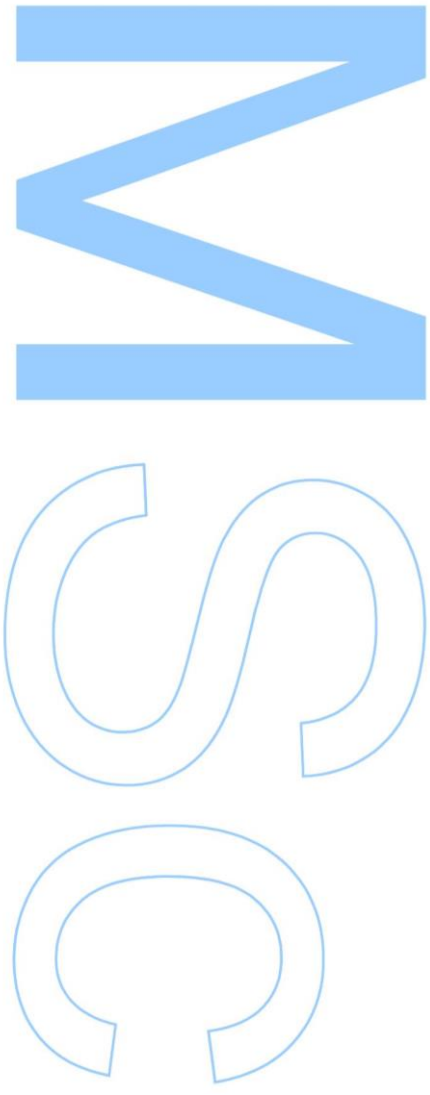
Towards Out-of-the-Box Wireless Sensor Networks

Gil de Castro Ferro
Mestrado Integrado em Engenharia de Redes e Sistemas Informáticos
Departamento de Ciência de Computadores
2015

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



U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

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

O Presidente do Júri,

Porto, _____/_____/_____

This thesis is dedicated to my beloved ones, for all the support and patience in all the most important moments of my life.

Gil de Castro Ferro

June 2015

Acknowledgments

I would like to thank my mentor, Professor Luís Lopes, for the opportunity of working with him in this project, giving me the opportunity to do research on one of my favorite fields of study.

A special thanks to my colleague Carlos Machado, for all the advice and expertise in some hardware issues.

To my family, for all the patience and support, and specially for the unconditional love.

Last but not least, to my friends, who shared most of my growth and happiness in this journey.

This work was supported by "Project RTS - Real Time Languages and Tools for Critical Real-Time Systems" (contract NORTE-07-0124-FEDER-000062)

Resumo

Programar Redes de Sensores-Atuadores Sem Fios (WSN) não é uma tarefa trivial, dada a variedade de configurações de *hardware* que são altamente dependentes da aplicação final. Além disso, os sistemas operativos e máquinas virtuais existentes, na maioria dos casos muito próximos do *hardware*, tornam esta tecnologia pouco apelativa e, por consequência, impedem a sua mais ampla disseminação .

Nesta tese apresentamos uma nova versão da arquitetura SONAR, que tem como objetivo minimizar o esforço necessário para configurar, programar e implantar uma WSN. Esta arquitetura fornece um serviço de `publish/subscribe` que pode ser utilizado pelos clientes para facilmente acederem às `streams` de dados geradas nos nós da rede, permitindo, ao mesmo tempo, a gestão da rede, incluindo a sua reprogramação dinâmica e `debug`. A arquitetura SONAR é composta por três camadas, que implementam uma interface de cliente ao estilo `shell`, um serviço de `broker` e um sistema operativo e máquina virtual que vêm pré-instalados nos nós da WSN.

Realizamos ainda um conjunto de testes que medem o impacto da nossa solução em termos de tempo, consumo de energia e memória nos nós da rede. Os resultados obtidos mostram que o impacto associado à utilização do nosso sistema operativo e máquina virtual é relativamente pequeno, para os parâmetros medidos.

Para testar a portabilidade da nossa camada de dados, portamos a nossa implementação de uma WSN baseada em Arduinos Mega2560 para outra baseada em Arduinos Uno. Estes últimos são significativamente mais restritos em termos de recursos e configurações de *hardware*. Esta experiência mostra que a quantidade de código que necessita de ser reescrito é muito pequena (apenas algumas linhas), sendo ainda de referir que essas alterações foram feitas em bibliotecas específicas para componentes de *hardware*, tendo-se mantido inalterado o código referente ao sistema operativo e à máquina virtual com exceção de algumas definições básicas de parâmetros.

Abstract

Programming Wireless Sensor-Actuator Networks (WSN) is a non-trivial task, given the multitude of hardware configurations that are highly dependent on the final application. Moreover, the existing operating systems and programming languages, mostly very close to the hardware, make the technology unappealing to the masses and therefore preclude its wider dissemination.

In this thesis we present a new version of the SONAR architecture, which aims to minimize the effort needed to configure, program and deploy a WSN. The architecture also provides a publish/subscribe service that can be used by clients to seamlessly access the data-streams generated by the sensing nodes while allowing, at the same time, the management of the network, including its dynamic reprogramming and debugging. SONAR is a three-layer architecture composed by a shell-like client interface, a broker service, and an operating system and virtual machine installed in the nodes of the WSN.

We perform a set of tests that measure the impact of our solution in terms of time, energy consumption and memory footprint on the devices. The results show that the resource footprint associated with our operating system and virtual machine is small in all the given parameters.

To test the portability of our data layer, we port the implementation from an Arduino Mega2560 based WSN to another one based on Arduino Uno devices. The latter are significantly more constrained in terms of resources and hardware configuration. This experiment shows that the amount of code re-write is very small (just a few lines) and these changes are done in libraries specific of hardware components. The operating system and virtual machine are untouched, except for the definition of basic constant parameters.

Acronyms

AJAX Asynchronous JavaScript and XML. 23

API Application Programming Interface. 9

DOM Document Object Model. 24

FRP Functional Reactive Programming. 8

GUI Graphical User Interface. 3

HTML HyperText Markup Language. 24

HTTP Hypertext Transfer Protocol. 14, 24, 25

IoT Internet of Things. 2, 11, 12

MQTT Message Queue Telemetry Transport. 13

MQTT-SN Message Queue Telemetry Transport for Sensor Networks. 13, 14

OS Operating System. 6, 9–13

QoS Quality of Service. 14

REST Representational State Transfer. 14

SONAR Sensor Observation aNd Actuation aRchitecture. 3–6, 19, 23, 24

STL SONAR Task Language. 4

STMP Spatio-Temporal Macroprogramming. 8

VM Virtual Machine. 6, 9, 10, 12, 13

WSN Wireless Sensor Network. X, 1–7, 9–14, 23

XSS Cross-Site Scripting. 24

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the past 25 years, advances in hardware manufacture and wireless communications provided the means to develop a new class of embedded devices, capable of interconnecting, sensing physical conditions and of interacting with the environment.

In this context WSN arise as a new paradigm of networks. A WSN can be defined as a distributed system composed of a varying number of embedded devices, usually called nodes, provided with a processing unit, a wireless communication interface, and a set of sensors/actuators, making these devices capable of sensing real physical environment or interacting with it [1].

Figure 1.1 depicts an example of a typical WSN scenario. A set of nodes with different sensors/actuators communicate via a radio interface, transmitting the sensed data to a *gateway* (data collection node), which is directly connected to computing resources. A set of clients can then access this data via a traditional network.

Given the characteristics of WSN nodes, the basic mode of operation of these networks differs significantly from traditional ones. In [2], the authors present three main differences:

1. Nodes are highly restricted in terms of energy, computational power and memory;

2. The design of a WSN is strongly driven by each particular application;

3. The deployment of WSN applications requires self-configuration of nodes and software updates in the network without human intervention.

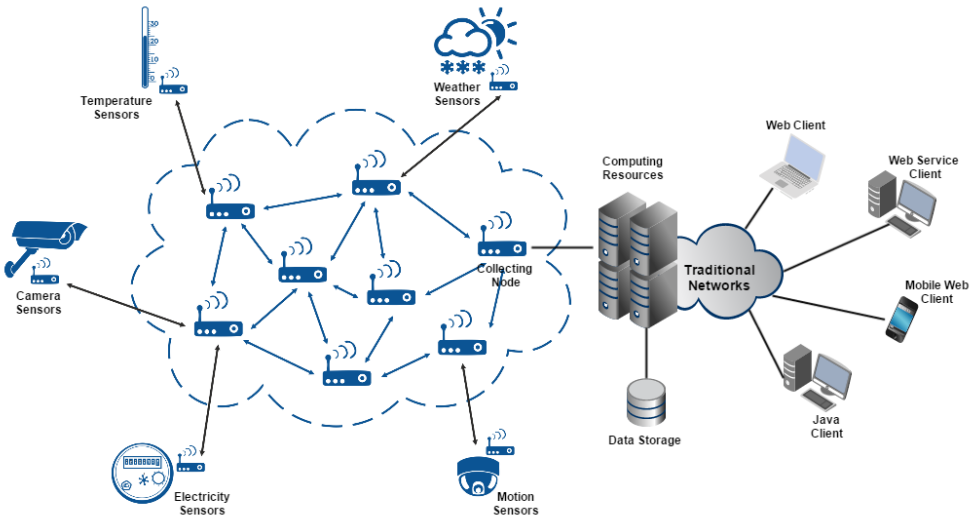


Figure 1.1: WSN typical scenario example.

The application areas of WSN goes from medical diagnosis, wildlife monitoring, traffic control, military systems, precision agriculture, among others [3]. Currently, there is a trend to increase the usage of WSN into new areas, alongside with development of the so-called Internet of Things (IoT) [4].

1.1 Context

Despite their wide range of applications, WSN are still rather cumbersome to use, especially by non-specialists. A few reasons can be identified for why this is so:

- The user needs a high level of expertise in configuring sensor devices, as well as low level programming;
- The wide offer of hardware platforms that can be used to deploy WSN contribute to a lack of portability of the applications;
- In most of the cases, the dynamic reprogramming or debugging of the deployment is a difficult task to do, impossible in some cases;
- The integration of a WSN with a traditional network is not trivial.

1.2 Motivation

Project Sensor Observation aNd Actuation aRchitecture (SONAR) [5, 6] purposes a three layer framework that allows a seamless deployment of a WSN. Figure 1.2 depicts the architecture used.

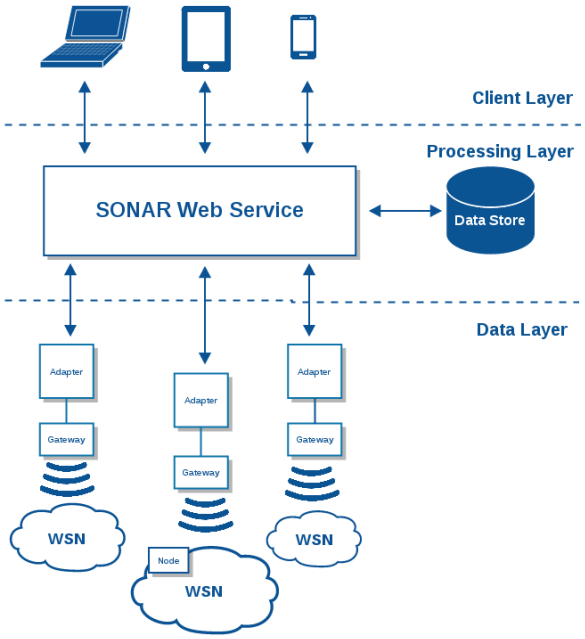


Figure 1.2: Previous architecture of SONAR

The Client Layer provides thin-clients implemented with a user-friendly Java Graphical User Interface (GUI) that provide hooks to access the Processing Layer to manage the deployments or to receive data streams. The management actions include: add/remove a periodic task to a deployment, change the period of a running task and view the data generated in a running task.

The Processing Layer is responsible for the reception and storage of the data generated in the Data Layer, as well as providing remote methods so the client can access this data. This layer also propagates management actions originating in the Client Layer to the underlying Data Layer.

The Data Layer is composed of a set of nodes and a single gateway (the initial implementation uses nodes based on Arduino Mega 2560 [7]). The nodes run a small operating system and virtual machine that schedule and run byte-code for tasks written in a domain-specific programming language, the SONAR Task Language (STL).

The initial SONAR architecture presented several limitations, first identified in [6]. These include:

- non-scalable storage of data generated by WSN deployments;
- complex management of pre-compiled modules required for different platforms; a hardware independent format is desirable;
- non-scalable, centralized processing of data in the Processing layer;

Some proposed solutions are forwarded in [6] but an implementation is lacking, together with an evaluation of performance in the time/energy axes.

1.3 Problem Statement and Proposed Solution

The development and integration of systems that make use of WSN is still a hard task to do, given the significant differences with traditional networks.

In this thesis we build on the work initiated in [6] and propose to answer the following relevant research questions:

1. *Is it possible to develop a framework for programming, configuring and deploying WSN in a seamless way, providing clients with the corresponding data-streams?*
2. *What is the impact of this approach on hardware/energy resources?*
3. *Can we do this in such a way that it is portable across WSN and allows dynamic reprogramming and debugging?*

To answer these questions, we divide the work done in this thesis in three main phases:

1. Rewrite the Client and Processing Layer of SONAR, implementing command-line-based thin-clients and a publish/subscribe broker;
2. Evaluate the impact of our approach in terms of computation overhead, energy consumption, memory usage, and code size, with respect to a native Arduino, C++-based solution;

3. Test the effort of porting our architecture to different hardware nodes.

With respect to the former implementation of SONAR, the work presented in this dissertation involved:

1. the redesign and implementation of the client and processing layers, namely publish/subscribe broker and clients, and an administration client;
2. rewriting part of the virtual machine installed in the nodes (data layer), and extensive testing;
3. significant changes to the SONAR Task Language (STL) compiler;
4. performance and resources usage evaluation of a prototype ported to a Arduino 2560 based WSN;
5. port the prototype to the Arduino Uno microcontroller.

1.4 Outline

The remainder of this thesis is organized as follows: Chapter 2 presents the state-of-the-art in software systems for WSN relevant for this work. Chapter 3 presents a high-level overview of the current SONAR architecture, detailing each layer and its components. Chapter 4 presents the technical aspects related to the modification of the architecture. Chapter 5 presents a formal description of the SONAR software that comes pre-installed in the nodes, depicting the programming language used to write new tasks, the operating system and virtual machine of these nodes, and the compiler we created to generate the byte-code representation of the written tasks. Chapter 6 presents the evaluation of the impact associated with our prototype, in terms of time, energy consumption, memory usage, and code size comparing to the same implementations using the Arduino native language. In Chapter 7 we describe the process of porting our prototype to Arduino Uno based nodes with greater memory constraints, focusing on the changes made, which we try to quantify. Finally, Chapter 8, presents a summary of the conclusions obtained in with this work, as well as a set of changes to be implemented as future work.

Chapter 2

Related Work

In this chapter we present a survey on the state-of-the-art relevant for this thesis.

Section 2.1 starts by analyzing the relevant work done in the field of Programming Languages for WSN. Section 2.2 describes projects that developed operating systems and virtual machines for WSN. Finally, Section 2.3 ends the chapter with existing publish/-subscribe systems for WSN. Along the chapter we try to establish a parallelism between our work and other approaches.

2.1 Programming Languages

Despite the advances observed in the last years, programming WSN applications still remains a hard task to do, given the wide range of existing programming languages, aside with the high heterogeneity of platforms. Most of the available languages work at low level and focus on the development of specific types of applications. With respect to the platforms, the configuration of the hardware is not a trivial task. Both these aspects introduce the need for expert users.

Currently, there is a considerable range of programming languages for WSN, each with a set of advantages/disadvantages in what concerns to the development of a specific application. Next, we present a brief description of three different languages, each one belonging to a different programming paradigm. To simplify the analysis, we define some taxonomy on the programming languages for WSN.

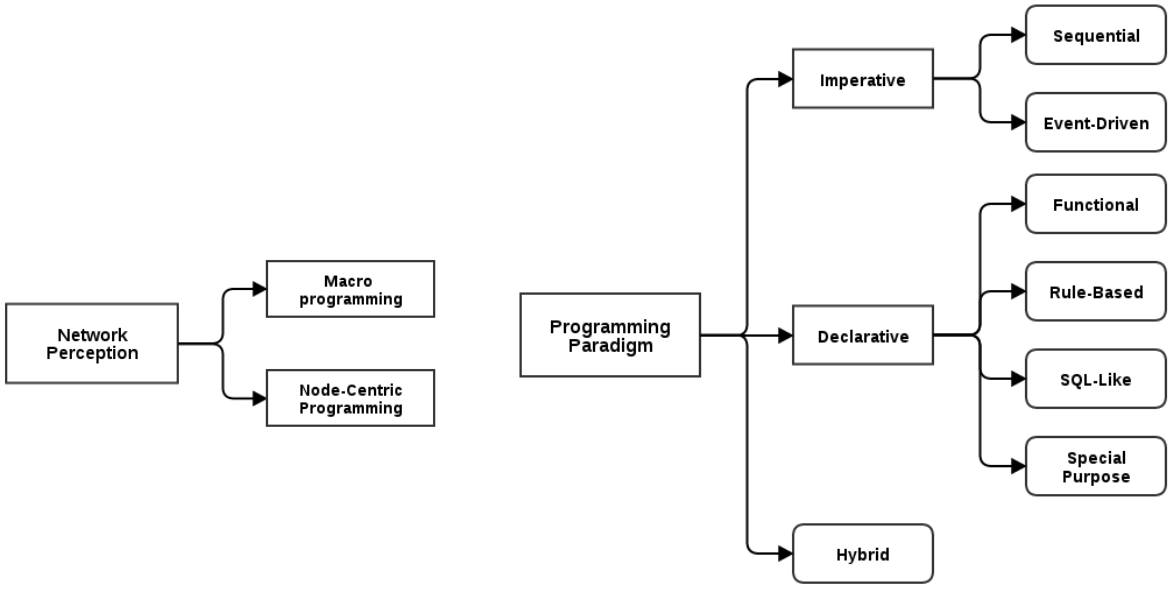
In [8], Gummadi et al. present a taxonomy that divides the programming languages in two main groups, according to the network perception: macroprogramming and node-centric programming. The former corresponds to an approach where application development is done without defining the behavior of each node individually, the latter corresponds to an approach where there is the need for the definition, and deployment of the behavior of each node.

A different taxonomy was purposed by Mottola et al. in [1]. They present an extensive survey on a set of different aspects related to programming WSN. In this article, they purposed a taxonomy related with the language aspects, divided in four categories: Communication, Computation Scope, Data Access Model, and Programming Paradigm. In the current context, we only analyze the Programming Paradigm dimension as it is the most relevant for this thesis when analyzing the following set of programming languages. According to Motolla's taxonomy, there are three main types of programming languages: Imperative, Declarative, and Hybrid. The first type corresponds to the languages where programs are described in terms of statements that change the program state, which can be divided in Sequential and Event-Driven languages; the second type corresponds to the the languages that expresses the logic of a program without describing the control flow, which can be divided in Functional, Rule-Based, SQL-Like and Special-Purpose; the third type corresponds to the programming languages that adopt a mixture of the two previous ones. Figure 2.1 presents a scheme describing the two taxonomies previously presented.

Next, we describe and categorize the set of programming languages that may be relevant in the current context of WSN, analyzing the network perception and the programming paradigm of each.

2.1.1 Programming Languages Overview

TinyDB [9] is a query processing system focused in the optimization of energy consumption, running in the top of TinyOS. It incorporates acquisitional techniques along with traditional query techniques, taking advantage from the fact that sensors possess control over when, where and how periodically the data is sensed and delivered to query processors. Madden et al. detail in [9] all the aspects related to the query language implemented in TinyDB, as well as the optimizations made in order to minimize the energy consumption, the query dissemination in the system and, finally, the model created for query execution and result collection.



(a) Network Perception (b) Programming Paradigms

Figure 2.1: Network Perception and Programming Paradigms

nesC [10] is an event-driven programming language that extends the C language, built atop of TinyOS. It is based in *components* that are assembled to form programs, as well as bidirectional *interfaces*, which specify the *components* behavior in terms of their *interfaces*. The *interfaces* specify the functions to be implemented by the interface's provider (*commands*) and by the user of the interface (*events*). nesC static links the *components* via their *interfaces*, increasing their runtime efficiency and robustness [11]. Gay el al. detail in [10] the design of nesC, as well as summary of their experience with it.

Regiment [12] is spatial macroprogramming language and runtime environment, with a compiler compiler that targets a lightweight intermediate representation called the Token Machine Language. It is based in the concept of Functional Reactive Programming, a programming paradigm that uses some building blocks of functional programming languages, like map, reduce, or filter. It was designed to support Spatio-Temporal Macroprogramming applications. In Regiment, the network is seen by the programmer as a set of spatial-distributed and time-varying signals, which represents the state of an individual node or region aggregate. Given this fact, the development of an application using Regiment is based in the usage of three main language concepts:

1. Signals, which are principal object the developer control;

2. Regions, which represent a collection of signals;

3. Nodes, which allows the developer to access the state of an individual node.

A Regiment program is translated to a node-level program using the language compiler. In this process, the program code is first reduced to an intermediate language called RQuery, which is finally translated to node-level code. In the process, the compiler performs many stages of node normalization, analysis and optimizations. Newton et al. present in [12] an overview of the Regiment language, describing in detail the respective compiler and deglobalization techniques, as well as an evaluation of performance of some event-detection in simulation.

Table 2.1 summarizes the information for the three programming languages presented that are representative of the state-of-the-art.

Prog. Language	**Network Perception**	**Prog. Paradigm**
TinyDB	Macroprogramming	Declarative, SQL-Like
nesC	Sensor-Based	Imperative, Sequential
Regiment	Macroprogramming	Declarative, Functional

Table 2.1: Programming Languages classification

2.2 Operating Systems and Virtual Machines

The restricted resources that characterizes WSN makes the usage of a traditional Operating System (OS) impracticable, given the fact that tradition OS are designed for devices with significantly more resources. These differences should be taken into account when developing a OS for nodes in a WSN.

There are a set of functionalities that an OS should provide and that include: resource abstractions for different hardware devices, interrupt management, task scheduling, concurrency control and networking support. Moreover, the OS should provide the application programmers high-level Application Programming Interface (API), independent of the underlying hardware [13].

Concurrently, virtual machines presented another important approach in developing WSN. Despite the great advantages of using OS, like performance optimization and the reduction

of energy consumption, the lack of interoperability and reprogramming of the network makes them not totally satisfactory. In this context, Virtual Machine (VM) allow a more flexible model for application development, eventually with some performance and energy efficiency penalties.

In the next two subsections we describe the most important works done in these areas, focusing in the main characteristics of each one. This part of the study will help understanding the current approaches that are being used when developing OS and VM for WSN.

2.2.1 Operating Systems Overview

TinyOS is one of the most used OS for WSN, which can nowadays be thought as a standard. It is a tiny multithreaded OS whose implementation tries to guarantee concurrent data flow among hardware devices, providing modularized components with a small processing and storage overhead. It follows an Event-based model designed to support high levels of concurrent applications in a small amount of memory, using a simple FIFO mechanism for task scheduling. Levis et al. [14] presents a complete description about the system where they analyze the system components, execution models and support for concurrency.

Contiki [15] is an open-source OS designed for networked embedded devices, implemented in the C language. A running instance of Contiki is composed for a kernel, a set of libraries, program loader and a set of processes. In Contiki all the communication between processes goes through the kernel, given the fact that it does not provide a hardware abstraction layer. However, it allows devices and applications to access the hardware directly. This OS allows preemptive multi-threading, implemented as a library on top of the event-based kernel, which can be optionally linked when implementing applications that require a multi-threading model of operation. Contiki's kernel is a lightweight event scheduler that dispatches events to running processes, with a periodical call to process polling handlers. One particularity of Contiki is the implementation of a lightweight uIP TCP/IP stack, allowing IPv4 and IPv6 addressing with a small footprint. Application development in Contiki is made using the C language and it allows an over-the-air programming of the entire network. Dunkel et al. [15] presents a full description of the Contiki, analyzing in detail the construction of the kernel and the preemptive multi-threading. The authors also analyze how Contiki handles the libraries and how it support communication, finishing the article presenting some bedtests using this OS.

LiteOS [16] is an OS for WSN, designed to provide a UNIX-like environment. It includes a hierarchical file system, wireless shell interface, kernel support for dynamic loading, native execution multithreaded applications, online debugging and a file system assisted communication stack. The system is composed of three main parts: LiteShell, LiteFS and the LiteOS Kernel. The former provides a UNIX-like command line interface to motes, running on the Base Station PC side; The second provides support both for files and for directory operations; the latter corresponds to a kernel design approach based on the usage of threads, which at the same time allows the user applications to handle events using callback functions. By default, LiteOS uses a priority-based scheduling scheme for task scheduling, but it is possible for the user to configure the system so it uses a round-robin scheduling scheme. Cao et al. [16] presents a detailed description about the design and implementation of LiteOS, its programming environment and some applications samples. They also present some evaluation results, as well as a brief description about some other features offered by LiteOS.

SOS [17] is an OS composed by a common kernel and a set of dynamic application modules, which can be loaded and unloaded at run time. It uses dynamic memory both in the kernel and the application modules, decreasing the complexity of writing tasks and increasing the temporal memory re-usage. The scheduling is based in a priority scheme, providing support for time-critical applications and, at the same time, moving the processing out of an interruption context. The implementation of an application for SOS is made using the standard C language, reducing the learning curve while taking benefits from the existing compilers, debuggers and analytic tools. C also provides an efficient execution, needed in a environment composed of hardware constrained devices. Han et al. [17] presents a detailed description of SOS architecture, focusing in a comparison with TinyOS. After that, some application samples are presented, ending with a system evaluation.

RIOT [18] is an OS which design and implementation was made focusing in the concept of the IoT, aiming to fulfill the gap between OS for WSN and traditional OS running on Internet hosts. It implements a modular microkernel architecture, with a minimal computational and memory overhead, inherited from FireKernel [19]. These characteristics makes it robust against bugs in single components. RIOT implementation supports multi-threading with a standard API and provides a TCP/IP network stack. The number of threads created is not limited by the OS itself, being only limited by the available memory in the device. Baccelli et al. [18] presents an analysis on the requirements of devices for the IoT, as a motivation to the development of RIOT. After that, they present a comparison on the existing OS, finishing with the description of RIOT characteristics.

2.2.2 Virtual Machines Overview

Squawk [20] is a Java micro edition VM specially designed for the usage in embedded devices. One of the main innovations of Squawk is the fact that almost all of the VM is written in Java, except for the Java interpreter and the binary bootloader. It provides OS level mechanisms for small devices, as well as easy to port applications and VM debugging. Given the fact that applications are written in Java, Squawk takes advantage from the fact that this programming language offers type safety, garbage collection and exception handling, contributing for a more robust and reliable application development. Applications running in this VM use Squawk byte-code, an optimized version of Java byte-code obtained by transformation. The optimizations made were essentially focused in minimizing space, in-place execution and to simplify the work of the garbage collector. Some optimizations were made in the core data structures used in order to save space. The execution model of Squawk allows the usage of green threads, which emulates multi-threaded environments without relying on any OS capabilities. Simon et al. [20] presents an extensive description of Squawk, detailing the internal implementation of this VM. They also present a set of examples on programming Sun SPOTs and some experimental results.

Maté [21] is a VM based in a byte-code interpreted that runs on TinyOS. Applications are implemented using a low level language in which transmission is based on the usage of capsules which containing up to 24 instructions, where an instruction is stored in a single byte. The limit of instructions in a capsule corresponds to the maximum number of instructions that fit in a single TinyOS packet. A complete program can be composed by more than one capsule, that can forwards themselves through the network with a single instruction. Maté uses two stacks: an operand stack and a second one corresponding to the address stack. It provides various ways of routing capsules: a built-in ad-hoc routing algorithm which is used by default and a set of mechanisms available for writing new routing algorithms. Given the hardware constrains in WSN, Maté was designed to run with the smallest hardware requirements possible. The size of Maté itself and all his subcomponents fit in 1kb of RAM and 16kb of instruction memory. Mueller et al. [21] presents a detailed description of Maté design, structure and implementation, as well as a brief description of the underlying OS, TinyOS. They also present some analysis of behavior and performance of the VM, discussing their own evaluation of Maté.

TinyReef [22] is a register-based VM for WSN. It runs on top of TinyOS and was designed with the premise that registered-based VM provides some advantages related with the fact that it requires less instructions to implement a task, e.g. smaller programs and

less update costs, which can compensate the offset introduced compared with stack-based VM. The architecture is divided in five main parts: a Program State, a Loader, an interpreter, a Instruction Set and an Event-Handler. The implementation of a program is based in a low-level approach, where programs are divided in two main parts: a Data segment, which stores all the static variables used and a Code segment, which contains the program instructions. Marques et al. [22] a description on the implementation of TinyReef, describing the main challenges of using VM in constrained devices and showing some basic samples of programs written for TinyReef.

VM* [23] is software system based in the usage of a VM and a OS, whose implementation was made with three main challenges in mind: 1) High heterogeneity of end systems in WSN; 2) Need of dynamic updates in deployed software; and 3) Supply of a rich programming interface while respecting the resource constrains in end devices. Programs are written in Java, with access to the I/O and sensors via native interfaces. The architecture of VM* includes six main parts: a component language for representing system software components, a set of tools for analyzing and compacting Java classes, a component based OS, a component based implementation of a subset of the VM, a set of tools for synthesizing the VM and the underlying OS and an incremental linker to add features to the system. Koshy et al. [23] a conceptual overview and implementation details of VM*, ending with some evaluation results obtained in tests.

2.3 Publish/Subscribe Systems

Message Queue Telemetry Transport for Sensor Networks (MQTT-SN) [24] is an extension of the open publish/subscribe protocol Message Queue Telemetry Transport (MQTT) [25], developed to use in the top of TCP/IP protocol. Originally it was called MQTT-S, with the S commonly confused with Security. It follows the design concept of MQTT, focusing in allowing operations on low-cost and low-power sensor-actuators devices, most of the integrated in non-TCP/IP networks. The main usage of this protocol is to provide a simple and scalable communication mean while allowing a seamless integration of the WSN into the traditional networks. In a MQTT-SN system, the running applications and devices can be both Publisher and Subscribers. The published message always passes in the Broker, even if they both reside in the same network. The construction of the widespread topics is based in a hierarchical scheme, e.g. wsn/sensors/room183/node1. It supports the usage of three levels of Quality of Service (QoS), not totally working when the main

article about this work was released. The communication between the devices/clients inside the WSN with the traditional network is made through the Broker, passing first for the network Gateway. This protocol allows more than one running gateway, providing more robustness. Hunkeler et al. [24] presents a full description of the MQTT-SN, as well as some performance bedtests.

Mires [26] is a Publish/Subscribe middleware for WSN. The design and implementation of this middleware was made in order to encapsulate the network, providing a seamless access to the available nodes through a Publish/Subscribe architecture and facilitating the development of applications to WSN. The implementation was made atop of TinyOS. In Mires, the Publish/Subscribe mechanism is composed of three main phases: 1) a running node, which wants to publish, advertise its available topics; 2) when data is available to be published, the node routes that data to the sink node; 3) after verifying that the received message has at least one subscriber, the sink node broadcasts that message down to the network nodes. Souto et al. [26] presents a full description of this Publish/Subscribe middleware, as well as environmental monitoring application used to validate their work.

TinyREST [27] is a Representational State Transfer (REST) middleware developed to allow the exchange of information between an WSN and the Internet, using an Hypertext Transfer Protocol (HTTP)-like approach. This system was built using MICAz [28] nodes, which runs TinyOS. TinyREST uses a multi-threaded lightweight HTTP-2-TinyREST gateway which is responsible to actuate as an interface between the sensors/actuators and the clients. All the interactions made with the network are based in three methods: GET, POST and SUBSCRIBE, which allows the users to, respectively, read the value of a sensor in a specific node (e.g. GET /gatewayIP/dinningroom/temperature), change the current value of a sensor/actuator (e.g. POST /gatewayIP/dinningroom/heater/on) and subscribe for an event of interest (e.g. SUBSCRIBE /gatewayIP/entrance/motion-sensor/change). Additionally, all of the three methods used to interact with the network can be forwarded by the gateway to a set of nodes, or broadcasted to the entire network, using an addressing mechanism. Luckenbach et al. [27] presents a full description of the implementation of TinyREST, as well as some robustness, data loss and communication delay bedtests made using the MICAz nodes.

MufFIN [29] is a generic middleware framework that allows for managing and programming Internet of Things smart objects and to provide the resulting data-streams through publish-subscribe web-services. The framework allows web clients to install code modules (filters) directly into the devices or, when this is not possible, in the middleware side. These

modules are organized in a chain of dependencies called a Data-Flow Network and allow the processing of data gathered by the back-end WSN. The data streams hence produced are provided in publish-subscribe web-services.

2.4 Summary

In this chapter we present an overview on the related work which may be relevant for this thesis. We presented an overview on the different programming languages paradigms, specifying a relevant language of each type. Later, we analyzed the most significant operating systems and virtual machines developed to run in WSN. In the last section of this chapter we analyzed a set of systems who implement a publish/subscribe architecture, highlighting some particularities in each one.

Chapter 3

SONAR Architecture

In this chapter, we present the architecture of SONAR, describing the components in each layer. The chapter is organized as follows: in Section 3.1, we start by presenting a typical application scenario where SONAR can be used; Section 3.2 presents a simple overview of the architecture, enumerating its layers and components. Sections 3.3, 3.4, and 3.5 describe, respectively, the Client layer, the Broker Layer, and the Data Layer.

3.1 Application Scenario

To clarify the main idea of SONAR, we present a typical application scenario of a WSN. Imagine a person who owns a greenhouse and wants to implement a system to monitor environmental variables and automate actions if certain conditions arise. The owner would buy a kit composed by a set of nodes, a gateway node, and a software pack to install in a common computer.

Each node can have a set of sensors, for example, temperature, humidity, and light sensors, and a set of pins available to connect actuators, for example a window vent, a sprinkler and a greenhouse awning motor. After installing the software, the owner places each node in a strategic place and connect the actuators to the pins in the nodes. At this point, he is able to connect the gateway node to the computer USB port and use the WSN. When the gateway is connected to the computer, it sends a message to the software who parses it and registers itself in the Broker, making the data produced in the deployment available through this component.

To manage the WSN the owner uses a simple shell interface that is also supplied with the software. He can now list all the running tasks, set the period of a task, and add or remove a task in the network. These tasks are written using a very simple specific domain programming language, STL, which we address later in this thesis.

To monitor the data produced at his WSN, the owner simply connects to the Broker of the publish/subscribe system. It then selects the data streams it is interested in from the WSN. This can be done for multiple WSN (e.g., more greenhouses, the garden, the house), and for many tasks running on the nodes of a WSN (e.g., temperature and humidity, movement detection, luminosity), providing a multiplicity of data streams that can be subscribed by users.

This is the level of seamlessness that we aim to address with our architecture, SONAR. In the following sections we describe the components of each layer of the architecture.

3.2 Architecture Overview

SONAR follows a typical 3 layer architecture, depicted in figure 3.1. It is based on a publish/subscribe architecture where a set of clients, connected to the Internet, access the data generated at each SONAR deployment through the SONAR Broker.

The data layer is composed by a set of nodes that come with a pre-installed operating system and virtual machine, and a gateway, that collects the data produced in the nodes. Each node can schedule and run multiple tasks. A task can be described as a program that periodically runs in the nodes, with no interruptions, and that generally produces a data-stream. The gateway acts as a simple forwarder, exchanging data from the nodes with the other layers.

A client, when connected to the Broker, can access a list containing information about the registered deployments. For each deployment, it is presented the set of running tasks, with the following parameters: task description, task period, and type of generated data. The client can subscribe the desired data streams and receives the respective data produced in the data layer.

The management of the each deployment is made using an administration client that connects directly to a deployment adapter. With this client, a user is able to manage his deployment, adding or removing tasks, or changing the period of a running task.

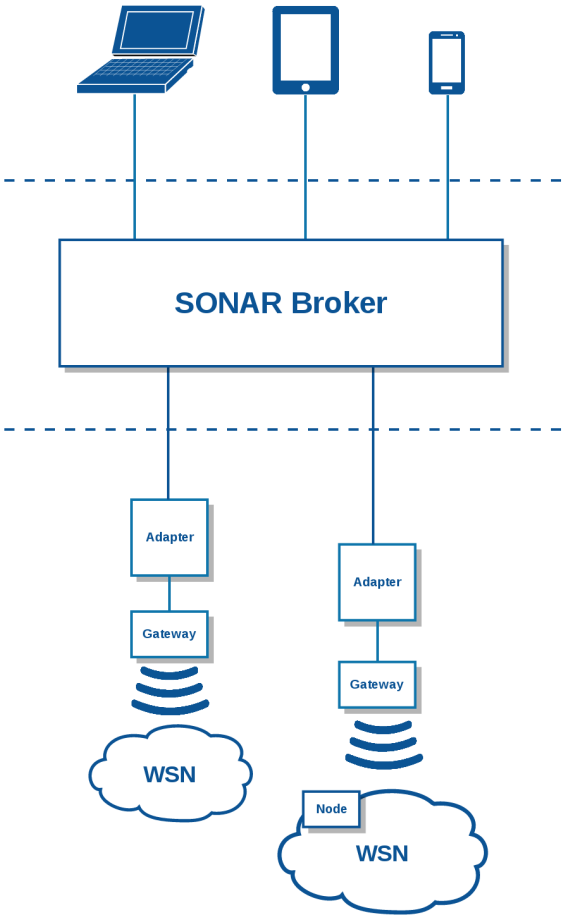


Figure 3.1: SONAR architecture.

3.3 Client Layer

The Client layer is composed by two different modules included in the software the user must install in his computer:

- a Publish/Subscribe Client, used to connect to a SONAR Broker (middle layer), allowing the user to list and subscribe the available deployments, receiving the data produced after the subscription;

- an Administration Client that allows authenticated users to access a deployment, through a component called Adapter, and to manage it, sending control messages.

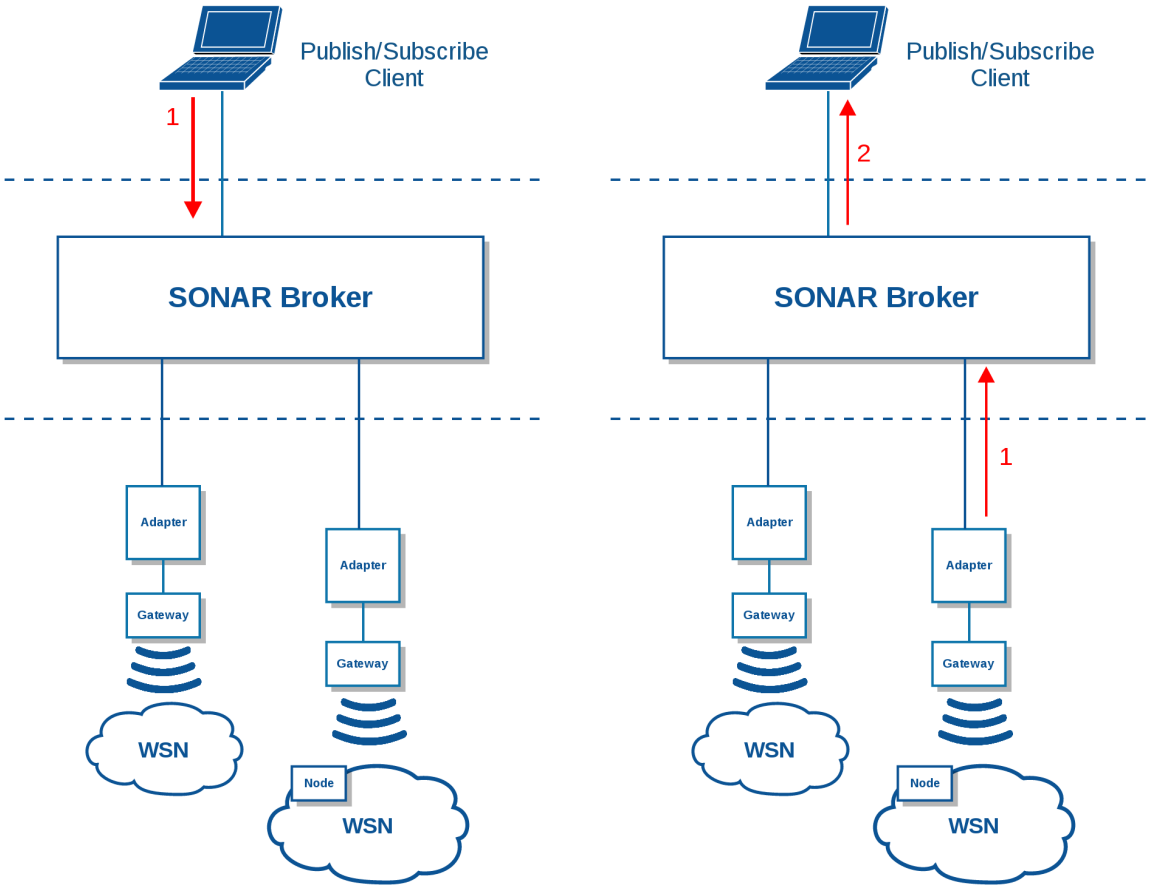
3.3.1 Publish/Subscribe Client

The Publish/Subscribe Client is a shell-based interface where a user can access the methods to list and subscribe a data stream, available in a SONAR Broker. Using this module, a user can send data messages, allowing him to list all the available deployments and respectively running tasks, subscribe and unsubscribe, tasks and query the Broker for a specific type of data being produced in all the available deployments. Figures 3.2a and 3.2b depict, respectively, the data flow of listing or subscribing a task and the data flow of publishing data generated at the Data layer to a client. To subscribe a data stream, the user sends a request to the broker indicating the ID of the desired data stream. The Broker parses that request and then responds to the client with a message confirming a successful subscription. When new data is available in the data layer, the respective adapter forwards that data to the Broker, who checks the clients subscribing these data-streams. It then sends the received data to each subscriber.

3.3.2 Administration Client

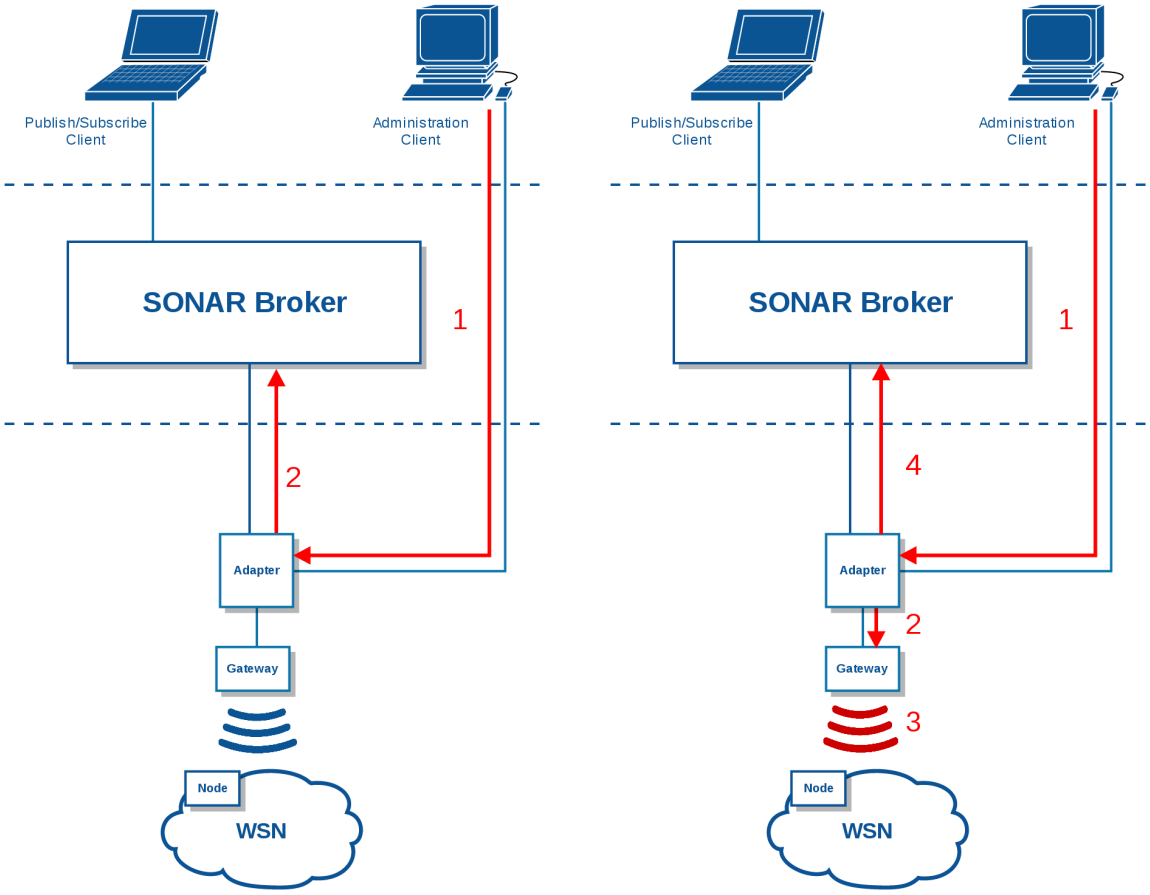
The Administration Client is the component used by the users to administer deployments, namely, to register and unregister it own deployment in a Broker, as well as to manage it, allowing him to add, remove or change the period of a running task. Figures 3.3a and 3.3b depict, respectively, the data flow in the process of register/unregister the deployment and the data flow when adding, removing, or changing the period of a task.

To register a deployment, the user sends a control message to the Adapter, who sends a register request to the Broker containing the MAC Address of the gateway. The Broker processes the request and response with a confirmation that the deployment had been registered. To manage his own deployment, the user sends a message control message to the Adapter containing the management command, which is then forwarded to the deployment gateway node. After receiving the command, the gateway radio the command to all the nodes in the deployment.



(a) Data flow of listing and subscribing a task (b) Data flow of publish mechanism

Figure 3.2: Data flow in the Publish/Subscribe Client



(a) Data flow in a deployment register

(b) Data flow in tasks management

Figure 3.3: Data flow in the Administration Client

3.4 Broker Layer

The Broker is the middle layer of our architecture, which maintains a connection with one (or more) SONAR Adapters, as well as with a set of SONAR Clients. This component is responsible for the three main tasks: registering deployments, publishing data provided by the data layer, and handling subscription requests from clients. To allow these functionalities, the Broker stores three main structures:

- a **tasks structure** that stores all the parameters of one data streams registered in the Broker. Each task representation is composed by four parameters: an ID that identifies inequivoquely the task; a period that indicates the periodicity of the task; a values description that presents the data produced in that task; the units description that presents the units of each value produced; and, the task info, an optional field used to store some extra information about the task.
- a **deployments table** that stores all the registered deployments. For each deployment, this table maps a pair of parameters: an general information field (optional), containing some generalist information about the deployment and a list of tasks, containing the ID of each task running in that deployment.
- a **subscribers table** that, for each available data stream, maps a list of subscribers IDs, identifying the clients subscribing that data stream.

3.4.1 Data Flow in SONAR Broker

There are two types of data flows related with the Broker: the first is related with the messages sent by the SONAR Clients, when listing, subscribing or unsubscribing a data stream; the second is related with the data produced at the data layer, which is received from the Adapter to be forwarded to the respective subscribers.

Client layer $\leftrightarrow$ Broker messages Figures 3.4 and 3.5 describe the messages flow exchanged between the Client and the Broker when the Client is subscribing a data stream.

The Client starts by listing all the available data stream. Figure 3.4 depicts the data flow involved

1. the client send a message to the Broker asking for all the available data streams;

2. the Broker receives the message, queries its deployments table and produces a message containing a summary with the available deployments and the respective data streams.

3. The Broker responds to the client with the message containing the parameters needed so that the Client can subscribe the desired tasks;

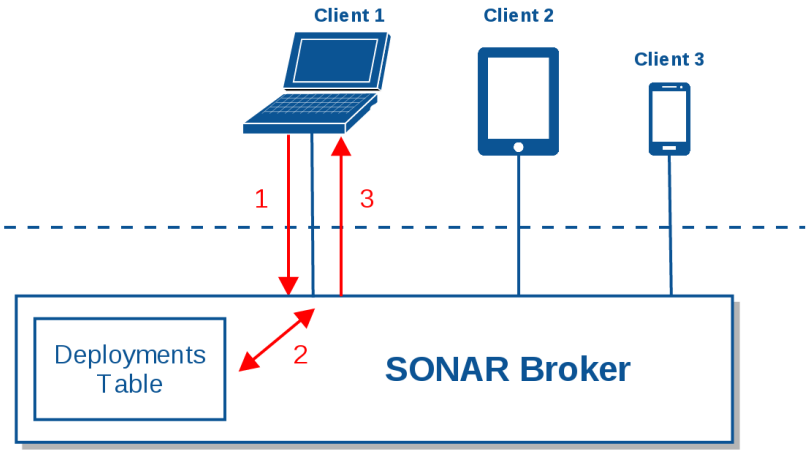


Figure 3.4: Available data streams

After receiving a list of the available data streams, the client is able to subscribe them (Figure 3.5):

1. The Client sends a message to the Broker containing the IDs of the data streams to be subscribed;

2. the Broker receives that message and add the client ID to the entry of the subscribers table;

3. the Broker responds to the Client, confirming that he is now subscribing that data streams.

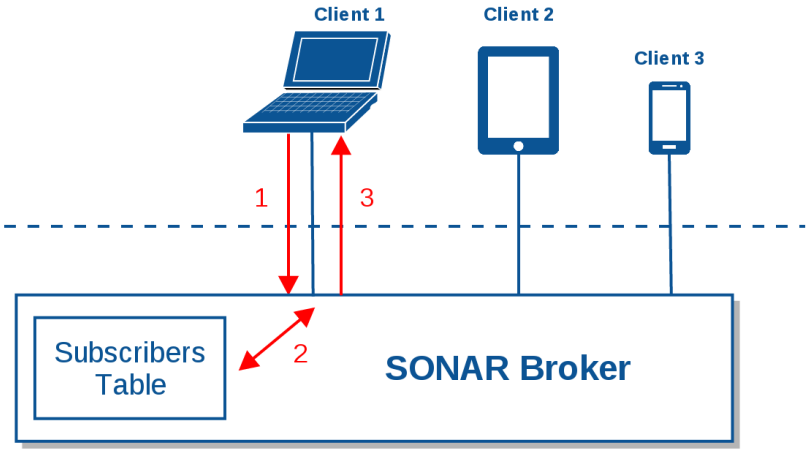


Figure 3.5: Subscribing/Unsubcribing a data stream

Data layer ↔ Broker messages Figure 3.6 describes the mechanism for publishing a message. When the Broker receives a message containing Data from one Adapter, it parses the message and retrieves two parameters: the identifier of the deployment where the data was generated and the identifier of the task who generated the data. Using that information, it queries its Subscribers Table and retrieve the identifiers of the Clients who are subscribing that task. Using these identifiers, the Broker forwards the messages to the respective clients.

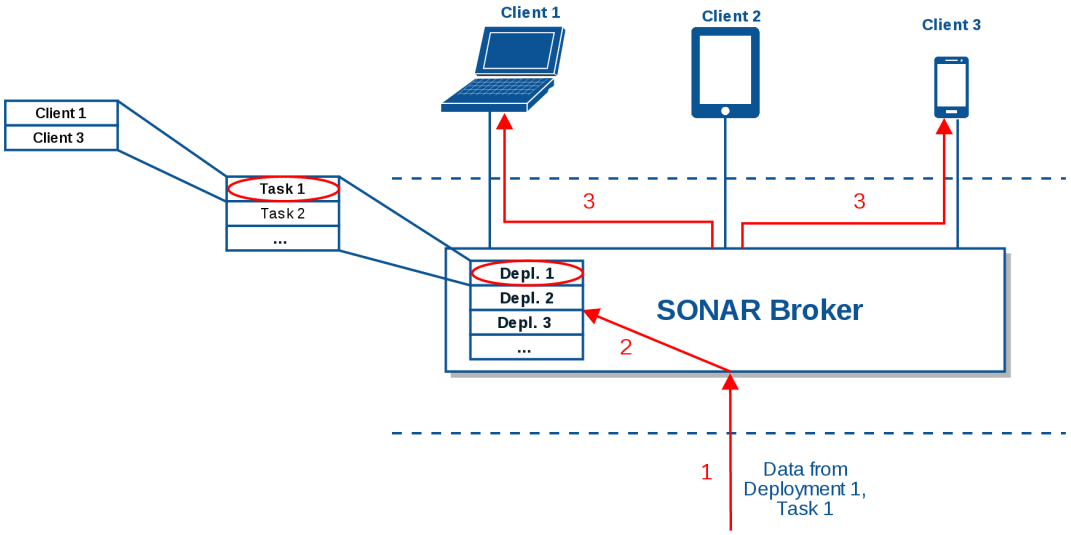


Figure 3.6: Data delivery to subscriber of a given task.

3.5 Data Layer

The Data layer abstract all the sensors and actuators present in each deployment. It is composed by three different components:

- SONAR Adapter;
- gateway nodes;
- a mesh of Nodes.

3.5.1 Adapter

The Adapter is a software component that establishes connections with other two components: the Administration Client and the Gateway. It is responsible for two main actions:

1. **Administration**: allows users to administrate the aspects related to his SONAR Wireless Sensor Network deployment using the Administration Client. This connection is only active when the administrator is using the Administration Client.
2. **Forwarding**: gathers coming messages from the Gateway with the data generated in each node and forward it to the Broker. This connection is always active when the deployment is running.

Figure 3.7 depicts the data flow associated with the forwarding function of the Adapter. Red arrows and radio signals indicate the path done by the data generated at the nodes. For each running task that produces data, each node sends that data to the deployment Gateway, who analyze the message and add parameters to unequivocally identify the origin of the message. After that, it simply forward the message to the adapter, who then forward it to the Broker.

3.5.2 Gateway and Nodes

The Gateway is a typical node equipped with a radio device capable of receiving and forwarding tasks to a previously configured set of nodes. It is responsible to radio the

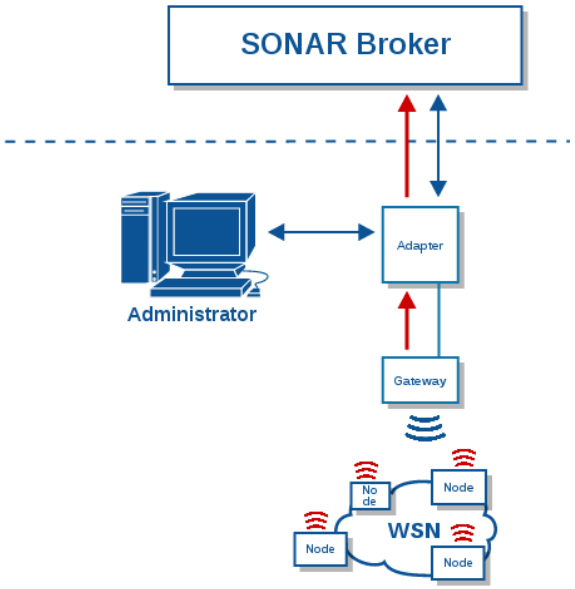
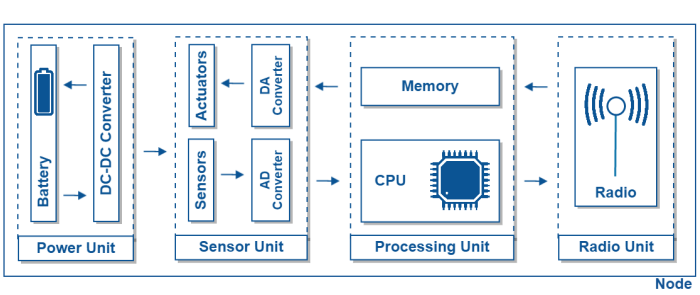


Figure 3.7: Data flow generated in nodes

commands and tasks sent by the administrator to all the nodes, as well as receiving the data produced in the nodes and forwarding it to the adapter. Figure 3.8a depicts a high level hardware representation of a typical node and Figure 3.8b presents a picture of a SONAR Gateway.



(a) Typical node scheme

(b) SONAR Gateway Sensor.

Figure 3.8: Typical node scheme and SONAR Gateway node

The nodes used in SONAR follow the same hardware configuration as the gateway, with the addition of some sensors and actuators. Currently, they contain temperature, humidity, and light sensors, as well as a LED functioning as an actuator.

In the next chapter we are going to formally describe the software installed in the gateway

and on the nodes, starting with the description of STL, and the node's operating system and virtual machine.

3.6 Summary

In this chapter we presented an high-level overview of the SONAR architecture, describing each layer and its components. We also described the control message and data flow message in the system.

Chapter 4

Client Layer and Broker Layer

In this chapter we describe the implementation of the SONAR Publish/Subscribe mechanism, focusing on the implementation of the Client and Broker layer. To simplify the analysis of the architecture, we will first describe the Broker Layer and its implementation, which is relevant to understand the implementation of the Client layer.

4.1 Broker Implementation

For the implementation of this Broker we used a simple Java Web Service with two terminals (Server Endpoints), one reserved for the client connections and the other reserved for the connections with SONAR Adapters. There are numerous ways to create communication channels between the web service and a client, each one with different advantages and disadvantages.

4.1.1 Communication model

In the process of implementing a Publish/Subscribe system, there is a fundamental question that significantly affects the communication model: the communication between the server and the clients is, in most of the cases, started by the server, a mechanism called "server push". In the traditional paradigm of communication between a server and its clients, all the connections are started by the client, which sends a request to the server and waits for the server to respond. After receiving the response, nothing happens until a new request is

sent to the server. In cases where a server is generating (or gathering from a service) data that it needs to send to some clients, this paradigm of communication is not applicable.

This is a well-known problem and there are a set of solutions to surpass this limitation. One solution that has been widely used is a mechanism called *long pooling* [30]. It is based on the usage of Asynchronous JavaScript and XML (AJAX) and relies in a client-side routine being constantly called, sending requests to the server asking if new data is available.

With the fifth revision of HyperText Markup Language (HTML), two new solutions where introduced as standards for this problem: Server-Sent Events (SSE) and WebSockets. Both solutions present a better performance that the *long pooling* mechanism, given the fact that none of these are constantly hitting the server with requests to verify if new data is available. Given that, we decided to use one of these mechanisms (initially, the Server-Sent Events; later, we changed to the WebSockets) to implement the Publish/Subscribe component of the SONAR architecture, avoiding the *long polling*. Next, we describe each of these mechanisms, analyzing the advantages and disadvantages of each one.

Server-Sent Events Server-Sent Events [31] were introduced in HTML5 specifications as a mechanism to allow a server to push notifications through an HTTP connection to a set of clients. These notifications are sent in the form of Document Object Model (DOM) events. Using this mechanism it is possible to implement a Publish/Subscribe service, where a client starts a connection to a server, chooses the desired data streams, and subscribes them. The complete process of sending data to the client is made using the pending request made by the client. Table 4.1 present the advantages and disadvantages of using SSE.

Advantages	Disadvantages
Transported over simple HTTP instead of a custom protocol;	Unidirectional communication channel (server to client only);
Built in support for timeouts and re-connection;	Browser support is more limited;
Simpler protocol.	Relies on client to verify origin (possibly more vulnerable to Cross-Site Scripting (XSS) attacks).

Table 4.1: Server-Sent Events - advantages and disadvantages

4.1.1.1 Web Sockets

Web Sockets are currently a part of HTML5 specification [32] as a protocol for a two-way communication between a server and a remote host client, via a full-duplex channel. To establish a WebSocket connection, the client sends a handshake request to the server, who responds with an ACK. Thereafter, the server leaves the connection open so that if an event occurs (in the case of SONAR, new data has been received from the Data layer), it can be sent out immediately; otherwise, the event would have to be queued for transmission until the client sends a new request to the server. Using this channel which is now open, both client and server are able to send data anytime it is needed.

Table 4.2 presents the main advantages and disadvantages of WebSockets.

Advantages	Disadvantages
Real time, full-duplex communication;	Protocol more complex than SSE;
Native support in most of the browsers, as an HTML5 standard;	Browsers must be fully HTML5 complaint;
Server can send data from multiple events subscribed by one client using a single connection;	Timeouts and re-connection must be manually managed.

Table 4.2: WebSockets - advantages and disadvantages comparison

4.1.2 Chosen Mechanism

After analyzing the advantages and disadvantages of using each of the above mechanisms to implement a Publish/Subscribe, we decided in first hand to use the Server-Sent Events. This decision arises from the fact that we only need to push messages in one way (from the web service to the client), coupled with the fact that Server Sent Events are a simpler protocol, which uses HTTP as the transport layer.

The first implementation tests with Server-Sent Events allowed us to send message from the web service to the connected clients, defining an EndPoint where the messages are generated. Although we can successfully push messages generated at the web service to the desired clients, it became clear that an increasing number of subscriptions by a client will possibly result in a worst performance, given the fact that each subscription is a pending request from the client to the server. At the same time, we decided that a shell-based client could be an interesting feature in SONAR, allowing the user to interact with

the project without a web browser.

Given these facts, we decided to abandon the implementation using Server Sent Events to a new implementation using Web Sockets. WebSockets allowed us to create a two-way communication channel where we can easily implement a simple shell client. Using that channel, given the fact that all the data is pushed by the web service, we can make use of the same channel to send data gathered from the different subscriptions made by a client.

4.2 Client Layer

Both the Publish/Subscribe Client and the Administration Client are implemented as Java Web Services, using Web Sockets Clients to communicate with the Broker and the Adapter, respectively. The usage of a Web Sockets allows the access to the Broker and to the Adapter anywhere using a common terminal with Internet access.

Both Client interfaces are implemented as a shell. Tables 4.3 and 4.4 present the available commands in the Publish/Subscribe Client and in the Administration Client.

Command	Description
`sonar ld`	List the available Deployments and Tasks
`sonar sub -t ID_TASK -d DEP`	Subscribe Task ID_TASK running in Deployment DEP
`sonar unsub -t ID_TASK -d DEP`	Unsubscribe Task DEP running in Deployment ID_TASK
`sonar find -t DATATYPE`	Query the Broker for tasks producing data with type DATATYPE
`help`	Show a list containing the available commands

Table 4.3: Commands available in the user interface.

The main advantage of this type of interfaces is that it allows the Publish/Subscribe Client to simply pipe a set of commands in order to treat the received data from a subscribed task. As an example, a client who owns a database server or an account in an online storage service like Amazon S3 [33] can simply write a small script that waits for data to be received in the standard input and sends that data to one of these databases. With a simple `pipe`, the client can chain the subscription command to a script, say `sendToDB.py` (Figure 4.2.1), receiving the data and storing it in the database. The command used for subscribing the task with ID 1 from deployment 1 and passing the data to `sendToDB.py`

Command	Description
`reg -a MAC`	Register the Deployment in the Broker.
`unreg -a MAC`	Unregister the Deployment in the Broker.
`list`	List the running Tasks
`task -p PER -b BC_PATH -d DESC_PATH`	Add a new task with period PER, byte-code file BC_PATH, and description file DESC_PATH;
`period -t ID_TASK -p PER`	Change the period of task ID_TASK to PER.
`kill -t ID_TASK`	Remove task ID_TASK from the runing tasks pool.
`reset`	Remove the tasks from the running tasks pool.

Table 4.4: Commands available in the administration client interface.

script would be:

```
$$ sonar sub -t 1 -d 1 | sendToDB.py
```

A simpler example where the user stores the raw data received in a text file.

```
$$ sonar sub -t 1 -d 1 > dump_t1_d1.txt
```

4.3 Summary

In this chapter we present the implementation of the Broker layer and Client layer components. We explained the significant technological choices made when implementing each of the components, analyzing some existing approaches. Next, we explained the decision of using a shell-like interface in both clients, presenting the list of commands available in both clients. We present a simple example of some chained command used to store the received data from a client subscription in a MySQL database.

Python Script Code 4.2.1 Python Script to insert data received in the STDIN to a MySQL Database

```
Import MySQLdb
connection = MySQLdb.connect(
                  host= "localhost",
                  user= "root",
                  passwd= "rootpasswd",
                  db= "room1.83_dep")
cursor = conn.cursor()

while true:
   data = raw_input().split(" ")
   deployment = data[0]
   task_id    = data[1]
   mac_sensor = data[2]
   data_type  = data[3]
   value      = data[4]

   try:
      cursor.execute("''INSERT INTO data VALUES (%s,%s,%s,%s,%s)"'',
                                                (deployment,
                                                task_id,
                                                mac_sensor,
                                                data_type,
                                                value))
   except:
      conn.rollback()

   conn.commit()
conn.close()
```

Chapter 5

Data Layer

In this chapter we present the specification and implementation of the SONAR data layer. This software runs in the gateway and nodes of the deployments and is pre-installed. It includes an operating system, a domain-specific programming language, and a virtual machine.

In Section 5.1 we describe the programming language used to implement tasks in SONAR, SONAR Task Language (STL). Thereafter, in Section 5.2 we describe the SONAR Virtual Machine and the compiler used to produce the byte-code executed in the nodes. Section 5.3 describes the operating system running both in the nodes and in the gateway. This chapter ends with Section 5.4 that describes the data flow associated with each node.

5.1 Programming Language

In this section we describe the syntax and semantics of the domain-specific programming language used to implement periodic tasks - the SONAR Task Language (STL).

5.1.1 Syntax

The syntax for tasks is described in Figure 5.1. The notation $\tilde{\alpha}$ is used to denote a sequence of pairwise distinct elements, α, of a given syntatic cathegory. A task T uses two sets of identifiers, s and a, to specify the available sensors and actuators in a given platform. Each

$$
\begin{array}{lll}
T & ::= & \textbf{sensors } \{s_1 : \sigma_1 \ldots s_n : \sigma_n\} \quad \textit{Tasks} \\
 & & \textbf{actuators } \{a_1 : \sigma_1 \ldots a_m : \sigma_m\} \\
 & & \textbf{init } \{\tilde{q}\} \\
 & & [\tilde{\tau}] \textbf{ loop } \{\tilde{r}\} \\
\sigma & ::= & \tilde{\tau} \mapsto \tau \quad \textit{Types} \\
\tau & ::= & \textbf{bool} \mid \textbf{int} \mid \textbf{float} \mid \textbf{void} \\
q & ::= & \tau\ x = v \quad \textit{Initializations} \\
r & ::= & x = e \quad \textit{Instructions} \\
 & \mid & a(\tilde{e}) \\
 & \mid & \textbf{radio } [\tilde{e}] \\
 & \mid & \textbf{if } e\ \{\tilde{r}\} \textbf{ else } \{\tilde{r}\} \\
 & \mid & \textbf{while } e\ \{\tilde{r}\} \\
e & ::= & s(\tilde{e}) \mid e\ op\ e \mid op\ e \mid (e) \mid v \quad \textit{Expressions} \\
v & ::= & x \mid u \quad \textit{Values} \\
u & ::= & \textit{bools} \mid \textit{ints} \mid \textit{floats} \quad \textit{Constants}
\end{array}
$$

Figure 5.1: The syntax of STL.

of these identifiers maps to a unique sensor or actuator in the hardware. 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 with the **init** block, used to initialize global task variables. This code is not executed, rather the compiler copies the initial values for each variable directly to the data segment of the byte-code generated for the program. The **loop** block, on the other hand, is the code executed for every (periodic) activation of the task. It is immediately preceded by the type of message sent back by the task to the gateway using the construct $[\tilde{\tau}]$. The task only sends messages of this type to the gateway and the type is checked against all **radio** statements in the task. The instructions available to the programmer include: assignment, actuation - $a(\tilde{e})$, sending a set of evaluated expressions to the gateway - **radio** $[\tilde{e}]$, a conditional execution construct - **if** $e\ \{\tilde{r}\}$ **else** $\{\tilde{r}\}$, and a while loop - **while** $e\ \{\tilde{r}\}$. The expressions are standard except for $s(\tilde{e})$ that is used to read a value from a given sensor.

As we said, for a given platform and configuration, the hardware description provided by the constructs sensors and actuators is the same. We use a preprocessing directive - use - to include this description at the top of all programming examples in this thesis (Figure 5.1.1).

STL Code 5.1.1 Hardware description for Arduino 2560 prototype WSN - file "ard2560.hw".

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

actuators {
  led        : bool -> void
}
```

The example in Figure 5.1.2 shows a STL program that at each activation reads the temperature and humidity and radios the values to the gateway. A similar implementation in Arduino C++ is presented in Appendix A.0.1. The example uses two sensors, designated as temperature and humidity, whose types are declared in hardware description file "ard2560.hw". Notice that the periodicity of the task is not included in the code. It is an external attribute set with the administration client when the task is sent to the gateway to be radioed to the nodes. In this way, users with administration access can dynamically change the period of running tasks using simple control messages.

STL Code 5.1.2 STL program that reads the temperature and humidity and radio the results to the gateway.

```
use "ard2560.hw"

init {
  float t = 0.0;
  float h = 0.0;
}

[float @ "temperature:Celsius",
 float @ "Humidity:Percentage"]
loop {
  t = temperature();
  h = humidity();
  radio [t,h];
}
```

The language specification is complete with both the operational and static semantics that together define how well-formed programs are executed.

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

$(S, A, V, \tilde{r})$. In the latter, S and A are of type $\mathsf{Set}(\text{Vars})$ 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 $\mathsf{Map}(\text{Vars}, \text{Values})$ keeps the values of the variables during the execution of the program. Thus, the initial state for the task:

$$\begin{aligned}&\textbf{sensors } \{s_1 : \sigma_1 \ldots s_n : \sigma_n\}\\ &\textbf{actuators } \{a_1 : \sigma_1 \ldots a_m : \sigma_m\}\\ &\textbf{init } \{\tilde{q}\}\ [\tilde{\tau}]\ \textbf{loop } \{\tilde{r}\}\end{aligned}$$

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

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

The reduction rules are presented in Figures 5.2 and 5.3, 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:

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

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. It then calls a built-in function $\mathsf{write}(a, \tilde{v})$ that actually performs the low-level operation for the program. When it returns the state of the program is $(V, \tilde{r})$. The reasoning is similar in rule (2) for expressions, where we read data from a sensor. Here, however, the value returned from the built-in function, $v = \mathsf{read}(s, \tilde{v})$, is the value of the expression. Rule (8) for instructions, another example, is invoked when the code sequence in the text block ends, the next state is $\perp$.

$$\frac{v = \mathsf{eval}(V, e)}{(V, \underline{x = e}\ \tilde{r}) \to (V + \{x : v\}, \tilde{r})} \tag{1}$$

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

$$\frac{\tilde{v} = \mathsf{eval}(V, \tilde{e}) \qquad \mathsf{send}(\tilde{v})}{(V, \underline{\textbf{radio}\ [\tilde{e}]}\ \tilde{r}) \to (V, \tilde{r})} \tag{3}$$

$$\frac{\mathsf{eval}(V, e) = \textbf{true}}{(V, \underline{\textbf{if}\ e\ \{\tilde{r}_1\}\ \textbf{else}\ \{\tilde{r}_2\}}\ \tilde{r}_3) \to (V, \tilde{r}_1\ \tilde{r}_3)} \tag{4}$$

$$\frac{\mathsf{eval}(V, e) = \textbf{false}}{(V, \underline{\textbf{if}\ e\ \{\tilde{r}_1\}\ \textbf{else}\ \{\tilde{r}_2\}}\ \tilde{r}_3) \to (V, \tilde{r}_2\ \tilde{r}_3)} \tag{5}$$

$$\frac{\mathsf{eval}(V, e) = \textbf{false}}{(V, \textbf{while}\ e\ \{\tilde{r}_1\}\ \tilde{r}_2) \to (V, \tilde{r}_2)} \tag{6}$$

$$\frac{\mathsf{eval}(V, e) = \textbf{true}}{(V, \textbf{while}\ e\ \{\tilde{r}_1\}\ \tilde{r}_2) \to (V, \tilde{r}_1\ \textbf{while}\ e\ \{\tilde{r}_1\}\ \tilde{r}_2)} \tag{7}$$

$$(V, \epsilon) \to \bot \tag{8}$$

Figure 5.2: Reduction rules for STL instructions.

$$\frac{\tilde{e} = e_1 \ldots e_n \qquad v_i = \mathsf{eval}(V, e_i), 1 \leq i \leq n}{\mathsf{eval}(V, \tilde{e}) = \tilde{v}} \tag{1}$$

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

$$\frac{v_1 = \mathsf{eval}(V, e_1) \qquad v_2 = \mathsf{eval}(V, e_2)}{\mathsf{eval}(V, e_1 \; op \; e_2) = v_1 \; \mathsf{op} \; v_2} \tag{3}$$

$$\frac{v = \mathsf{eval}(V, e)}{\mathsf{eval}(V, op \; e) = \mathsf{op} \; v} \tag{4}$$

$$\mathsf{eval}(V, x) = V(x) \tag{5}$$

$$\mathsf{eval}(V, v) = v \tag{6}$$

Figure 5.3: Reduction rules for STL expressions.

5.1.2 Static Semantics

The static semantics of a task is provided in the form of a type system (Figure 5.4). The rules are fairly standard and use a typing environment Γ that keeps track of the types for identifiers. The rules are written as $\Gamma \vdash r$ for instructions, meaning that the instruction is well-formed, and $\Gamma \vdash e : \tau$ for expressions, meaning that expression e has type τ. Some rules have side effects, in which the environment Γ is enriched with new entries and becomes Γ', as in $\Gamma \vdash \cdots \dashv \Gamma'$. An example is rule (4): $\Gamma \vdash [\tilde{\tau}] \dashv \Gamma, \mathit{radiates} : \tilde{\tau}$ (Γ' is Γ plus the type collected from the *radiates* construct). Besides this rule, three others are worthy of note. Rule (10) checks that messages sent by the task have types that match the one declared in the *radiates* construct. Rule (12) 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 τ'. The logic is similar for rule (9), where the type system just checks that the instruction $a(\tilde{e})$ is well formed (instructions do not evaluate to values). Rules (15), (16), and (17) are axioms and allow booleans, integers, and floating point values to be typed.

$$\frac{\begin{array}{l}\emptyset \vdash \textbf{sensors } \{s_1 : \sigma_1 \dots s_n : \sigma_n\} \dashv \Gamma_1 \\ \emptyset \vdash \textbf{actuators } \{a_1 : \sigma_1 \dots a_m : \sigma_m\} \dashv \Gamma_2 \\ \emptyset \vdash \textbf{init } \{q_1 \ \dots \ q_l\} \dashv \Gamma_3 \\ \Gamma_1, \Gamma_2, \Gamma_3, \mathit{radiates} : \tilde{\tau} \vdash \textbf{loop } \{\tilde{r}\}\end{array}}{\vdash \begin{array}{l}\textbf{sensors } \{s_1 : \sigma_1 \dots s_n : \sigma_n\} \\ \textbf{actuators } \{a_1 : \sigma_1 \dots a_m : \sigma_m\} \\ \textbf{init } \{q_1 \ \dots \ q_l\} \\ [\tilde{\tau}] \textbf{ loop } \{\tilde{r}\}\end{array}} \tag{1}$$

$$\emptyset \vdash \textbf{sensors } \{s_1 : \sigma_1 \dots s_n : \sigma_n\} \dashv \{s_1 : \sigma_1 \dots s_n : \sigma_n\} \tag{2}$$

$$\emptyset \vdash \textbf{actuators } \{a_1 : \sigma_1 \dots a_m : \sigma_m\} \dashv \{a_1 : \sigma_1 \dots a_m : \sigma_m\} \tag{3}$$

$$\frac{\Gamma \vdash q_1 \dashv \Gamma_1 \quad \Gamma \vdash q_l \dashv \Gamma_l}{\Gamma \vdash \textbf{init } \{q_1 \dots q_l\} \dashv \Gamma_1, \dots, \Gamma_l} \tag{4}$$

$$\frac{\emptyset \vdash v : \tau}{\Gamma \vdash \tau \ x = v \dashv \Gamma, x : \tau} \qquad \frac{\Gamma \vdash \tilde{r}}{\Gamma \vdash \textbf{loop } \{\tilde{r}\}} \tag{5,6}$$

$$\frac{\Gamma \vdash r_1 \ \dots \ \Gamma \vdash r_n}{\Gamma \vdash \tilde{r}} \qquad \frac{\Gamma \vdash x \colon \tau \quad \Gamma \vdash e \colon \tau}{\Gamma \vdash x = e} \tag{7,8}$$

$$\frac{\Gamma \vdash a : \tilde{\tau} \mapsto \textbf{void} \qquad \Gamma \vdash \tilde{e} : \tilde{\tau}}{\Gamma \vdash a(\tilde{e})} \tag{9}$$

$$\frac{\Gamma \vdash \tilde{e} : \tilde{\tau} \qquad \Gamma(\mathit{radiates}) = \tilde{\tau}}{\Gamma \vdash \textbf{radio } [\tilde{e}]} \tag{10}$$

$$\frac{\Gamma \vdash e \colon \textbf{bool} \quad \Gamma \vdash \tilde{r}_1 \quad \Gamma \vdash \tilde{r}_2}{\Gamma \vdash \textbf{if } \ e \ \{\tilde{r}_1\} \textbf{ else } \{\tilde{r}_2\}} \tag{11}$$

$$\frac{\Gamma \vdash s : \tilde{\tau} \mapsto \tau' \qquad \Gamma \vdash \tilde{e} : \tilde{\tau}}{\Gamma \vdash s(\tilde{e}) : \tau'} \tag{12}$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x \colon \tau} \tag{13}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \ \dots \ \Gamma \vdash e_n : \tau_n}{\Gamma \vdash e_1 \dots e_n : \tau_1 \dots \tau_n} \tag{14}$$

$$\emptyset \vdash v \colon \textbf{bool} \quad \emptyset \vdash v \colon \textbf{int} \quad \emptyset \vdash v \colon \textbf{float} \tag{15,16,17}$$

Figure 5.4: Type system for STL.

$$
\begin{array}{lll}
p & ::= & h\ d\ b & \textit{Program} \\
h & ::= & i_1\ i_2 & \textit{Header} \\
d & ::= & \tilde{v} & \textit{Data Segment} \\
v & ::= & \textit{bools} \mid \textit{ints} \mid \textit{floats} & \textit{Values} \\
b & ::= & \tilde{r} & \textit{Text Segment} \\
r & ::= & \textbf{ld}\ i \mid \textbf{st}\ i \mid \textbf{wrt}\ i_1\ i_2 \mid \textbf{rd}\ i_1\ i_2 & \textit{Instructions} \\
 & & \mid \textbf{rad}\ i \mid \textbf{bf}\ i \mid \textbf{jp}\ i \mid \textbf{ret} & \\
 & & \mid \textit{bop} \mid \textit{uop} &
\end{array}
$$

Figure 5.5: Byte-code syntax.

5.2 Compiler and Virtual Machine

In this section we give the specification for the SONAR Virtual Machine (SVM), one of the modules pre-installed in the nodes. The virtual machine executes STL tasks, translated into byte-code by a compiler. We begin by defining the byte-code format and then give the translation function for the STL source code.

5.2.1 Formal Description

The byte-code is composed of 4 segments: header, data, stack, and text (Figure 5.5). The header contains the total size of the byte-code as well as the offset to the beginning of the text segment. The stack segment is allocated between the data and text segment, growing towards the lower addresses. Its size is calculated at compile time since there are no calls to user defined functions. The data segment provides space for all the variables in a STL program. Constants and the initial values of global variables are stored there by the compiler. The data segment can be seen as the only activation record required for the virtual machine since, again, there are no calls to user functions or user functions in tasks. All variables, of types **bool**, **int**, and **float**, 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 have a 1 byte opcode and eventually 1 or 2 extra bytes for arguments. There are instructions for loading a value to the stack (**ld**), storing a value from the stack (**st**), sending an actuation command (**wrt**), reading a sensor (**rd**), sending a message over the radio (**rad**), the usual control flow (**bf**, **jp**, **ret**) and, the usual integer and floating-point arithmetic and logic and relational operators (**bop**, **uop**).

Byte-code instructions map almost one-to-one with reduction rules from the operational semantics.

This correspondence is important for proving that the virtual machine correctly executes the byte-code, but this is a problem we will not address here.

The translation function receives a syntactic term and returns a pair of sequences (D, B) (Figures 5.6 and 5.7). The first, D, is the contribution of the term to the data segment, the latter, B, is the contribution to the text segment. The top level translation function $[\![\cdot]\!]$, for STL tasks, breaks the translation into a sequence of pairwise concatenations (operator ":") and uses appropriate translation functions for each syntactic category. We use the same $[\![\cdot]\!]$ to simplify the notation, but these should be seen as distinct functions. The translation function uses 3 sets which hold integer identifiers for sensors and actuators, S and A, and data segment offsets for variables (set Var) and constants (set Const), V and U, defined as follows:

$$
\begin{aligned}
S &= \{(s_i, i) \mid s_i \in \widetilde{s : \tau}\} \\
A &= \{(a_i, i) \mid a_i \in \widetilde{a : \tau}\} \\
V &= \{(x, i) \mid x \in \text{Var} \wedge i = \text{offset}(x)\} \\
U &= \{(u, i) \mid u \in \text{Const} \wedge i = \text{offset}(u)\}
\end{aligned}
$$

The translation function is quite straightforward. The translation of an actuation command, $a(\tilde{e})$, is simply the translation of the arguments $\tilde{e}$, followed by a **wrt** instruction with the integer identifier for the actuator $A(a)$ and the number of expressions, $|\tilde{e}|$, as the arguments. Similarly, reading a sensor, $s(\tilde{e})$, translates into the translation of the expressions followed by a **rd** instruction with the integer identifier for the sensor $S(s)$ and the number of expressions, $|\tilde{e}|$, as the arguments. Likewise, the translation for **radio** $[\tilde{e}]$ is simply the translation of the expressions to be sent, followed by a **rad** instruction with the number of expressions, $|\tilde{e}|$, as the argument.

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 $[\![T]\!] = (D, B)$ and set its initial state to:

$$[D|\underbrace{0 \ldots 0}_{k}|B]_0$$

$$
\begin{aligned}
[\![T]\!] =& [\![\textsf{sensors } \{\widetilde{s : \tau}\}]\!] : \\
& [\![\textsf{actuators } \{\widetilde{a : \tau}\}]\!] : \\
& [\![\textsf{init } \{\tilde{q}\}]\!] : \\
& [\![[\tilde{\tau}] \textsf{ loop } \{\tilde{r}\}]\!] : \\
& (\epsilon, \textsf{ret}\,) \\
[\![\textsf{sensors } \{\widetilde{s : \tau}\}]\!] =& (\epsilon, \epsilon) \\
[\![\textsf{actuators } \{\widetilde{a : \tau}\}]\!] =& (\epsilon, \epsilon) \\
[\![\textsf{init } \{\tilde{q}\}]\!] =& [\![\tilde{q}]\!] \\
[\![\tau\ x = v\ \tilde{q}]\!] =& (v, \epsilon) : [\![\tilde{q}]\!] \\
[\![[\tilde{\tau}] \textsf{ loop } \{\tilde{r}\}]\!] =& [\![\tilde{r}]\!] \\
[\![r\ \tilde{r}]\!] =& [\![r]\!] : [\![\tilde{r}]\!] \\
[\![x = e]\!] =& [\![e]\!] : (\epsilon, \textsf{st } : V(x)) \\
[\![a(\tilde{e})]\!] =& [\![\tilde{e}]\!] : (\epsilon, \textsf{wrt } : A(a) : |\tilde{e}|) \\
[\![\textsf{radio } [\tilde{e}]]\!] =& [\![\tilde{e}]\!] : (\epsilon, \textsf{rad } : |\tilde{e}|) \\
[\![\textsf{if } e\ \{\tilde{r}_1\} \textsf{ else } \{\tilde{r}_2\}]\!] =& [\![e]\!] : (D', B') \\
& \text{where} \\
& (D_1, B_1) = [\![\tilde{r}_1]\!] \\
& (D_2, B_2) = [\![\tilde{r}_2]\!] \\
& D' = D_1 : D_2 \\
& j_1 = 2 + |B_1| \\
& j_2 = |B_2| \\
& B' = \textsf{bf } : j_1 : B_1 : \textsf{jp } : j_2 : B_2 \\
[\![\textsf{while } e\ \{\tilde{r}\}]\!] =& [\![e]\!] : (D, B') \\
& \text{where} \\
& (D, B) = [\![\tilde{r}]\!] \\
& j = 2 + |B| \\
& B' = \textsf{bf } : j : B : \textsf{jp } : -j - 2 \\
[\![\epsilon]\!] =& (\epsilon, \epsilon)
\end{aligned}
$$

Figure 5.6: Translation to bytecode (part I).

$$
\begin{aligned}
[\![e_1, \ldots, e_n]\!] &= [\![e_1]\!] : \cdots : [\![e_n]\!] \\
[\![s(\tilde{e})]\!] &= [\![\tilde{e}]\!] : (\epsilon, \textbf{rd} : S(s) : |\tilde{e}|) \\
[\![e_1 \ \textbf{bop} \ e_2]\!] &= [\![e_1]\!] : [\![e_2]\!] : (\epsilon, \textbf{bop}) \\
[\![\textbf{uop} \ e]\!] &= [\![e]\!] : (\epsilon, \textbf{uop}) \\
[\![x]\!] &= (\epsilon, \textbf{ld} : V(x)) \\
[\![u]\!] &= (u, \textbf{ld} : U(u)) \\
[\![\epsilon]\!] &= (\epsilon, \epsilon)
\end{aligned}
$$

Figure 5.7: Translation to bytecode (part II).

where k is the maximum stack size computed by the compiler and included in the bytecode. The computation proceeds according to the reduction rules presented in Figure 5.8, of the form:

$$\frac{c_1 \ \ldots \ c_n}{S \rightarrow S'}$$

where the c_i are preconditions or actions that must be fulfilled to make the transition from the current state, S, to a given state, S', possible. For example, when the current instruction (the one the program counter j is indexing) is **ld** (1st rule), 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** (2nd rule), 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* (3rd rule) and *actuators* (4th rule) provide the pointers to built-in functions associated with the identifiers. For example, the **rd** instruction (3rd rule) 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, $v_1 \ldots v_n$, are all stored at the top of the stack. The rule evolves by calling a built-in function $f \leftarrow sensor[i]$ with the arguments taken from the stack, $f(v_1 \ldots v_n)$ and placing the result of the call, v, at the top of the stack. The function *send* (5th rule) is a built-in that sends data over the radio. Finally, instructions *bop* and *uop* (9th and 10th rules) actually encapsulate a set of rules that include the usual arithmetic, relational and logical binary and unary operators.

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

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

$$\frac{\begin{array}{c} B[j] = \textbf{rd} \quad B[j+1] = i \quad B[j+2] = n \\ f \leftarrow \mathit{sensors}[i] \\ v \leftarrow f(v_1, \ldots, v_n) \end{array}}{[D|v_1, \ldots, v_n, S|B]_j \rightarrow [D|v, S|B]_{j+3}}$$

$$\frac{\begin{array}{c} B[j] = \textbf{wrt} \quad B[j+1] = i \quad B[j+2] = n \\ g \leftarrow \mathit{actuators}[i] \\ g(v_1, \ldots, v_n) \end{array}}{[D|v_1, \ldots, v_n, S|B]_j \rightarrow [D|S|B]_{j+3}}$$

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

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

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

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

$$\frac{B[j] = \mathit{bop}}{[D|v_2, v_1, S|B]_j \rightarrow [D|v_1 \ \mathit{bop} \ v_2, S|B]_{j+1}}$$

$$\frac{B[j] = \mathit{uop}}{[D|v, S|B]_j \rightarrow [D|\mathit{uop} \ v, S|B]_{j+1}}$$

$$\frac{B[j] = \textbf{ret}}{[D|S|B]_j \rightarrow \bot}$$

Figure 5.8: Transition rules for SVM.

5.3 Operating System

A node in a SONAR deployment may run multiple periodic tasks that generate data streams. Users with administration access to the deployments can program tasks, compile them and inject them in the deployment via the WSN Adapter (a Web service) which then forwards the tasks to the gateway to be radioed to the nodes. A simple protocol, implemented on top of the MAC layer, allows tasks, eventually divided into multiple blocks, to be sent over-the-air to the nodes. On arrival, the tasks are reassembled and installed in the nodes. Other control messages are also forwarded from the gateway to the nodes. From the nodes, the gateway receives data messages that it forwards to the deployment's Adapter to be forwarded to the SONAR Broker. Thus, the gateway does not run tasks, it acts simply as 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. Algorithm 1 shows this basic component. The gateway is initialized by attaching two handlers for interrupts signaling radio (from the nodes) and serial port (from the Adapter) 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.

Each node in a SONAR deployment has 2 pre-installed components: a small operating system and the SONAR virtual machine. The operating system is responsible for processing incoming control messages, for managing memory resources for tasks and for scheduling them EDF-style. Nodes keep information about tasks in a table. For each task, an entry in the table stores: a boolean - indicating if the entry is valid; three integers - the identifier, the period and the next activation of the task, respectively; and, an array of bytes - the byte-code for the task. The identifier is attached to messages sent by the task to the gateway so that the latter can distinguish to which stream the data it is receiving belongs to. This information is used to schedule the tasks and to prepare their execution with the SVM. The operating system executes a loop as described in Algorithm 2. The currently active task is identified by its integer index in the task table, denoted *curr* in the following algorithms.

A brief initialization attaches handlers for radio reception and real-time clock interrupts. 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

Algorithm 1 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
```

Algorithm 2 The node main loop

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

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.

Algorithm 3 Run current task

```
function RUN
    if ¬TABLEEMPTY() then
        (D, S, B) ← GETBYTES(curr)
        RUNSVM(D, S, B)
        t ← RTCTIME()
        p ← GETPERIOD(curr)
        SETNEXTACTIV(curr, t + p)
    end if
end function
```

Procedure RUN (Algorithm 3) 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 byte-code so that any state is 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 4 Select next task

```
function SCHEDULE()
    if ¬TABLEEMPTY() then
        min ← MAX_INT
        for 0 < i < MAX_TASKS do
            if TASKVALID(i) then
                t ← GETNEXTACTIV(i)
                if t ≤ min then
                    min ← t
                    curr ← i
                end if
            end if
        end for
    end if
end function
```

The SCHEDULE procedure (Algorithm 4) 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.

Algorithm 5 Sleep until next task activation

```
function sleep()
    if ¬tableEmpty() then
        t ← getNextActiv(curr)
        rtcAlarm(t)
    end if
    microSleep()
end function
```

Procedure Sleep (Algorithm 5) 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 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 milliseconds we find this assumption adequate for practical purposes.

Finally, procedure Listen (Algorithm 6) 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 4 types of control messages: TASK - sends the identifier, the period and the byte-code for a new task to be executed in the node; PERIOD - sends the identifier and the new period for a running task in the node; KILL - sends the identifier of a task to be invalidated in the node, and; RESET - that invalidates all tasks running on a node. When a new task is reassembled and copied to the task table, its next activation is set to getNextActiv($curr$) + δ, where δ 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.

5.4 Implementation and Data Flow

Figure 5.9 depicts an high level overview of the implementation and the data flow in a deployment gateway.

Algorithm 6 Handle Incoming Radio Message

```
function handleRadioInterrupt()
    interrupted ← True
end function

function listen()
    if interrupted then
        msg ← readRadioRCVBuffer()
        tag ← getTag(msg)
        switch ( tag )
            case TASK :
                i ← getId(msg)
                p ← getPeriod(msg)
                b ← getBytes(msg)
                addTask(i, p, b)
            case PERIOD :
                i ← getId(msg)
                p ← getPeriod(msg)
                changePeriod(i, p)
            case KILL :
                i ← getId(msg)
                removeTask(i)
            case RESET :
                for i = 0 ... TableSize − 1 do
                    removeTask(i)
                end for
        end switch
        interrupted ← False
    end if
end function
```

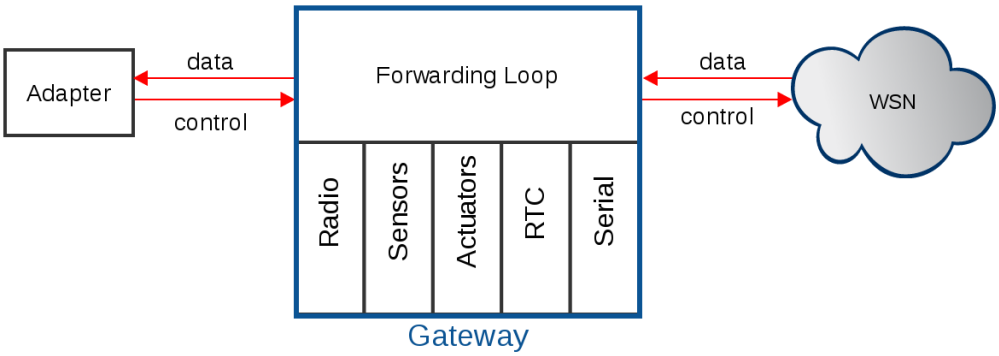


Figure 5.9: Message flow in SONAR gateway

The gateway software is composed by two main parts:

- a forwarding loop, which listens to control messages received from the deployment adapter and data messages received from the deployment nodes;
- a set of hardware libraries, used to access and control the sensors, actuators and other harware modules present in the board.

As presented in the same figure, the gateway is always listening to two different type of data: control messages, sent by the adapter, which contains the management commands sent by the Administration Client and data messages, sent by the deployment nodes, containing the data produced in each running task.

Figure 5.10 depicts the same high level overview of the implementation and the data flow, this time about the nodes.

Each node is composed by three main components:

- a receiving and scheduling loop, which is responsible for listening to new data radioed by the gateway and containing the control messages sent by the Administration Client, as well as by the scheduling of the running tasks in the node;
- a virtual machine, used to run the byte-code tasks stored at the node;
- a set of hardware libraries, used to access and control the sensors, actuators and other harware modules present in the board.

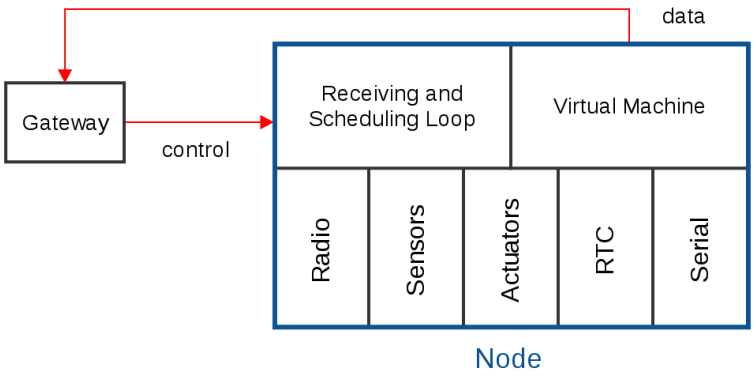


Figure 5.10: Message flow in SONAR nodes

As shown in the figure, a node receives control messages from the gateway via the XBee antenna and processes it. These messages are used to change to reprogram the tasks in the node, allowing to add or remove a task, change the period of a task and remove all the tasks.

All the data produced in a running task is directly sent to the gateway by the virtual machine, which uses the hardware libraries present in the node.

5.5 Summary

In this chapter we presented the formal description of the the data layer components. We started by formally presenting the STL syntax, depicting thereafter some STL examples and the respective reduction rules for the language instructions. Next we described the operating systems pre-installed in SONAR nodes, depicting the most important routines that compose both the nodes and the gateway. We finished this chapter with the formal description of SONAR virtual machine, as well as the description of the STL compiler.

Chapter 6

Setup, Evaluation and Discussion

In this chapter we analyze the impact of using our virtual machine and operating system in terms of energy consumption and memory footprint. We start by detailing the setup process of our system, presenting an example on how to use all the components. Next, we describe the hardware configuration of the nodes, a relevant point in the energy consumption. Thereafter, we present the experimental results obtained, followed by the analysis and discussion of the data gathered. We end this chapter with a brief summary.

6.1 Setup

We start this chapter by depicting the full process of using the SONAR prototype, explaining all the steps in the initialization process, the data flow happening when the Administration submits a new task, and the data flow related with the subscription and reception of the respective data by the Publish/Subscribe client.

6.1.1 Initialization

To start using SONAR, the user must own a computer with an USB interface, as simple as a Raspbery-Pi, where he can connect the gateway node. This computer must have access to the Internet, allowing the deployment to send the data to the Broker Web Service.

Using the previously installed SONAR software, the user starts the Adapter Web Service,

which automatically connects to a SONAR Broker, and connects the gateway to the computer. After finishing the boot, the gateway sends a message to the Adapter containing some information that needs to register the deployment in the Broker.

At this point, the user can start the Administration Client. It starts by opening a connection with the Adapter, then displaying in the user interface a list containing the available commands.

Before starting the management of the deployment, the user must register his own deployment in the Broker using the `reg` command, which sends the necessary information to the Broker in order to insert the deployment in the Broker running deployments list. Now, the initialization process is finished and the user can fully use his own SONAR deployment.

6.1.2 Managing the running tasks

To add a new task, the user must use the `task` command, providing the paths to previously created byte-code and description files.

Using this information, the Adapter starts two distinct data flows: one targeting the Broker, sending the information that a new task had been added to the deployment and another to the gateway, containing the byte-code that must be radioed to the nodes.

The same data flows are generated when the user submits the `period`, `kill`, and `reset` commands, affecting the nodes in the Data layer and changing the information about the tasks in the Broker.

6.1.3 Client subscription

To subscribe a set of tasks, a user must start the Publish/Subscribe client Web Service, which opens a connection to the Broker, and send a `ld` command. This command contacts the Broker and retrieves all the available deployments (and the respective tasks) that are connected in that moment.

Using that information, the client is able to subscribe a task using the `sub` command, which tells the Broker that this Client intends to receive all the data produced by a given task. The Broker receives that information and adds the ID of the client in the Tasks Table. From this moment, until the client closes the connection or unsubscribes the task, every

time the Broker receives data from that task, it automatically forwards it to the Client.

6.2 Node Configuration

In the current configuration of our prototype, each node is composed by an Arduino Mega2560 board connected to an Arduino Wireless Protoshield, equipped with a XBee Series 2 antenna, a SHT-15 temperature and humidity sensor, a Light-Dependent Resistor (LDR), one red LED and a Adafruit Chronodot Real-Time Clock. Figure 6.1 presents the hardware scheme of the nodes, depicting the connection between all the hardware components.

6.2.1 Jumper Connections

As depicted in Figure 6.1, we are currently using 3 jumpers from the wireless shield to the Mega2560 board: one jumper from pin 3 to pin 18, which connects the XBee Rx to the Arduino Tx_0, used to wake up the node when a new message is received from the gateway; a second jumper from pin 2 to pin 19, which connects the XBee TX to the Arduino Rx_0, used to pass to the antenna all the data produces in the tasks; and a third jumper from pin 8 to pin 20, which connects the Chronodot SQW pin to the Arduino SDA pin, used to wake up the node when a task must be executed.

6.2.2 XBee setup

To get a full description of the nodes in our prototype, it is relevant to describe how the XBee antennas are configured, since it significantly affect the energy consumption of this device.

When developing the software for our nodes, one important decision made was related with the network organization, which determines how a node communicate with the others. Currently, we are taking benefits from the fact that the XBee antennas allows the usage of a mesh architecture, with multi-hop communication and message reliability. It was a feature we thought that would be important given the fact that in some deployments, a node can be physically placed in locals that are out-of-reach to a direct communication with

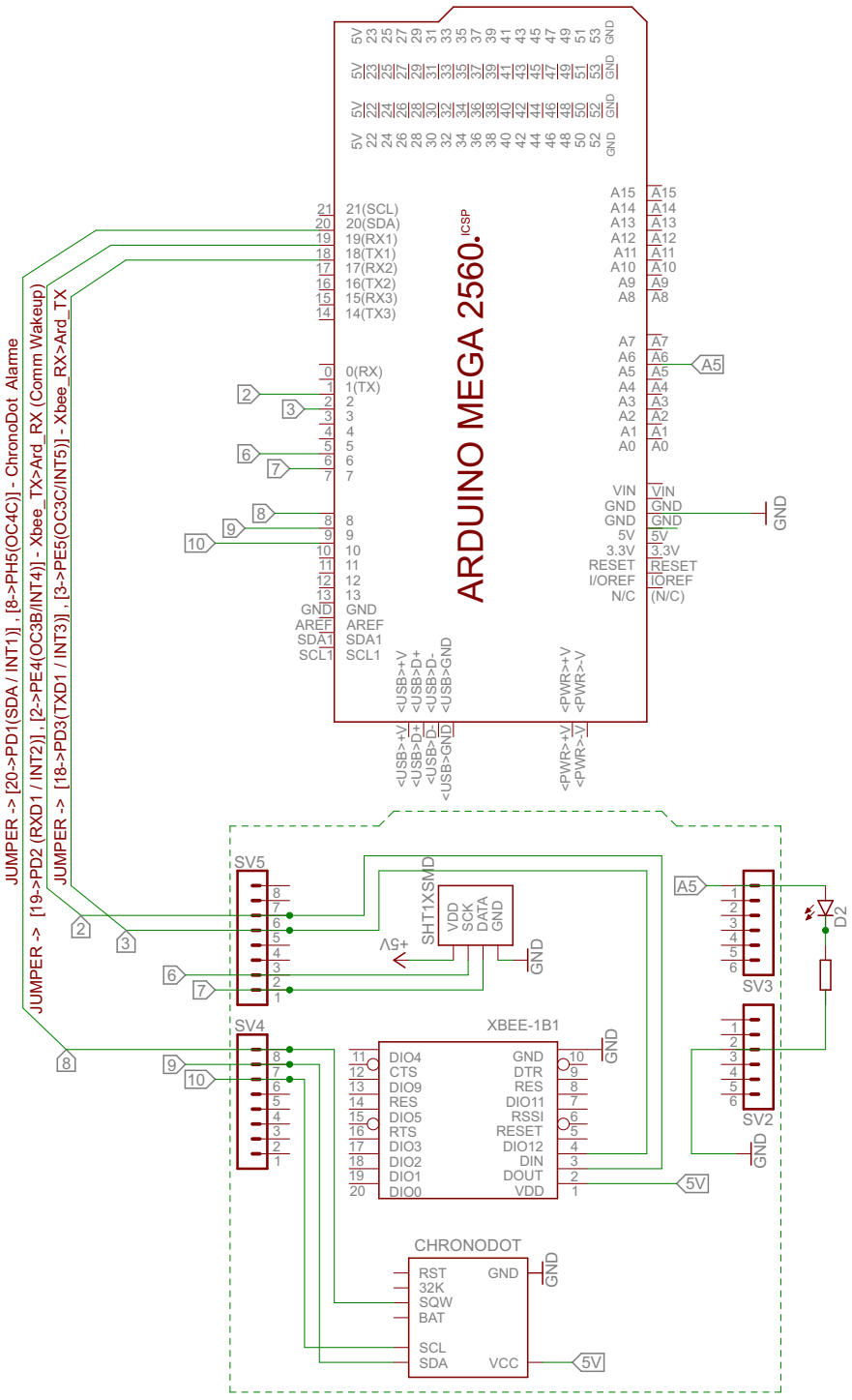


Figure 6.1: SONAR node scheme - Arduino Mega 2560

the gateway. The mesh architecture can be very useful in this situations, using message forwarding through the network and allowing this nodes to send data to the gateway.

Despite very useful, the mesh architecture brings an increase in the energy consumption, since all the nodes have to be configured has `routers`. This could in some cases be very undesirable, since a node marked as `router` disallows the transition of the XBee antenna to a `sleep` state, enabling power saving. A full description about all the possible XBee antenna configurations can be accessed in [34].

Other important parameters of our XBee configuration are:

- The RF interface is configured to use the `Highest Power Level`, with `Boost Mode Enabled`, increasing the amount of energy consumed by the antenna;
- There is no encryption in the radio messages and the network do no use any type of security, which would increase the energy consumption.

Although there are a bigger set of XBee configuration parameters, we are only analyzing a reduced number parameters since these are the ones that influence the most the energy consumption of this device.

6.3 Evalutation

The following tests have the purpose to determine the energy consumption and computation overhead of our prototype. The tasks test radio transmission, access to sensors and actuators and computation. Each test was implemented both in STL, running on top of SVM in each node, and directly in Arduino's native C/C++/Wiring. To measure the timings and the energy consumption, we connected a multimeter in series with one node (Figure 6.2) and registered the electric current variation associated with the execution of each task. The multimeter we used was a TENMA 72-7732A [35]. A Keysight InfiniiVision MSO-X 2002A oscilloscope [36] was also used for some time related measurements.

6.3.1 Experimental Results

The first tasks test radio transmissions. Figure 6.3.1 presents the STL code for a task that radioes 64 bytes, simulating a case where, for example, a task is programmed to

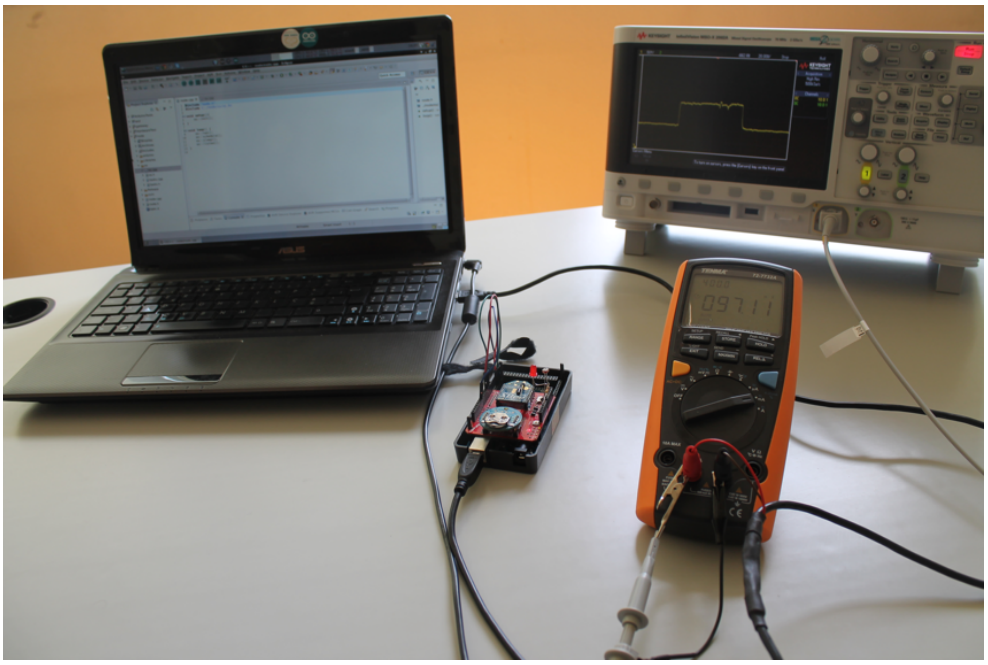


Figure 6.2: Multimeter and oscilloscope setup.

radio 16 sensor readings (floating point values) to the gateway. The 64 bytes refer only to the payload of the messages that carry an additional 10 bytes of header information. Appendix A.0.2 presents a similar implementation in Arduino C++.

STL Code 6.3.1 Transmission of data.

```
use "ard2560.hw"

[float @ "float1:sensor1",
 float @ "float2:sensor2",
 float @ "float3:sensor3",
 ...,
 float @ "float16:sensor16"]
loop {
  radio [ 2.0 , 2.0 , 2.0 , 2.0 ,
          2.0 , 2.0 , 2.0 , 2.0 ,
          2.0 , 2.0 , 2.0 , 2.0 ,
          2.0 , 2.0 , 2.0 , 2.0   ];
}
```

Figure 6.3 (blue line) shows current intensity vs. time when running the task with a period of 10 seconds. The first records the successful reception of the message by the node (our code puts the red LED on for 1.0 second). After the reception, one more peak (around $t = 20$) is visible, corresponding to the moments where the node radioed the 64 bytes.

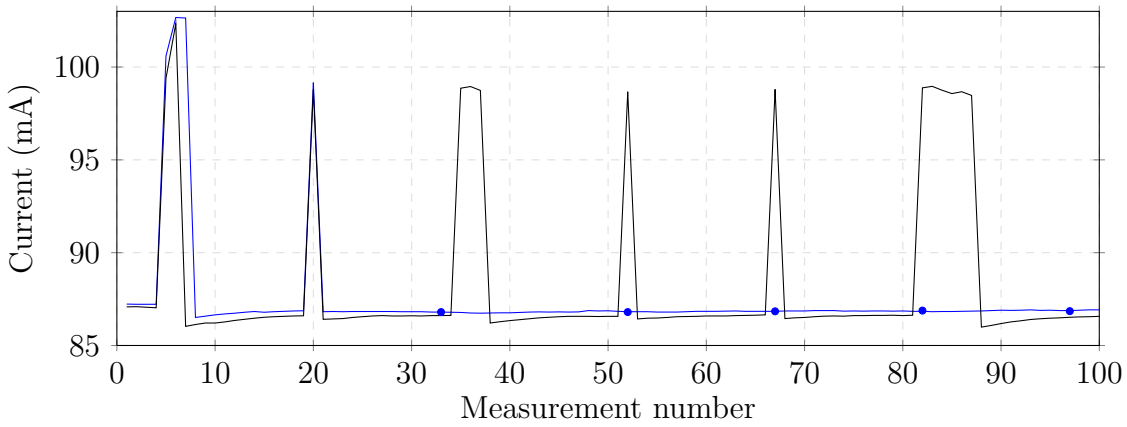


Figure 6.3: Radio Tests - 64 bytes Message

We would expect five more peaks (marked in the plot with a blue dot) within that time interval, given the period of the task. Their absence is due to a sampling problem related with the number of measurements the multimeter can execute per second. We ran the task again with a delay of 500 ms inserted and, sure enough, the other peaks became visible (black line). This delay was used only to allow the graphical visualization of the peaks. All the measurements given here were performed *without* the delay. Also, given the limited time resolution of the multimeter, we decided to use the digital oscilloscope to make all timing measurements. In this figure, the last, wide peak is due to a retransmission.

Figure 6.4 depicts the data obtained in one execution of this task. When the message is sent over the radio, a slight increase in the voltage ($\approx$ 0.1V) is detected, allowing us to time a full execution at 135ms.

Similar measurements were done for messages carrying 4, 8, 16, 32 and 64 bytes to assess how power varies with message size. Figure 6.5 presents the correlation between the size of the message and the time it takes to be received in the node. As it shows, there is a almost linear relation between those two parameters with a factor of 1.05 ms/B, plus an overhead of about 59 ms.

To test the access to sensors, we wrote a STL task (Figure 6.3.2) that accesses the temperature and humidity sensors in sequence. A similar program was written in C++ for Arduino (Appendix A.0.3). Figure 6.6 shows 6 executions of the STL task (black line) and of the Arduino code (red line). The graph plots current intensity vs. time when running the task with a period of 10 seconds. The fact that the peaks for the red and black lines are out of phase is due to overhead in the reception and initial scheduling of the task,

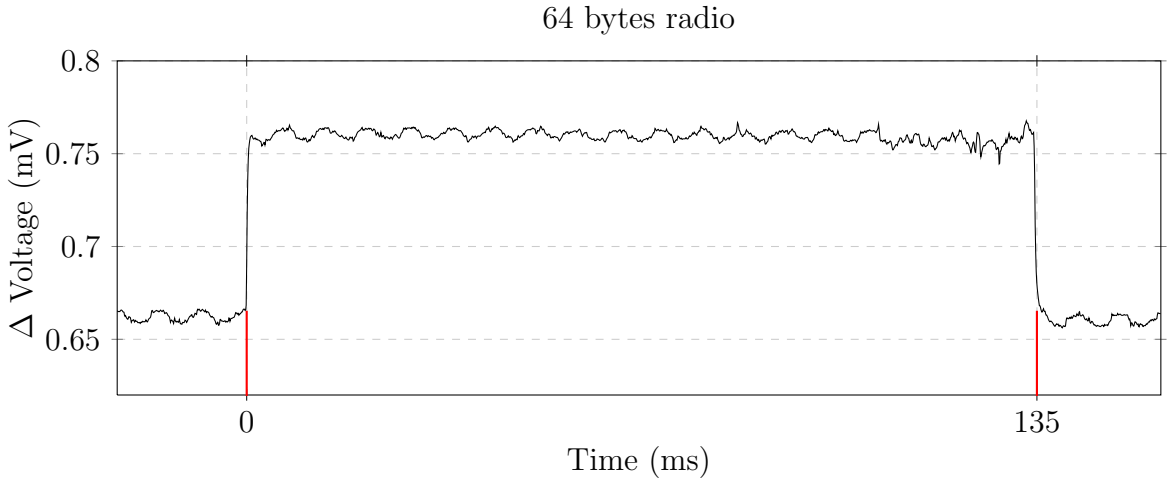


Figure 6.4: Oscilloscope data of task execution

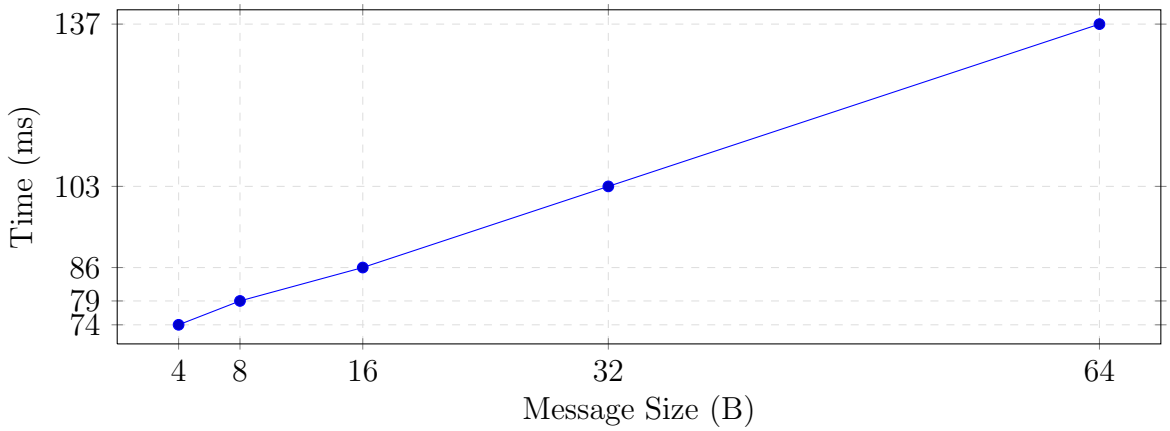


Figure 6.5: Message Size Comparison - Registered Time

otherwise the approximate periodicity is observed. The first peak is, again, due to the reception of the task in the node. This color code - black for STL and red for Arduino - will be used for all figures henceforth.

STL Code 6.3.2 Access to sensors.

```
use "ard2560.hw"

[ ]
loop {
  float t = temperature();
  float h = humidity();
}
```

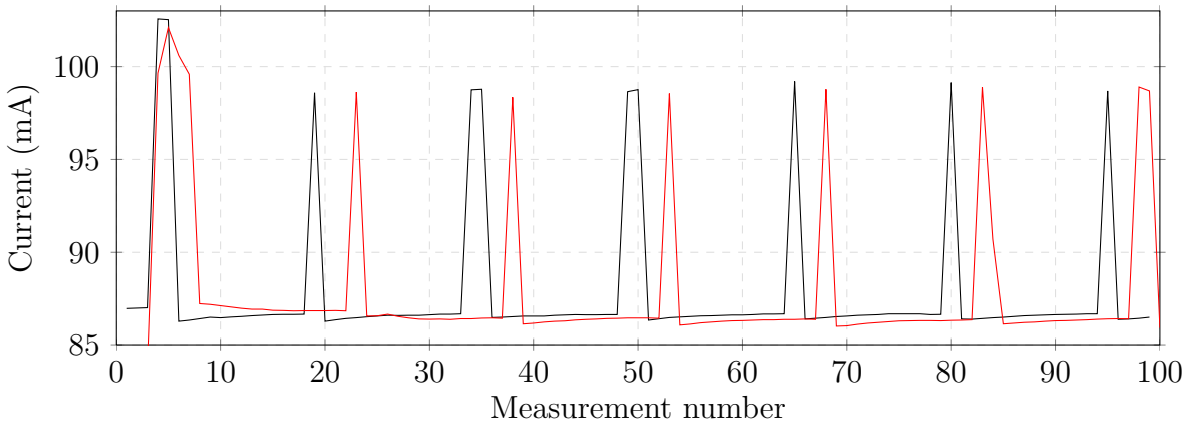


Figure 6.6: Sensors Access - Temperature and Humidity Task

The third task tests computation within the microprocessor. The task computes the 1000th term of the *logistic map* [37], a famous simple map that produces a series of numbers between 0 and 1. Figure 6.3.3 shows the STL code and Figure 6.7 shows the execution of 6 such tasks. Appendix A.0.4 presents a similar implementation in Arduino C++. A careful measurement with the oscilloscope, for this and the other examples, allowed us to conclude that the peaks are well approximated by a square wave with a maximum current of 98.9mA.

Finally, a task tests the triggering of actuators by alternatively activating and deactivating the external red LED. Figure 6.3.4 presents the code for the task. Appendix A.0.5 presents the same code written in Arduino native language, using the same RTC library as in SONAR. Figure 6.8 shows the execution of 6 tasks. The task activations correspond to the observed peaks in the graph, except for the first one. After a task is executed the current

STL Code 6.3.3 Computation of the logistic map.

```
use "ard2560.hw"

[]
loop {
  float x = 0.2;
  float k = 4.0;

  int i = 0;
  while (i < 1000) {
    x = k * x * (1.0 - x);
    i = i + 1;
  }
}
```

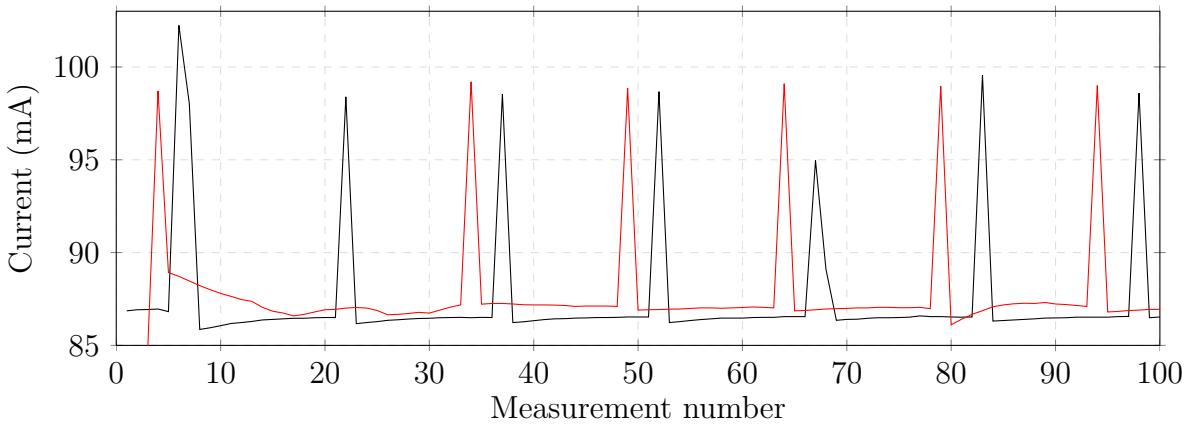


Figure 6.7: Computation - while loop

stabilizes in one of 2 levels, corresponding to LED disconnected and LED connected, with a difference of 4.3mA.

The intensity of the current in the Arduino Mega 2560 board varies between a base value, when the board is in sleep mode, not running a task, and a peak value, when a task is being executed by the virtual machine. The same values are observed for the corresponding Arduino programs. Table 6.1 shows the base and peak values for current, voltage and instantaneous power in the experiments. The Arduino 2560 provides a set of sleep modes with different levels of energy savings and hardware components turned off. Our prototype uses the IDLE mode, which is not the most power efficient, but allows us to wake up the board in time to properly receive asynchronous messages from the gateway. In order to save as much power as possible while in IDLE mode, we disable also: the analog-to-digital converter, the peripheral interface, three different timers, and the two wire interface.

STL Code 6.3.4 Access to external LED (actuators).

```
use "ard2560.hw"

init {
  bool state = false;
}

[]
loop {
  led(state);
  state = !state;
}
```

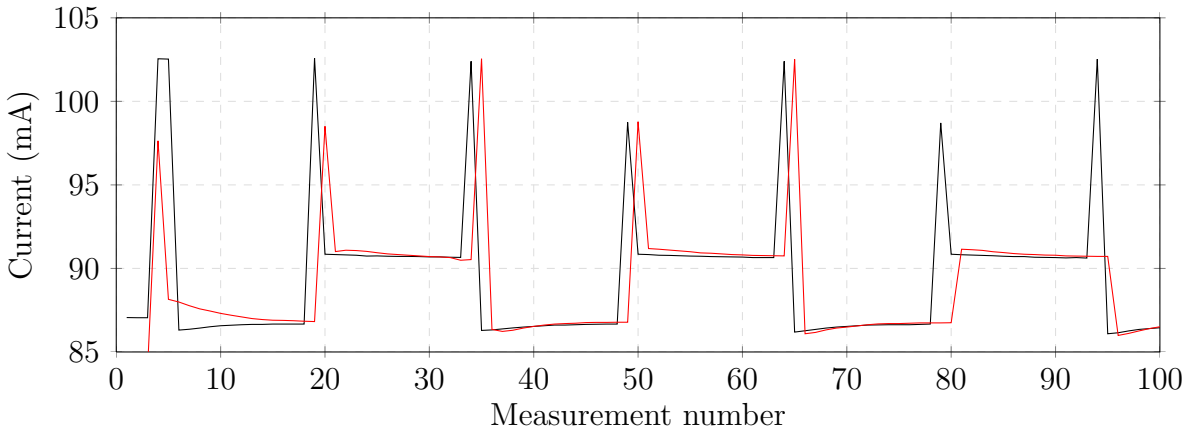


Figure 6.8: Actuators Test - External Red LED

From these base and peak values, and from the execution times of the tasks, we can compute the total energy spent by a SONAR task and by the corresponding native Arduino program. The values were computed from the measurements using the following equations for the instantaneous power and energy consumption:

$$P = V \times I$$
$$E = P \times \Delta t$$

These equations are adequate as we measured the profile of the tasks to be well approximated by rectangles of height equal to the peak intensity and width equal to their execution time. Table 6.2 shows, for each test: the size in byte of the STL task and then, the time and energy consumed to execute both the STL tasks and the equivalent Arduino program.

	base value	peak value
current (mA)	86.3	98.9
voltage (V)	5.0	5.0
power (mW)	432	495

Table 6.1: Base and peak values

task	size (B)	time (ms)		energy (mJ)		$\frac{\text{STL}}{\text{Ard}}$
		STL	Ard	STL	Ard	
Computation	121	160	27	79.1	13.4	5.9
Temperature	37	275	247	136.0	122.1	1.1
Humidity	37	274	80	135.5	39.6	3.4
Luminosity	37	37	11	18.3	5.4	3.4
Actuator	71	38	13	18.8	6.4	2.9
Radio - 4B	36	74	36	36.6	17.8	2.1
Radio - 8B	50	80	40	39.6	19.8	2.0
Radio - 16B	54	89	49	44.0	24.2	1.8
Radio - 32B	78	105	66	51.9	32.6	1.6
Radio - 64B	126	135	99	66.8	49.0	1.4

Table 6.2: Energy consumption

6.3.2 Discussion

The analysis of the measured data allowed us to conclude some interesting facts about the current prototype. The SONAR operating system and the virtual machine in the nodes introduce the highest overhead for tasks that are purely computational, by a factor of 5.9, for a cycle with 1000 iterations. However, a closer look at the ratio between the execution times in STL and Arduino (Figure 6.9) shows that part of this overhead includes an initial setup time by the node's operating system. In fact, as the number of iterations grows, the contribution of this initial overhead gets diluted and the real ratio between STL and Arduino (native) operations stabilizes at around 4.5 (the bars represent 95% confidence intervals). This is expected, and is due to the fact that we are running byte-code tasks on top of a virtual machine, rather than native code generated from C++ programs. We believe that optimizations of the byte-code generator and of the virtual machine implementation will diminish this gap, but it will otherwise be always present. It is the price of portability and dynamic reprogramming. The difference for other tests is far more modest, with access to sensors and actuators around 3 times slower and radio transmission of any size around 2. Though we throughly analyzed the code that accesses

the temperature sensor, we cannot yet explain the lower overhead (only a factor of 1.1 slower) relative to the other sensors and actuators (globally around 3 times slower). As the energy consumed is proportional to the time the task takes to execute, clearly the optimizations must focus on this aspect, all other being the same for STL tasks and for Arduino programs. We believe, however, that even in this unoptimized state our prototype compares well with Arduino native code with the added benefits of simplified programming and dynamic reprogramming.

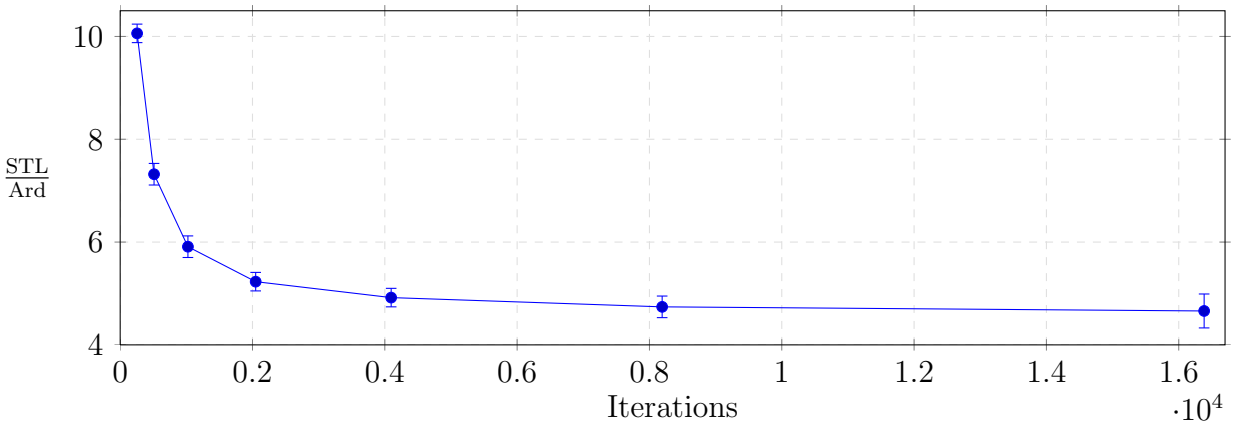


Figure 6.9: SVM overhead vs. size of problem.

The possibility to dynamically reprogram the network allow us to change the running tasks at a small energy cost, avoiding simultaneously the need to flash the board.

When programing in the Arduino native language, this process is much more complex, forcing the user to upload the code to the flash memory. Figure 6.10 presents the profile obtained using the oscilloscope when flashing one node to change the running program in Arduino native language. This data was obtained then flashing the board with the code used to test the temperature sensor access.

The first two visible peaks represents the moment when the board is reseting, a mandatory step so it can access the bootloader. After accessing the bootloader, there is a visible plateau where the board is waiting for a command indicating that there is a new program to be flashed to the board (presented in the plot from $t = 0.750$ to $t = 1.000$ approximately). Next, the flash starts and the data is written to the board (from $t = 1.000$ to $t = 2.500$ approximately), followed by a period where it is verified that all the data had been correctly written. The whole process took 4294ms to occur, being the board fully active for a period of 4024ms. Applying the formulas previously presented, flashing the board to upload a new program consumed about 1991.9mJ.

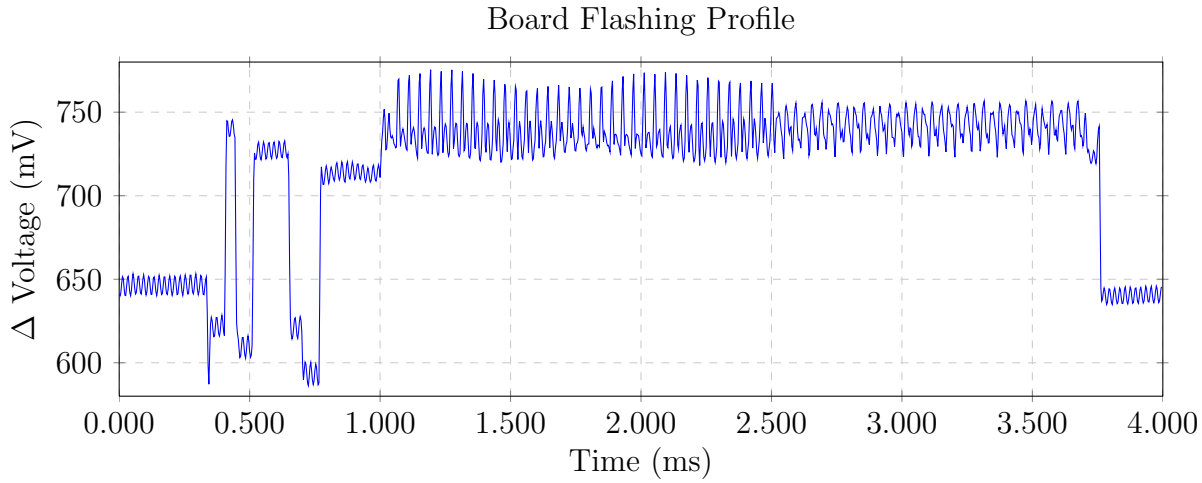


Figure 6.10: Board Flash Profile

To perform the same action using our prototype, the only consumption is the reception of the message, plus a very small overhead when the code is being stored and scheduled. In this case, it took us 100ms to receive the message, store the code and schedule it, with a consumption of about 49.5mJ, 40 times less that flashing the board in Arduino.

6.4 Summary

In this chapter we describe the usage of SONAR and present the benchmarking of our prototype. We describe the setup used to measure the energy consumption and execution time of each tested task, presenting all the plots obtained in each execution. We compare the values obtained using our prototype with the values obtained when running a task written using the Arduino native language. The ratio between these values allowed us to study the impact of our prototype in terms of energy consumption and execution time.

Chapter 7

Port to Arduino Uno

In this chapter we describe the process of porting our prototype to a new node based on a Arduino Uno board. We start by presenting an overview of the parameters we analyzed before starting to port our solution. Next, we describe the implementation of a hardware test tool we created to provide a simple and fast way to test the hardware components present on the board. We finish this chapter by describing the changes made to port our prototype to the Uno board, detailing the main problems found and quantifying the amount of code added or re-written.

7.1 Overview

In order to port the current solution to a new sensor based on the Arduino Uno, two main steps were taken to adapt all the needed code which makes our solution possible to run in a different node. First, we analyzed the used pins in Arduino Mega2560 and mapped it in the Arduino Uno pins, focusing in the used Universal Asynchronous Receiver/Transmitter (UART) connections used. Second, given the memory constrains of the Arduino Uno, we analyzed the memory consumption of our prototype and parameterized the operating system and virtual machine to reduce the maximum number of tasks allowed to run simultaneously in a node

Figure 7.1 presents a picture of both boards: the Mega2560 and the Uno. Both boards have the same hardware configuration, despite the differences in the jumpers connections, depicted in the picture.

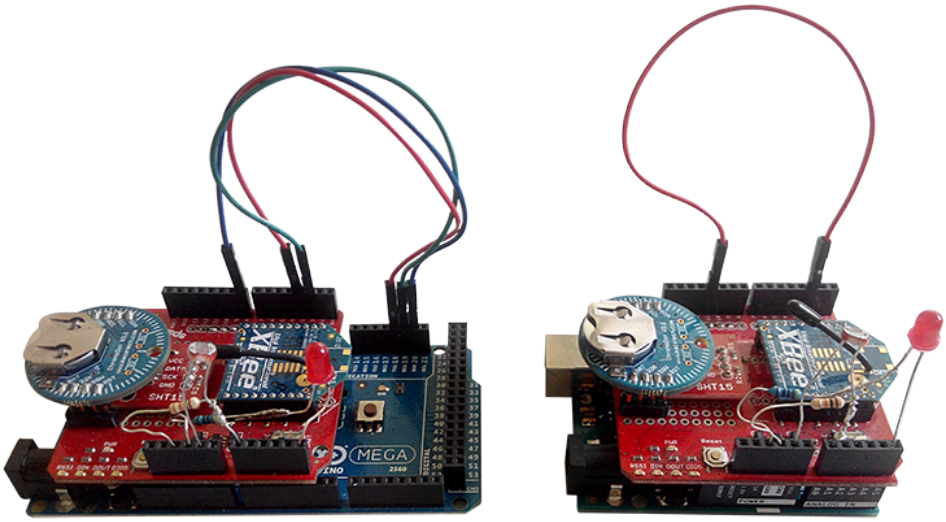


Figure 7.1: SONAR nodes - Mega2560 and Uno

7.2 Hardware Test Tool

At this point we concluded that currently, the SONAR nodes were using two UART pins in order to allow external interrupt events: one reserved to the XBee event, used to wake the nodes when a control message is being received and one for the RTC alarm, used to wake up the node when a deployed task must start to execute. At this point, one major difference between the Arduino Mega2560 and the Arduino Uno arises: the former has four UART pins and the latter only has one. This lead us to make some significant changes in the way our nodes implement the external events to wake up.

In order to test all the needed changes, we decided to implement a simple test project that allows us to interact with each module of our node, using the same hardware libraries we were using before. In the next two subsection we describe the implementation process of this tool.

7.2.1 XBee external event

The first change we have done was the XBee connection to the board. In the previous implementation, the XBee antenna was connected to the Arduino Mega2560 board through jumpers using the serial communication pins 18 and 19 (RX1 and TX1 respectively). Given

the current wire welding, we would need to use software serial communication to allow the serial communication between the XBee antenna and the Uno board. Since version 1.0, Arduino offers by default a library that emulates the serial communication on other digital pins, the SoftwareSerial [38] library. Using this library, we mapped two serial communication pins in digital pin 2 and 3, connecting the XBee TX pin to the Uno RX pin and the XBee RX pin to the Uno Tx pin. With this configuration, we created a simple example that tries to send some floating point values to the gateway using unicast radio messages. The gateway correctly received the message sent, proving that the XBee antenna and the Uno board are correctly connected and working.

7.2.2 RTC alarm

After changing the XBee connection and event in the Arduino Uno board, we started to solve the RTC alarm problem. Given the fact that at this point we did not have any more serial communication pins available in our board, we searched for a different way of allowing events to wake up the board using the RTC. After some research, we found a simple solution that can be used to overcome this restriction: Pin Change Interrupt (PCInt). A Pin Change Interruption can be enabled on any of the Arduino Uno signal pin, allowing the trigger of events ON CHANGE of the pin value. To test this feature we used the same library, PinChangeInt [39]. With this library, we were able to enable software interrupts in a given digital pin, allowing to receive the alarm event generated at the RTC.

At this point we analyzed the circuit connections used in the previous implementation (Figure 6.1) and decided to use pin 8 as the target pin. With this choice, we were able to maintain the same welded wires and allow the RTC to wake the board using pin CHANGE interrupt.

To test if this approach works with our node configuration, we have written a very simple test in C++ where we simply start the node, schedule an alarm for 5 seconds later, put the node in sleep mode and wait for the alarm to wake it. With this test we were able to verify that the chosen configuration allows the RTC to correctly generate wake up events using PCInt.

7.2.3 Gathering all together

After testing each of the previous changes, we tried to incorporate all the features in a simple tool that allows the test the complete hardware components present in the board. For this tool, we uses the same libraries used in SONAR, in order to guarantee that no library conflicts occurs.

In the process of gathering all together, an error arises when incorporating the PCInt library and the SoftwareSerial library. After some code analysis, we discovered that the error was related with the fact that the SoftwareSerial library uses some PCInt definition to work properly. More specifically, both libraries are defining a set of interrupt vectors, which enter in conflict. To solve this problem, we analyzed the set of pins that each of this vectors is using, getting the following results:

- ISR (PCINT0_vect) pin change interrupt for digital pins from 8 to 13
- ISR (PCINT1_vect) pin change interrupt for analogical pins from 0 to 5
- ISR (PCINT2_vect) pin change interrupt for digital pins from 0 to 7

Given the pin configuration we previously used, the solution to this problem was to simply split which vector pins are used by each library: PCInt library defined the PCINT0_vect and PCINT1_vect, allowing the usage of interrupts in pin 8, and the SoftwareSerial library defined the PCINT2_vect, allowing the creation of software serial communication in pins 2 and 3. With these changes, we were able to use both libraries simultaneous without conflicts, finishing the implementation of the hardware test tool.

Figure 7.2 shows an example of the interface: the user is asked which hardware component he wants to test and then the output is retrieved by the selected component.

7.3 SONAR porting

The Hardware Test Tool allowed us to prove that it is possible to use all the hardware components simultaneously using the Arduino Uno. At this point, we started to adapt all the software code to run in the board. We started by adding the PCInt and SoftwareSerial libraries, allowing the XBee to correctly communicate with the board and the RTC to

```
Welcome to the Hardware Test Tool for Arduino Uno
Select one of the following options:

    1 - Read SHT-15 temperature
    2 - Read SHT-15 humidity
    3 - Read LDR luminosity
    4 - Read RTC time
    5 - Turn on LED for 1.5 seconds
    6 - Test board IDLE mode
    7 - Test XBee Radio antenna

Option: 2
Humidity: 46.89 %
```

Figure 7.2: Hardware Test Tool Example

	Section Size (kB)			Total Size (kB)	
#Tasks	.data	.test	.bss	FLASH	SRAM
8	0.48	23.3	2.31	23.3	2.79
4	0.48	23.3	1.57	23.3	1.95
2	0.48	23.3	1.04	23.4	1.53

Table 7.1: Memory usage: max. of 8, 4, and 2 tasks

trigger the wake up events when a task must be executed. With these changes, we expected to be able to run the full implementation using the Arduino Uno.

To test the port, we compiled all the code with the new libraries added. We adjusted the number of maximum tasks to 4 and 2, in order to save some SRAM space. Table 7.1 presents all the values obtained when compiling the code.

As shown in 7.1, the 2kB of SRAM available in Arduino Uno do not allow to schedule 8 tasks simultaneously without some code optimization. However, reducing the maximum number of tasks to 4 or 2 reduces the SRAM usage to about 1.95kB and 1.53kB respectively, which fits in the Uno parameters (98% and 77% of total Uno SRAM size). Despite it was possible to run the software with 4 tasks, we decided to configure the maximum number of tasks as 2 for our porting test.

7.3.1 Port Tests

After uploading the software to the Uno board, we used the previously presented STL programs to test if everything is working correctly. When the node is turned on, it broadcasts a message to its PAN announcing himself as a new node. As expected, the gateway receives that message and stores the node Mac Address as a new deployment node.

At this point we tried to radio a task from the gateway to verify if the node is receiving and executing it correctly. Contrarily to what we expected, the node did not react to the task reception, turning on the LED. In order to verify if the message was reaching the board causing it to wake from the sleep mode, we again connected the node to the multimeter and the oscilloscope. After resending the task to the node, we registered a peak in the voltage, proving that the node is waking from sleep and receiving some radio message.

As we started to debug this problem, we discovered that it was related with the fact that when a message is received through the XBee, the board is waking but for an unknown reason the interruption was not being correctly generated. We suspected that this problem may be related with the SoftwareSerial connections previously created, so we decided to try a different approach. Another way to allow the XBee to wake the board is using the same interrupts we had previously used with the RTC, an PCInt. Given the current board wiring already presented, we changed the XBee interruption attachment from pin 3 to pin 11, which we connected with a jumper, and added another PCInt in pin 11, allowing the board to wake when a radio message is received.

After these changes, we were able to correctly receive some messages from the gateway: `MAC`, `kill` and `period` commands. When testing the reception of new tasks, another problem occurred: when trying to send new tasks, the node only receives and executes them if the byte-code size of that task is smaller than 55 bytes. After inspecting the SoftwareSerial library, we found that this problem was related with the fact that the a SoftwareSerial port uses a 64 bytes reception buffer, which will not be sufficient for receiving tasks with size greater than 54 bytes (plus the 10 bytes header always sent). Given the fact that this is a software limitation, we solved the problem by changing the communication protocol created to divide the tasks in packets, reducing the maximum size of each packet to 54 bytes. After this change, we tested again the reception of tasks with size greater than 54 bytes and everything worked correctly, finishing the port to the Arduino Uno board. The schematic representation of the new node is presented in Figure 7.3.

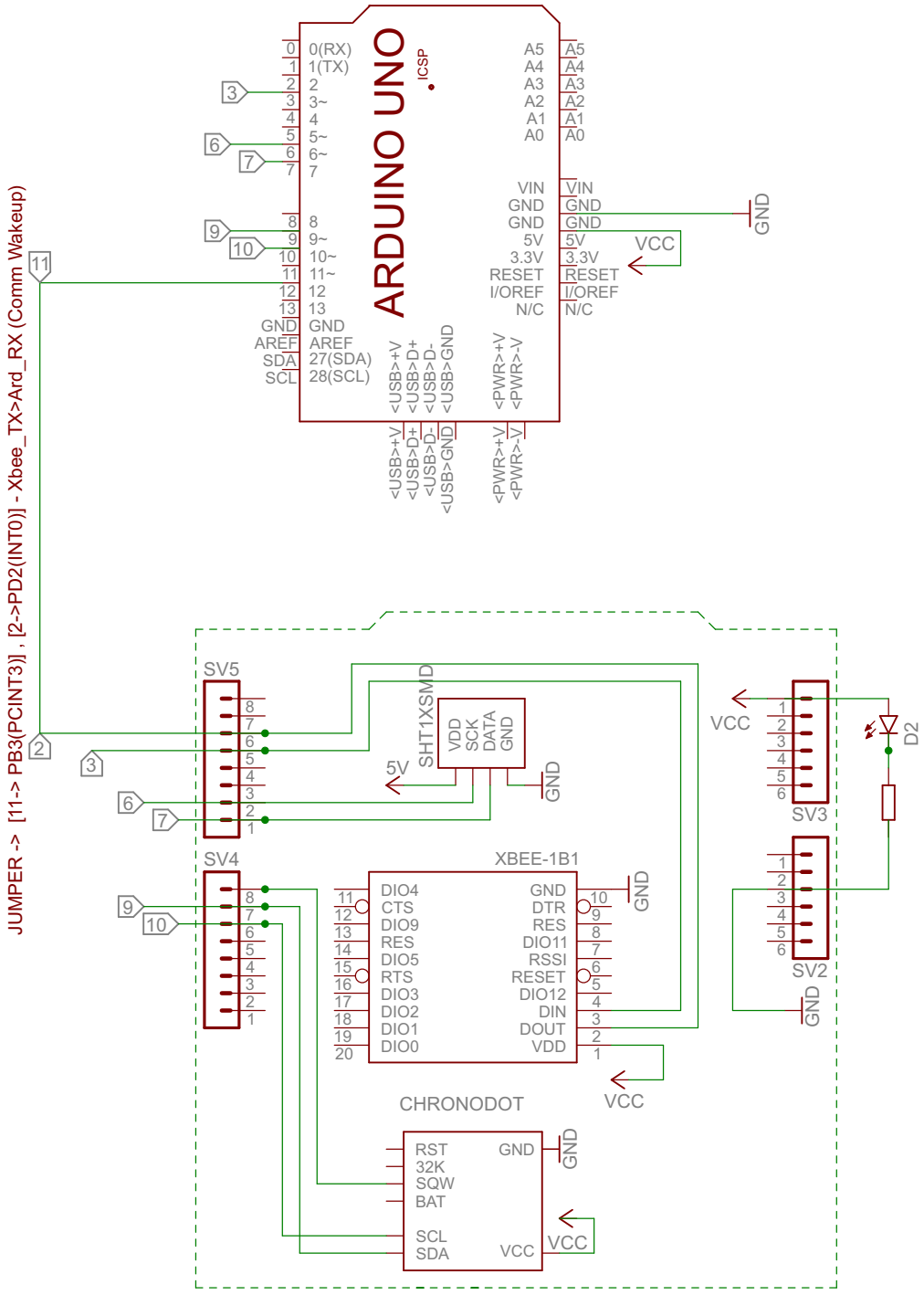


Figure 7.3: SONAR node scheme - Arduino Uno

7.3.2 Port effort

The effort to port all the available code to a new Arduino board with less memory capacity was essentially related with the new connections and the limitation introduced by the different speed when using SoftwareSerial communications. Table 7.2 present a brief quantification of changes needed, in terms of lines of code, connections and libraries added.

	Gateway		Node	
	Added	Changed	Added	Changed
Lines of Code	0	3	13	4
Libraries	0	0	2	2
Wire Connections	0	0	1	3

Table 7.2: Port effort quantification: lines of code, libraries, and wire connections

As described in the previous section, the biggest difficult associated with the port was the connection scheduling, allowing all the hardware components to correctly communicate with the others.

Although in this case we opted to port our solution to a similar platform with less memory and UART pins, the easiness of the process suggest that the effort to port our prototype to a completely different platform with similar capacities would reside mostly in discovering hardware libraries for all the components and adjust some parameters to fit the platform memory constrains.

7.4 Summary

In this chapter we presented a port of SONAR to the Arduino Uno platform. We described in detail the creation of a simple hardware test tool, used to test all the hardware connections in the new board, as well as all the changes in the code to adapt the prototype to the new board, with less SRAM memory capacity and only one available UART pin. To test the port we used the same STL tasks presented in Chapter 6, leading to a set of errors related with the SoftwareSerial Library. We then solved this issue and successfully ported the system to Uno. We finish this chapter by trying to quantify the effort of this port, analyzing the amount of code we needed to rewrite and the changes in the connections.

Chapter 8

Conclusions and Future Work

In this chapter we present the conclusions of this thesis as well as some ideas about future work that can improve our architecture.

8.1 Conclusions

In this thesis we addressed the problem of providing to non-specialist users a framework for programming, configuring and deploying a WSN in a seamless way. We presented our solution to this problem, SONAR, a three-layered Publish/Subscribe architecture implemented over a set of interconnected Web Services, describing the implementation of each component that composes this layers.

The SONAR Data layer presents a new approach in the usage of Virtual Machines to develop WSN, since it allows dynamic reprogramming and debugging of the tasks running in the nodes, as well as intranode multitasking. To manage the running tasks, each node comes with a pre-installed operating system to manage memory and schedule tasks alongside with the virtual machine to run tasks. The use of a virtual machine associated with a domain specific language provides application portability and dynamic reprogramming of WSN. Moreover, given the fact that the data layer is based on a virtual machine, an important feature is related with the portability of this approach across WSN, that needs to be as simple as possible.

To prove the portability of our prototype, we have ported it from a node based in the

Arduino Mega2560 to a node based in th Arduino Uno, which presents significantly more memory constrains. We described all the porting process and quantified the amount of code rewritten.

To finish the study of our prototype, we studied the impact of using a virtual machine and an operating system in terms of hardware resources and energy consumption, two important factors in a WSN. We created a set of tests in which we connected a multimeter and an oscilloscope to the Mega2560 nodes, measured the energy consumption and the execution time of running tasks with our approach and compared it with a similar implementation done using the Arduino native language.

After finishing the evaluation of the impact, the data we gathered in the tests allowed us to conclude that this approach presents a small memory footprint on the nodes: we easily fitted 8 running tasks in a board with just 8kB of SRAM, leaving enough space to increase the number of running tasks. In the Uno board, we fitted 4 running tasks in a 2kB SRAM. About the computational overhead introduces by the usage of the virtual machine and operating system, the tests led us to conclude that the biggest overhead between the execution of tasks written in STL and Arduino (native language) is reached in tasks that are purely computational, in a factor of about 4.5. All the remainder of the tests presented a far more modest ratio, with access to sensors and actuators around 3 times slower and radio transmissions about 2 times slower. All of this overheads were already expected due to the fact that we are running byte-code on top a virtual machine rather than native code generated and optimized from C++.

The take away message from these experiments is thus:

- we fit in the available memory even without optimization (we use standard libs) and in small devices;
- we have little computational overhead;
- we are far more simple to program with;
- we can simply reprogram networks at minimal energy cost, with no flashing.

The work developed under this thesis led to the submission of an article called "Towards Out-of-the-Box Programming of Wireless Sensor-Actuator Networks" [40] to the conference "18th IEEE International Conference on Computational Science and Engineering".

8.2 Future Work

There are a set of features we are currently working to add in our prototype.

First we want to extend our language so it can support the usage of arrays and some aggregation primitives. The possibility of adding data aggregation within a deployment is an important feature to reduce the energy consumption of the overall network. Coupled with this feature comes the need to change the communication model inside the Data layer, allowing each node to send the sensed data to another node of the network, rather than only to the gateway.

In order to minimize the energy consumption of the network, we are studying a new approach in the way our nodes receive new tasks. The current state of our prototype assumes a communication model where the administrator can submit a task to his deployment anytime, producing immediate changes in the running tasks. The real-time changes produced by the addition of a new task is allowed by the fact that the XBee antennas in each node are constantly listening to new messages, disabling the possibility to put them in a sleep state. This approach is very useful in applications where time is a critical component, making possible to apply changes in the network at any moment and receiving the data in a small period of time since the changes have been submitted. Shifting this working model to an approach where the application is more flexible in the time it takes to receive the new tasks and apply the changes to the deployment, we can significantly save energy by putting the XBee antennas in a sleep state, programming it to periodically wake up to verify if new tasks were added to the network. In this approach, the gateway would become more complex, being responsible for queuing the new tasks added by the administrator, as well as the information about which nodes have already waked up and received this task. At the same time, it would allow us to introduce a deeper sleep of state in the nodes, saving even more energy.

References

[1] L. Mottola and G. P. Picco, "Programming wireless sensor networks: Fundamental concepts and state of the art," *ACM Computing Surveys (CSUR)*, vol. 43, no. 3, p. 19, 2011.

[2] L. Lopes, F. Martins, and J. Barros, "Programming wireless sensor networks," in *Middleware for Network Eccentric and Mobile Applications*. Springer Berlin Heidelberg, 2009, pp. 25–41.

[3] K. Römer, O. Kasten, and F. Mattern, "Middleware challenges for wireless sensor networks," *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 6, no. 4, pp. 59–61, 2002.

[4] Q. Zhu, R. Wang, Q. Chen, Y. Liu, and W. Qin, "IOT Gateway: BridgingWireless Sensor Networks into Internet of Things," in *8th International Conference on Embedded and Ubiquitous Computing (EUC)*, Dec 2010, pp. 347–352.

[5] E. Neto, R. Mendes, and L. Lopes, "An architecture for seamless configuration, deployment, and management of wireless sensor-actuator networks," in *3rd International Conference on Sensor Networks (SENSORNETS 2014)*, SciTePress. SciTePress, January 2014.

[6] E. Neto, "Towards Out-of-the-Box Programming of Wireless Sensor Networks," Master's thesis, Faculdade de Ciências da Universidade do Porto, July 2014.

[7] Arduino Mega 2560 Overview. Last seen in January, 2015. http://arduino.cc/en/Main/arduinoBoardMega2560

[8] R. Gummadi, O. Gnawali, and R. Govindan, "Macro-programming wireless sensor networks using kairos," in *Distributed Computing in Sensor Systems*. Springer, 2005, pp. 126–140.

[9] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "TinyDB: an acquisitional query processing system for sensor networks," *ACM Transactions on database systems (TODS)*, vol. 30, no. 1, pp. 122–173, 2005.

[10] D. Gay, P. Levis, R. Von Behren, M. Welsh, E. Brewer, and D. Culler, "The nesC language: A holistic approach to networked embedded systems," in *Acm Sigplan Notices*, vol. 38, no. 5. ACM, 2003, pp. 1–11.

[11] nesC Language Reference Manual. Last seen in January, 2015. http://www.tinyos.net/api/nesc/doc/ref.pdf

[12] R. Newton, G. Morrisett, and M. Welsh, "The regiment macroprogramming system," in *6th International Conference on Information Processing in Sensor Networks*. ACM, 2007, pp. 489–498.

[13] T. V. Chien, H. N. Chan, and T. N. Huu, "A comparative study on operating system for wireless sensor networks," in *International Conference Advanced Computer Science and Information System (ICACSIS)*. IEEE, 2011, pp. 73–78.

[14] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer *et al.*, "TinyOS: An operating system for sensor networks," in *Ambient intelligence*. Springer, 2005, pp. 115–148.

[15] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *29th Annual IEEE International Conference on Local Computer Networks (LCN 2004)*. IEEE, 2004, pp. 455–462.

[16] Q. Cao, T. Abdelzaher, J. Stankovic, and T. He, "The liteOS operating system: Towards unix-like abstractions for wireless sensor networks," in *International Conference on Information Processing in Sensor Networks (IPSN'08)*. IEEE, 2008, pp. 233–244.

[17] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava, "A dynamic operating system for sensor nodes," in *3rd International Conference on Mobile systems, applications, and services*. ACM, 2005, pp. 163–176.

[18] E. Baccelli, O. Hahm, M. Günes, M. Wählisch, T. Schmidt *et al.*, "Riot os: Towards an os for the internet of things," in *32nd IEEE International Conference on Computer Communications (INFOCOM 2013)*, 2013.

[19] H. Will, K. Schleiser, and J. Schiller, "A real-time kernel for wireless sensor networks employed in rescue scenarios," in *IEEE 34th Conference on Local Computer Networks (LCN 2009)*, Oct 2009, pp. 834–841.

[20] 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 *2nd International Conference on Virtual execution environments*. ACM, 2006, pp. 78–88.

[21] P. Levis and D. Culler, "Maté: A tiny virtual machine for sensor networks," in *ACM Sigplan Notices*, vol. 37, no. 10. ACM, 2002, pp. 85–95.

[22] I. L. Marques, J. Ronan, and N. S. Rosa, "Tinyreef: a register-based virtual machine for wireless sensor networks," in *Sensors, 2009 IEEE*. IEEE, 2009, pp. 1423–1426.

[23] J. Koshy and R. Pandey, "Vmstar: synthesizing scalable runtime environments for sensor networks," in *3rd International Conference on Embedded networked sensor systems*. ACM, 2005, pp. 243–254.

[24] U. Hunkeler, H. L. Truong, and A. Stanford-Clark, "Mqtt-s - a publish/subscribe protocol for wireless sensor networks," in *3rd International Conference on Communication Systems Software and Middleware and Workshops (COMSWARE 2008)*. IEEE, 2008, pp. 791–798.

[25] D. Locke, "Mq telemetry transport (mqtt) v3. 1 protocol specification," *IBM developerWorks Technical Library*, 2010.

[26] E. Souto, G. Guimarães, G. Vasconcelos, M. Vieira, N. Rosa, C. Ferraz, and J. Kelner, "Mires: a publish/subscribe middleware for sensor networks," *Personal and Ubiquitous Computing*, vol. 10, no. 1, pp. 37–44, 2006.

[27] T. Luckenbach, P. Gober, S. Arbanowski, A. Kotsopoulos, and K. Kim, "Tinyrest: A protocol for integrating sensor networks into the internet," in *Proc. of REALWSN*. Citeseer, 2005, pp. 101–105.

[28] MICAz datasheet. Last seen in January, 2015. http://www.openautomation.net/uploadsproductos/micaz_datasheet.pdf

[29] B. Valente and F. Martins, "A Middleware Framework for the Internet of Things," in *The Third International Conference on Advances in Future Internet (AFIN 2011)*. Xpert Publishing, August 2011, pp. 139–144.

[30] N. Sharma, "Push technology–long polling," *http://www.ijcsmr.org/vol2issue5/paper398.pdf*, 2013.

[31] (2012, December) Server Sent Events - W3C Standard. Visited in November, 2014. http://www.w3.org/TR/2009/WD-eventsource-20091029/

[32] (2012, September) Web Sockets - W3C Standard. http://tools.ietf.org/pdf/rfc6455.pdf

[33] Amazon S3 Web Service. Visited in June, 2015. https://aws.amazon.com/s3/?nc2=h_ls

[34] (2009) XBee Datasheet. Last seen in 14 February, 2015. https://www.sparkfun.com/datasheets/Wireless/Zigbee/XBee-Datasheet.pdf

[35] (2006) TENMA 72-7732A Datasheet. Last seen in February, 2015. http://www.farnell.com/datasheets/1653479.pdf

[36] (2014) Keysight InfiniiVision MSO-X 2002A Datasheet. Last seen in June, 2015. https://www.upc.edu/sct/es/documents_equipament/d_332_dsox2012a.pdf

[37] J. C. Sprott, *Chaos and Time-Series Analysis*. Oxford University Press, 2003.

[38] SoftwareSerial Library. Last seen in May, 2015. http://www.arduino.cc/en/Reference/SoftwareSerial

[39] PinChangeInt Library. Last seen in May, 2015. https://code.google.com/p/arduino-pinchangeint/

[40] G. Ferro, R. Silva, and L. Lopes, "Towards Out-of-the-Box Programming of Wireless Sensor-Actuator Networks," 2015, manuscript submitted for publication.

Appendix A

Code Blocks

In this Appendix we present the similar implementation of the tasks presented in Chapter 6. All the examples are using the same libraries used in our prototype, which significantly reduce the amount of code written in each example.

Arduino Code A.0.1 Arduino Sensors and Radio Test Program

```
#include "Libraries/RTC/RTC.h"
#include "Libraries/SHT15/SHT15.h"
#include "Libraries/Radio/Radio.h"

#define SHTx_DATA_PIN  7
#define SHTx_CLOCK_PIN 6

volatile bool wake = false;
SHT15 sht15_obj;
SendBuf send_buffer;
float sensor_values[2];

void setup() {
  attachInterrupt(3, wakeAlarm, FALLING);

  sht15_obj(SHTx_DATA_PIN, SHTx_CLOCK_PIN);
  radio::get_my_address();

  time_t time = rtc::time();
  rtc::alarm(time + 5);
  sleep();
}

void loop() {
  sensor_values[0] = sht15_obj.getTemperature();
  sensor_values[1] = sht15_obj.getHumidity();

  send_buffer = radio::new_send_buf(radio::myaddress, 1, 2, sensor_values);
  radio::send_data(send_buffer);

  rtc::alarm(time + 5);
  sleep();
}

void wakeAlarm() {
  wake = true;
}

void sleep() {
  wake = false;

  set_sleep_mode (SLEEP_MODE_IDLE);
  sleep_enable();

  power_adc_disable();
  power_spi_disable();
  power_timer0_disable();
  power_timer1_disable();
  power_timer2_disable();
  power_twi_disable();

  sleep_mode();
  sleep_disable();

  power_all_enable();
}
```

Arduino Code A.0.2 Arduino Radio 64 bytes Program

```
#include "Libraries/RTC/RTC.h"
#include "Libraries/Radio/Radio.h"

volatile bool wake = false;
SendBuf send_buffer;
float radio_values[16];

void setup() {
  attachInterrupt(3, wakeAlarm, FALLING);

  radio::get_my_address();

  for(int i=0; i<16; i++) {
    radio_values[i] = 2.0;
  }

  time_t time = rtc::time();
  rtc::alarm(time + 5);
  sleep();
}

void loop() {

  send_buffer = radio::new_send_buf(radio::myaddress, 1, 16, radio_values);
  radio::send_data(send_buffer);

  rtc::alarm(time + 5);
  sleep();
}

void wakeAlarm() {
  wake = true;
}

void sleep() {
  wake = false;

  set_sleep_mode (SLEEP_MODE_IDLE);
  sleep_enable();

  power_adc_disable();
  power_spi_disable();
  power_timer0_disable();
  power_timer1_disable();
  power_timer2_disable();
  power_twi_disable();

  sleep_mode();
  sleep_disable();

  power_all_enable();
}
```

Arduino Code A.0.3 Arduino Sensors Test Program

```
#include "Libraries/RTC/RTC.h"
#include "Libraries/SHT15/SHT15.h"

#define SHTx_DATA_PIN  7
#define SHTx_CLOCK_PIN 6

volatile bool wake = false;
SHT15 sht15_obj;

void setup() {
  attachInterrupt(3, wakeAlarm, FALLING);

  sht15_obj(SHTx_DATA_PIN, SHTx_CLOCK_PIN);

  time_t time = rtc::time();
  rtc::alarm(time + 5);
  sleep();
}

void loop() {
  float temperature = sht15_obj.getTemperature();
  float humidity    = sht15_obj.getHumidity();

  rtc::alarm(time + 5);
  sleep();
}

void wakeAlarm() {
  wake = true;
}

void sleep() {
  wake = false;

  set_sleep_mode (SLEEP_MODE_IDLE);
  sleep_enable();

  power_adc_disable();
  power_spi_disable();
  power_timer0_disable();
  power_timer1_disable();
  power_timer2_disable();
  power_twi_disable();

  sleep_mode();
  sleep_disable();

  power_all_enable();
}
```

Arduino Code A.0.4 Arduino Computation Test Program

```
#include "Libraries/RTC/RTC.h"

volatile bool wake = false;
bool state = true;

void setup() {
  attachInterrupt(3, wakeAlarm, FALLING);

  time_t time = rtc::time();
  rtc::alarm(time + 5);
  sleep();
}

void loop() {
  float x = 0.2;
  float k = 4.0;

  int i=0;
  while(i<1000) {
    x = k * x * (1.0 - x);
    i++;
  }

  rtc::alarm(time + 5);
  sleep();
}

void wakeAlarm() {
  wake = true;
}

void sleep() {
  wake = false;

  set_sleep_mode (SLEEP_MODE_IDLE);
  sleep_enable();

  power_adc_disable();
  power_spi_disable();
  power_timer0_disable();
  power_timer1_disable();
  power_timer2_disable();
  power_twi_disable();

  sleep_mode();
  sleep_disable();

  power_all_enable();
}
```

Arduino Code A.0.5 Arduino Actuator Test Program

```
#include "Libraries/RTC/RTC.h"

volatile bool wake = false;
bool state = true;
int ledPin = A5;

void setup() {
  attachInterrupt(3, wakeAlarm, FALLING);
  pinMode(ledPin, OUTPUT);

  time_t time = rtc::time();
  rtc::alarm(time + 5);
  sleep();
}

void loop() {
  if(state) {
    digitalWrite(ledPin, HIGH);
  }
  else {
    digitalWrite(ledPin, LOW);
  }
  state = !state;

  rtc::alarm(time + 5);
  sleep();
}

void wakeAlarm() {
  wake = true;
}

void sleep() {
  wake = false;

  set_sleep_mode (SLEEP_MODE_IDLE);
  sleep_enable();

  power_adc_disable();
  power_spi_disable();
  power_timer0_disable();
  power_timer1_disable();
  power_timer2_disable();
  power_twi_disable();

  sleep_mode();
  sleep_disable();

  power_all_enable();
}
```