommand().getRegion();  
        for (int index = 0; index < region.length(); index++) {  
            conn.send(command, region[i]);  
        }  
    }  
}
```

Figure 4.7: The code for the command forwarder class in the gateway.

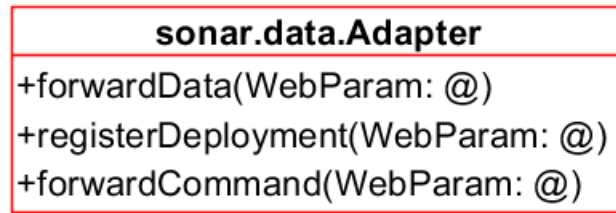


Figure 4.8: The interface of the adapter.

### 4.3 The Processing Layer

As stated above, the processing layer is composed of three components: a data-store, a web service and a task pool.

The data store schema is described in Figure 4.9. The tables provides a generic abstraction for a deployment. A top-level table - **Deployment**, keeps information about the deployment, e.g., its name, an optional description and the underlying hardware platform. It indexes three further tables that contain information about the sensors - **Sensor**, and actuators - **Actuator**, present in the nodes, and the tasks - **Task**, managed by the processing layer and associated with the deployment. Tasks described in the **Task** table may be active (running in the task pool) or inactive (on hold, waiting for a client to activate them). This table also keeps the Java byte-code associated with each task object so that, in the event of a web server crash, the tasks can be restarted on recovery (field **binaryTask**). The actual data produced by the nodes in the deployment is stored in a single table - **Data**, accessed through the sensor table, that keeps the data indexed by the sensor identifier and by the time stamp. Initially the database is empty and new data are going to populate the database as new deployments register themselves with the processing layer. Each gateway module contains the code necessary to register the deployment with the processing layer and to build the tables in the data-base. Each deployment is uniquely identified by a key, the MAC address of the device that runs the gateway module. In the SunSPOT platform this is the MAC address of the basestation.



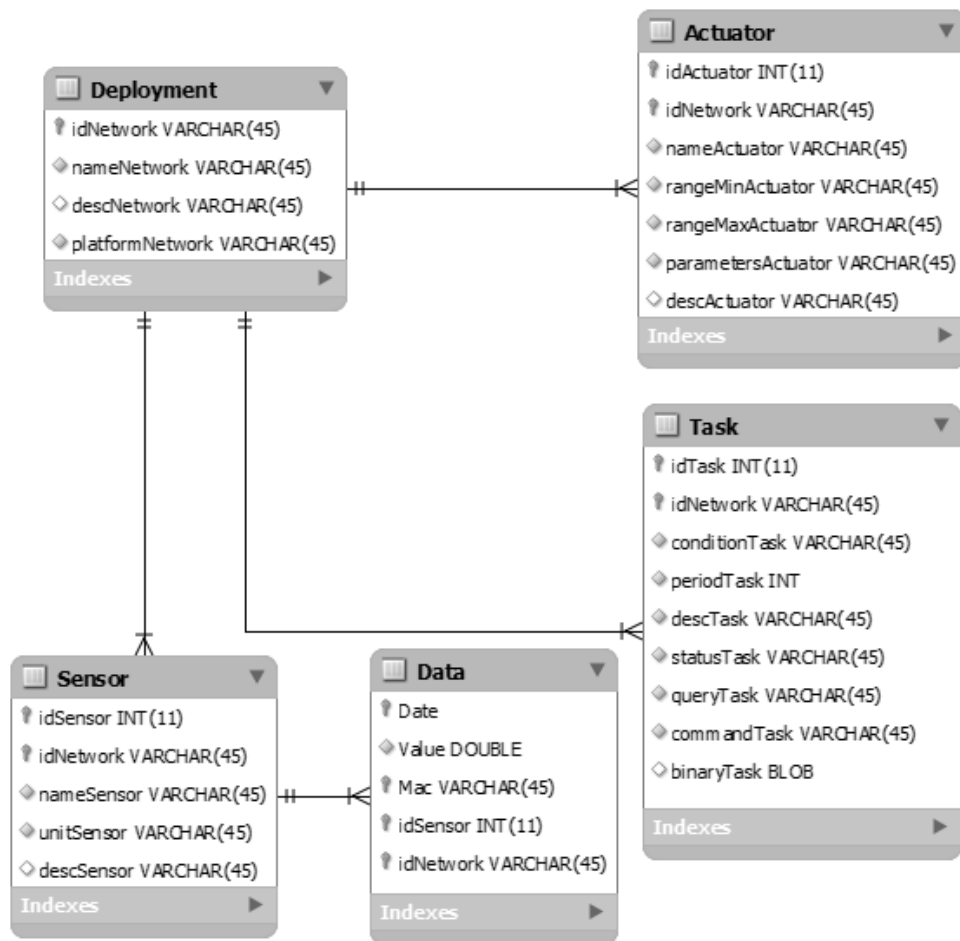


Figure 4.9: EER diagram for the SONAR data-store.

The web service controls the data flow in the architecture. It is a Java web service that implements the interface given in Figure 4.10. The methods in the service allow for:

- `mysqlConn`: connecting with the data-store;
- `registerDeployment`: registering a deployment with the processing layer;
- `getDeploymentList`: consulting the registered deployments;
- `getDeployment`: select one registered deployment;
- `storeData`: storing data in the data-store;

- `createTask`: creates a new task;
- `destroyTask`: destroy a specific task;
- `runTask`: executes a task;
- `pauseTask`: pauses a task;
- `refreshTask`: refreshes the status of a task;
- `getRegion`: finds the nodes that satisfy a given set of boolean conditions;
- `executeCommand`: sends an actuation command to the data layer.

Upon initialization the web service checks for tasks marked as active in the data-base tables. For each record found, it gets the corresponding serialized object, deserializes it and adds it to the pool of tasks. After this step it continues with the server loop (Algorithm 2). This initialization step is important as, in case the server crashes, all the active tasks prior to the crash event, can be restarted automatically. This is of course possible because a serialized copy of each task is maintained in the data-base.

<b>sonar.processing.WebService</b>
+ <code>mysqlConn(WebParam : @)</code> + <code>registerDeployment(WebParam : @)</code> + <code>getDeploymentList() : String[][]</code> + <code>getDeployment(WebParam : @) : String[][]</code> + <code>storeData(WebParam : @)</code> + <code>createTask(WebParam : @) : int</code> + <code>destroyTask(WebParam : @) : boolean</code> + <code>runTask(WebParam : @) : boolean</code> + <code>pauseTask(WebParam : @) : boolean</code> + <code>refreshTasks(WebParam : @) : String[][]</code> + <code>getRegion(WebParam : @) : LinkedList&lt;String&gt;</code> + <code>executeCommand(WebParam : @)</code>

Figure 4.10: The interface of the SONAR web-service.

---

**Algorithm 2** Initializing SONAR.

---

```

function MAIN( )
  INITPOOL( )
  records ← DBGETACTIVETASKS()

  for each record in records do
    bytes ← GETBYTES(record)
    task ← DESERIALIZE(bytes)
    POOLADD(task, taskId)
  end for
  SERVERLOOP( )
end function

```

---

The task pool is managed through the web service interface. The tasks running on the pool can be periodic or one-shot, as specified by clients. Tasks are created by clients using the method `createTask` in the web service that registers the task in the appropriate table for the deployment in the data-store (Algorithm 3).

---

**Algorithm 3** Registering a task with SONAR.

---

```

function CREATETASK(task, taskId)
  ▷ task - the task to be created
  ▷ taskId - the identification of the task to be created
  bytes ← SERIALIZE(task)
  DBSTORE(taskId, bytes)
end function

```

---

When tasks are first created they are always inactive. To activate a task the method `runTask` is used that attaches the task to one of the threads in the pool (Algorithm 4).

---

**Algorithm 4** Running a task in SONAR.

---

```

function RUNTASK(taskId)
  ▷ taskId - identification of the task to be run
  record ← DBMAKEACTIVE(taskId)
  bytes ← GETBYTES(record)
  task ← DESERIALIZE(bytes)
  POOLADD(task, taskId)
end function

```

---

Typically, tasks periodically query the data store for data returned from the deployment and process it. As a result of that processing, actuation commands for a region of the deployment may be issued. These are forwarded to the adapter using the `forwardCommand` method in the web service interface. Clients may permanently remove tasks by invoking the method `removeTask` with the task identifier (Algorithm 5).

---

**Algorithm 5** Removing a task from SONAR.

---

```

function DESTROYTASK(taskId)
  ▷ taskId - identification of the task to be killed
    POOLREMOVE(taskId)
    DBREMOVE(taskId)
end function

```

---

Besides the task being removed from the thread pool (whether it is active), the record of the task in the data-store will also be removed. Finally, a task may also be temporarily paused by a client - `pauseTask`, and executed again - `runTask`.

## 4.4 The Client Layer

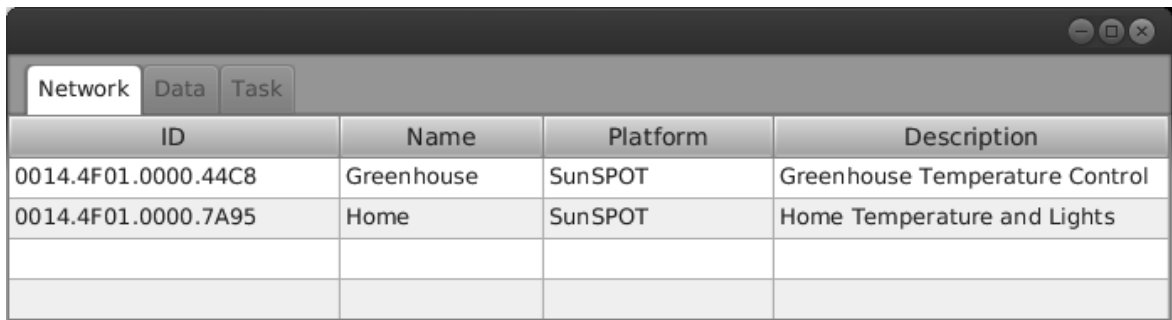
The client layer is implemented as a Java-based GUI. When it is started it connects to the SONAR web service and requests information about all registered networks. This information is retrieved from the data-store via the web service.

### 4.4.1 The Network Tab

The list of registered networks is provided in the “network tab” of the GUI (Figure 4.11). When a user selects one network from this list, the GUI is adapted to account for the different sensors and actuators supported by the network and for the current tasks associated with it. The user may then choose to visualize data, in the data tab, or manage the tasks, in the task tab.

### 4.4.2 The Data Tab

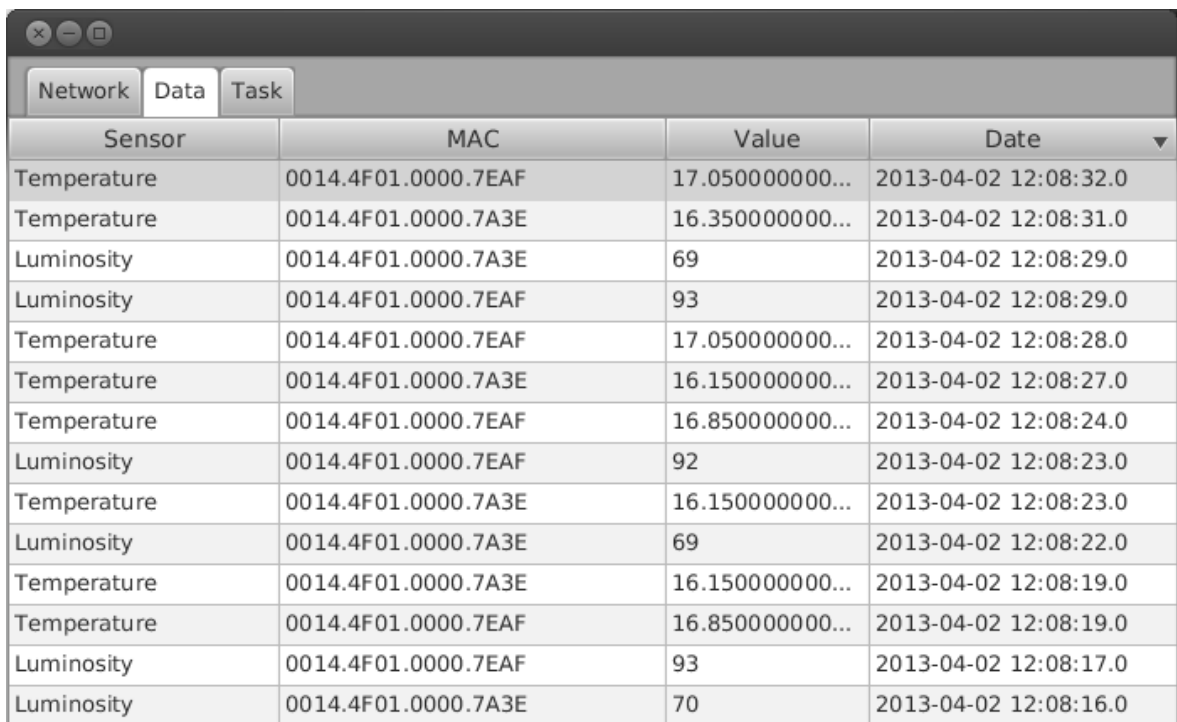
In this prototype, the “data tab” is quite simple, listing all readings that have been sent by nodes in the network. Each entry describes the sensor that reported the



ID	Name	Platform	Description
0014.4F01.0000.44C8	Greenhouse	SunSPOT	Greenhouse Temperature Control
0014.4F01.0000.7A95	Home	SunSPOT	Home Temperature and Lights

Figure 4.11: The network tab.

reading, the MAC address of the corresponding node, the reading, and the time-stamp (Figure 4.12). The readings are time-stamped in the gateway, when the data is received from the nodes and before it is forwarded to the processing layer.



Sensor	MAC	Value	Date
Temperature	0014.4F01.0000.7EAF	17.050000000...	2013-04-02 12:08:32.0
Temperature	0014.4F01.0000.7A3E	16.350000000...	2013-04-02 12:08:31.0
Luminosity	0014.4F01.0000.7A3E	69	2013-04-02 12:08:29.0
Luminosity	0014.4F01.0000.7EAF	93	2013-04-02 12:08:29.0
Temperature	0014.4F01.0000.7EAF	17.050000000...	2013-04-02 12:08:28.0
Temperature	0014.4F01.0000.7A3E	16.150000000...	2013-04-02 12:08:27.0
Temperature	0014.4F01.0000.7EAF	16.850000000...	2013-04-02 12:08:24.0
Luminosity	0014.4F01.0000.7EAF	92	2013-04-02 12:08:23.0
Temperature	0014.4F01.0000.7A3E	16.150000000...	2013-04-02 12:08:23.0
Luminosity	0014.4F01.0000.7A3E	69	2013-04-02 12:08:22.0
Temperature	0014.4F01.0000.7A3E	16.150000000...	2013-04-02 12:08:19.0
Temperature	0014.4F01.0000.7EAF	16.850000000...	2013-04-02 12:08:19.0
Luminosity	0014.4F01.0000.7EAF	93	2013-04-02 12:08:17.0
Luminosity	0014.4F01.0000.7A3E	70	2013-04-02 12:08:16.0

Figure 4.12: The data visualization tab.

### 4.4.3 The Task Tab

All the management tasks associated with a network are listed in the “task tab” (Figure 4.13). SONAR tasks are very simple. To specify one, the user must select a frequency associated with the task, what sensors are interesting, their sampling frequencies and what method is used to evaluate them (e.g., time window, average), a boolean expression on those readings, and a set of actuation commands that must be sent to a specific region, in which the boolean expression evaluates to true. Tasks are implemented internally using a small domain specific programming language. The example in Figure 4.13 shows one such task, for a WSN that manages a greenhouse. Every 5 minutes, it reads the last 20 minutes of temperature data, takes the average and checks whether the value is above 30 Celsius. If so, it then sends actuation commands to all nodes with temperatures above 30 Celsius to activate the pins that switch on the sprinkler system and open the ventilation windows. Users may also define one-shot tasks using the keyword **once** rather than **repeat**. Also, besides **average**, several data models may be used to determine the value of a physical variable for each sensor, e.g., **last** - the last value stored, and **median**.

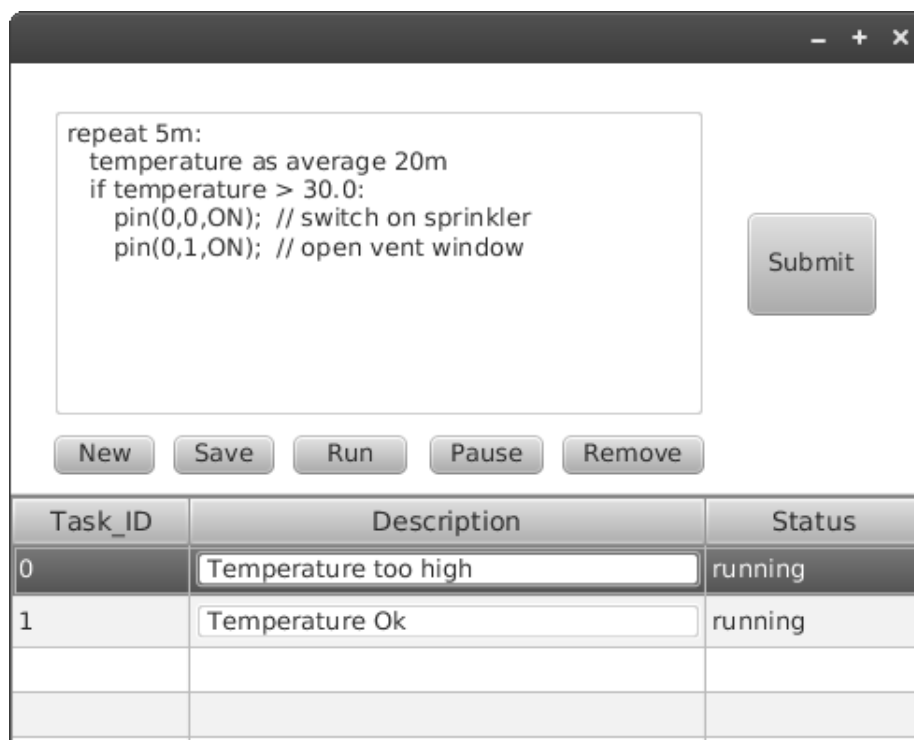


Figure 4.13: The task management tab.

Users may not edit tasks explicitly. Rather, a wizard is provided to guide users in the specification of tasks, without the need to write code. It is the wizard that then automatically generates the code for the tasks.

Submitted tasks are compiled by the client and an internal representation, similar to an annotated AST, is produced. Finally, a request is sent to the SONAR web server to register and start running the task, as described in Section 4.3. When the client receives an acknowledgment from this request, it adds the task to the list in the task tab.

## 4.5 Summary

In this chapter we detailed our first prototype of SONAR. This first prototype, achieved its primary goal, being an out-of-the-box solution for building, setup and deploying WSN, since with just three steps, the end-user would be able to both manage and interact with its WSN, through an easy-to-use java client GUI. In addition, this version of the prototype was also published in proceedings of the conference SensorNets 2014 [38].

In the next chapter we discuss the several weaknesses of this prototype, and present the solutions that we have implemented, in order to overcome them.

# Chapter 5

## Discussion

In this chapter we detail the weaknesses in the approach that we took, in order to implement the prototype, described in the Chapter 4. We propose solutions for these problems whose implementation will be described in forthcoming chapters.

### 5.1 Weaknesses

A subsequent analysis of the prototype presented in the Chapter 4, showed several weaknesses in the approach that we took in its implementation. These weaknesses are related to the following components:

- task pool;
- data store;
- interpreter and reader.

The following points detail such weaknesses:

1. The interpreter and readers are both provided by SONAR, being presented as pre-compiled modules or scripts. Managing such modules/scripts is no easy task since a user could build different types and combinations of deployments. It implies that for each sensor available and possible combination of them we should have had a set of pre-compiled modules/scripts, proving it to be non-scalable in a long term;



2. The task pool, presented in the Section 4.3, has been found with the very same problem, since managing such high number of different tasks, all of them querying the data-store periodically, would also be non-scalable;
3. In this prototype, the implementation of the communication with the data-store, was very restricted, since it was prepared to only connect to a MySQL database, and only one repository at a time. This situation restricts SONAR, since a typical case scenario a user with different types of database for distinct deployments, was not being addressed;
4. Finally, we did not address access control problems, such as the relationship between users and deployments. This relationship is important, since we have to define an access policy for users to deployments, to preserve the privacy of data and/or the integrity of the deployments.

## 5.2 Solutions

As mentioned in the above section, our first prototype's implementation was rather inflexible, in terms of data store; did not address access control problems regarding the access to deployments and; it is non-scalable, when considering the task pool and both the interpreter and readers components. Such issues are solved with a new implementation of the prototype, as follows:

1. Development of a generic interface for the SONAR service, to communicate with the different types of data store. This interface, provides an abstraction for the data store, in such a way that the user can refer to different data-stores products, in contrast of being restricted to the usage of the MySQL. This means that the user can access different data stores with different deployments.
2. The task pool is removed from the architecture. Thus, instead of using the task pool to schedule and execute all the tasks, we decided to implement a tiny operating system that runs on every node of the deployment. This operating system manages the hardware resources, and handles the scheduling of the tasks, in the nodes. Also, through transferring tasks to nodes, the amount of communication between the SONAR service and the data store would diminish significantly. This means that the tasks wouldn't have to query the data store every single time they would need information, since a small time window of the produced values, will be locally available in the nodes.

3. The interpreter and readers are also removed from the architecture, since we decided to design and implement a virtual machine. The virtual machine, allows us to abstract the hardware of the platform being used and, the effort required to port the virtual machine to other platforms, is lesser than having to manage different types and combinations of pre-compile modules/scripts. In addition, a domain specific programming language was developed to program tasks. Its compiler produces a byte-code, which is an input for the virtual machine.
4. The usage of the builder application is not necessary anymore. This means that the user does not have to generate the data layer of the deployment, since the operating system and the virtual machine components, would come factory installed.
5. Regarding the access control concern, we decided to implement an authentication and administration system functionality. In order to properly implement it, we decided that it would be for the best to redesign the interface of the user application. This way the new functionalities would be integrated from scratch. Also, with the redesigned client's interface, the user would be able to manage and interact with multiple deployments at the same time.

### 5.3 Summary

Based on facts presented in the Section 5.1, it was decided that instead of having different types of pre-compiled modules/scripts, we should design and implement new components. These components would provide us, a better abstraction of the hardware platform, and higher scalability. For that we designed an operating system, a virtual machine, and a domain specific programming language. In the following chapters we present and detail the new components: SONAR Task Language (STL); SONAR Operating System (SOS) and; SONAR Virtual Machine (SVM).

# Chapter 6

## The SONAR Task Language

In this chapter we describe the syntax and semantics of the domain-specific programming language used to implement periodic tasks in SONAR. We call it the Sonar Task Language (STL).

### 6.1 Syntax

The syntax for tasks is described in Figure 6.1. A task  $T$  uses two sets of identifiers,  $\tilde{s}$  and  $\tilde{a}$ , to specify the available sensors and actuators in a given platform. Each of these identifiers maps to a unique sensor or actuator in the hardware as implemented by the underlying virtual machine. This declaration is thus similar for all tasks running on the same hardware configuration and in a more concrete syntax would simply be included by the programmer using a compiler directive.

The code that is actually specific for the task starts by specifying the type of the messages sent back by the task to the gateway using the construct **radiates**  $[\tilde{\tau}]$ . The task only sends messages of this type to the gateway and the type is checked against all **radio** statements in the task.

The **data** block is used to initialize task variables. This code is not executed, rather the compiler will copy the initial values directly to the data segment of the bytecode generated for the program. The **text** block, on the other hand, is the code executed for every (periodic) activation of the task. The instructions available to the programmer include: assignment, actuation -  $a(\tilde{e})$ , sending a set of evaluated expressions to the gateway - **radio**  $[\tilde{e}]$ , and a standard conditional execution construct - **if**  $e$   $\{\tilde{r}\}$  **else**  $\{\tilde{r}\}$ .

$T ::= \mathbf{sensors} \{s_1 : \tau_1 \dots s_n : \tau_n\}$ $\quad \mathbf{actuators} \{a_1 : \tau_1 \dots a_m : \tau_m\}$ $\quad \mathbf{radiates} [\tilde{\tau}] \mathbf{data} \{\tilde{q}\} \mathbf{text} \{\tilde{r}\}$	<i>Tasks</i>
$\tau ::= \mathbf{bool} \mid \mathbf{int} \mid \mathbf{float} \mid \mathbf{void}$ $\quad \mid \tilde{\tau} \mapsto \tau$	<i>Types</i>
$q ::= \tau \ x = v$	<i>Initializations</i>
$r ::= x = e$ $\quad \mid a(\tilde{e})$ $\quad \mid \mathbf{radio} [\tilde{e}]$ $\quad \mid \mathbf{if} \ e \ \{\tilde{r}\} \ \mathbf{else} \ \{\tilde{r}\}$	<i>Instructions</i>
$e ::= s(\tilde{e}) \mid e \ op \ e \mid op \ e \mid v$	<i>Expressions</i>
$v ::= x \mid u$	<i>Values</i>
$u ::= \mathit{bools} \mid \mathit{ints} \mid \mathit{floats}$	

Figure 6.1: The syntax of STL.

The expressions are standard except for  $s(\tilde{e})$  that is used to read a value from a given sensor.

The example in Figure 6.2 shows a STL program that turns on and off a sprinkler according to ambient temperature and humidity, and that also radios temperature and humidity. The example uses two sensors, designated as `temperature` and `humidity`, and an actuator `sprinkler`, whose types are declared in the hardware description in the first two constructs. The task always radios two floats when sending data to the gateway. After the initialization of `sprinklerOff`, each activation of the task reads the temperature and the humidity, checks a simple condition and decides whether or not the sprinkler should be kept on or off. Before stopping until the next activation it radios the sampled temperature and humidity to the gateway.

## 6.2 Operational Semantics

The operational semantics is defined through a reduction relation  $\rightarrow$  on the program state. The latter is defined as either the halted task,  $\perp$ , or, if the task is active, as a

```
sensors {
  temperature: void -> float ,
  humidity    : void -> float
}

actuators {
  sprinkler   : bool -> void
}

radiates [ float , float ]

data {
  bool  sprinklerOff = true;
  float temp = 0.0;
  float hum  = 0.0;
}

text {
  temp = temperature();
  hum  = humidity();
  if temp > 30 and hum < 50 {
    if ( sprinklerOff ) {
      sprinkler( true );
      sprinklerOff = false;
    }
  } else {
    if ( not sprinklerOff ) {
      sprinkler ( false );
      sprinklerOff = true;
    }
  }
  radio [temp,hum];
}
```

Figure 6.2: STL program that turns on and off a sprinkler according to ambient temperature and humidity.

$$\frac{v = \text{eval}(V, e)}{(V, \underline{x = e} \tilde{r}) \rightarrow (V + \{x : v\}, \tilde{r})} \quad (1)$$

$$\frac{\tilde{v} = \text{eval}(V, \tilde{e}) \quad a \in A \quad \text{write}(a, \tilde{v})}{(V, \underline{a(\tilde{e})} \tilde{r}) \rightarrow (V, \tilde{r})} \quad (2)$$

$$\frac{\tilde{v} = \text{eval}(V, \tilde{e}) \quad \text{send}(\tilde{v})}{(V, \underline{\text{radio} [\tilde{e}]} \tilde{r}) \rightarrow (V, \tilde{r})} \quad (3)$$

$$\frac{\text{eval}(V, e) = \mathbf{true}}{(V, \underline{\text{if } e \{ \tilde{r}_1 \} \text{ else } \{ \tilde{r}_2 \} \tilde{r}_3}) \rightarrow (V, \tilde{r}_1 \tilde{r}_3)} \quad (4)$$

$$\frac{\text{eval}(V, e) = \mathbf{false}}{(V, \underline{\text{if } e \{ \tilde{r}_1 \} \text{ else } \{ \tilde{r}_2 \} \tilde{r}_3}) \rightarrow (V, \tilde{r}_2 \tilde{r}_3)} \quad (5)$$

$$(V, \epsilon) \rightarrow \perp \quad (6)$$

Figure 6.3: Reduction rules for STL instructions.

tuple  $(S, A, V, \tilde{r})$ . In the latter,  $S$  and  $A$  are of type  $\text{Set}(\text{Vars})$  and keep the identifiers for the built-in functions declared at the beginning of an STL program and that provide access to sensors and actuators, respectively.  $V$ , of type  $\text{Map}(\text{Vars}, \text{Values})$  keeps the values of the variables during the execution of the program. Thus, the initial state for the task:

**sensors**  $\{s_1 : \tau_1 \dots s_n : \tau_n\}$   
**actuators**  $\{a_1 : \tau_1 \dots a_m : \tau_m\}$   
**radiates**  $[\tilde{r}]$  **data**  $\{\tilde{q}\}$  **text**  $\{\tilde{r}\}$

is the tuple  $(S_0, A_0, V_0, \tilde{r})$ , where:

$$\begin{aligned} S_0 &= \{s_1, \dots, s_n\} \\ A_0 &= \{a_1, \dots, a_m\} \\ V_0 &= \{(x : v) \mid \tau x = v \in \tilde{q}\} \end{aligned}$$

The reduction rules are presented in Figures 6.3 and 6.4, where the identifiers  $s$  and  $a$  are built-in functions, as well as the function **radio**. We also simplify the notation somewhat by not including  $S$  and  $A$  explicitly in the state, i.e., we represent the tuple  $(S, A, V, \tilde{r})$  tuple as the shorter version  $(V, \tilde{r})$ . The rules have the structure:

$$\frac{c_1 \dots c_n}{(V_1, \tilde{r}_1) \rightarrow (V_2, \tilde{r}_2)}$$

where the  $c_i$  are preconditions or actions that must be fulfilled to make the transition from the current state,  $(V_1, \tilde{r}_1)$ , to a given state,  $(V_2, \tilde{r}_2)$ , possible. For example, rule (2) for instructions executes  $a(\tilde{e})$  statements, underlined and the next in the code sequence. It evaluates the expressions  $\tilde{e}$  into values  $\tilde{v}$  first, checks whether the identifier  $a$  is a valid actuator built-in function,  $a \in A$ , and finally calls  $a(\tilde{v})$ . When  $a$  returns the state of the program will be  $(V, \tilde{r})$ . The reasoning is similar in rule (2) for expressions, where we read data from a sensor. Here, however, the value returned from the sensor,  $v = f(\tilde{v})$ , is the value of the expression. Rule (6) for instructions, another example, is invoked when the code sequence in the text block ends, the next state is  $\perp$ .

## 6.3 Static Semantics

The static semantics of a task is provided in the form of a type system (Figure 6.5). The rules are fairly standard and use a typing environment  $\Gamma$  that keeps track of the types for identifiers. The rules are written as  $\Gamma \vdash r$  for instructions, meaning

$$\frac{\tilde{e} = e_1 \dots e_n \quad v_i = \mathbf{eval}(V, e_i), 1 \leq i \leq n}{\mathbf{eval}(V, \tilde{e}) = \tilde{v}} \quad (1)$$

$$\frac{\tilde{v} = \mathbf{eval}(V, \tilde{e}) \quad s \in S \quad v = \mathbf{read}(s, \tilde{v})}{s(\tilde{e}) = v} \quad (2)$$

$$\frac{v_1 = \mathbf{eval}(V, e_1) \quad v_2 = \mathbf{eval}(V, e_2)}{\mathbf{eval}(V, e_1 \text{ op } e_2) = v_1 \text{ op } v_2} \quad (3)$$

$$\frac{v = \mathbf{eval}(V, e)}{\mathbf{eval}(V, \text{op } e) = \text{op } v} \quad (4)$$

$$\mathbf{eval}(V, x) = V(x) \quad (5)$$

$$\mathbf{eval}(V, v) = v \quad (6)$$

Figure 6.4: Reduction rules for STL expressions.

that the instruction is well-formed, and  $\Gamma \vdash e : \tau$  for expressions, meaning that expression  $e$  has type  $\tau$ . Some rules have side effects, in which the environment  $\Gamma_1$  is enriched with new entries and becomes  $\Gamma_2$ , as in  $\Gamma_1 \vdash \dots \dashv \Gamma_2$ . An example is rule  $\Gamma \vdash \mathbf{radiates} [\tilde{\tau}] \dashv \Gamma, \mathbf{radiates} : \tilde{\tau}$  ( $\Gamma$  collects the type declared in the **radiates** construct). Besides this rule, three others are worthy of note. Rule (12) checks that messages sent by the task have types that match the one declared in the **radiates** construct. Rule (14) checks that the sensor,  $s$ , is of type  $\tilde{\tau} \mapsto \tau'$ , that the arguments  $\tilde{e}$  match the type  $\tilde{\tau}$  to infer that the value returned by  $s(\tilde{e})$  is of type  $\tau'$ . The logic is similar for rule (11) except that the type system just checks that the instruction  $a(\tilde{e})$  is well formed (instructions do not evaluate to values). The rules (16), (17) and (18) are axioms and allow values booleans, integers and floating point values to be typed.



$$\begin{array}{l}
\emptyset \vdash \mathbf{sensors} \{s_1 : \tau_1 \dots s_n : \tau_n\} \dashv \Gamma_1 \\
\emptyset \vdash \mathbf{actuators} \{a_1 : \tau_1 \dots a_m : \tau_m\} \dashv \Gamma_2 \\
\emptyset \vdash \mathbf{radiates} [\tau_1 \dots \tau_k] \dashv \Gamma_3 \\
\emptyset \vdash \mathbf{data} \{q_1 \dots q_l\} \dashv \Gamma_4 \\
\Gamma_1, \Gamma_2, \Gamma_3, \Gamma_4 \vdash \mathbf{text} \{\tilde{r}\}
\end{array}
\tag{1}$$

$$\begin{array}{l}
\mathbf{sensors} \{s_1 : \tau_1 \dots s_n : \tau_n\} \\
\mathbf{actuators} \{a_1 : \tau_1 \dots a_m : \tau_m\} \\
\vdash \mathbf{radiates} [\tau_1 \dots \tau_k] \\
\mathbf{data} \{q_1 \dots q_l\} \\
\mathbf{text} \{\tilde{r}\}
\end{array}$$

$$\frac{\Gamma \vdash s_1 : \tau_1 \dashv \Gamma_1 \quad \Gamma \vdash s_n : \tau_n \dashv \Gamma_n}{\Gamma \vdash \mathbf{sensors} \{s_1 : \tau_1 \dots s_n : \tau_n\} \dashv \Gamma_1, \dots, \Gamma_n}
\tag{2}$$

$$\frac{\Gamma \vdash a_1 : \tau_1 \dashv \Gamma_1 \quad \Gamma \vdash a_m : \tau_m \dashv \Gamma_m}{\Gamma \vdash \mathbf{actuators} \{a_1 : \tau_1 \dots a_m : \tau_m\} \dashv \Gamma_1, \dots, \Gamma_m}
\tag{3}$$

$$\Gamma \vdash \mathbf{radiates} [\tau_1 \dots \tau_k] \dashv \Gamma, \mathbf{radiates} : \tau_1 \dots \tau_k
\tag{4}$$

$$\frac{\Gamma \vdash q_1 \dashv \Gamma_1 \quad \Gamma \vdash q_l \dashv \Gamma_l}{\Gamma \vdash \mathbf{data} \{q_1 \dots q_l\} \dashv \Gamma_1, \dots, \Gamma_l}
\tag{5}$$

$$\frac{\emptyset \vdash v : \tau}{\Gamma \vdash \tau x = v \dashv \Gamma, x : \tau}
\tag{6}$$

$$\frac{\Gamma \vdash \tilde{r}}{\Gamma \vdash \mathbf{text} \{\tilde{r}\}} \quad \frac{\Gamma \vdash r_1 \dots \Gamma \vdash r_n}{\Gamma \vdash \tilde{r}}
\tag{7,8}$$

Figure 6.5: Type system for STL (part I).

$$\frac{\Gamma \vdash x : \tau \quad \Gamma \vdash e : \tau}{\Gamma \vdash x = e} \quad (9)$$

$$\frac{\Gamma \vdash a : \tilde{\tau} \mapsto \mathbf{void} \quad \Gamma \vdash \tilde{e} : \tilde{\tau}}{\Gamma \vdash a(\tilde{e})} \quad (10)$$

$$\frac{\Gamma \vdash \tilde{e} : \tilde{\tau} \quad \Gamma(\mathbf{radiates}) = \tilde{\tau}}{\Gamma \vdash \mathbf{radio} [\tilde{e}]} \quad (11)$$

$$\frac{\Gamma \vdash e : \mathbf{bool} \quad \Gamma \vdash \tilde{r}_1 \quad \Gamma \vdash \tilde{r}_2}{\Gamma \vdash \mathbf{if} e \{ \tilde{r}_1 \} \mathbf{else} \{ \tilde{r}_2 \}} \quad (12)$$

$$\frac{\Gamma \vdash s : \tilde{\tau} \mapsto \tau' \quad \Gamma \vdash \tilde{e} : \tilde{\tau}}{\Gamma \vdash s(\tilde{e}) : \tau'} \quad (13)$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \quad (14)$$

$$\emptyset \vdash s : \tau \dashv s : \tau \quad \emptyset \vdash a : \tau \dashv a : \tau \quad (15,16,17)$$

$$\emptyset \vdash v : \mathbf{bool} \quad \emptyset \vdash v : \mathbf{int} \quad \emptyset \vdash v : \mathbf{float} \quad (18,19,20)$$

Figure 6.6: Type system for STL (part II).

# Chapter 7

## The SVM and SOS

In this chapter we detail the SONAR Virtual Machine (SVM) and the SONAR Operating System (SOS), the modules pre-installed in the nodes.

### 7.1 The SONAR Virtual Machine

The SVM executes STL tasks, translated into byte-code by a compiler. Thus, we begin by defining the byte-code format, then give the translation function for the source code, and finally the state specification of the SVM.

#### 7.1.1 Byte-code

The byte-code is composed of 3 segments: header, data, and text (Figure 7.1). The header contains the sizes of the data and text segments as well as the maximum run-time stack size. The stack segment exists only at run-time and is allocated between the data and text segment, growing towards the lower addresses. The data segment contains program constants and slots for task variables. It is in fact the only activation record required for the virtual machine, since there are no calls to user functions or user functions in tasks. All constants or variables use 4 bytes in the data segment in this version, but this can and should be optimized to minimize the size of the byte-code. The text segment is composed of instructions that, in general, map almost one-to-one on the operational semantics. For example, instructions **ld** (**st**) move data between the data segment and the stack segment and match the rules (5), in Figure 6.4, and (1), in Figure 6.3, respectively. On the other hand, instruction **ret** has no correspondence

$p ::= h d b$	<i>Program</i>
$h ::= i_1 i_2 i_3$	<i>Header</i>
$d ::= \tilde{v}$	<i>Data Segment</i>
$v ::= \text{bools} \mid \text{ints} \mid \text{floats}$	<i>Values</i>
$b ::= \tilde{r}$	<i>Text Segment</i>
$r ::= \mathbf{ld} \ i \mid \mathbf{st} \ i \mid \mathbf{wrt} \ i_1 \ i_2 \mid \mathbf{rd} \ i_1 \ i_2$	<i>Instructions</i>
$\mid \mathbf{rad} \ i \mid \mathbf{bf} \ i \mid \mathbf{jp} \ i \mid \mathbf{ret}$	
$\mid \mathbf{bop} \mid \mathbf{uop}$	

Figure 7.1: Byte-code syntax.

in the syntax, it marks the end of the **text** block and is used to stop the machine. Instructions have a 1 byte opcode and eventually 1 or 2 extra bytes for arguments. In addition, a simplified explanation of all instructions available for this machine, is provided hereafter:

- **ld**  $i$ : loads 4 bytes at offset  $i$  in data segment, to top of stack.
- **st**  $i$ : stores 4 bytes at top of stack at offset  $i$  in data segment.
- **wrt**  $i_1 \ i_2$ : selects actuator  $i_1$  and write with  $i_2$  parameters on top of stack.
- **rd**  $i_1 \ i_2$ : select sensor  $i_1$  and read value from it given  $i_2$  internal parameters in the stack. The value returned is left on top of the stack.
- **rad**  $i$ : send  $i$  values on top of stack to radio.
- $i$ : **branch on the value on top of the stack by offset  $i$  or continue with the next instruction.**
- **jp**  $i$ : **jump to offset  $i$ , counting from end of instruction.**
- **ret**: **stop machine.**
- **bop**, **uop**: **binary and unary boolean, integer and floating point instructions.**

$$\begin{aligned}
\llbracket T \rrbracket &= \llbracket \mathbf{sensors} \{ \widetilde{s} : \tau \} \rrbracket : \\
&\quad \llbracket \mathbf{actuators} \{ \widetilde{a} : \tau \} \rrbracket : \\
&\quad \llbracket \mathbf{radiates} [\widetilde{r}] \rrbracket : \\
&\quad \llbracket \mathbf{data} \{ \widetilde{q} \} \rrbracket : \\
&\quad \llbracket \mathbf{text} \{ \widetilde{r} \} \rrbracket : \\
&\quad (\epsilon, \mathbf{ret}) \\
\llbracket \mathbf{sensors} \{ \widetilde{s} : \tau \} \rrbracket &= (\epsilon, \epsilon) \wedge S = \{ s_i : i \mid s_i \in \widetilde{s} \} \\
\llbracket \mathbf{actuators} \{ \widetilde{a} : \tau \} \rrbracket &= (\epsilon, \epsilon) \wedge A = \{ a_i : i \mid a_i \in \widetilde{a} \} \\
\llbracket \mathbf{radiates} [\widetilde{r}] \rrbracket &= (\epsilon, \epsilon) \\
\llbracket \mathbf{data} \{ \widetilde{q} \} \rrbracket &= \llbracket \widetilde{q} \rrbracket \\
\llbracket \tau x = v \widetilde{q} \rrbracket &= (v, \epsilon) : \llbracket \widetilde{q} \rrbracket \wedge V[x] \leftarrow i, i \leftarrow i + 4 \\
\llbracket \mathbf{text} \{ \widetilde{r} \} \rrbracket &= \llbracket \widetilde{r} \rrbracket \\
\llbracket r \widetilde{r} \rrbracket_r &= \llbracket r \rrbracket_r : \llbracket \widetilde{r} \rrbracket_r \\
\llbracket x = e \rrbracket_r &= \llbracket e \rrbracket_e : (\epsilon, \mathbf{st} : V(x)) \\
\llbracket a(\widetilde{e}) \rrbracket_r &= \llbracket \widetilde{e} \rrbracket_e : (\epsilon, \mathbf{wrt} : A(a) : |\widetilde{e}|) \\
\llbracket \mathbf{radio} [\widetilde{e}] \rrbracket_r &= \llbracket \widetilde{e} \rrbracket_e : (\epsilon, \mathbf{rad} : |\widetilde{e}|) \\
\llbracket \mathbf{if} e \{ \widetilde{r}_1 \} \mathbf{else} \{ \widetilde{r}_2 \} \rrbracket_r &= \llbracket e \rrbracket_e : (D', B') \\
&\quad \text{where} \\
&\quad (D_1, B_1) = \llbracket \widetilde{r}_1 \rrbracket_r \\
&\quad (D_2, B_2) = \llbracket \widetilde{r}_2 \rrbracket_r \\
&\quad D' = D_1 : D_2 \\
&\quad j_1 = 2 + |B_1| \\
&\quad j_2 = |B_2| \\
&\quad B' = \mathbf{bf} : j_1 : B_1 : \mathbf{jp} : j_2 : B_2
\end{aligned}$$

Figure 7.2: Translation to bytecode (part I).

### 7.1.2 Translation

Each translation function receives a syntactic term and returns a pair of sequences  $(D, B)$  (Figure 7.2). The first,  $D$ , is the contribution to the data segment and has type  $\text{Array}(\mathbf{bool} \cup \mathbf{int} \cup \mathbf{float})$ . The latter,  $B$ , is the contribution to the text segment and has type  $\text{Array}(\mathbf{byte})$ . The top level translation function  $\llbracket \cdot \rrbracket$ , for STL tasks, breaks the translation into a sequence of pairwise concatenations (operator “.”) and uses appropriate translation functions for each syntactic category. We used the same notation  $\llbracket \cdot \rrbracket$  to simplify the notation except in the case of instructions  $\llbracket \cdot \rrbracket_r$  and of expressions  $\llbracket \cdot \rrbracket_e$ . Besides the maps  $S$  and  $A$  (c.f., Sub-Section 6.2) we use  $V$  and  $U$ , of types  $\text{Map}(\text{Vars}, \mathbf{int})$  and  $\text{Map}(\mathbf{bool} \cup \mathbf{int} \cup \mathbf{float}, \mathbf{int})$ , respectively, to map variables and constants into data segment offsets. The current free offset is kept in variable  $i$ , which is initially 0.

$$\begin{aligned}
 \llbracket e_1, \dots, e_n \rrbracket_e &= \llbracket e_1 \rrbracket_e : \dots : \llbracket e_n \rrbracket_e \\
 \llbracket s(\tilde{e}) \rrbracket_e &= \llbracket \tilde{e} \rrbracket_e : (\epsilon, \mathbf{rd} : S(s) : |\tilde{e}|) \\
 \llbracket e_1 \mathbf{bop} e_2 \rrbracket_e &= \llbracket e_1 \rrbracket_e : \llbracket e_2 \rrbracket_e : (\epsilon, \mathbf{bop}) \\
 \llbracket \mathbf{uop} e \rrbracket_e &= \llbracket e \rrbracket_e : (\epsilon, \mathbf{uop}) \\
 \llbracket x \rrbracket_e &= (\epsilon, \mathbf{ld} : V(x)) \\
 \llbracket u \rrbracket_e &= (u, \mathbf{ld} : U(u)) \wedge U[u] \leftarrow i, i \leftarrow i + 4 \\
 \llbracket \epsilon \rrbracket_e &= (\epsilon, \epsilon)
 \end{aligned}$$

Figure 7.3: Translation to bytecode (part II).

The translation is quite forward with the data segment being used to store all variables and constants. The data segment works as the only activation record required for the execution of the task. This simplicity of layout also allows us to use a single instruction,  $\mathbf{ld}$ , to move both variable contents and constants to the stack. An apparently unintuitive translation assigns sensor and actuator identifiers to integers, rather than functions as in the operational semantics. However, these integers are indexes into arrays of built-in functions in the virtual machine, the concrete realizations of the  $f_i$  and  $g_i$  functions of the operational semantics.

$$\frac{B[j] = \mathbf{ld} \quad B[j+1] = i \quad v \leftarrow D[i]}{[D|S|B]_j \rightarrow [D|v, S|B]_{j+2}} \quad (7.1)$$

$$\frac{B[j] = \mathbf{st} \quad B[j+1] = i \quad D' \leftarrow D + \{i : v\}}{[D|v, S|B]_j \rightarrow [D'|S|B]_{j+2}} \quad (7.2)$$

$$\frac{B[j] = \mathbf{rd} \quad B[j+1] = i \quad B[j+2] = n \\ f \leftarrow \mathit{sensors}[i] \\ v \leftarrow f(v_1, \dots, v_n)}{[D|v_1, \dots, v_n, S|B]_j \rightarrow [D|v, S|B]_{j+3}} \quad (7.3)$$

$$\frac{B[j] = \mathbf{wrt} \quad B[j+1] = i \quad B[j+2] = n \\ g \leftarrow \mathit{actuators}[i] \\ g(v_1, \dots, v_n)}{[D|v_1, \dots, v_n, S|B]_j \rightarrow [D|S|B]_{j+3}} \quad (7.4)$$

$$\frac{B[j] = \mathbf{rad} \quad B[j+1] = n \quad \mathit{send}(v_1, \dots, v_n)}{[D|v_1, \dots, v_n, S|B]_j \rightarrow [D|S|B]_{j+2}} \quad (7.5)$$

$$\frac{B[j] = \mathbf{bf} \quad B[j+1] = i}{[D|\mathbf{false}, S|B]_j \rightarrow [D|S|B]_{j+2+i}} \quad (7.6)$$

$$\frac{B[j] = \mathbf{bf} \quad B[j+1] = i}{[D|\mathbf{true}, S|B]_j \rightarrow [D|S|B]_{j+2}} \quad (7.7)$$

$$\frac{B[j] = \mathbf{jp} \quad B[j+1] = i}{[D|S|B]_j \rightarrow [D|S|B]_{j+2+i}} \quad (7.8)$$

$$\frac{B[j] = \mathbf{bop}}{[D|v_2, v_1, S|B]_j \rightarrow [D|v_1 \mathbf{bop} \ v_2, S|B]_{j+1}} \quad (7.9)$$

$$\frac{B[j] = \mathbf{uop}}{[D|v, S|B]_j \rightarrow [D|\mathbf{uop} \ v, S|B]_{j+1}} \quad (7.10)$$

$$\frac{B[j] = \mathbf{ret}}{[D|S|B]_j \rightarrow \perp} \quad (7.11)$$

Figure 7.4: Transition rules for SVM.

### 7.1.3 Semantics

The state of the virtual machine is represented as the term  $[D|S|B]_j$ , where  $j$  is the program counter and is used to travel the instructions in the text segment. The halted machine is represented by a special state denoted  $\perp$ . To run a task  $T$  in the virtual machine we use the translation function to get its byte code  $\llbracket T \rrbracket = (D, B)$  and set its initial state to  $[D|\epsilon|B]_0$ . The computation proceeds according to the reduction rules presented in Figure 7.4. For example, when the current instruction (the one the program counter  $j$  is indexing) is **ld**, rule (1), the next byte contains  $i$ , the offset of a variable or a constant in the data segment, and we use it to access the value (denoted as  $v \leftarrow D[i]$ ). The new state has the same data and text segments but the stack has the value  $v$  on top of it, and the program counter was updated to  $j + 2$ . Similarly for **st**, rule (2), we have a value  $v$  in the stack in the current state and we make a transition to a state where that value has been removed from the stack and copied to position  $i$  in the data segment (denoted as  $D' = D + \{i : v\}$ ). The arrays *sensors* (rule (3)) and *actuators* (rule (4)) provide the pointers to built-in functions associated with the identifiers. The function *send* (rule (5)) is a built-in that sends data over the radio. Tasks never listen for data. For example, in rule (3), the **rd** instruction has two arguments:  $i$ , the index that identifies the built-in sensor function to be called, and  $n$ , the number of arguments that function takes. The latter,  $v_1 \dots v_n$ , are all stored at the top of the stack. The rule evolves by calling a built-in function  $f \leftarrow \text{sensor}[i]$  with the arguments taken from the stack,  $f(v_1 \dots v_n)$  and placing the result of the call,  $v$ , at the top of the stack. Instructions **bop** and **uop** actually encapsulate a set of rules that include the usual arithmetic, relational and logical binary and unary operators.

## 7.2 The SONAR Operating System

A node in the SONAR architecture may run multiple periodic tasks at the request of clients. A user composes a task and compiles it in a SONAR client application. It can then submit the task to be executed by the nodes in a given deployment. Such a request is sent via the SONAR middleware, which then forwards it to the adapter of the deployment in question and finally to the gateway where the bytecode is radioed to the mesh using a simple protocol implemented on top of XBee (Figure 3.5). The protocol uses broadcast to send tasks. If a task is smaller than the XBee payload (usually 92 bits) then a single broadcast is sufficient. Otherwise, the task is broken in numbered blocks and these are sent in independent broadcast events over an interval.



Only tasks that can be fully re-assembled in the nodes are scheduled for execution, by the SOS. At this point we do not try to check whether tasks are received correctly by all nodes.

### 7.2.1 Gateway

---

**Algorithm 6** The gateway program

---

```

function MAIN()
  ATTACH(RADIO_RCV, HANDLERADIOMSG)
  ATTACH(SERIAL_RCV, HANDLESERIALMSG)
  loop
    MICROSLEEP()
    switch ( src )
      case RADIO:
        msg ← READRADIORCVBUFFER()
        FORWARDTOADAPTER(msg)
      case SERIAL:
        msg ← READSERIALRCVBUFFER()
        FORWARDTONODES(msg)
    end switch
  end loop
end function

function HANDLERADIOMSG()
  src ← RADIO
end function

function HANDLESERIALMSG()
  src ← SERIAL
end function

```

---

The gateway does not run tasks. It is simply a message forwarder: it receives data messages from nodes in the deployment and passes them to the adapter, and receives control messages (including new tasks) from the adapter and radios them to the nodes in the deployment (Figure 3.2). Algorithm 6 shows this basic component. The gateway is initialized by attaching two handlers for interrupts signaling radio and serial port data reception. It then sleeps most of the time. When one of the interrupts is detected,

the corresponding handler is executed and a flag is set to identify the source. The remainder of the loop then processes the incoming message.

## 7.2.2 Nodes

Nodes, conversely, may run multiple tasks kept on a table ( $Table : Array(Entry)$ ). Each entry in this table contains a boolean that says whether the entry is valid, the task's period (an integer), its next activation time (another integer), and the stored state ( $Entry : \mathbf{bool} \times \mathbf{int} \times \mathbf{int} \times State$ ). The latter is composed of an array of bytes that keeps the data, stack and text segments ( $State : Array(\mathbf{int} \cup \mathbf{float}) \times Array(\mathbf{int} \cup \mathbf{float}) \times Array(\mathbf{byte})$ ). The task currently active is identified by its integer index, denoted *curr* in the following algorithms. To manage and schedule the tasks each node has a tiny operating system that executes a loop as described in Algorithm 7.

---

**Algorithm 7** The node main loop

---

```

function MAIN()
  ATTACH(RADIO_RCV, HANDLERADIOMSG)
  loop
    RUN()
    SCHEDULE()
    SLEEP()
    LISTEN()
  end loop
end function

```

---

A brief initialization attaches a handler for an interrupt that signals radio data reception, and initializes the current task (initially the built-in task). The node then enters the loop and executes the following procedures: `RUN`, that executes the current task; `SCHEDULE` - that selects the next task to be executed; `SLEEP` - that sleeps until the next task must be activated, and, finally - `LISTEN` - that listens for incoming radio commands that may have been received while executing elsewhere in the loop. The first 3 procedures are executed only if there are valid tasks in the table, i.e., the predicate `TABLEEMPTY` evaluates to false.

Procedure `RUN` (Algorithm 8) gets the stored state for the current task, its data, stack and text segments, and runs the task in the SVM. Note that changes to variables in a task are made directly in the data segment of the bytecode so that any state is

---

**Algorithm 8** Run current task

---

```

function RUN
  if  $\neg$ TABLEEMPTY() then
     $(D, \epsilon, B) \leftarrow$  GETBYTES(curr)
    RUNSVM(D,  $\epsilon$ , B)
     $t \leftarrow$  RTCTIME()
     $p \leftarrow$  PERIOD(curr)
    SETNEXTACTIVATION(curr,  $t + p$ )
  end if
end function

```

---

preserved in between successive activations of the task. The virtual machine preserves the invariant that the stack  $S$  is empty when a task begins to execute and when it exits. Finally, the procedure adjusts the next activation time for the task by adding its period to the current time given by the Real-Time Clock (RTC).

---

**Algorithm 9** Select next task

---

```

function SCHEDULE()
  if  $\neg$ TABLEEMPTY() then
     $min \leftarrow$  MAX_INT
    for  $0 < i <$  MAX_TASKS do
      if TASKVALID(i) then
         $t \leftarrow$  GETNEXTACTIVATION(i)
        if  $t \leq min$  then
           $min \leftarrow t$ 
           $curr \leftarrow i$ 
        end if
      end if
    end for
  end if
end function

```

---

The SCHEDULE procedure (Algorithm 9) computes the index of the (valid) task with the closest activation time. This becomes the next task to be executed by the operating system. Otherwise the predicate TABLEEMPTY will evaluate to true.

Procedure SLEEP (Algorithm 10) computes the time until the next task activation and programs an alarm to wake up the node. The node then goes to sleep. This specification builds on the underlying assumption that tasks, being so small, execute

---

**Algorithm 10** Sleep until next task activation
 

---

```

function SLEEP()
  if  $\neg$ TABLEEMPTY() then
     $t \leftarrow$  GETNEXTACTIVATION(curr)
    RTCALARM(t)
  end if
  MICROSLEEP()
end function

```

---

in only a tiny fraction of their corresponding periods. In other words, if a task has a period  $p$  and an execution time, per activation, of  $t$ , then  $t \ll p$ . Otherwise we make no effort to schedule tasks within their periods. Since  $t$  is in the order of microseconds we find this assumption adequate for practical purposes.

Finally, procedure LISTEN (Algorithm 11) checks for any incoming messages while the main loop was running. We assume that the nodes have the means to receive and to buffer messages asynchronously, by programming an appropriate handler to process the corresponding hardware interrupts. If a message is received, its tag is checked to identify its type and it is processed accordingly. At this point, there are 3 types of control messages: `ADD_TASK` - sends the identifier, the period and the bytecode for a new task to be executed in the nodes; `CHANGE_PERIOD` - sends the identifier and the new period of a task to be updated in the nodes, and; `REMOVE_TASK` - sends the identifier of a task to be invalidated in the nodes. When a new task is reassembled and copied to the task table, its next activation is set to `GETNEXTACTIVATION(curr) +  $\delta$` , where  $\delta$  is a delay introduced to make sure that the task is schedulable in the next loop run, i.e., its activation time is in the future when the `SCHEDULE` procedure is called.

### 7.3 Implementation

We have implemented a full prototype of the specification for the data layer in a WSN mesh composed of Arduino Mega 2560 devices, each with an extra shield with a XBee Series 2 radio, an SHT-15 temperature and humidity sensors, and an Adafruit Chronodot RTC. In particular, for this implementation, we used the C/C++ Wiring programming language, which is the default programming language for the Arduino devices. One of the devices acts as the gateway and is connected to a desktop computer through a USB connection. The computer runs the adapter web service which links

---

**Algorithm 11** Handle Incoming Radio Message
 

---

```

function HANDLERADIOINTERRUPT()
  interrupted ← TRUE
end function

function LISTEN()
  if interrupted = TRUE then
    msg ← READRADIORCVBUFFER()
    tag ← GETTAG(msg)
    switch ( tag )
      case ADD_TASK :
        i ← GETID(msg)
        p ← GETPERIOD(msg)
        b ← GETBYTES(msg)
        ADDTASK(i, p, b)
      case CHANGE_PERIOD :
        i ← GETID(msg)
        p ← GETPERIOD(msg)
        CHANGEPERIOD(i, p)
      case REMOVE_TASK :
        i ← GETID(msg)
        REMOVETASK(i)
    end switch
    interrupted ← FALSE
  end if
end function

```

---

the deployment with the SONAR middleware, running on a server. The binaries for the gateway and for the nodes (including the OS and the SVM) are loaded into the devices before they are deployed physically.

Table 7.1 shows the memory footprint and the total number of code lines for both the gateway and the nodes in this implementation. In this case the SVM in the nodes is configured to support a maximum of 8 tasks each with a maximum byte-code size of 128 bytes. At this stage no effort was made to optimize the code both in terms of size and energy consumption. For example, since speed is not a major constraint, we can take advantage of the typical Harvard architecture of these devices and transfer some of the data structures currently stored in the SRAM to the Flash, which is usually substantially larger.

	Flash (256 kb)	SRAM (8 kb)	#lines
gateway	10.6 kb (4%)	1.3 kb (16%)	330
node	20.3 kb (8%)	2.2 kb (26%)	1325

Table 7.1: Memory consumption for the gateway and nodes.

This SVM configuration, although quite generous, is actually quite compact and fits easily in the Mega 2560 and almost fits in the smallest AVR Atmel micro-controller, the ATMega32 (32 kb Flash, 2 kb SRAM) - 2.2kb used vs. 2kb available SRAM. A SVM configuration with 4 tasks instead of 8 would fit nicely with a 1.6 kb SRAM footprint. However, it is important to notice that of these 2.2kb used, only 1.1kb represents the space occupied by the task table, which is the only significant structure, in terms of memory space, created by our code. These numbers show that half of those 2.2kb, are allocated for data structures used by Arduino libraries. This leaves room for further optimizations, as some of these libraries are overkill for our needs. Also, of the 1325 lines of code in the nodes, 43% correspond to the OS, 27% to the SVM, and just 30% correspond to hardware specific code, e.g., modules for sensors, actuators, radio, and real-time clock. These numbers give us confidence that porting (and optimizing) this data-layer to more resource constrained devices will not present major problems.

# Chapter 8

## The Processing and Client Layers

In this chapter we describe the changes introduced in the processing and client layers.

### 8.1 Processing Layer

As stated in section 5.1, the task pool being directly handled by the SONAR service, proved to be a non-scalable solution because of the potential large number of queries to the data-store, when considering a high number of running tasks. Also, our first prototype had a critical limitation in terms of data store since it was only able to use MySQL databases. In order to solve such a limitation we designed and implemented a more generic solution, which would be capable of supporting any type of data store. This solution is a generic common interface that allows the usage of different types of data-store by abstracting away the specific implementation of each data-store from the SONAR service.

#### 8.1.1 The Data Store

The data store is one important component in SONAR, since it must be able to abstract and represent all the data generated by deployments. Taking that into consideration, and the new components of the data layer, we also decided to reformulate the relational representation of the data store, in order to acquire a more accurate and reliable representation of the deployment's data and in the process also simplify it, as it is depicted in Figure 8.1. In this new representation, the tables provides a generic abstraction for a deployment, focusing on their tasks and the data they generate. A

top-level table - **Deployment**, keeps information about the deployment, e.g., its name, an optional description and the underlying hardware platform. It indexes one further table that contain information about the tasks - **Task**, scheduled and executed by the SOS and the SVM. Tasks described in the **Task** table may be active (running in the nodes) or killed (removed from the nodes). This table also keeps the deployment, which the task is associated with and some additional information, e.g., its periodicity, status and an optional description. The actual data produced by the tasks, is stored in a single table - **Data**, accessed through the task table, that keeps the data indexed by the task and the deployment identifiers.

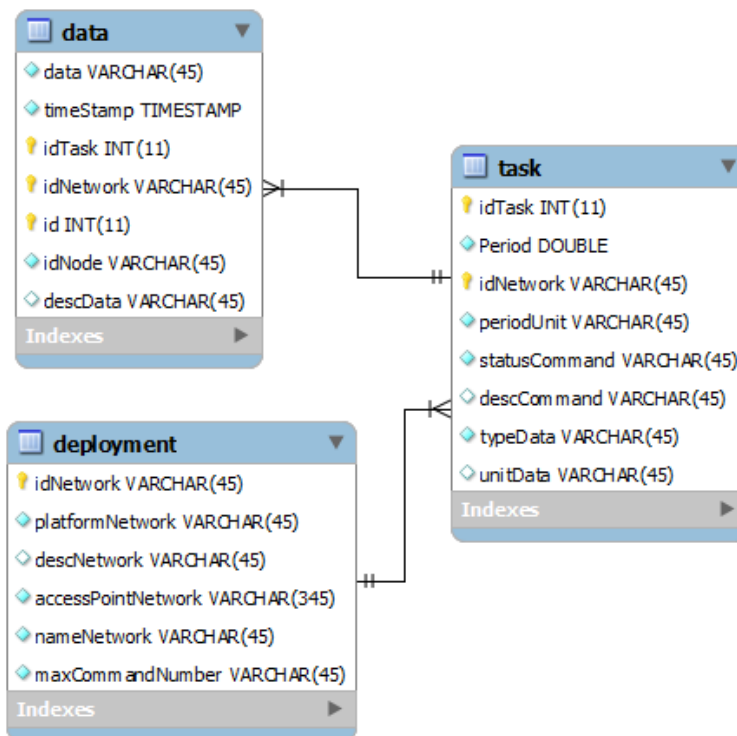


Figure 8.1: EER diagram for the new SONAR data-store

The changes in the relational representation of the data-store, was strictly necessary because we think SONAR as an out-of-the-box solution, i.e., all the components required would come pre-installed. Thus, all the information concerning the hardware/software configuration of the deployment were redundant and could be removed from the representation. This led to a better representation of the produced data and also a simplified one.



### 8.1.2 The Generic Interface for communication with data store

In Section 5.1 we mentioned a critical limitation concerning the data store component since our original implementation supported only MySQL databases, which could restrict the usage of SONAR to only one very specific case scenario, contradicting the primary goal of the project. Not only that but the number of possible data stores, being used simultaneously was restricted to only one. However different users might want to store the data from their deployments in different data stores of different types. Thus in consideration of such limitations, we decided to implement a generic interface, which would be able to support any type of data store, e.g., relational, wide column based and document based.

The generic interface component was designed to mediate the communication between the SONAR service and the data store, abstracting away the specific implementation from the SONAR service. In order to implement such an interface, we used two design patterns: Data Access Object (DAO) and; Factory Method Pattern (FMP).

The DAO pattern provides an abstraction layer between the business logic tier (business object) and the persistent storage tier (data source). Business objects access data sources via data access objects. This abstraction layer encapsulates the persistent storage specific product implementation [39].

Gamma et al., define the FMP as [40]:

*Define an interface for creating an object, but let the subclasses decide which class to instantiate. The Factory method lets a class defer instantiation to subclasses.*

The generic interface implementation is represented using UML Classes Diagrams, depicted in the Figures 8.2, 8.3, 8.4 and 8.5. Notice that each of the aforementioned design patterns, are being directly used as the name of the classes in the UML Classes Diagrams.

The Figures 8.2, 8.3 and 8.4 directly represent the usage of the DAO design pattern, where the relational database model depicted in the Figure 8.1, is directly mapped into objects that contains the same attributes types and names as those aforementioned. Such objects are also meant to be instantiated, since they represent the concrete data that populates the data store, or is retrieved from it.

The interfaces, are responsible for abstracting the concrete implementation of queries

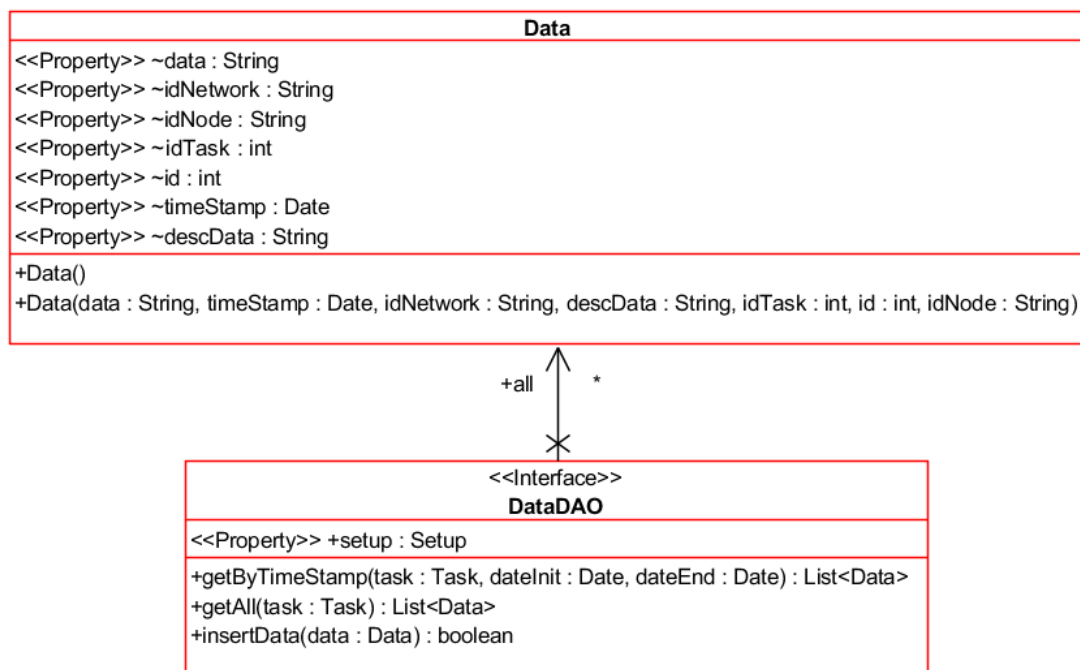


Figure 8.2: DAO for Data mapped object

to data stores. Together with the Factory Method design pattern implementation, which is depicted in the Figure 8.5, each implementation of the interfaces is handled individually, meaning that given the proper implementation of some specific database query language, the factory is capable of returning the desired implementation and the web service does not have to worry about specific details when using methods to insert or retrieve data from the different data stores.

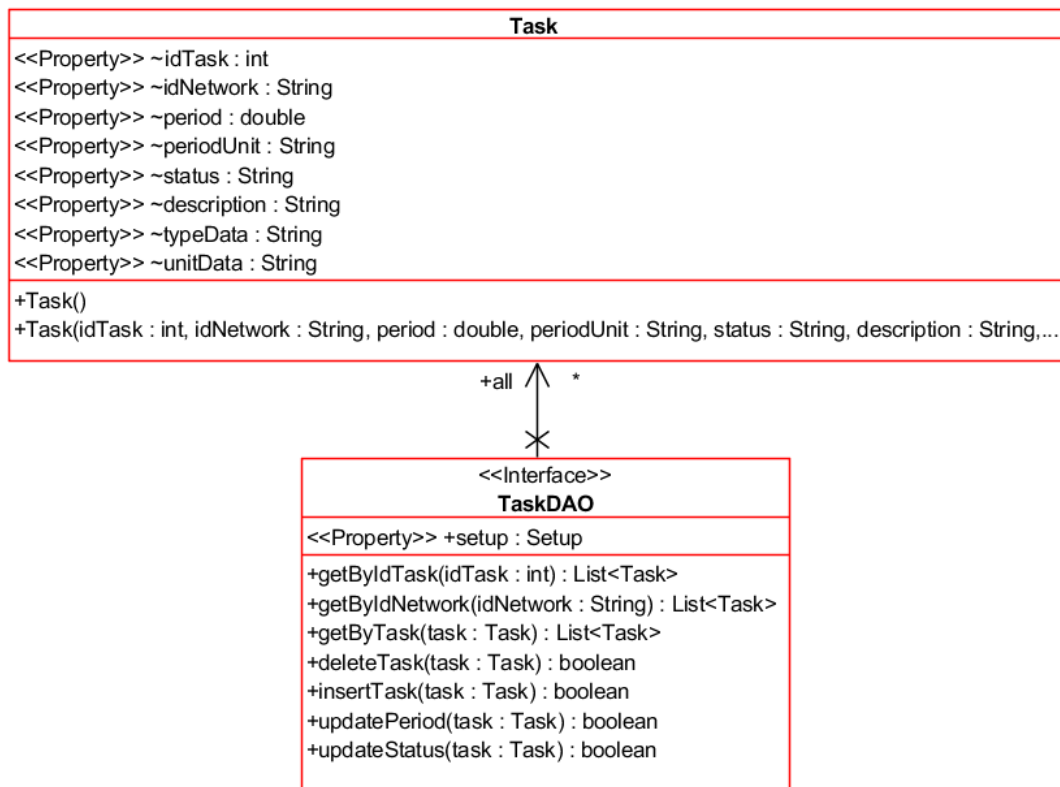


Figure 8.3: DAO for Task mapped object

## 8.2 The Client Layer

In Section 4.4 we presented our first implementation of the client layer. This implementation was focused on having a user-friendly application in which an end-user would be able to manage and interact with deployments. However, our application did not address access control problems. This means that all users were capable of interacting and managing all registered deployments in the data store. However, it is reasonable to think that each user has its own deployment(s), and should only be allowed to manage these.

As a consequence of the aforementioned facts, we decided to redesign the client's GUI. In particular, with the new design, a user would only be allowed to manage and interact with its own deployment(s). In order to achieve such a goal, we decided to implement an authentication and system administration functionalities.

The admin functionalities are only accessible through the client's GUI at the moment

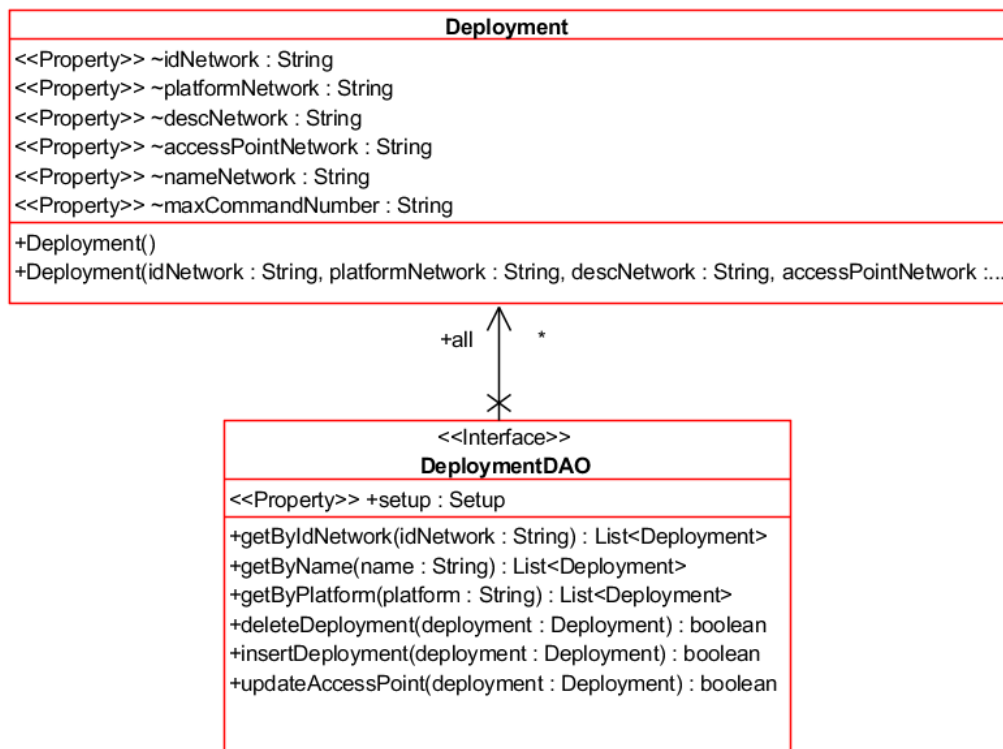


Figure 8.4: DAO for Deployment mapped object

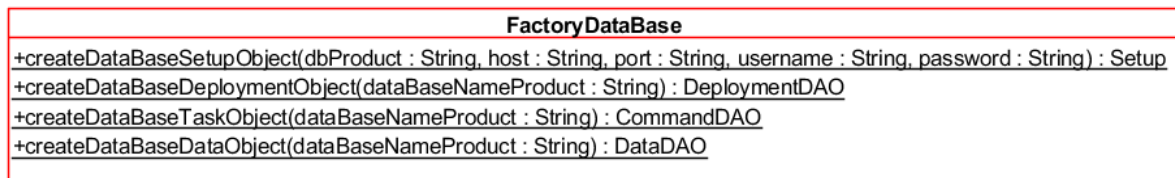


Figure 8.5: Factory Method for retrieve the appropriate implementation

of the login, using the system's predefined user name and password. The admin's role is to manage the mappings between data stores, users and deployments. In order to accomplish this, the admin manages a local database, further detailed in Section 8.2.3.

On the other hand, new users are created by the admin, and provided with a login. This login, is used to correctly identify which deployments are available to the logged user, and only those are going to be manageable by him.

### 8.2.1 Deployment List View

When the client is started, Figure 8.6, it offers the possibility for the user to login with a user name and password, and choose which SONAR-server he wants to connect with.

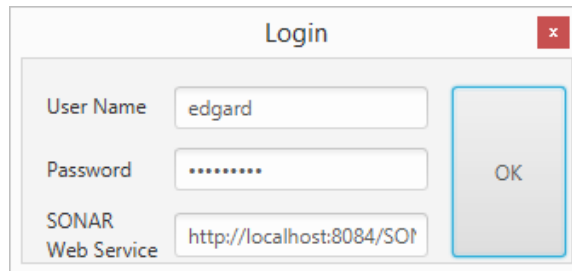


Figure 8.6: Login and SONAR-service connect window.

Afterwards, the main window of the client is instantiated as depicted in Figure 8.7. The deployment list contains all the deployments that belong to the logged user. There may be other users that can access the same deployments, since the admin is allowed to map more than one user to the same deployment. In addition, the admin has access to all available deployments, in the data store. This is necessary, since the admin needs to manage the mapping between users and deployments.

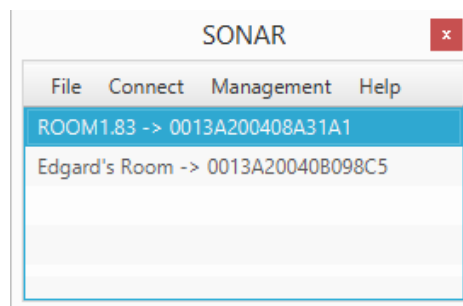


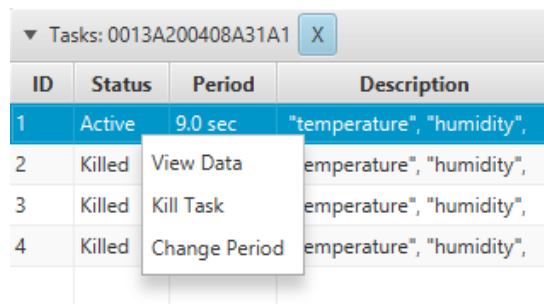
Figure 8.7: Deployment view.

When the deployment is selected, the user can double-click and the task view window is instantiated, showing all tasks in that deployment. However, in the case of the admin, a management window would be presented, allowing him to manage which

data-store and users can access this deployment. The admin's privileges are further explained in Section 8.2.3.

## 8.2.2 Task View

The task view, Figure 8.8, is used to manage the deployment's tasks. It allows the user to add more tasks, view their respective data, change their periods or even killing them.



The screenshot shows a window titled "Tasks: 0013A200408A31A1" with a close button (X). Below the title bar is a table with the following columns: ID, Status, Period, and Description. The table contains four rows of data. A context menu is open over the second row (ID 2, Status Killed), showing three options: "View Data", "Kill Task", and "Change Period".

ID	Status	Period	Description
1	Active	9.0 sec	"temperature", "humidity",
2	Killed	View Data	temperature", "humidity",
3	Killed	Kill Task	temperature", "humidity",
4	Killed	Change Period	temperature", "humidity",

Figure 8.8: Task view: Context Menu.

In this view the available actions, concerning the selected task, are always dependent on their own status. A task can have two possible status:

- Active: The task is being executed in all nodes of the deployment.
- Killed: The task was killed in all nodes of the deployment.

Suppose a task has its status as active, the user can kill it, change its period or view its produced data. Although when a task has its status as killed, the user can not activate it again, since the task has already been removed from all nodes. This means that the only possible action, in this case, is to visualize their past data, as it is depicted in the Figure 8.9.

Regardless the status of all current tasks, the user can add more tasks to the deployment. Although the number of concurrent active tasks is limited by the SVM running on the nodes. Thus, assuming the user has not reached its maximum number, and he has the task's byte code generated by the compiler of the STL, he has to right-click with their mouse button on an empty row and the option Add Task will be prompted

▼ Data: 0013A200408A31A1 X				
Description	MAC	Value	Unit	Date
temperature	0013A200408A31C5	26.08	Celsius	2014-04-22T03:42:20+01:00
humidity	0013A200408A31C5	52.77	%	2014-04-22T03:42:20+01:00
temperature	0013A200408A31C5	26.05	Celsius	2014-04-22T03:42:29+01:00
humidity	0013A200408A31C5	52.70	%	2014-04-22T03:42:29+01:00
temperature	0013A200408A31C5	26.07	Celsius	2014-04-22T03:42:38+01:00
humidity	0013A200408A31C5	52.64	%	2014-04-22T03:42:38+01:00
temperature	0013A200408A31C5	26.10	Celsius	2014-04-22T03:42:47+01:00
humidity	0013A200408A31C5	52.62	%	2014-04-22T03:42:47+01:00
temperature	0013A200408A31C5	26.13	Celsius	2014-04-22T03:42:56+01:00
humidity	0013A200408A31C5	52.43	%	2014-04-22T03:42:56+01:00
temperature	0013A200408A31C5	26.17	Celsius	2014-04-22T03:43:05+01:00
humidity	0013A200408A31C5	52.40	%	2014-04-22T03:43:05+01:00
temperature	0013A200408A31C5	26.20	Celsius	2014-04-22T03:43:14+01:00
humidity	0013A200408A31C5	52.37	%	2014-04-22T03:43:14+01:00
temperature	0013A200408A31C5	26.20	Celsius	2014-04-22T03:43:23+01:00
humidity	0013A200408A31C5	52.28	%	2014-04-22T03:43:23+01:00
temperature	0013A200408A31C5	26.20	Celsius	2014-04-22T03:43:32+01:00
humidity	0013A200408A31C5	52.37	%	2014-04-22T03:43:32+01:00

Figure 8.9: Data view.

to him. By selecting it, the user will have to select the task's byte code and type its periodicity. Afterwards, the submitted task will be active and running in all nodes of the deployment.

### 8.2.3 Admin Mode

When logged in as admin, the deployment list view displays all the previously registered deployments in the admin's local database. Also, the Management menu is available, as depicted in the Figure 8.10.

The Management menu, contains some possible actions to be performed by the admin. Those consists on registering new deployments, users and/or database repositories, as it is depicted in the Figure 8.10.

Before mapping a data store to a deployment, the admin needs to register the data

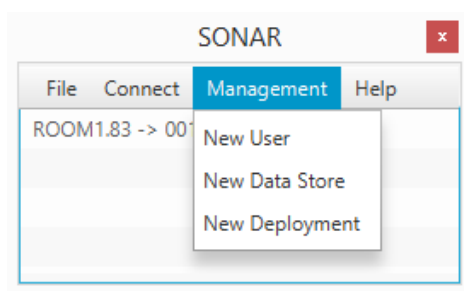


Figure 8.10: The Management menu.

store. This mapping process must be done, since a deployment needs a data repository to store its produced data, and a user needs to have access to that same data repository. The registration of this new data store is done by selecting the option New Data Store, and the window depicted in Figure 8.11 will be instantiated. The admin must fill all the fields, in order to properly register its new database repository. Otherwise, an error message will appear informing him of the fact and he will not be allowed to conclude the registration process. Afterwards, the admin may register a new user and/or a new deployment, as depicted in Figures 8.12, 8.13, respectively.

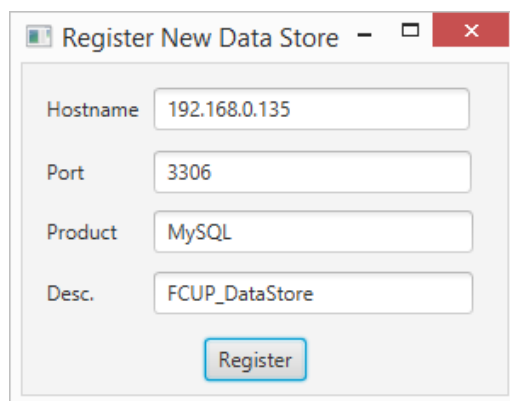
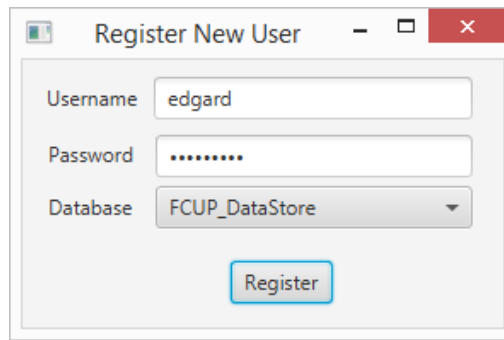


Figure 8.11: The register data store window.

Finally, after having registered all data repositories, users and deployments, the admin is allowed to authorize users to manage and interact with deployments. It is done by double-clicking over a selected deployment, in the deployment list. This will instantiate the Authorize window, as it is depicted in the Figure 8.14.





Register New User

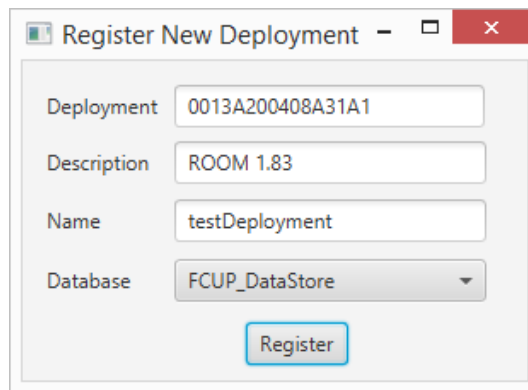
Username: edgard

Password: .....

Database: FCUP\_DataStore

Register

Figure 8.12: The register user window.



Register New Deployment

Deployment: 0013A200408A31A1

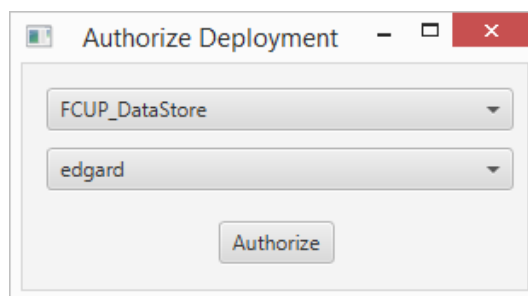
Description: ROOM 1.83

Name: testDeployment

Database: FCUP\_DataStore

Register

Figure 8.13: The register deployment window.



Authorize Deployment

FCUP\_DataStore

edgard

Authorize

Figure 8.14: The Authorize window.

Management actions performed by the admin, are stored in an internal database, called Admin's Database, kept in the same machine as the SONAR service. Fig-

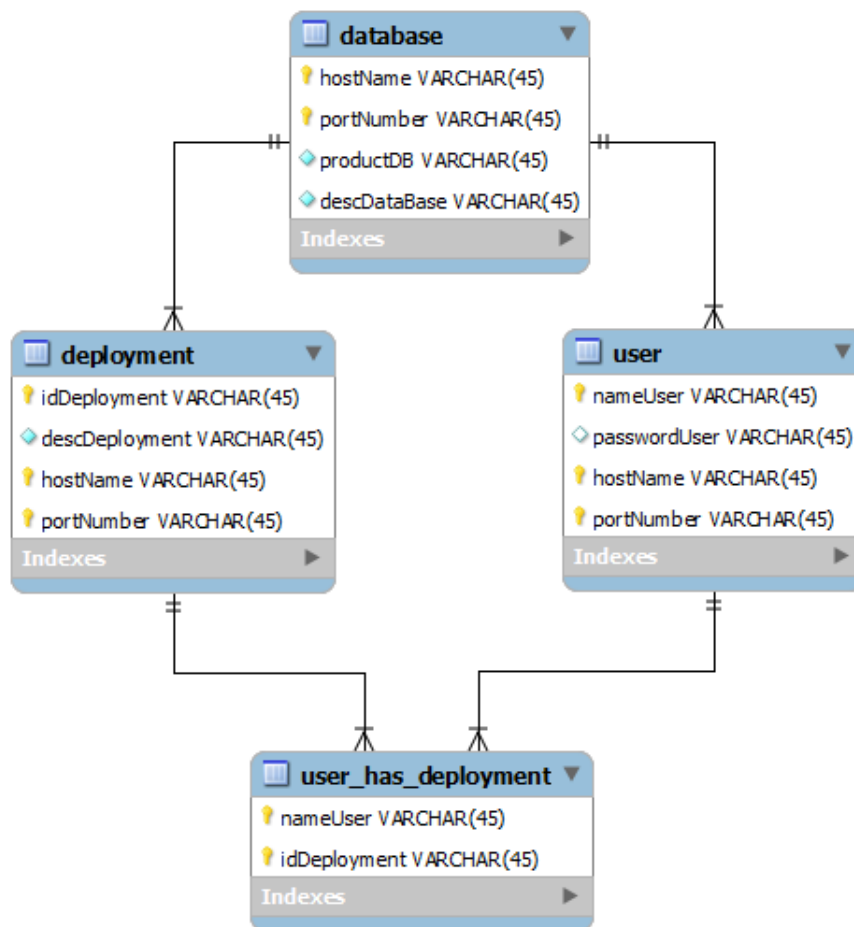


Figure 8.15: EER diagram of the admin's database.

Figure 8.15, represents the EER diagram of that database, and the relationship among its components. The admin's database, is responsible for maintain the relationship among users, databases, and deployments. Also, for the communication with this database, we use the generic interface, depicted in the Section 8.1.2.

## Chapter 9

# Conclusions and Future Work

In this thesis we have presented SONAR, a multi-platform middleware architecture for WSN. In particular, SONAR differs from the most solutions on the market, because it focuses on the (non-specialist) end-user. SONAR offers a complete generic and transparent solution, both in terms of software and hardware. In terms of software, the end-user is provided with a graphical user interface, that allows him to monitor the data produced by sensor nodes and to interact with the deployments, through periodic tasks. These tasks are written using the STL, a simple platform independent, domain specific language. These tasks are compiled into byte-code, and injected in the deployment nodes, where they are executed by a compact virtual machine, the SVM. In addition, these tasks are also dynamic loaded on the sensor nodes. This means that, differently from the static programmed applications, they can be reprogrammed or deleted, from the sensor nodes, without having to stop and reprogram all the nodes, on the WSN. This is made possible by the usage of a compact operating system, the SOS. It manages and runs the tasks on the nodes, using the SVM. In addition, both SOS and SVM can be ported to different hardware platforms with a minimal effort. The current prototype that we have implemented with the new revision of the data layer, has also been submitted to the conference MASS 2014 [41].

However, SONAR is still a work in progress project. The approach we take is not without problems. We deliberately make assumptions that diminish the span of applications that can be programmed for a given deployment in order to optimize the architecture for a common use case. One particular example is the current impossibility of collaboration between nodes, e.g., in the aggregation of data or processing. This possibility is particularly important in very large networks that have to scale while keeping power consumption within bounds. Future work involves evaluating the

impact of introducing data aggregation, or online processing primitives in the STL and the SVM. Also, proving the type-safety of the STL, and the correctness of the SVM and of the SOS, would provide users with the reassurance that compiled tasks would not produce run-time errors. This would be a significant result, given the difficulty in debugging deployed WSN applications.

Finally, from a personal point of view, I believe that this project contributed to further expand my knowledge in the area of embedded systems, namely to WSN. In addition, I was also presented with different challenges, which helped me to improve my technical knowledge and my interpersonal skills. These skills helped me to understand the importance of working as team member, and made me realize that no science can be done by just one man.

# Bibliography

- [1] I. F. Akyildiz, S. Weilian, Y. Sankarasubramaniam, and C. Erdal. Wireless sensor networks: a survey. *Computer networks*, 38(4):393–422, 2002.
- [2] University of Cantabria. Smart City Project. Available at <http://www.smartsantander.eu/>. [Online; accessed 12-June-2014].
- [3] Libelium World. E-health sensor platform. Available at [www.libelium.com/130220224710](http://www.libelium.com/130220224710), 2014. [Online; accessed 12-June-2014].
- [4] DeltaZ. ARES Project. Available at [http://www.libelium.com/ehealth\\_monitor\\_medical\\_drug\\_preservation\\_waspmote/](http://www.libelium.com/ehealth_monitor_medical_drug_preservation_waspmote/). [Online; accessed 12-June-2014].
- [5] Harvard Sensor Network Labs. CodeBlue: Wireless Sensors for Medical Care. Available at <http://fiji.eecs.harvard.edu/CodeBlue>. [Online; accessed 12-June-2014].
- [6] Grupo Austen. Siega System Project. Available at <http://www.siegasystem.com/>. [Online; accessed 12-June-2014].
- [7] M.M. Wang, J.N. Cao, J. Li, and D. K. Sajaal. Middleware for wireless sensor networks: A survey. *Journal of computer science and technology*, 23(3):305–326, 2008.
- [8] E. Avilés-López and J. A. García-Macías. TinySOA: a Service-Oriented Architecture for Wireless Sensor Networks. *Service Oriented Computing and Applications*, 3(2):99–108, 2009.
- [9] K. Aberer, M. Hauswirth, and A. Salehi. A Middleware for Fast and Flexible Sensor Network Deployment. In *Very Large Data-Bases (VLDB'06)*, pages 1199–1202. ACM Press, 2006.

- [10] T. Gross, T. Egla, and N. Marquardt. Sens-ation: a Service-Oriented Platform for Developing Sensor-Based Infrastructures. *International Journal of Internet Protocol Technology (IJIPT)*, 1(3):159–167, 2006.
- [11] P. Gibbons, B. Karp, Y. Ke, S. Nath, and S. Seshan. IrisNet: an architecture for a worldwide sensor Web. *Pervasive Computing*, 2(4):22–33, 2003.
- [12] R. Alonso, D. Tapia, J. Bajo, and et al. Implementing a Hardware-Embedded Reactive Agents Platform Based on a Service-Oriented Architecture over Heterogeneous Wireless Sensor Networks. *Ad-Hoc Networks*, 11(1):151–166, 2013.
- [13] D. Tapia, R. Alonso, F. De La Prieta, C. Zato, and et al. SYLPH: An Ambient Intelligence Based Platform for Integrating Heterogeneous Wireless Sensor Networks. In *IEEE International Conference on Fuzzy Systems (FUZZ 2010)*, pages 1–8. IEEE Press, July 2010.
- [14] R. Khoury, T. Dawborn, B. Gafurov, and et al. Corona: Energy-Efficient Multiquery Processing in Wireless Sensor Networks. In *Database Systems for Advanced Applications*, volume 5982 of *LNCS*, pages 416–419. Springer Berlin Heidelberg, 2010.
- [15] Oracle. SunSPOT Project. Available at <http://www.sunspotworld.com/>, 2004.
- [16] Wikipedia. Virtual machines — Wikipedia, the free encyclopedia, 2004. [Online; accessed 12-June-2014].
- [17] P. Levis and D. Culler. Maté: A Tiny Virtual Machine for Sensor Networks. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLoS X)*, pages 85–95. ACM Press, 2002.
- [18] R. Newton and M. Welsh. Region Streams: Functional Macroprogramming for Sensor Networks. In *First International Workshop on Data Management for Sensor Networks (DMSN'04), Toronto, Canada*, 2004.
- [19] D. Simon, C. Cifuentes, D. Cleal, J. Daniels, and D. White. Java on the Bare Metal of Wireless Sensor Devices – The Squawk Java Virtual Machine. In *Proceedings of VEE'06*. ACM Press, June 2006.
- [20] The TinyOS Documentation Project. Available at <http://www.tinyos.org>.

- [21] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. *Local Computer Networks, Annual IEEE Conference on*, 0:455–462, 2004.
- [22] C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava. A dynamic operating system for sensor nodes. In *Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services (MobiSys'05)*, pages 163–176, New York, NY, USA, 2005. ACM Press.
- [23] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, and R. Han. MANTIS OS: An Embedded Multithreaded Operating System for Wireless Micro Sensor Platforms. *ACM/Kluwer Mobile Networks & Applications (MONET), Special Issue on Wireless Sensor Networks*, 10(4):563–579, August 2005.
- [24] A. Eswaran, A. Rowe, and R. Rajkumar. Nano-RK: An Energy-Aware Resource-Centric Operating System for Sensor Networks. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS'05)*, December 2005.
- [25] L. Lopes, F. Martins, J. Barros. Programming Wireless Sensor Networks. In B. Garbinato, H. Miranda, L. Rodrigues, editor, *Middleware for Network Eccentric and Mobile Applications*, pages 25–41. Springer-Verlag, 2009.
- [26] World Wide Web Consortium. Web services glossary. Available at <http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/>, 2004.
- [27] T. Erl. *Service-oriented architecture: a field guide to integrating XML and web services*. Prentice Hall PTR, 2004.
- [28] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer. Simple Object Access Protocol (SOAP), 2000.
- [29] E. Christensen, F. Curbera, G. Meredith, S. Weerawarana, et al. Web Services Description Language (WSDL), 2001.
- [30] E. Cerami. *Web services essentials: distributed applications with XML-RPC, SOAP, UDDI & WSDL*. " O'Reilly Media, Inc.", 2002.
- [31] Wikipedia. Web service — Wikipedia, the free encyclopedia, 2004. [Online; accessed 12-June-2014].
- [32] MySQL. Available at <http://www.mysql.com/>. [Online; accessed 12-June-2014].

- [33] PostgreSQL. Available at <http://www.postgresql.org/>. [Online; accessed 12-June-2014].
- [34] Apache Cassandra. Available at <http://cassandra.apache.org/>. [Online; accessed 12-June-2014].
- [35] MongoDB Inc. MongoDB. Available at <http://www.mongodb.org/>. [Online; accessed 12-June-2014].
- [36] Amazon Simple Storage Service. Available at [aws.amazon.com/s3/](http://aws.amazon.com/s3/).
- [37] RXTX Project. Available at <http://mfizz.com/oss/rxtx-for-java>.
- [38] E. Neto, R. Mendes, and L. Lopes. An architecture for seamless configuration, deployment, and management of wireless sensor-actuator networks. In *SENSOR-NETS*, pages 73–81, 2014.
- [39] Clifton Nock. *Data access patterns: database interactions in object-oriented applications*. Addison-Wesley Boston, 2004.
- [40] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.
- [41] R. Silva, E. Neto, and L. Lopes. Towards out-of-the-box programming of wireless sensor-actuator networks. In *11th IEEE International Conference on Mobile Ad-hoc and Sensor Systems (IEEE MASS'14)*. (submitted).